Retargetable Postpass Optimisation by Integer Linear Programming

Dissertation

Zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

von

Diplom-Informatiker
Daniel Kästner

Oktober, 2000

Tag des Kolloquiums:	22.12.2000
Dekan:	Prof. Dr. R. Schulze-Pillot
Gutachter:	Prof. Dr. R. Wilhelm Prof. Dr.-Ing. L. Thiele
Vorsitzender:	Prof. Dr.-Ing. P. Slusallek

Abstract

Embedded Systems are subject to severe cost restrictions but impose high performance requirements on typical applications. This has lead to the development of specialised irregular hardware architectures for which traditional code generation and optimisation techniques fail to achieve a satisfactory code quality. Therefore assembly programming is still common practice. One reason for this is the interdependence of code generation phases, the so-called phase-coupling problem. Integer linear programming (ILP) allows to integrate several phases in a homogeneous problem description and to solve them simultaneously. In this thesis, two well-structured ILP formulations for the problem of global phase-coupled code optimisation for irregular architectures are presented. The relationship between the design of the hardware architecture and the appropriate ILP modelling style is investigated. In order to speed up the computations, ILP-based approximations are developed that provide a very high solution quality and reduce the computation time significantly compared to the exact solution. The optimisations are implemented in a retargetable framework for postpass optimisations and analyses, called PROPAN. The relevant properties of the target architecture are concisely specified in a novel architecture description language TDL. From the TDL description a hardware-sensitive postpass optimiser is automatically generated that performs efficiency-increasing transformations on assembly code using integer linear programming. The system has been retargeted to several representative standard digital signal processors. Practical experiments demonstrate the applicability of this approach.

Zusammenfassung

Eingebettete Systeme unterliegen engen Kostenschranken, typische Anwendungen stellen jedoch hohe Leistungsanforderungen. Dies hat zur Entwicklung spezialisierter irregulärer Hardwarearchitekturen geführt, für die traditionelle Codeerzeugungs- und -optimierungsverfahren keine zufriedenstellenden Ergebnisse erzielen. Anwendungsprogramme für eingebettete Systeme werden daher oft in Assembler programmiert. Eine Ursache hierfür ist die gegenseitige Abhängigkeit der Codeerzeugungsphasen, das sogenannte Phasenkopplungsproblem. Ganzzahlige lineare Programmierung (ILP) jedoch ermöglicht es, verschiedene Teilprobleme in einer homogenen Problembeschreibung zu integrieren und gemeinsam zu lösen. In der vorliegenden Arbeit werden zwei strukturierte ILP-Formulierungen zur globalen, phasengekoppelten Codeoptimierung für irreguläre Architekturen vorgestellt. Der Zusammenhang zwischen dem Hardwareentwurf der Zielarchitektur und der geeigneten Modellierungsmethode für ganzzahlige lineare Programme wird herausgestellt. Zur Beschleunigung der Berechnungen werden ILP-basierte Approximationen entwickelt, die eine sehr hohe Codequalität erzielen und die Berechnungszeit im Vergleich zur exakten Lösung deutlich senken. Die Optimierungen wurden in einem retargierbaren System für Postpassoptimierungen und -analysen, genannt Propan, implementiert. Eine neue Architekturbeschreibungssprache Tdl ermöglicht eine kurze und prägnante Spezifikation der relevanten Hardwareeigenschaften der Zielarchitektur. Aus der Tdl-Beschreibung wird ein hardwaresensitiver Postpassoptimierer generiert, der durch Einsatz ganzzahliger linearer Programmierung effizienzsteigernde Transformationen von Assemblercode durchführt. Das System wurde für verschiedene repräsentative Standardprozessoren retargiert. Die experimentellen Ergebnisse belegen die Anwendbarkeit dieses Verfahrens.

Extended Abstract

During the last years, the markets for telecommunication, embedded systems, and multimedia applications have been rapidly growing. The distinct cost sensitivity of these markets in connection with the stringent time constraints of real-time applications have led to the development of specialised, irregular hardware architectures designed to efficiently execute typical applications of digital signal processing. In the area of general-purpose processors, compiler technology has reached a high level of maturity. For irregular architectures however, the code quality achieved by traditional high-level language compilers is often not satisfactory. Generating efficient code for irregular architectures requires highly optimising techniques that have to be aware of specific hardware features of the target processor. Since such techniques are usually not provided by standard compilers, many digital signal processing applications are developed in assembly language. The increasing complexity of typical applications and the shrinking design cycles of embedded processors render this approach increasingly unacceptable. Therefore, there is an urgent need for retargetable code generation and optimisation techniques that can be quickly adapted to different target architectures and can provide a high code quality.

In this thesis the PROPAN system is presented as a retargetable framework for high-quality code optimisations and machine-dependent program analyses at assembly level. In the past, research on retargetability has mainly focused on closed compilation systems. Using such a system in industry however mostly requires replacing the existing compiler infrastructure which causes high costs. Thus the use of retargetable compilers in industry is rare. To the best of our knowledge, PROPAN is the first postpass framework where the issues of easy retargetability and of high-quality code optimisations have been combined. Due to the postpass orientation, PROPAN can be integrated in existing tool chains with moderate effort and allows to improve the code quality of existing compilers. Thus the costs associated with changing the compiler infrastructure in a company can be avoided. The retargetability concept of PROPAN is based on a combination of generic and generative mechanisms. We have developed a new machine description language called TDL which allows to specify the hardware resources of the target processor, its instruction set, and its assembly language in a concise way. Apart from the assembly orientation the main innovation of TDL is the generic modelling of irregular hardware constraints that allows them to be exploited in generic search-based optimisation algorithms. From the TDL description a parser for the assembly language and an architecture database are generated. The architecture database consists of a set of ANSI C files containing data structures representing all relevant information about the target architecture and functions to initialise, access and manipulate this information. The core system of PROPAN has been implemented in a generic, i. e., machine-independent way; if target-specific information is required, the generated architecture database is referenced. For each target architecture the architecture database is linked with the generic core system yielding a dedicated hardware-sensitive postpass optimiser.

Generating high-quality code for irregular architectures requires taking the phase-coupling problem into account. In general, the code generation process is subdivided into several phases. For complexity reasons they are usually addressed separately by heuristic methods. While the heuristic approaches perform well for regular architectures, suboptimal combinations of suboptimal partial results can lead to poor code quality for irregular architectures. In this thesis a new approach is presented that unifies a well-known subset of the code generation phases at assembly level. The use of integer linear programming allows a generic homogeneous modelling of the problem of phase-coupled code optimisation taking into account irregular hardware characteristics of individual target architectures. The theory of integer linear programming has lead to sophisticated solution techniques such that powerful tools are available for computing solutions of integer linear programs. In our work we have developed extensions of two well-structured ILP formulations for phase-coupled instruction scheduling, register assignment and resource allocation. We have investigated the structure of both ILP formulations and the relationship between ILP modelling styles and hardware architectures. One formulation is based on an order-indexed modelling. It supports an efficient integration of the register assignment task for architectures with severely restricted instruction-level parallelism. The other formulation uses a time-indexed modelling. It provides an efficient integration of the problems of instruction scheduling and resource allocation. It is well suited for architectures where a large amount of alternative functional units has to be taken into account. In contrast to most other exact code generation methods the scope of the optimisations presented in this thesis is not restricted to single basic blocks. In order to speed up the computation process we have developed dedicated ILP-based approximations. Thus the generated integer linear programs can be solved either exactly providing provably optimal solutions, or by the use of the approximative methods. The basic idea of the approximations is the iterative solution of partial relaxations of the original problem. They allow to reduce the computation time significantly and still provide a very high solution quality.

The PROPAN framework has been retargeted to a wide range of architectures. It has been used to generate ILP-based postpass optimisers for two widely used modern digital signal processors with considerably different hardware characteristics, the Analog Devices ADSP-2106x [Ana95] and the Philips TriMedia TM1000 [Phi97]. PROPAN can also be used as a platform for generic program analyses and user-supplied hardware-dependent program optimisations. It is integrated in a framework for calculating worst-case execution time guarantees for real-time systems [FKL+99]; in this context a TDL specification of the Infineon TriCore μC/DSP [Inf00] has been developed. For the Infineon C16x [Sie96] microprocessor family PROPAN has been used as a platform for implementing hardware-sensitive postpass optimisations that are part of a commercial postpass optimiser [Abs00a]. Various aspects of this thesis have been presented in a number of publications [KL98, KL99, Käs99a, KW99, FKL+99, Käs00a, Käs00b].

Ausführliche Zusammenfassung

Innerhalb der letzten Jahre hat ein starkes Wachstum der Märkte für Telekommunikation, eingebettete Systeme und Multimediaanwendungen stattgefunden. Die ausgeprägte Kostensensitivität dieser Märkte in Verbindung mit strengen Zeitschranken, denen Realzeitanwendungen unterliegen, haben zur Entwicklung spezialisierter, irregulärer Hardwarearchitekturen geführt. Diese werden speziell zur effizienten Ausführung von Algorithmen der digitalen Signalverarbeitung entworfen. Auf dem Gebiet der Allzweckprozessoren hat die Compilertechnologie eine hohe Reife erreicht. Für irreguläre Architekturen jedoch ist die von traditionellen Hochsprachencompilern erzeugte Codequalität in der Regel nicht ausreichend. Die Erzeugung effizienten Maschinencodes für irreguläre Architekturen erfordert hochoptimierende Techniken, die an spezielle Hardwareeigenschaften des Zielprozessors angepaßt sein müssen. Da solche Verfahren in Standardcompilern üblicherweise nicht eingesetzt werden, werden viele Anwendungen der digitalen Signalverarbeitung in Assembler programmiert. Aufgrund der steigenden Komplexität typischer Anwendungsprogramme und der schrumpfenden Produktzyklen eingebetteter Prozessoren wird dies jedoch zunehmend inakzeptabel. Daher besteht ein dringender Bedarf an retargierbaren Codeerzeugungs- und -optimierungstechniken, die schnell an unterschiedliche Zielarchitekturen angepaßt werden können und in der Lage sind, eine hohe Codequalität zu erreichen.

In der vorliegenden Arbeit wird das Propan-System als ein retargierbares System für hochleistungsfähige Codeoptimierungen und maschinenabhängige Programmanalysen auf Assemblerebene vorgestellt. In der Vergangenheit hat sich die Forschung im Bereich retargierbarer Techniken hauptsächlich auf geschlossene Übersetzungssysteme konzentriert. Der industrielle Einsatz solcher Systeme impliziert jedoch in der Regel einen Austausch der Compilerinfrastruktur, was mit hohen Kosten verbunden ist. Daher sind retargierbare Compiler in der Industrie kaum anzutreffen. Unserer Kenntnis nach ist Propan das erste postpass-orientierte System, das die Ziele leichter Retargierbarkeit und suchbasierter Codeoptimierungstechniken kombiniert. Aufgrund der Postpassorientierung kann Propan mit geringem Aufwand in existierende Toolketten integriert werden und ermöglicht die Verbesserung der Codequalität existierender Compiler. Dadurch werden die mit dem Austausch der Compilerinfrastruktur verbundenen Kosten vermieden. Das Retargierbarkeitskonzept von Propan beruht auf der Kombination generischer und generativer Techniken. Im Rahmen dieser Arbeit wird eine neue Maschinenbeschreibungssprache, Tdl, vorgestellt, die die Spezifikation der Hardwareressourcen eines Zielprozessors, seines Instruktionssatzes und der verwendeten Assemblersprache in kurzer und prägnanter Form ermöglicht. Abgesehen von der Assemblerorientierung besteht die wichtigste Innovation von TDL in der generischen Modellierung irregulärer Hardwareeigenschaften, die es ermöglicht, diese in generischen suchbasierten Optimierungsalgorithmen zu berücksichtigen. Aus der TDL-Beschreibung wird ein Parser für die spezifizierte Assemblersprache und eine Architekturdatenbank in Form von ANSI-C Dateien generiert. Darin werden Datenstrukturen definiert, in denen alle relevanten Informationen über die Zielarchitektur repräsentiert sind sowie Funktionen zum Zugriff, zur Initialisierung und zur Manipulation dieser Datenstrukturen. Das Kernsystem von PROPAN ist generisch, d. h. maschinenunabhängig implementiert. Werden architekturspezifische Informationen benötigt, wird die generierte Architekturdatenbank referenziert. Für jede Zielarchitektur wird die Architekturdatenbank mit dem generischen Kernsystem zusammengelinkt und es ergibt sich ein spezieller hardwaresensitiver Postpassoptimierer.

Zur Erzeugung einer hohen Codequalität für irreguläre Architekturen ist die Berücksichtigung des Phasenkopplungsproblems erforderlich. Im allgemeinen wird der Codeerzeugungsprozeß in verschiedene Teilaufgaben untergliedert. Aus Komplexitätsgründen werden diese üblicherweise getrennt voneinander durch heuristische Verfahren gelöst. Die Codequalität der heuristischen Verfahren ist für reguläre Architekturen in der Regel ausreichend. Bei irregulären Architekturen jedoch kann es aufgrund der suboptimalen Kombination suboptimaler Teillösungen zur Erzeugung ineffizienten Maschinencodes kommen. In der vorliegenden Arbeit wird ein neuer Ansatz vorgestellt, der verschiedene Teilaufgaben der Codeerzeugung auf Assemblerebene vereinigt. Die Verwendung ganzzahliger linearer Programmierung ermöglicht eine generische homogene Modellierung des Problems der phasengekoppelten Codeoptimierung unter Berücksichtigung irregulärer Hardwareeigenschaften der jeweiligen Zielarchitektur. Die Theorie der ganzzahligen linearen Programmierung hat zu hochentwickelten Lösungstechniken geführt, so daß leistungsfähige Tools zum Lösen ganzzahliger linearer Programme verfügbar sind. Im Rahmen dieser Arbeit werden Erweiterungen zweier strukturierter ILP-Formulierungen für phasengekoppelte Instruktionsanordnung, Registerzuteilung und Ressourcenallokation vorgestellt. Die Struktur beider ILP-Formulierungen wird untersucht und der Zusammenhang zwischen ILP-Modellierung und Hardwareentwurf der Zielarchitektur herausgestellt. Eine Formulierung basiert auf einer reihenfolge-bezogenen Indizierung. Sie ermöglicht eine effiziente Modellierung des Registerzuteilungsproblems und ist besonders für Architekturen mit eingeschränkten Parallelverarbeitungskapazitäten geeignet. Die andere Formulierung verwendet eine zeitpunktbezogene Modellierung. Sie erlaubt eine effiziente Koppelung von Instruktionsanordnung und Ressourcenallokation und ist für Architekturen mit einer großen Anzahl alternativer funktionaler Einheiten geeignet. Im Gegensatz zu früheren exakten phasengekoppelten Codeerzeugungsverfahren sind die in dieser Arbeit vorgestellten Optimierungen nicht auf Basisblockgrenzen beschränkt. Ein weiterer Schwerpunkt liegt auf der Entwicklung ILP-basierter Approximationen durch die

die Berechnungen beschleunigt werden können. Die erzeugten ganzzahligen linearen Programme können somit entweder exakt gelöst werden, wodurch beweisbar optimale Ergebnisse erzielt werden, oder durch Einsatz der approximativen Techniken. Die Grundidee der Approximationen besteht in der schrittweisen Lösung partieller Relaxationen des Originalproblems. Sie erlauben eine deutliche Reduktion der Berechnungszeit und erzielen dennoch eine sehr hohe Codequalität.

Das PROPAN System wurde für verschiedene repräsentative Standardprozessoren retargiert. Es wurde zur Erzeugung ILP-basierter Postpassoptimierer für zwei moderne digitale Signalprozessoren mit sehr unterschiedlichen Architektureigenschaften eingesetzt, den Analog Devices ADSP-2106x SHARC [Ana95] und den Philips TriMedia TM1000 [Phi97]. PROPAN kann auch als Plattform für generische Programmanalysen und benutzerdefinierte hardwareabhängige Programmoptimierungen eingesetzt werden. PROPAN ist in ein System zur Berechnung von Laufzeitgarantien in Realzeitsystemen integriert [FKL+99]; in diesem Zusammenhang wurde eine TDL-Beschreibung des Infineon TriCore μC/DSP [Inf00] entwickelt. Für die Infineon C16x-Mikroprozessorfamilie [Sie96] wurde PROPAN als Ausgangsbasis zur Implementierung hardwaresensitiver Postpassoptimierungen eingesetzt, die Teil eines kommerziellen Postpassoptimierers sind [Abs00a]. Verschiedene Aspekte dieser Arbeit wurden in einer Reihe von Veröffentlichungen vorgestellt [KL98, KL99, Käs99a, KW99, FKL+99, Käs00a, Käs00b].

Acknowledgements

I would like to thank my advisor Prof. Dr. Reinhard Wilhelm for his invaluable advice and his guidance and support throughout my course of graduate study. His comments often unveiled new interesting aspects and perspectives. He always left me a lot of freedom and contributed much to an enjoyable and productive working atmosphere.

I like to thank Christian Ferdinand for his advice and many fruitful suggestions and comments. This work has greatly benefited from his insight and experience. The implementation uses several modules developed by Marc Langenbach; special thanks goes to him for years of excellent cooperation, the pleasant and productive working atmosphere, and many stimulating discussions. For careful proof reading of this thesis in different stages I thank Reinhold Heckmann, Marc Langenbach, Christian Ferdinand, Stephan Diehl, Friedrich Eisenbrand and Florian Martin.

I thank Henrik Theiling for adapting his control flow reconstruction algorithm to the assembly setting. I am also grateful to Nicolas Fritz and Stephan Wilhelm who were the first users of the PROPAN system. They suffered through several bugs and shortcomings and their comments helped improving the system. A word of thanks also goes to Steven Bashford for interesting discussions and to all members of the chair of programming languages and compiler construction for the inspiring and cooperative working atmosphere.

Also, I wish to thank the Deutsche Forschungsgemeinschaft for supporting this research by a graduate fellowship in the Graduiertenkolleg “Effizienz und Komplexität von Algorithmen und Rechenanlagen” at Saarland University.

I would like to thank Prof. Dr. Kurt Mehlhorn and the Max Planck Institute for Informatik of Saarbrücken for granting access to their CPLEX installation and their compute server. I gratefully acknowledge the support of Hans Rieder from the Fraunhofer IZFP who made available the `g21k` compiler for the SHARC and provided industry-relevant hand-crafted assembly programs for the experimental evaluation. I like to thank the Philips research group Eindhoven, especially Joachim Trescher and Zbigniew Chamski, for making available the Philips tmcc compiler and the development environment for the TM1000.

Finally I would like to thank my parents Hans-Dieter and Josefa and my girlfriend Sylvie for their patience and their support.

Contents

1. Introduction

During the last years, the markets for telecommunication, embedded systems, and multimedia applications have been rapidly growing. The distinct cost sensitivity of those markets in connection with the stringent time constraints of real-time applications have led to the development of specialised, irregular hardware architectures designed to efficiently execute typical applications of digital signal processing. Common characteristics of those architectures are, e. g., heterogeneous register files, support for low-overhead looping, and restricted interconnectivity of functional units and register sets. In the area of general-purpose processors, compiler technology has reached a high level of maturity. For irregular architectures however, the code quality achieved by traditional high-level language compilers often is not satisfactory [SCL96, ZVSM94]. Generating efficient code for irregular architectures requires highly optimising techniques which have to be aware of specific hardware features of the target processor. Since such techniques are usually not provided by standard compilers, many digital signal processing applications are developed in assembly language [SCL96]. Assembly programming is a time consuming and error-prone task that suffers from bad portability and bad maintainability thus reducing the productivity of software development. Due to the increasing complexity of embedded applications and the shrinking design cycles of embedded processors, the usage of high-level programming languages becomes more and more imperative. As a consequence a growing demand for compilation techniques that can produce high-quality code for irregular architectures has risen.

Most digital signal processors dispose of special functionality; identifying special functionality however requires bringing much architectural information into the compiler. Developing a dedicated compiler for each processor is prohibitive due to the short design cycles and the cost sensitivity of the embedded markets. Thus retargetable code generation and optimisation techniques that can be quickly adapted to different target architectures and are able to generate high-quality code for each individual target have become an active area of research [MG95].

The process of generating code for high-level language programs can be subdivided into several phases: code selection, register allocation, instruction scheduling, register assignment, and functional unit binding. Since most of these subtasks are NP-complete problems, in classical code generation methods they are addressed in separate phases by heuristic algorithms. Unfortunately the code generation phases are interdependent; decisions made in one phase impose constraints to the subsequently addressed phases. For regular architectures the quality of the classical heuristic methods is satisfactory. For irregular architectures however the interdependencies between the phases usually lead to a significant decrease of code quality [ZSWS95] due to the suboptimal combination of suboptimal partial results. In heuristic phase-coupling methods a heuristic algorithm for one code generation phase typically is extended to address other subtasks based on heuristic estimates of the phase interactions. As a consequence the quality of the generated code strongly depends on the choice of the heuristics. The quality of the individual heuristics and their combined effects in turn strongly depend on the target architecture. Thus there is a conflict between the goals of retargetability and heuristic generation of high-quality code.

Search-based techniques such as integer linear programming allow to model the interactions of code generation phases in an exact way. In the last decade, the use of integer programming models has increased significantly which is mostly due to the advances in algorithms for solving integer programs and the availability of reliable software packages [JNS97]. Computing an optimal solution of an integer linear program is NP-complete [GJ79]. Nevertheless many large instances of such problems can be solved. Recent research has lead to an understanding of polyhedral properties that can be used to develop well-structured formulations permitting efficient computations [CWM94, Bal98]. Other advances have made it possible to improve the efficiency of ILP solving techniques by curtailing the necessary enumeration process [JNS97, Eis00]. The use of integer linear programming for phase-coupled code generation however is still rare.

Although during the last years several retargetable research compilers have been developed they are rarely used in industry. To the best of our knowledge the only commercially available retargetable compiler developed for embedded systems is the CHESS compiler [LVPK+95]. One reason is the problem of simultaneously realizing the goals of retargetability and of generating high-quality code. Another reason is that using such a system in industry mostly requires replacing the existing compiler infrastructure which causes high costs. Therefore postpass techniques are a very attractive solution since they allow to improve the quality of previously generated machine or assembly code without requiring to change the complete compilation system. The legacy compiler can be kept and yet the efficiency of the generated software is increased leading to a large cost benefit. Another advantage of postpass methods is that they make optimisations of compiler-intrinsic functions possible. Compiler-intrinsic functions are often used to embed manually written assembly code into high-level language programs. For high-level language compilers they usually represent barriers across which no program optimisation is possible. However this does not concern postpass approaches since they work on machine level anyway. Moreover the machine level is a natural stage for optimisations aiming at exploiting hardware-specific functionality of digital signal processors. Previous studies [KL99] and industrial experience [Abs00a] have shown that postpass methods can be integrated into existing tool chains with moderate effort.

In embedded systems, the software often has to meet specific requirements which necessitate complex program analyses. In real-time systems, e. g., it must be guaranteed that tight time constraints will be met by every execution of a program. The calculation of tight time bounds for modern architectures is impeded by the use of caches and pipelines. The consequence is that hardware-dependent program analyses have to be performed which—as e. g. the static cache behaviour prediction of [Fer97]—are necessarily postpass analyses. The results of such analyses can also be used for program optimisations, e. g. by cache-sensitive task scheduling algorithms [KT98, KT99].

1.1. The PROPAN System

Our contribution to this situation is the Propan system (*Postpass-oriented Retargetable OPtimiser and ANalyser*) that has been designed as a retargetable framework for postpass optimisations and analyses. To the best of our knowledge, Propan is the first system where the issues of machine description driven retargetability and of search-based postpass optimisations have been combined.

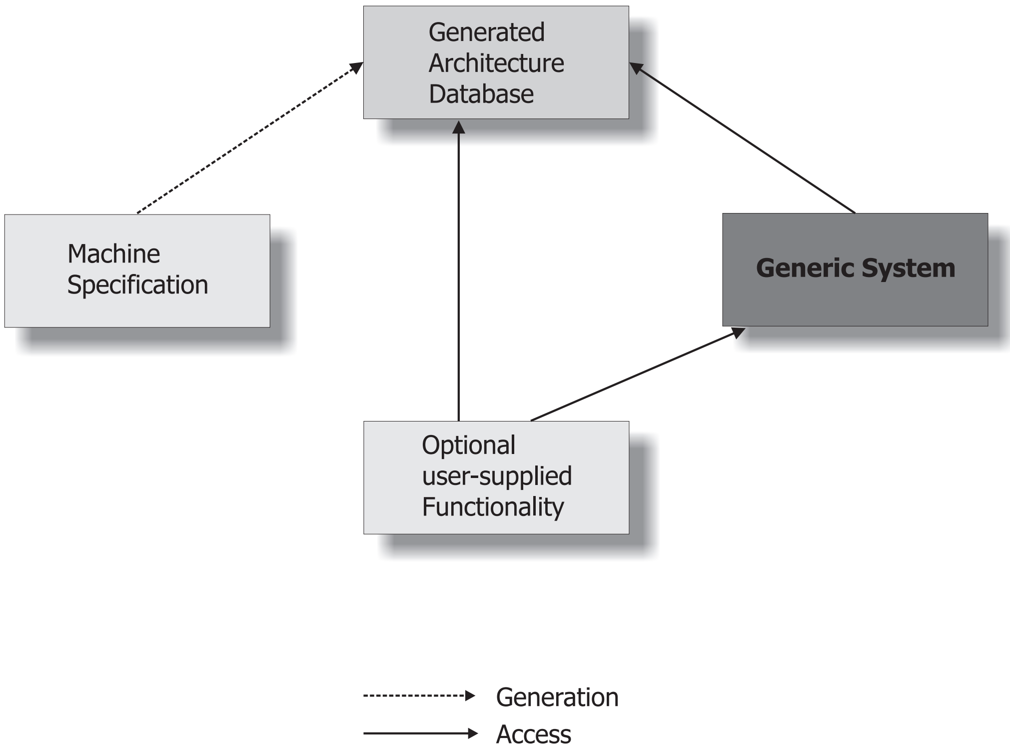


Figure 1.1.: The retargetability concept.

The retargetability concept of Propan is based on the combination of generic and generative techniques. An illustration is given in Fig. 1.1 in an abstract form. The core system is composed from generic and generated program parts. Generic program parts are independent from the target architecture and can be used for different processors without any modification. If hardware-specific knowledge is required all information is retrieved in a standardised way from an architecture 'database' that is generated from a central machine description. The program parts that change with the specific target architecture are automatically generated from the machine description. Thus retargeting the system to a different architecture only requires an appropriate specification of the target machine. Additionally the integration of user-supplied functionality is supported. This may include dedicated program analyses or hardware-specific program transformations. Those external program parts can communicate with the core system via a well-defined file interface and can access the architecture database generated from the machine description.

A detailed overview of the PROPAN framework is shown in Fig. 1.2. The input of PROPAN consists of a TDL-description of the target machine and of the assembly programs that are to be analysed or optimised. TDL (*Target Description Language*) is a dedicated machine description language that allows to specify the hardware resources of the target processor, its instruction set, the assembly language and irregular hardware constraints. Apart from the assembly orientation the main innovation of TDL is the generic modelling of the irregular hardware constraints that allows them to be exploited in generic search-based optimisation algorithms. The TDL specification is processed once for each target architecture; from the TDL description a parser for the specified assembly language and an architecture database are generated. The architecture database consists of a set of ANSI-C files where data structures representing all specified information about the target architecture and functions to initialise, access and manipulate them are defined. The core system of PROPAN is generic; if hardware-specific knowledge is required the architecture database is referenced. For each target architecture, the generic core system is linked with the generated files yielding a dedicated hardware-sensitive postpass optimiser.

The parser reads the input programs and computes their control flow graphs that are represented in a generic intermediate language called CRL (*Control Flow Representation Language*) [Lan99]. The input format is not restricted to assembly files; it is also possible to specify the output format of disassemblers reading executable files, or textual representations of compiler-specific intermediate formats. The CRL interface serves as the main interface for all optimisation and analysis algorithms including additional user-supplied algorithms. From the control flow graph, the necessary program representations as, e. g., the data dependence and the control dependence graphs are calculated by generic algorithms. If required, a register renaming algorithm is executed that replaces references to physical registers of the input program by references to virtual registers. This way spurious data dependences limiting the available parallelism are removed.

The central part of the PROPAN system is the modelling of phase-coupled code optimisation by integer linear programming. A set of fundamental code generation subtasks is identified that can still be addressed on assembly level and phase-coupling methods are investigated that allow to compute high-quality solutions in a generic way without the necessity of reimplementing parts of the source code. In-

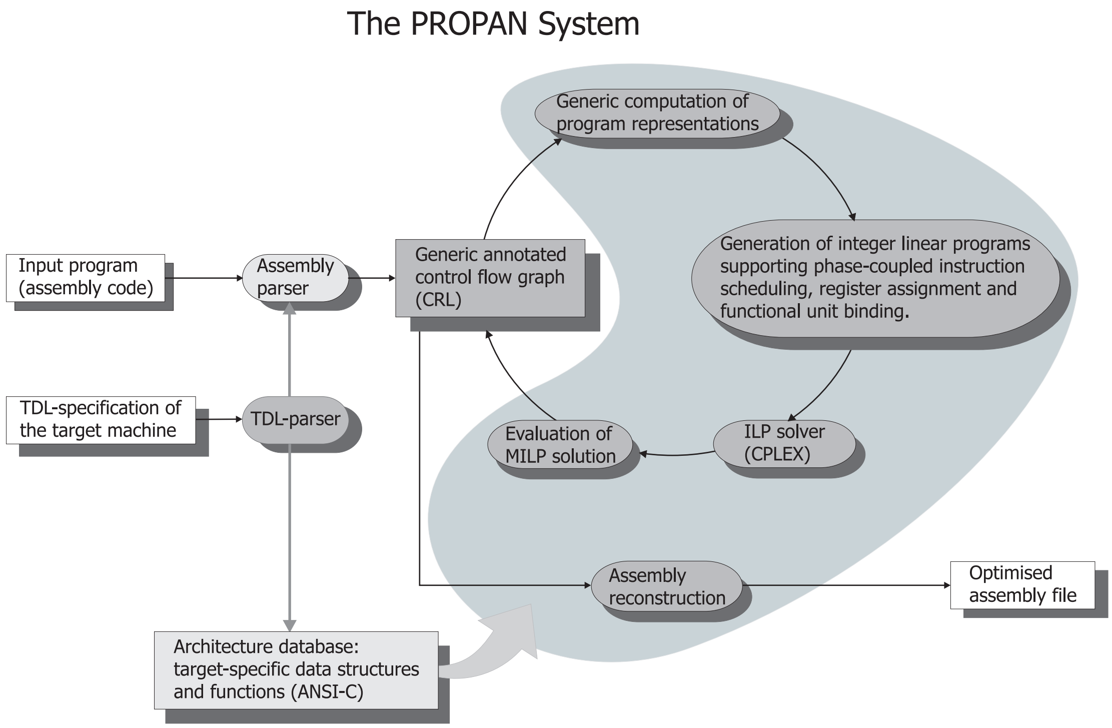


Figure 1.2.: The PROPAN system.

teger linear programming allows a homogeneous problem description that can fully integrate the tasks of instruction scheduling, register reassignment, and functional unit binding, or subsets thereof. Our approach is based on two well-structured ILP models developed in the area of architectural synthesis [GE92, GE93, Zha96]. We have adapted those models to the postpass optimisation problem and extended them to incorporate irregular hardware characteristics and exceed basic block boundaries. Among those models, the most appropriate formulation can be chosen individually for each target architecture. For each input program a dedicated integer linear program is generated that models the execution of the program on the specified target architecture[1]. In contrast to most previous approaches for search-based code generation the optimisation scope is not restricted to basic blocks. A novel superblock concept allows to extend the optimisation scope across basic block and loop boundaries. For each superblock, an individual integer linear program is generated.

Since integer linear programming is an NP-complete problem, the time for computing a provably optimal solution can grow high for large input programs. Therefore a set of ILP-based approximations has been developed that can reduce the computation time significantly and still obtain high-quality solutions. The basic idea of the ILP-based approximations is the iterative solution of heuristically

[1]The incorporation of the register assignment task however can be considered promising only for the SILP formulation (see Chap. 5).

determined relaxations of the original problem. In the last step of the optimisation process the solutions of the integer linear programs are evaluated and the optimised assembly file is generated.

1.2. Overview of this thesis

In the next chapter an overview of code generation and optimisation is given. The fundamental code generation phases are introduced and the problem of code generation for irregular architectures is illustrated. Chap. 3 gives a classification of modern microprocessors that focuses on design characteristics and application areas of digital signal processors as one class of processors characterised by irregular architectures. Chap. 4 gives a short introduction into integer linear programming where the most important mathematical concepts required for the scope of this thesis are summarised. The ILP models used for code generation are presented in Chap. 5. Proofs of important properties of their polytope structure are given, the incorporation of the control flow structure of programs is described and the dependence between ILP modelling styles and hardware architectures is pointed out. In Chap. 6 novel ILP-based approximations are presented that allow to reduce the computation time while still retaining a high solution quality. Chap. 7 is dedicated to the superblock mechanism that allows to extend the optimisation scope across basic block boundaries. After the definition of the underlying concepts, the required extensions of the ILP models are presented and evaluated. In Chap. 8 the hardware description language TDL is presented in detail. Special attention is paid to the generation of integer linear constraints from a specification of architectural irregularities in the form of logical conditions. Implementation details of the PROPAN system are presented in Chap. 9. An overview of the modelled architectures is given in Chap. 10, followed by the evaluation of our experimental analyses.

Surveys of relevant publications specifically related to individual chapters of this thesis are given at the beginning of the corresponding chapters. A separated survey of retargetable compilation systems and of code generation and optimisation frameworks for irregular architectures is given in Chap. 11.

Finally, Chap. 12 concludes and gives an outlook to future research. The appendix contains a summary of the instruction sets of the Analog Devices ADSP-2106x SHARC and the Philips TriMedia TM1000, together with excerpts from their TDL specifications. A list of symbols is given at the end of this thesis.

2. The Code Generation Problem

Compilers for high-level programming languages aim at transforming input programs written in a certain source language into a semantically equivalent program in some target language, usually the machine or assembly code of a target processor. Conceptually, the process of compiling can be subdivided into several phases. In an initial phase, often called the compiler frontend, the syntactic structure and static semantic properties of the source program are computed. The results of this phase comprise either messages about syntactic or semantic errors in the program or an appropriate representation of the syntactic structure and the static semantic properties of the program. Subsequently many compilers perform efficiency-increasing program transformations on this representation that, to a large degree, are machine independent. This includes tasks like constant folding, elimination of common subexpressions, elimination of dead code, loop-invariant code motion, etc. The corresponding phase is often called the middle-end of the compiler. Then a synthesis phase takes the intermediate representation produced by the middle-end and converts it into semantically equivalent target machine code. An important part of the synthesis phase is the task of code generation which consists of several subtasks:

- The goal of *code selection* is to map the intermediate representation to a semantically equivalent sequence of machine operations.

- The task of *register allocation* is to map the values of the intermediate representation to physical registers in order to minimise the number of memory references during program execution. It consists itself of two subtasks: *Register allocation proper* attempts to decide which variables and expressions of the intermediate representation are mapped to registers and which ones are kept in memory. The second subtask is called *register assignment*; its goal is to determine the physical registers that are used to store the values that have been previously selected to reside in registers.

- *Instruction scheduling* is the task of reordering the produced instruction stream in order to minimise pipeline stalls and to exploit the available instruction-level parallelism.

- *Resource allocation* is concerned with binding operations to machine resources, e. g. functional units or buses; it is also called *functional unit binding*.

This task is only relevant for architectures where operations can be explicitly assigned to functional units and where the binding can impose constraints to register assignment and instruction scheduling.

All code generation tasks have high worst case complexity; in general, code selection, register allocation, register assignment and instruction scheduling are NP-complete [GJ79] problems. Therefore, traditional code generation approaches rely on heuristic methods. Furthermore, a modular decomposition of code generation is advisable for reasons of software complexity. Thus in traditional approaches all code generation phases are addressed isolatedly by separate algorithms. Unfortunately this can lead to a suboptimal combination of suboptimal partial solutions resulting in a very poor code quality, especially for irregular architectures.

The remainder of this chapter is organised as follows: in Sec. 2.1 an overview of the fundamental program representations is given that are computed from the intermediate representation of the program and constitute the input of most subtasks of code generation. Subsequently the individual code generation phases are presented in more detail and the phase coupling problem is described. In Sec. 2.3, the problems associated with the compilation for irregular architectures and their consequences to the code generation process are summarised.

2.1. Fundamental Program Representations

The program representations introduced in this chapter can be defined either on source level, or on machine level. For the scope of this thesis it is more convenient to choose the machine-level representations. Therefore, it is necessary to introduce the concept of machine operations. In the terminology of [LDS80, Bas95] a *micro-operation*, or *machine operation*, is an elementary operation that can be executed by the target processor. The notion of machine *operation* has to be distinguished from the concept of machine *instructions*. In some architectures exhibiting intraprocessor parallelism, especially in VLIW[1] architectures, several machine operations can be combined to form one machine instruction. The execution of all operations contained in the same instruction is started in parallel. In the following a short summary of the most important concepts is given; more detailed explanations and additional literature references can be found in [WM95, Bas95].

The control flow graph of a procedure indicates which instructions can be executed one after the other. Whether this actually occurs during program execution may depend on conditions which in general cannot be evaluated at compilation time.

Definition 2.1 (Control Flow Graph) *The control flow graph of a procedure is a directed graph $G_C = (N_C, E_C, n_A, n_\Omega)$ with node and edge labels. For each instruction i of the procedure there is a node $n_i \in N_C$ that is marked by i. The edges*

[1] Very Long Instruction Word

(n, m, λ) denote the control flow of the procedure; $\lambda \in \{T, F, \epsilon\}$ is the edge label. The subgraphs representing loops, conditional branches and sequential program flow are shown in Fig. 2.1. Edges belonging to unconditional branches lead from the

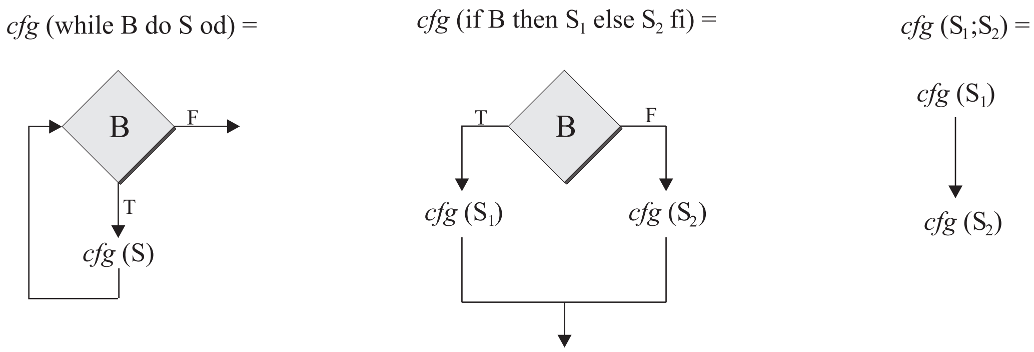


Figure 2.1.: Control flow graph for composed statements. A subgraph $G'_C = (N'_C, E'_C, n'_A, n'_\Omega)$ is inserted as follows: All incoming edges lead to n'_A, all outgoing edges to the successor node uniquely determined by the execution context.

node of the branch to the branch destination. The node $n_A \in N_C$ is the uniquely determined entry point into the procedure; it belongs to the first instruction to be executed. n_Ω denotes the end node that is reached by any path through the control flow graph. Nodes with more than one predecessor are called joins *and nodes with more than one successor are called* forks.

Definition 2.2 (Path) *A path π from node n_1 to node n_k in a directed graph $G = (N, E)$ is a sequence of edges, beginning with a node $n_1 \in N$ and ending in $n_k \in N$ where $\pi = (n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k)$ and $(n_i, n_{i+1}) \in E$ for $i = 1, \ldots, k-1$. The length of π is defined as the number of edges on π, i. e. $l(\pi) = k - 1$.*

Definition 2.3 (Basic Block) *A basic block in a control flow graph is a path of maximal length which has no joins except possibly at the beginning and no forks except possibly at the end.*

If the first instruction of a basic block is executed, then in case of error-free execution (no runtime errors, exceptions, etc.) all other operations of the basic block are executed as well.

Definition 2.4 (Basic Block Graph) *The basic block graph $G_B = (N_B, E_B, b_A, b_\Omega)$ of a control flow graph $G_C = (N_C, E_C, n_A, n_\Omega)$ is formed from G_C by combining each basic block into a node. Edges of G_C leading to the first node of a basic block, lead to the node of that basic block in G_B. Edges of G_C leaving the last node of a basic block, lead out of the node of that basic block in G_B. The node b_A denotes the uniquely determined entry block of the procedure; b_Ω denotes the exit block that is reached at the end of any path through the procedure.*

When performing global analyses or optimisations the control structure of the procedure has to be taken into account. As an example, moving an operation from the *then*-block of a conditional branch to the *else*-block must be prevented in order to preserve program semantics. The control structure of a procedure is represented by the control dependence graph G_{CD}. Defining the control dependence graph requires several other definitions to be given first.

Definition 2.5 (Dominator) *Let a control flow graph $G_C = (N_C, E_C, n_A, n_\Omega)$ be given. A node $n \in N_C$ dominates a node $m \in N_C$, $n\ \Delta_d\ m$, if and only if each path from the entry node n_A of the procedure to m contains the node n. Each node dominates itself.*

Definition 2.6 (Immediate Dominator) *Let a control flow graph $G_C = (N_C, E_C, n_A, n_\Omega)$ be given. A node $n \in N_C$ is an immediate dominator of $m \in N_C$, if and only if*

- $n\ \Delta_d\ m$
- $\nexists z:\ n\ \Delta_d\ z\ \wedge\ z\ \Delta_d\ m\ \wedge\ z \neq n\ \wedge\ z \neq m.$

Definition 2.7 (Dominator Tree) *The dominator tree T_d of a control flow graph is a tree containing all nodes of the control flow graph G_C. Its root is the entry node n_A of the procedure. There is an edge between n and m if and only if n is immediate dominator of m.*

Definition 2.8 (Postdominator) *Let a control flow graph $G_C = (N_C, E_C, n_A, n_\Omega)$ be given. A node $n \in N_C$ postdominates a node $m \in N_C$, $n\ \Delta_p\ m$, if and only if each path from m to n_Ω contains the node n. A node never postdominates itself.*

Definition 2.9 (Immediate Postdominator) *Let a control flow graph $G_C = (N_C, E_C, n_A, n_\Omega)$ be given. A node $n \in N_C$ is an immediate postdominator of $m \in N_C$, if and only if*

- $n\ \Delta_p\ m$
- $\nexists z:\ n\ \Delta_p\ z\ \wedge\ z\ \Delta_p\ m\ \wedge\ z \neq n.$

Definition 2.10 (Postdominator Tree) *The postdominator tree T_p of a control flow graph is a tree containing all nodes of the control flow graph G_C. Its root is the exit node n_Ω of the procedure. There is an edge between n and m if and only if n is the immediate postdominator of m.*

The postdominator tree can be calculated as the dominator tree of the inverse control flow graph; an algorithm is given in [Käs97, ASU86].

Definition 2.11 (Control Dependence) *Let $G_C = (N_C, E_C, n_A, n_\Omega)$ be a control flow graph. A node $n \in N_C$ has control dependence on $m \in N_C$, $n\ \delta_c^a\ m$, if the following conditions hold:*

1. $(n, a, \lambda) \in E_C$ *where* $\lambda \in \{T, F, \epsilon\}$,

2. m *does not post dominate* n, $\neg(m\ \Delta_p\ n)$, *and*

3. *there is a path* $p = n, a, \ldots, m$, *such that for all* $z \in p$ *where* $z \neq n$, $z \neq m$ *holds:* $m\ \Delta_p\ z$.

A node m *is control dependent on* n *if and only if* $n\ \delta_c^a\ m$.

Definition 2.12 (Control Dependence Graph) *The control dependence graph* G_{CD} *of a control flow graph* $G_C = (N_C, E_C, n_A, n_\Omega)$ *is a directed graph* $G_{CD} = (N_{CD}, E_{CD})$ *with edge labels, such that* $(n, m, \lambda) \in E_{CD} \Leftrightarrow n\ \delta_c^a\ m$ *and* $(n, a, \lambda) \in E_C$ *and* $\lambda \in \{T, F\}$.

Definition 2.13 (Control Equivalence) *Let a control dependence graph* $G_{CD} = (N_{CD}, E_{CD})$ *be given. Two nodes* $n_1, n_2 \in N_{CD}$ *are* control equivalent *if they have the same predecessor* m *in the control dependence graph and the edges* (m, n_1, λ_1) *and* (m,n_2, λ_2) *have the same label, i. e.,* $\lambda_1 = \lambda_2$.

During instruction scheduling the operations of a procedure are reordered with the goal of improving the efficency of the program. The program semantics must not be changed by the reordering. In order to preserve the program semantics the control dependences must be respected, but additionally also the data dependences of the operations have to be taken into account. Data dependences are determined by the ordering of reading respectively writing accesses to the components of the machine state, as e. g. registers, or memory cells. Writing accesses are termed as *definitions*, reading accesses as *uses*. The data dependences of a procedure are represented by the data dependence graph.

Definition 2.14 (Data Dependence Graph) *Let* $G_C = (N_C, E_C, n_A, n_\Omega)$ *be a control flow graph. Its data dependence graph is a directed graph* $G_D = (N_D, E_D)$ *with node and edge labels whose nodes are labelled by the operations of the procedure. The set of edges is defined as* $E_D \subseteq N_D \times N_D \times \mathcal{R} \times T$ *where* $\mathcal{R}$ *denotes the storage resources of the target processor and* $\mathcal{T} = \{t, a, o\}$ *denotes the type of the data dependence. An edge runs from the node of an operation* i *to the node of an operation* j, *if* i *has to be executed before* j, *i. e. if there is a path from* i *to* j *in the control flow graph and if*

- i *defines a resource* r, j *uses it and the path from* i *to* j *does not contain other definitions of* r *(true dependence):* $(i, j, r, t) \in E_D$

- i *uses a resource,* j *defines it and the path from* i *to* j *does not contain any definitions of* r *(anti dependence):* $(i, j, r, a) \in E_D$

- i *and* j *use the same resource and the path from* i *to* j *does not contain any uses nor any definitions of* r *(output dependence):* $(i, j, r, o) \in E_D$.

Let E_D^t *denote the set of all true dependences,* E_D^a *the set of all anti dependences and* E_D^o *the set of all output dependences, then the edge set of the data dependence graph can be rewritten as follows:*

$$E_D \;=\; E_D^t \;\cup\; E_D^a \;\cup\; E_D^o.$$

Programs that contain loops must be handled with care, since their data dependence graph may contain cycles. Each data dependence must be classified as *loop-carried* if the dependence is caused by the repeated execution of a loop body, i. e. the operation instances belong to different loop iterations, or otherwise as *loop-independent*.

Definition 2.15 (Loop) *Let a basic block graph* G_B *and its dominator tree* T_d *be given. A loop* $G_L = (N_L, E_L, h_L)$ *is a subgraph of* G_B *where* $N_L \subseteq V_G$ *and* $E_L \subseteq N_L \times N_L$ *such that* $E_L \subseteq E_G$*.* G_L *must satisfy two conditions:*

- *There must be a unique entry point, the loop header* h_L*, that dominates all blocks of the loop, i. e.* $h_L \,\Delta_d\, b \;\forall b \in N_L$*.*
- *There must be at least one path starting from the loop header* $h_L \in E_L$ *that leads back to itself.*

The edges $(b, h_L) \in E_L$ *are denoted* backward edges *of* G_L*; all others* forward edges*. The* body of the loop G_L *is defined as* $N_L - \{h_L\}$*.*

A loop $G_{L'}$ is said to enclose another loop G_L, if $N_L \subseteq N_{L'}$. The loop nesting depth l_L of a loop G_L is defined to be the number of loops $G_{L'} \neq G_L$ such that $G_{L'}$ encloses G_L. The data dependences are classified as loop-independent or loop-carried with the help of the *reduced transitive hull* G_B^+ of the basic block graph.

Definition 2.16 (Reduced Transitive Hull of the Basic Block Graph) *Let the basic block graph* $G_B = (N_B, E_B, b_A, b_\Omega)$ *of a procedure be given and let* E_B^- *denote the set of all forward edges in* E_B*. Then the reduced transitive hull of the basic block graph is defined as* $G_B^+ = (N_B, E_B^+, b_A, b_\Omega)$*. There is an edge* (b, g) *in* E_B^+ *if and only if one of the following conditions is met:*

- *there is a path from* b *to* g *in* E_B^-*, or*
- *there is a loop* $G_L = (N_L, E_L, h_L)$ *with* $b \in N_L$*,* $g \notin N_L$ *such that there is a path from* h_L *to* g *in* E_B^-*.*

As an illustration, a basic block graph and its reduced transitive hull are shown in Fig. 2.2. The shaded blocks form a loop L whose header is block h_L.

From the control flow graph G_C and the reduced transitive hull of the basic block graph, the *reduced transitive hull of the control flow graph* G_C^+ can be derived. All incoming edges into a block $b \in G_B$ are represented by edges leading into the first node of E_C that belongs to b; all outgoing edges of b are represented

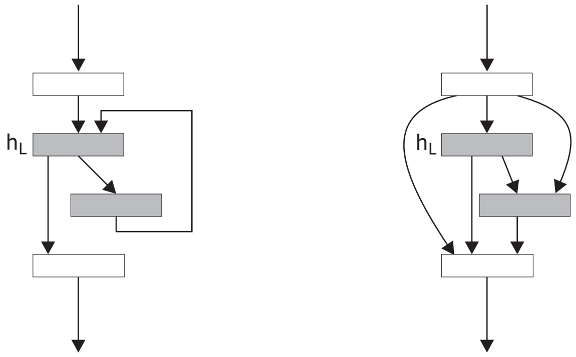


Figure 2.2.: A basic block graph and its reduced transitive hull.

by outgoing edges of the node for the last instruction in b. The edges between instructions of the same basic block are the same as in G_C. Then, each dependence from an operation i to another operation j ($(i, j, r, \tau) \in E_D$) where the instruction containing i is no predecessor of the instruction containing j in G_C^+ is a loop-carried dependence.

2.2. The Code Generation Phases

2.2.1. Code Selection

The input of the code selection phase is the intermediate representation of the input program, typically in the form of expression trees. The implementation of a code selector can be simplified by using code selector generators with appropriate descriptions of the target machine and its correspondence to the intermediate representation. The basic idea of code selector generators can be formalised by the theory of tree parsing and tree automata [FSW94, WM97]. The instruction set of the target machine is described by a regular tree grammar. The right-hand side of a rule describes the meaning of an operation in the form of tree patterns of the intermediate representation. The terminals correspond to the nodes of the intermediate representation, the non-terminals denote storage resources of the target machine. From such a machine grammar, expression trees of the intermediate representation can be derived. The derivation tree of an expression tree describes a semantically equivalent sequence of machine operations. In order to deal with ambiguous machine grammars, the rules are annotated with costs such that among different derivation trees for the same expression tree the cheapest one can be selected. Another theoretical foundation has been given by the theory of term rewriting systems [PL88, Emm92]. Early implementations based on [GG78, Hen84] used LR-parsing techniques driven by a specification of the target machine by a context-free grammar. The code selector was generated by a parser generator. The limitation however was that the code selection for ambiguous instruction sets could not be modelled conveniently. A solution of this problem is to combine pattern matching algorithms with dynamic programming to determine locally optimal operation sequences [AG85, WW89, HD89a] or extend the pattern matcher to directly selecting locally optimal operation sequences [PLG88, HD89b]. Examples for contemporary code generators are BEG [Emm89], Twig [AGT89], iburg [FHP92], and OLIVE [SPA97].

A drawback of the code selection by tree parsing is caused by the necessity of processing expression or syntax trees. If due to common subexpressions, the intermediate representation takes the form of a directed acyclic graph (DAG), usually heuristics are used to break up the DAG into a forest of trees. For architectures with a complex instruction set or with irregular hardware features the independent covering of the expression trees may result in a decrease of code quality [LDKT95]. This is covered in more detail in Sec. 2.2.4.

2.2.2. Register Allocation and Assignment

There is a large and continuously increasing gap between the processing speed of the CPU and the memory access time [Fer97]. Therefore the program execution can be accelerated by keeping the largest possible number of values of live program variables and live intermediate results, often called *symbolic registers*, in fast processor registers. Liveness of a variable or result means that its current value will potentially be needed again later during the program execution. The number of simultaneously live symbolic registers usually exceeds the number of physical registers. Hence, a resource optimisation problem results. Those values should be kept in processor registers that produce the highest benefits for the execution time. *Register allocation* proper attempts to determine which symbolic registers are mapped to physical registers and which ones are stored in memory. The task of *register assignment* is to select a particular physical register. For architectures with heterogeneous register files the register assignment problem includes the problem of selecting an appropriate register bank. Often the term register allocation is used to denote both the phases of register allocation proper and register assignment.

The input of the register allocation is an intermediate representation of the program where each operation and each modified variable is associated with a symbolic register. The same physical register must never be assigned to two different symbolic registers if they are simultaneously live and might contain different values. In the following the most important definitions and concepts of register allocation are summarised; a more detailed introduction is given in [WM97].

Definition 2.17 (live, life range) *A symbolic register r is* live *at a program point p, if there is a program path from the entry node of the procedure to p that contains a definition of r and there is a path from p to a use of r on which r is not defined. The* life range *of a symbolic register r is the set of the program points at which r is live.*

Definition 2.18 (interference, register interference graph) *Two life ranges of symbolic registers* interfere*, if one of them is defined during the life range of the* *other. The* register interference graph *is an undirected graph whose nodes are life ranges of symbolic registers and whose edges connect the nodes of interfering life ranges.*

Typically, register allocation is performed by graph colouring algorithms as presented in [Cha82, CH90]. The problems of register allocation and assignment are translated into the problem of colouring the register interference graph by k colours where k denotes the number of available physical registers. Different colours must be assigned to directly connected nodes. Since for $k > 2$ the problem of deciding whether an arbitrary graph can be coloured by k colours is NP-complete, [Cha82, CH90] suggest the use of heuristics. The basic idea of the graph colouring method is the following: if the interference graph contains a node n with a degree less than k, then n can definitively be assigned a colour that is different from the colours of all its neighbours. The node n is removed from G and a new graph G' is obtained that contains one node and several edges fewer. If no k colouring can be found, heuristics have been proposed to introduce spill code [Cha82], or to split life ranges [CH90]. While the algorithms of [Cha82, CH90] are restricted to basic block level, global register allocation algorithms exceed basic block boundaries and take the control flow structure of the program into account. Examples of global register allocation algorithms based on heuristics are the packing algorithm of [Ben94], the probabilistic register allocation of [PF92] and the optimistic graph colouring approach of Briggs et al. [BCT94].

2.2.3. Instruction Scheduling

Most contemporary microprocessors offer intraprocessor parallelism, e. g., parallel functional units and/or pipelines. Instruction scheduling attempts to reorder the (sequential) machine operation sequences produced by previous phases in order to exploit these parallel capabilities. A program dependence analysis determines data and control dependences in the program. These limit the ways the operations of the program can be reordered.

The data dependence graph defines a partial order among the operations of the input program. A precedence relation $\prec$ on N_D can be defined where

$$i \prec j \;\Leftrightarrow i \xrightarrow[G_D]{+} j$$

Thus, $i \prec j$ holds if operation j depends directly or indirectly on operation i. The resulting problem is to rearrange the instructions of the input program so that the execution time is minimised, but no precedence constraints are violated. In its simplest form, instruction scheduling corresponds to the classical problem of *precedence constrained scheduling*. Let a set $\mathcal{T}$ of tasks of length 1 be given, m machines, an arbitrary partial order $\prec$ on $\mathcal{T}$, and an upper bound T on the schedule length. The goal is to find a schedule $\sigma : \mathcal{T} \rightarrow \{1, \ldots, T\}$, so that for all $t \in \{1, \ldots, T\}$ where $|\{i \in \mathcal{T} : \sigma(i) = t\}| \leq m$ holds:

$$i \prec j \qquad \Rightarrow \qquad \sigma(i) < \sigma(j)$$

This optimisation problem is already NP-complete for $m = 2$, if each task has to be executed by a dedicated machine [GJ79]. In the problem of instruction scheduling the tasks correspond to machine operations and the machines represent parallel functional units of the underlying processor, e.g. ALUs, multipliers, etc.

For real-world hardware architectures, the problem usually is more complex. One reason is that the assumption that each task has length 1 in general is not valid. If the assumption holds and each task can be executed by any of the available m machines the scheduling problem can be solved in polynomial time if $m = 2$. This problem already becomes NP-complete if both tasks lengths 1 and 2 are allowed. In instruction scheduling each task has to be executed by one specific machine and the task length in general can be any nonnegative integer. Another complication is that many architectures exhibiting instruction-level parallelism dispose of several identical functional units that represent instances of one functional unit type. In this case additional resource constraints have to be taken into account such that in the classification of [GJ79] the instruction scheduling problem corresponds to precedence and resource constrained scheduling. The complexity further rises if the assignment of operations to functional units is not uniquely determined, and if irregular restrictions of parallelism have to be taken into account.

Several heuristic scheduling methods are in use for instruction scheduling, e.g. *list scheduling* [LDS80, Fis81, Gas89], *trace scheduling* [Fis81], *region scheduling* [GS90], and *percolation scheduling* [Nic85]. The list scheduling algorithm [LDS80] starts with an empty list of instructions. A microoperation is inserted into the last instruction of the list if it satisfies the following three conditions:

- It is ready, i. e. all predecessors in the data dependence graph have already been scheduled.
- Its priority is the highest among all ready microoperations. Different heuristics have been proposed to determine the priority, e. g., the time an operation remains in the data ready set, or the length of the longest path from an operation in the data dependence graph (*highest-level-first heuristic*).
- The insertion into the last instruction is feasible, i. e. all operations already contained in it can be executed in parallel to the current operation.

If there is no ready operation that can be inserted into the last instruction, a new instruction is appended to the list. List scheduling is the most common algorithm for local instruction scheduling; its worst-case time complexity is $\mathcal{O}(n^2)$ for n microoperations [LDS80].

An algorithm for global instruction scheduling is Fisher's *trace scheduling* algorithm. The basic idea of this approach is to schedule the operations of consecutive basic blocks jointly in order to increase the available parallelism. Basic blocks that are frequently executed directly after one another should be addressed jointly. For this, the code generator retrieves information about the execution frequencies of the basic blocks in a procedure by measurements or heuristic estimates. The algorithm decomposes the control flow graph of a procedure into disjoint subpaths. First it considers the most frequently executed block. Then it decides whether a preceding or subsequent basic block is scheduled jointly with the current block. During the scheduling of the resulting trace, operations can be moved from the original basic block beyond control flow forks and joins. In basic blocks leading into or out of this trace, compensation code has to be inserted. Then the most frequently executed block that has not been scheduled yet is selected and the algorithm iterates until all basic blocks of the procedure have been scheduled.

2.2.4. The Phase Coupling Problem

The classical code generation methods described above address each code generation subtask in a separate phase. Unfortunately the code generation tasks are interdependent; decisions made in one phase impose constraints to the subsequently executed phases. For regular architectures the quality of the heuristic methods is satisfactory. For irregular architectures however the interdependencies between the phases usually lead to a significant decrease of code quality [ZSWS95]. In the following some basic interdependencies between the code generation phases are described; the next section then focuses specifically on code generation for irregular architectures.

The goal of code selection is to select the cheapest instruction sequence for a given subgraph of the intermediate representation. Memory accesses increase the cost of the instruction sequence. Therefore the code selector will use as many registers as possible. Since code selection usually takes place before register allocation and assumes an infinite number of registers it is in conflict with register allocation that has to cope with a limited number of registers. Since the costs of spill code are not considered during code selection, the chosen operation sequence may turn out to be disadvantageous due to the insertion of spill code. Moreover the conventional heuristic of breaking up directed acyclic graphs into trees suitable for tree-based code selection may introduce unnecessary stores of intermediate values.

A similar conflict exists between register allocation and instruction scheduling. For instruction scheduling it is profitable to use many different registers since this leads to a reduction of the anti and output dependences. Those dependences are often termed *false dependences* since they are caused by the reuse of physical registers and are not dictated by the program semantics. Nevertheless they restrict the available parallelism. Since the goal of register allocation is to minimise the number of memory accesses it will try to reuse as many registers as possible whereas for instruction scheduling it would be better to use different registers as long as this does not lead to the insertion of spill code. Thus, if register allocation is performed first, it can limit the available parallelism by introducing false data dependences. If instruction scheduling precedes register allocation, the number of simultaneously live values can be increased so much that many of these values have to be stored in main memory which may deteriorate the code quality considerably.

Several heuristic methods have been developed that take into account the requirements of other code generation subtasks with the help of estimations. A detailed overview of those methods is given in Sec. 11.2. Since for regular architectures heuristic methods usually produce satisfactory results, exact phase-coupling methods integrating different code generation phases in a homogeneous approach have not been estimated promising. The possible increase of code quality has not been considered worth the increase of compilation time associated with more powerful solution algorithms. However for irregular architectures the situation is different.

2.3. Code Generation for Embedded Processors

While for regular architectures the classical heuristic approaches produce satisfactory results, empirical studies have shown that the code quality achieved for irregular architectures often is insufficient [ZSWS95, PCL96]. One class of processors that is characterised by irregular architectures are the digital signal processors (see Chap. 3) often used in embedded systems. Important requirements of those processors are high computation performance on the one hand, but low cost and low power consumption on the other hand. This is best achieved by architectural specialisation usually leading to irregular architectures. It has turned out that the code quality of traditional high-level language compilers for many digital signal processors is not acceptable. Therefore in the area of digital signal processing still assembly programming is in common use. Given the high economic importance of DSPs this seems surprising and can only be explained historically. The originally small size of the programs running on DSPs permitted assembly programming, so the specific compiling problems for DSPs did not receive much attention. However during the last years the size of DSP applications and the time-to-market pressure have constantly been rising. Since assembly programming is a very time consuming and error prone task and leads to bad portability and bad maintainability, the productivity of software development is seriously affected. The consequence is that an urgent demand for the use of high-level languages has arisen that, in general, cannot be met in a satisfactory way for efficiency reasons.

There are several reasons for the inefficiency of traditional code generation approaches in the area of digital signal processing. One problem is that the primary metric of many established efficiency-increasing transformations like function inlining, loop unrolling or software pipelining is performance rather than code size. However code size plays an important role in DSP compilation, since DSP architectures often have limited on-chip program memory. The code size has to be considered as a barrier for the parallelism-increasing transformations as in the approaches of [BCE$^+$98] so that their potential is limited. Another problem is the architectural irregularity of many embedded processors. In the presence of irregular hardware features the efficiency of traditional heuristics decreases whereas the impact of phase coupling problems increases. Often digital signal processors dispose of special functionality, but the ability of traditional compilers to capture the existence of specialised functionality is very restricted. Identifying special functionality requires bringing much architectural information into the compiler. However this is very problematic since the market for embedded processors is characterised by fast design cycles and severe cost restrictions. Developing a dedicated compiler for each new architecture takes too much time and is too expensive. The consequence is that retargetable compilation techniques have to be developed. This means that it must be possible to quickly adapt an existing compiler to considerably different target architectures while still retaining a high code quality.

In the remainder of this chapter the consequences of irregular hardware features for the code generation process are outlined in more detail. Different approaches for retargetable compilation have been developed in the past; a classification of those approaches based on [Leu97, Sud98] is given subsequently. Chap. 3 gives a classification of contemporary microprocessors that focuses on the aspect of irregularity and presents typical features of embedded processors in more details. An overview of approaches addressing the conflicting goals of retargetability and high-quality code generation is given in Chap. 11.

2.3.1. Code Generation for Irregular Architectures

Typical characteristics of irregular architectures are distributed heterogeneous register files, irregularly restricted instruction-level parallelism and restricted interconnectivity of functional units and register sets (see Chap. 3 for more details).

A consequence of such architectural restrictions is that breaking up directed acyclic graphs of the intermediate representation into trees suitable for code selection by tree parsing can impose limitations to the achievable code quality [LDKT95, Bas95]. The expression trees are constructed from DAGs by associating common subexpressions with temporary storage locations. Since the trees are addressed separately, in the code selected for one tree a storage location may be chosen that turns out to be disadvantageous for other expression trees. In the presence of complex data paths, accessing that value might require a set of data transfer operations across a route of register sets. Moreover the heuristic generation of expression trees can prevent the selection of complex machine operations if the matching tree pattern is distributed across several trees.

Complex data routes also aggravate the phase coupling problem between code selection and register allocation. If spilling has to take place across a route of register sets it can lead to a significant decrease of the quality of the selected code. Thus the effects of register allocation have to be taken into account during code selection. Most tree-based code selectors are implemented by combining pattern matching with dynamic programming algorithms. A presupposition of the dynamic programming approach is that the costs of the subtrees of a node are independent of one another. However in order to efficiently compute the spill costs, knowledge about the locations of the program values and the evaluation order of the subtrees is required. In consequence the costs of the subtrees are not independent any more such that a necessary precondition for applying dynamic programming is violated

[Bas95].

In order to overcome those restrictions several algorithms have been developed that address code selection for directed acyclic graphs — which in [Set75, AJU77] has been proven to be an NP-complete problem even for simple architectures. The approaches of [BL99, LDKT95, FHMK94] are presented in Chap. 11 in more detail.

Another restriction of code selector implementations by dynamic programming is that the potential instruction-level parallelism of an architecture cannot be taken into account [Bas95]. Again this is due to the presupposition that an optimal code sequence for an input tree can be constructed from optimal code sequences of the subtrees [ASU86, AGT89]. Due to the possibility of parallel execution the costs of one subtree may depend on the costs of other subtrees of a given node. Additionally there is a phase coupling problem between code selection and instruction scheduling: a cheapest operation sequence chosen during code selection may turn out to be suboptimal if it prevents the parallel execution of other operation sequences.

The phase coupling problem between register assignment, instruction scheduling and functional unit binding also plays an important role. Due to the restricted connectivity of register files and functional units the register assignment may constrain the selection of functional units and vice versa. Additionally the functional unit binding strongly influences the parallelism available for instruction scheduling. On the other hand the scheduling freedom may also depend on the register assignment, e. g. due to encoding restrictions as in the ADSP-2106x SHARC. Both the tasks of register assignment and instruction scheduling in general are already NP-complete problems when addressed separately.

2.3.2. Retargetable Code Generation

Due to the short design cycles in the market for embedded systems, it is necessary to develop retargetable compilation techniques. This means that with minor modifications a compiler for one processor must be able to generate high-quality code for a different hardware architecture. There are different approaches to achieve this goal; in the following a classification is given that is based on [Leu97, Sud98]. Depending on the type of its retargetability, a compiler can be assigned to one of the following classes:

processor-specific: The code generation techniques of the compiler are specifically tailored to a fixed target processor. In order to generate code for another architecture a large part of the compiler backend has to be re-implemented.

portable: The code generation methods of the compiler are implemented in a modular way such that an adaptation to different target processors is possible that reuses a large part of the existing code generation methods. The adaptation may require rewriting certain parts of the compiler source code or providing a set of target-specific code generation functions.

retargetable: The compiler uses an external target machine description that can be provided by the developer. All information about the target architecture that is needed for code generation is automatically derived from the machine description. A further distinction can be made depending on whether the retargeting may imply rewriting compiler source code. If this is the case the compiler is said to be user-retargetable, otherwise it is called machine-independent.

parameterisable: The compiler is tailored to a specific class of processors sharing the same basic architecture. An external machine description is used which only consists of numerical parameters such as register file sizes, the number of functional units, or different operation latencies.

Some well-known portable compilers primarily for general purpose processors are `gcc` [Sta98] and `lcc` [FH95]. Both are located at the edge between portable and user-retargetable compilers. `gcc` requires an exhaustive target machine description in which the hardware description is intermixed with the implementation of the code selector. The frontend of `lcc` and parts of its backend are architecture-independent, only some program parts required for code generation have to be implemented separately. The code selector is generated from a specification of the instruction set in the form of a regular tree grammar by the code selector generator iburg [FHP92]. Examples of retargetable compilers are CHESS [LVPK$^+$95] or Express [HGG$^+$99]; a parameterisable compiler is part of the Trimaran system [tri98]. A more detailed description of each of those approaches together with a summary of further retargetable compilers is given in Chap. 11.

3. A Classification of Microprocessors

A traditional classification scheme of microprocessors is based on the basic design of their instruction set architecture. It is distinguished between complex instruction set computers (CISCs), reduced instruction set computers (RISCs) and very long instruction word (VLIW) architectures. Those classes can be characterised as follows [WM95]:

- The design goal of CISCs is to close the 'semantic gap' between high-level programming languages and machine languages. They are characterised by:
  - a large number of complex addressing modes to support efficient accesses to different data structures,
  - manifold versions of operations for operands of different length and combinations of different sorts of operands,
  - different execution times for instructions,
  - few processor registers,
  - a microprogrammed control logic.
- The basic idea of RISC processors is to increase the execution speed of machine instructions by simplifying them. They are characterised by
  - the ability to execute one machine operation per clock cycle,
  - restricting memory accesses to dedicated load/store instructions,
  - few addressing modes,
  - a hard-wired control logic.
- VLIW architectures are designed to provide explicit statically determined instruction-level parallelism. A fixed number of machine operations can be composed to form a VLIW instruction; the execution of operations from the same instruction is started in parallel. Arranging the operations to exploit a high degree of instruction-level parallelism is the task of the compiler.

A special subclass of RISCs or CISCs are the superscalar processors which provide multiple instruction pipelines in order to allow multiple instructions to be issued simultaneously during each clock cycle [Fly95, HP96]. In contrast to VLIW architectures the parallelism is not necessarily exposed to the compiler.

With the emergence of embedded systems, during the last years the acceptance for computationally intensive compilation techniques that facilitate an improvement of code quality at the expense of higher compilation time has increased. This change cannot be explained in a satisfactory way by the traditional classification scheme; more insight can be given by an application-based classification. Many processors used in embedded systems incorporate features of all previously mentioned architectural design styles — but they are subject to irregular restrictions and application-specific extensions. The impact of those irregular architectural features on the task of code generation is very important and represents a basic motivation of this thesis. Based on their application domain, contemporary microprocessors can be classified in two coarse categories: general-purpose processors (GPPs) and application-specific processors (ASPs).

- GPPs are designed to efficiently execute a wide range of different applications, e. g. the typical workload of a personal computer. Since they must provide the functionality required by all of these applications, general-purpose processors have a low level of specialisation. Cheaper or less power-intensive general-purpose processors which are used in industrial applications are often denoted as microcontrollers. Most traditional RISC or CISC processors belong to the class of general-purpose processors.

- Application-specific processors (ASPs) are designed to efficiently execute a narrow domain of applications. Important requirements are high computation performance, low cost, and low power consumption. This is best achieved by architectural specialisation commonly resulting in very specialised and irregular architectures. The negative consequence is that compiling for these architectures becomes a very difficult task. There exist mainly 3 classes of ASPs: an application specific integrated circuit (ASIC) is designed to implement a given target algorithm completely in hardware. An application specific instruction set processor (ASIP) is a programmable architecture where hardware and instruction set are designed together to implement a very specific algorithm. Usually the program is stored in Read-Only Memory. DSPs are specialised programmable microprocessors designed for applications that require extensive real-time numerical computations.

Because of the restricted programmability of ASICs and ASIPs in the following we will focus on digital signal processors. DSPs can be partitioned into fixed and floating-point DSPs. The earliest digital signal processors used fixed-point arithmetic, and in fact fixed-point DSPs still dominate today [LBSL97]. In fixed-point processors, numbers are represented either as integers (integer arithmetic) or as fractions between -1.0 and $+1.0$ (fractional arithmetic). The programmer is responsible for correctly scaling the result of the fractional operations. Another class of DSP processors primarily use floating-point arithmetic where numbers are represented by the combination of a mantissa and an exponent. The mantissa is usually a signed fractional value with a single implied integer bit assumed to be equal to 1. In floating point processors, scaling is automatically done; thus they are easier to program than fixed point processors. On the other hand, the additional hardware in floating-point DSPs results in higher cost and power consumption than in fixed-point DSPs.

There are several reasons for the differentiation between general-purpose processors and application-specific processors. One reason simply becomes evident when comparing the sizes and the prices of the processors. The die sizes of DSPs usually range from 25mm^2 to 60mm^2 whereas the die size of GPPs lies between 160mm^2 (Pentium) and 250mm^2 (SuperSparc). Example prices of fixed-point DSPs in 1997 were \$35.89 for the Motorola DSP56166 or \$43.95 for the Texas Instruments TMS320C541. As for floating-point DSPs, the price of the TMSC44 in 1997 was \$158.40 and the ADSP-21062 costed \$249.00. Modern general-purpose processors were considerable more expensive; for a 200 MHz Intel Pentium Pro with a 256K L2 cache \$487 had to be paid in 1998.

The performance characteristics of contemporary general-purpose processors are appropriate for most application domains of DSPs. Nevertheless in most cases power consumption and cost make the use of GPPs prohibitive for use in DSP applications. In [Cam98] a formalisation of the architectural specialisation is proposed that confirms the superiority of DSPs over GPPs with respect to applications of digital signal processing. The power consumption of a CMOS device can be defined as

$$P = CV^2 f \Delta N$$

where C denotes the capacitance, V the CPU core voltage, f the core clock frequency, and ΔN the number of gates that change state during a given clock cycle. More complex architectures require more gates. More specialised architectures have fewer gates toggles per task. Thus an architecture specialisation metric can be defined based on how many clock cycles and gates are required to complete a given functional task. In [Cam98] the ratio

$$AS = \frac{Performance}{PowerConsumption}$$

is proposed as a metric of architecture specialisation. The performance is approximated by the number of operations executed per second ($P = Of$). Then several observations can be made:

- Increasing the clock rate has no influence on the performance per power ratio.

$$AS = \frac{Performance}{PowerConsumption} = \frac{Of}{CV^2 f \Delta N} = \frac{O}{CV^2 \Delta N}$$

- A voltage decrease significantly improves performance per power for the same architecture.

$$\frac{AS_1}{AS_2} = \frac{\frac{O}{CV_1^2 \Delta N}}{\frac{O}{CV_2^2 \Delta N}} = \frac{V_2^2}{V_1^2}$$

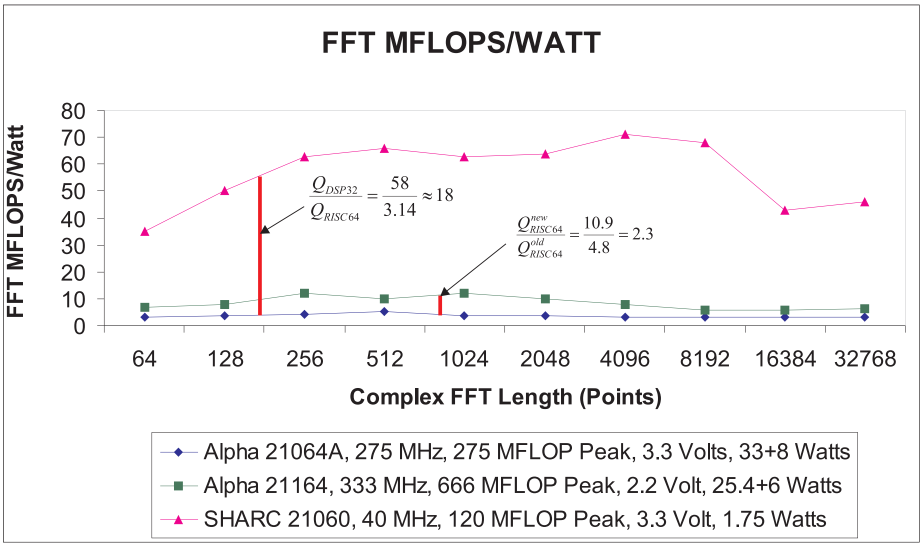


Figure 3.1.: Performance per power.

In Fig. 3.1 a comparison between the performance per power ratios of two alpha processors and the digital signal processor ADSP-21060 is shown for a fast fourier transformation algorithm. The ratios have been obtained by determining the number of operations required to execute the algorithm and by measuring the power consumption. The diagram shows that the specialisation ratio AS of the SHARC is 18 times higher than that of the Alpha 21064A. Note that two different architectures are compared such that the operation per second ratio is not a completely accurate basis for a performance comparison. However since due to specialised instructions for digital signal processing the instruction set of the SHARC is more complex than that of the Alpha, the comparison is conservative.

The study of [Cam98] indicates that contemporary GPPs do supply the performance required for embedded applications, but cannot compete with specialised DSPs in terms of performance per power. Another conclusion is that advances within the same architectural style have a relatively low effect on the architectural specialisation so that it can be expected that the relative lead of the DSPs will prevail. This conclusion is based on the central assumption that the performance capacities of digital signal processors are fully exploited. In the past this has usually been achieved by implementing the applications in assembly language. Due to the increasing software complexity and shrinking time-to-market cycles, assembly programming becomes increasingly unacceptable and the need for using high-level programming languages arises. A presupposition of using high-level languages is that there are compilation techniques that allow to produce high-quality code even for irregular architectures.

Recently the Intel Native Signal Processing Initiative has been launched aiming at integrating typical DSP functionality like multiply-accumulate units into general-purpose processor designs and thus creating "hybrid" architectures. However it is questionable that systems based on general-purpose processors with their intrinsic time-sharing nature will be able to meet the real-time requirements of typical DSP applications [LBSL97]. The majority of DSPs are not met in standard desktop computers but on systems which require the combination of low cost, reduced power consumption and high performance.

3.1. Applications of Digital Signal Processors

The emergence of digital signal processors is primarily driven by the explosion in the markets of embedded systems, telecommunication and multimedia. Various studies have pointed out the growth of embedded systems. According to Dataquest Interactive, 57% of all 32-bit microprocessors sold in 1995 were employed in such systems. The global DSP-market is expected to grow 40 % per year while the personal computer market is starting to level off at 14%. The share of general-purpose processors produced annually is only 6% while the application specific processors represent 94% of the total number of processors produced every year.

Typical DSP applications share a number of well defined characteristics. First the input data is usually a sequence of samples from some sort of digitised signal, originating from microphones, sensors, antennas, video cameras, etc. Second, applications require extensive numeric computation which must be performed unter very stringent time constraints. This happens because they usually run on real-time systems which demand very high throughput and very low latency. Common application areas are low-cost embedded systems like, e. g., speech synthesis and recognition, high speed modems, digital cellular phones, disk drives (servo control), etc. But there are also numerous high-performance applications: image compression and decompression, medical and seismic imaging, radar/sonar, etc.

Embedded systems are subject to severe cost restrictions. The cost of a microprocessor increases non-linearly with its die size. Thus generating high-density code is an important issue in code generation for digital signal processing since any reduction in ROM area translates to a non-linear reduction in cost. In order to guarantee that code density and performance requirements are safely met, system designers usually hand-program the embedded system in assembly language. In order to understand the reasons why the code quality of conventional compilers mostly is not acceptable, we have to look in some more detail at the architectural characteristics of digital signal processors.

3.2. Characteristics of Digital Signal Processors

Digital signal processors are programmable microprocessors specialised for applications of digital signal processing. Therefore most DSPs share some common features to support repetitive, numerically intensive tasks:

- Dedicated multiply-accumulate (MAC) units allow to perform multiplication and accumulation in a single cycle. MAC operations are used in vector products, digital filters, correlations, fourier transforms, etc.

- Multiple access memory architectures provide high bandwidths between processor and memory. This is essential to achieve good performance for repetitive data-intensive operations frequently occurring in DSP applications. A common design goal is to achieve a throughput of one instruction per clock cycle. This implies being able to complete several memory accesses per clock cycle. Therefore, memory space is often divided into program memory and data memory which can be accessed simultaneously using separate buses (Harvard architecture). Moreover the data memory space often is subdivided into multiple banks. This allows for variables in different memory banks to be accessed in parallel using dedicated addressing units. Often the memory accesses can be executed in parallel to arithmetic operations. A common restriction is that multiple memory accesses are only available for certain instructions such that the available parallelism is irregularly restricted.

- A typical characteristic of many DSPs is the availability of specialised addressing modes. There are three commonly used modes: linear, circular and reverse arithmetic addressing. In the linear addressing mode, the address of the data is computed by adding/subtracting an offset to/from an address register, which is often implemented as post-modify operation. In circular addressing the final address is determined by incrementing or decrementing an address register and then taking the remainder of the division of this intermediate value by a constant N (e.g. for circular buffers). In reverse arithmetic addressing, the carry bits of the increment or decrement operation are propagated in the reverse order (from the most significant to the least significant bit). This addressing mode specifically supports fast implementations of fourier transformations.

- Branch instructions in DSPs often use control bits which are automatically set by arithmetic operations. This permits the branch condition to be detected early in the pipeline, an approach which combined with typically short pipelines leads to small interlock penalties.

- DSP architectures usually have arithmetic instructions that can be residually controlled. These are instructions whose execution behaviour depends on specific bit values stored in a control register that were set by another instruction. Depending on a control bit, the result of an arithmetic instruction may be sign extended or not. Another application of residual control is predicated (guarded) execution, i. e. instructions that are only executed depending on certain bit or register values.

- Another typical feature are hardware loops, often denoted as zero-overhead loops. The key difference between hardware and software loops is that hardware loops do not need explicit instructions for incrementing or decrementing counters, checking the loop condition, or branching back to the top of the loop.

- Often the interconnectivity between registers and functional units is restricted. The main reason is that broad interconnectivity results in increased cost and a degradation of individual instruction performance. In many DSP architectures there are instructions whose operands have to be located in specific register files (heterogeneous register architectures). In GPPs the register usage usually is not restricted. This considerably simplifies the code generation problem since it decouples the tasks of instruction selection from register allocation (homogeneous register architectures) and instruction scheduling from register assignment.

- Many digital signal processors have strongly encoded instruction formats. Achieving a throughput of one instruction per clock cycle requires the ability to fetch one instruction per cycle. Thus each instruction usually has to fit into a single memory word. The consequence is that the number of bits used for encoding the instructions has to be minimised. The necessary number of bits can be reduced in several ways, e. g. by

  - reducing the number of addressing modes. Not all combinations of operations and addressing modes are feasible, e. g. immediate memory accesses often are restricted to a small set of instructions.
  - restricting the set of feasible source and destination operands. In the extreme case there are operations with implicit operands.
  - using mode bits. An example for a processor using mode bits is the TMS320C5x [Tex98a], where there are no separate arithmetic and logical shift operations. A shift mode bit in a control register determines whether the shift instruction represents an arithmetic or a logical shift.

  The consequence is an increased irregularity of the instruction set, but the narrower instruction word width usually reduces overall processor and system cost.

Most of those hardware characteristics can be exploited at machine operation level. Therefore it is a natural approach to address efficency-increasing transformations related to those characteristics in a postpass stage, i. e. after the generation of assembly code.

4. A Short Introduction to Integer Linear Programming

In this chapter some basic definitions and theorems of linear and integer linear programming as well as the underlying mathematical concepts are shortly summarised. Only the concepts that are required for the understanding of this thesis are presented; detailed surveys can be found in [NW88, NW89, PS82, Sch86, Wil93a, Wil93b, JNS97].

4.1. General Overview

Integer linear programming deals with the problem of maximising or minimising a linear function subject to linear inequality and equality constraints where some or all of the variables are required to be integral. A rich variety of problems can be represented by such discrete optimisation models. Typical applications concern the management and efficient use of scarce resources to increase productivity. Examples are planning problems as production scheduling, machine sequencing, or capital budgeting, portfolio analysis and design problems such as VLSI circuit design and the design of automated production systems. Scientific applications include problems in molecular biology, high energy physics and X-ray crystallography [NW88, NW89]. In the last decade, the use of integer programming models has increased significantly which is mostly due to the advances in algorithms for solving integer programs and the availability of reliable software packages [JNS97]. Computing an optimal solution of an integer linear program is NP-complete [GJ79]. Nevertheless many large instances of such problems can be solved. Recent research has lead to an understanding of properties that make some ILP formulations easily solvable. Those formulations are called structured [CWM94] since it is the structure of their constraints that permits efficient computations. Using structured formulations has lead to a considerable success in the ILP approach to NP-complete problems as, e. g., the travelling salesman problem [LLKS85] or in 0-1-programming [CJP83]. Recent advances have also made it possible to improve the efficiency of ILP solving techniques by curtailing the necessary enumeration process [JNS97, Eis00].

4.2. Mathematical Foundations

Definition 4.1 *Let $S \subseteq \mathbb{R}^n$. A point $x \in \mathbb{R}^n$ is a* convex combination *of points of S if there is a finite set $\{x^i\}_{i=1}^t \in S$ and a $\lambda \in \mathbb{R}^t$ with $\sum_{i=1}^t \lambda_i = 1$, $x = \sum_{i=1}^t \lambda_i x^i$ and $\lambda_i \geq 0$ for all $i = 1, \ldots, t$. If $\lambda_i > 0$ for all $i = 1, \ldots, t$, x is called strict convex combination of S. The convex hull of S, $conv(S)$, is the set of all convex combinations of points in S.*

Definition 4.2 *A set of points $x^1, \ldots, x^k \in \mathbb{R}^n$ is called* linearly independent, *if the unique solution of $\sum_{i=1}^k \lambda_i x^i = 0$, for $\lambda_i \in \mathbb{R}$ is $\lambda_i = 0$ for all $i = 1, \ldots, k$.*

Definition 4.3 *A set of points $x^1, \ldots, x^k \in \mathbb{R}^n$ is called* affinely independent *if the unique solution of $\sum_{i=1}^k \alpha_i x^i = 0$, $\alpha_i \in \mathbb{R}$, $\sum_{i=1}^k \alpha_i = 0$ is $\alpha_i = 0$ for all $i = 1, \ldots, k$.*

Definition 4.4 *A* polyhedron *$P \subseteq \mathbb{R}^n$ is a set of points satisfying a finite number of linear inequalities; $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$, where (A, b) is an $m \times (n+1)$-matrix. A polyhedron is called* rational, *if there is an $m' \times (n+1)$-matrix (A', b') with rational coefficients such that $P = \{x \in \mathbb{R}^n \mid A'x \leq b'\}$.*

Definition 4.5 *A point $x \in P$ is called* vertex *of the polyhedron P, if x cannot be represented as a strict convex combination of two different points from P.*

Definition 4.6 *A polyhedron $P \subseteq \mathbb{R}^n$ is* bounded *if there is an $\omega \in \mathbb{R}_+$, such that $P \subseteq \{x \in \mathbb{R}^n \mid -\omega \leq x_j \leq \omega$ for $j = 1, \ldots, n\}$. A bounded polyhedron is called a* polytope.

Definition 4.7 *A polyhedron P is of* dimension *k, $dim(P) = k$, if the maximum number of affinely independent points in P is $k + 1$.*

Given a polyhedron $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ the question arises which of the inequalities $a^i x \leq b_i$ are necessary in the description of P and which can be omitted.

Definition 4.8 *An inequality $a^i x \leq b_i$ is called* valid *for P, if it is satisfied by all points in P.*

Definition 4.9 *If $a^i x \leq b_i$ is a valid inequality for P and $F = \{x \in P \mid a^i x = b_i\}$, then F is called a* face *of P and we say that $a^i x \leq b_i$ represents the set F. A face F is said to be* proper, *if $F \neq \emptyset$ and $F \neq P$. A face F of P is a* facet *of P if $dim(F) = dim(P) - 1$.*

In [NW88] it is shown that in order to describe P it is sufficient to characterise its facets. For each facet F of P one of the inequalities representing F is necessary to describe P. Each inequality $a^r x \leq b_r$, representing a face of P of dimension $d < dim(P) - 1$ is irrelevant to the description of P. To each polyhedron P there is a finite set of linear inequalities forming a minimal representation of P [NW88]. When describing a polyhedron one should always try to find such a minimal representation.

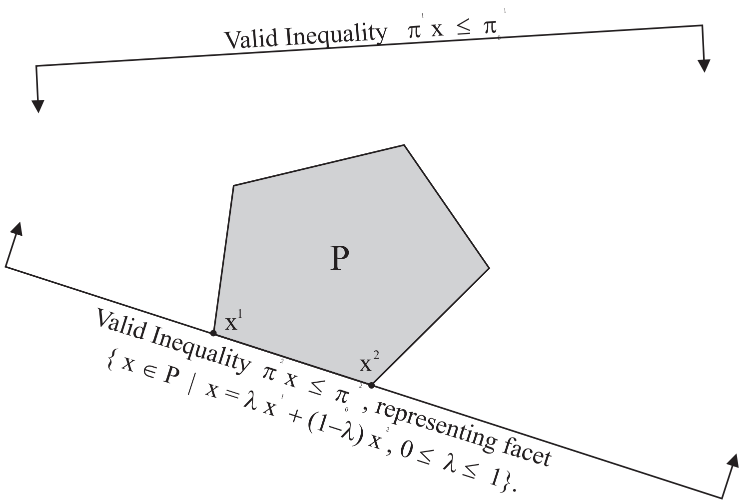


Figure 4.1.: Valid inequalities and faces.

4.2.1. The Theory of Linear Programming

A good understanding of the theory and the algorithms of linear programming is essential for understanding integer programming. Linear programming algorithms are often used as a subroutine in integer programming algorithms to obtain bounds on the optimal value of the integer program. A deeper connection between linear and integer programming is that corresponding to any integer programming problem there is a linear programming problem that has the same solution as the integer problem.

In the following the terminology is shortly presented and some central duality theorems are given. The duality theory provides necessary and sufficient optimality conditions as well as means to determine bounds on the optimal value of a linear program.

In the problem of linear programming a linear objective function is given that is to be minimised (or maximised) subject to a finite number of linear inequalities.

$$\begin{array}{rrcl} min & z_{LP} & = & c^T x \\ & Ax & \geq & b \\ & x & \in & \mathbb{R}^n_+ \end{array} \tag{4.1}$$

where

$$c \in \mathbb{R}^n, \; b \in \mathbb{R}^m, \; A \in \mathbb{R}^{m \times n}.$$

The set $S = \{x \in \mathbb{R}^n_+ \mid Ax \geq b\}$ is called *feasible region*, an $\hat{x} \in S$ is called *feasible solution*. The feasible region of a linear program is a convex polyhedron. An instance of the problem is said to be *feasible* if $S \neq \emptyset$. The function $z_{LP} = c^T x$ is called *objective function*, a feasible point $\hat{x} \in S$ that minimises z_{LP} is said to be an *optimal solution*. If x^* is an optimal solution, $z_{LP}(x^*) = c^T x^*$ is called *optimal* *solution value*. An instance of (4.1) is *unbounded*, if for all $\omega \in \mathbb{R}$ there is an $\hat{x} \in S$ with $c^T\hat{x} < \omega$. Computing a solution of an instance of (4.1) means calculating an optimal solution or showing that it is unbounded or infeasible. If all coefficients are rational, each feasible instance of a linear program has at least one optimal rational solution or is unbounded.

If a linear program is given the questions arise how the optimality of a solution can be proven and whether a lower bound on the optimal value of the objective function can be determined. In this context the problem (4.1) is denoted as the *primal problem*. The *dual problem* of (4.1) is defined as follows:

$$\begin{array}{rrcl} max & z'_{LP} & = & y^T b \\ & y^T A & \leq & c \\ & y & \in & \mathbb{R}^m_+ \end{array} \tag{4.2}$$

The vector of the objective function coefficients of the dual corresponds to the right-hand side vector of the primal problem and vice versa; the coefficient matrix of the dual is the transposed coefficient matrix of the primal. Each linear program can be dualised this way. The dual of the dual (4.2) is the primal problem such that it does not matter which of them is denoted as the dual and which as the primal problem.

Example 4.1 Let the following primal problem be given:

$$\begin{array}{rrcrcrcl} min & 36x_1 & + & 72x_2 & + & 24x_3 & & \\ & x_1 & + & 2x_2 & + & x_3 & \geq & 6 \\ & 2x_1 & + & 3x_2 & & & \geq & 3 \\ & 4x_1 & - & x_2 & + & x_3 & \geq & -9 \\ & -x_1 & + & x_2 & + & x_3 & \geq & 15 \\ & & & & & x_1, x_2, x_3 & \geq & 0 \end{array}$$

Then the dual problem reads as follows:

$$\begin{array}{rrcrcrcrcl} max & 6y_1 & + & 3y_2 & - & 9y_3 & + & 15y_4 & & \\ & y_1 & + & 2y_2 & + & 4y_3 & - & y_4 & \leq & 36 \\ & 2y_1 & + & 3y_2 & - & y_3 & + & y_4 & \leq & 72 \\ & y_1 & - & & + & y_3 & + & y_4 & \leq & 24 \\ & & & & & & & y_1, y_2, y_3, y_4 & \geq & 0 \end{array}$$

□

Proposition 4.1 (Weak Duality) *Let $\overline{x}$ be a feasible solution of the primal problem*

$$\begin{array}{rrcl} min & c^T x & & \\ & Ax & \geq & b \\ & x & \in & \mathbb{R}^n_+ \end{array}$$

and $\overline{y}$ a feasible solution of the dual problem

$$\begin{array}{rcl} max & y^T b & \\ & y^T A & \leq & c \\ & y & \in & \mathbb{R}^m_+ \end{array}$$

then

$$\overline{y}^T b \leq c^T \overline{x}.$$

Proposition 4.2 (Strong Duality) *Let x^* be a feasible solution of the primal problem and y^* a feasible solution of the dual problem. The equation $c^T x^* = y^{*T} b$ holds if and only if x^* is an optimal solution of the primal and y^* is an optimal solution of the dual problem.*

Corollary 4.1 *Let P and D be a pair of dual linear programs as defined in (4.1) and (4.2).*

1. *A linear program has an optimal solution if and only if P and D have (at least) one feasible solution. The optimal values of the objective functions of P and D are identical.*
2. *If P has no lower bound, D is infeasible.*
3. *If D has no upper bound, P is infeasible.*

The duality properties play an important role, e. g., in the branch-and-bound algorithm presented in Chap. 4.3.

There are several well-known algorithms for linear programming: the simplex algorithm, the ellipsoid method of Khachiyan [Kha80], and the projective algorithm of Karmarkar [Kar84]. The most prominent algorithm is the *simplex method* designed by Dantzig [Dan51]. Although its worst-case complexity is exponential, on the average the method is very efficient. Practical experience shows that the number of necessary pivot steps is about linear. The primal and dual simplex algorithms are components of many software systems for linear and integer linear programming including the CPLEX library [ILO99] used in the implementation of this thesis. The ellipsoid algorithm and the projective algorithm have polynomial time complexity; however the ellipsoid method is considered computationally impractical. More detailed discussions of those algorithms can be found in [NW88], [NW89], [PS82], [Chv83], [Sch86], or [Sch93].

4.2.2. The Theory of Integer Linear Programming

A *mixed integer linear program* (MIP) is an optimisation problem of the form

$$\min \{c^T x + h^T y \mid Ax + Gy \geq b;\ x \in \mathbb{Z}^n_+,\ y \in \mathbb{R}^p_+\} \tag{4.3}$$

where

$$c \in \mathbb{R}^n,\ b \in \mathbb{R}^m,\ h \in \mathbb{R}^p,\ A \in \mathbb{R}^{m \times n},\ G \in \mathbb{R}^{m \times p}$$

The special case of a MIP where there are no continuous variables is called pure integer linear program (ILP). It has the following form:

$$\begin{aligned} \min \quad z_{IP} &= c^T x \\ x &\in P_F \cap \mathbb{Z}^n \end{aligned} \tag{4.4}$$

where

$$P_F = \{x \mid Ax \geq b,\ x \in \mathbb{R}^n_+\},\ c \in \mathbb{R}^n,\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n}$$

In the scope of this thesis we can assume that $A \in \mathbb{Z}^{m \times n}$ and $b \in \mathbb{Z}^m$.

The optimal solution of an integer linear program can be computed by solving the following problem [NW88]:

$$\begin{aligned} \min \quad z_{IP} &= c^T x \\ x &\in P_I \end{aligned} \tag{4.5}$$

where

$$P_I = conv(\{x \mid x \in P_F \ \cap \mathbb{Z}^n\}).$$

A representation of P_F and P_I (equations 4.4, 4.5) is given for the two-dimensional case in Fig. 4.2. The integer points within P_F represent the feasible solutions of the integer linear program; depending on the objective function at least one of them is an optimal solution. The feasible region of (4.4) only consists of the integer points whereas the feasible region of (4.5), P_I, consists of the convex hull of those points.

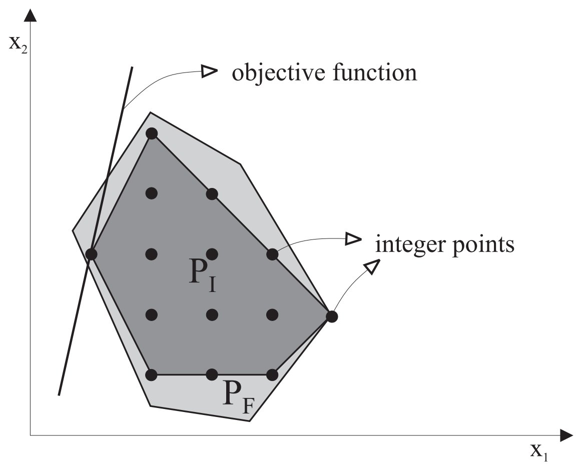


Figure 4.2.: Feasible region.

Since P_F is a rational polyhedron, each linear function can be optimised over P_F in polynomial time by linear programming algorithms. However in the general case this observation cannot directly be exploited when solving integer linear programs. Often no system of linear equations describing P_I is known, and in general the number of inequalities that are necessary to describe the convex hull P_I is extremely large [NW89]. Nevertheless it is an important observation that can be exploited to increase the efficiency of ILP models and algorithms.

Definition 4.10 (Relaxation) *Let an optimisation problem Q with feasible region $X(Q)$ be given. An optimisation problem Q^R is called a* relaxation *of Q, if for the feasible region $X(Q^R)$ holds:*

$$X(Q) \subset X(Q^R).$$

If all integrality constraints of an ILP are removed, the resulting linear program is called the *LP-relaxation* of the ILP. The LP-relaxation is defined as follows:

$$\begin{array}{rrcl} \min & z_R & = & c^T x \\ & x & \in & P_F \end{array} \tag{4.6}$$

where

$$P_F = \{x \mid Ax \geq b,\ x \in \mathbb{R}^n_+\},\ c \in \mathbb{R}^n,\ b \in \mathbb{Z}^m,\ A \in \mathbb{Z}^{m \times n}$$

Since $P_I \subseteq P_F$, it follows that $z_R \leq z_{IP}$. If $P_F = P_I$, the polyhedron P_F is said to be integral. In this case $z_R = z_{IP}$ holds and the optimal solution can be calculated in polynomial time by solving the LP-relaxation. Therefore, when formulating an integer linear program one should always try to find a set of constraints describing P_F as an integral polyhedron.

Unfortunately, most problems do not directly translate into integral polyhedra and it is not known how the required additional linear inequalities have to be formulated—and there might be an exponential number of them [NW89]. So, in general $P_I \subset P_F$ and the LP relaxation provides a lower bound for the objective function. The efficiency of many algorithms for solving integer linear programs depends on the quality of this bound. The better P_F approximates the feasible area P_I the tighter is the bound. Thus, for efficiently solving an ILP formulation it is extremely important that P_F is close to P_I (see Chap. 4.3) [CWM94].

One important algebraic property that can be used to check the integrality of a polyhedron is the total unimodularity of the constraint matrix. If the coefficient matrix of a set of constraints can be proven to be totally unimodular, the corresponding polyhedron is integral.

Definition 4.11 *A square, integer matrix $B \in \mathbb{Z}^{n \times n}$ is called* unimodular *if for its determinant B holds $|B| = \pm 1$. An integer matrix $A \in \mathbb{R}^{m \times n}$ is called* totally unimodular *if every square, nonsingular submatrix of A is unimodular.*

Theorem 4.1 *Let A be a totally unimodular matrix and let b be an integral vector. If $P = \{x \in \mathbb{R}^n_+ \mid Ax \geq b\}$ is not empty, then P is an integral polyhedron.*

Theorem 4.2 *Let A be a totally unimodular matrix, and let b, b', d and d' be integral vectors. If $P = \{x \in \mathbb{R}^n_+ \mid b' \leq Ax \leq b, d' \leq x \leq d\}$ is not empty, then P is an integral polyhedron.*

A stronger connection is established by the theorem of Hoffman and Kruskal:

Theorem 4.3 *Let A be an integral matrix. Then A is totally unimodular, if and only if for* each *integral vector b the polyhedron $\{x \in \mathbb{R}^n_+ \mid Ax \leq b\}$ is integral.*

The following definitions and theorems are helpful in checking total unimodularity.

Theorem 4.4 *Let $A \in \{0, 1, -1\}^{m \times n}$. If A has no more than two nonzero entries in each column, and if $\sum_i a_{ij} = 0$ if column j contains two nonzero coefficients, then A is totally unimodular.*

Theorem 4.5 *The following statements are equivalent:*

- *A is totally unimodular.*
- *The transpose of A, A^T, is totally unimodular.*
- *A matrix obtained by multiplying a row (column) of A by -1 is totally unimodular.*
- *A matrix obtained by permuting rows (columns) of A is totally unimodular.*
- *A matrix obtained by duplicating rows (columns) of A is totally unimodular.*

Theorem 4.6 *Let $A \in \{0, 1, -1\}^{m \times n}$. Then A is totally unimodular, if and only if for every $J \subseteq N = \{1, \ldots, n\}$ there exists a partition J_1, J_2 of J such that*

$$\left| \sum_{j \in J_1} a_{ij} - \sum_{j \in J_2} a_{ij} \right| \leq 1 \quad \textit{for } i = 1, \ldots, m.$$

Definition 4.12 *Let $A \in \{0, 1\}^{m \times n}$. Then A is called an* interval matrix *if in each column the 1's appear consecutively; that is if $a_{ij} = a_{kj} = 1$ and $k > i + 1$, then $a_{ij} = 1$ for all l with $i < l < k$.*

Theorem 4.7 *Interval matrices are totally unimodular.*

The proofs of theorems (4.1 – 4.7) are given in [NW88].

Advances in solving ILP formulations have been made in several ways. By formally analysing the structure of the constraints, tight descriptions of the feasible region P_F can be developed that approximate P_I more closely. Moreover the formulation can be tightened by incorporating valid inequalities which are due to the integrality of decision variables. This way better bounds on the objective function can be found and the efficiency of ILP formulations increases. The availability of good bounds plays an important role for example in the branch-and-bound algorithm. Finally the increased understanding of the constraints has led to better relaxation- and branching strategies [Wil93a, CWM94].

4.3. The Branch-And-Bound Algorithm

The most prominent algorithm for solving integer linear programs is the branch-and-bound method [NW88, Sch86]. By using controlled enumeration and relaxation and/or duality, the feasible region S of the optimisation problem IP is partitioned in a set of disjoint subsets $\{S^i \mid i = 1, \ldots, k\}$ and the problem is solved over each of those subsets. In this section the general branch-and-bound method for the problem of integer linear programming is presented.

Definition 4.13 *Let a set S be given. The sets $\{S^i\}_{i=1}^k$ are said to be a* division *of S, if $\bigcup_{i=1}^k S^i = S$. A division is called* partition*, if $S^i \cap S^j = \emptyset$ for $i, j = 1, \ldots, k,\ i \neq j$.*

Proposition 4.3 *Let $\{S^i\}_{i=1}^k$ be a division of S and $z_{IP}^i = \min\{c^T x \mid x \in S^i\}$. Then for $z_{IP} = \min\{c^T x \mid x \in S\}$ holds*

$$z_{IP} = \min_{i=1,\ldots,k} z_{IP}^i.$$

Proposition 4.3 represents the starting point of a *divide and conquer* algorithm. A problem is partitioned into subproblems that are recursively solved by the same algorithm [Meh88]. Then the solution of the entire problem is composed from the solutions of the subproblems.

Carried to the extreme, this can lead to a total enumeration of all elements of S—which is only possible if S contains very few elements. In order for this approach to be practicable the number of investigated partitions has to be as small as possible and the computation time at each node of the enumeration tree has to be kept low. In the branch-and-bound method for the problem of integer linear programming, the computation effort for the subproblems is restricted by solving relaxations or the dual problems. This way it is not necessary to compute the optimal solution of an integer linear problem of its own at each node of the enumeration tree. Let RP^i be a relaxation of IP^i, z_R^i the optimal objective function value of RP^i, and DP^i the dual of IP^i. The following Proposition 4.4 provides criteria for pruning the enumeration tree. An example of a pruned enumeration tree is given in Fig. 4.3 where the children of each node represent the partitions of the problem associated with the parent node. If the algorithm can establish that no further partitioning of a problem is necessary the branch-and-bound tree is pruned at the corresponding node.

Proposition 4.4 *The enumeration tree can be pruned at the node corresponding to S^i if any of the following conditions holds:*

1. *RP^i is infeasible.*
2. *There is an optimal solution x_R^i of RP^i that satisfies $x_R^i \in S^i$ and $z_R^i = c^T x_R^i$.*
3. *$z_R^i \geq \overline{z_{IP}}$, where $\overline{z_{IP}}$ is the value of some feasible solution of IP.*

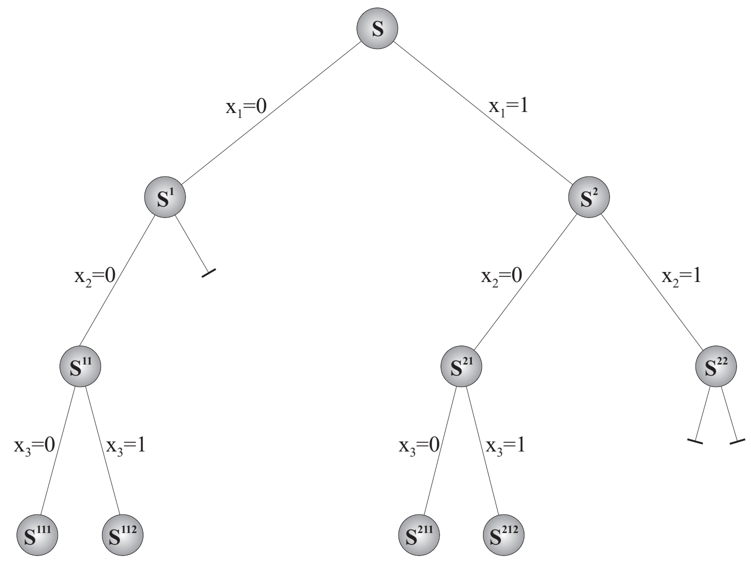


Figure 4.3.: Example of a branch-and-bound tree.

4. *The problem DP^i is unbounded from above.*
5. *DP^i has a feasible solution z^i_{DP} where $z^i_{DP} \geq \overline{z_{IP}}$.*

Explanation 1

1. *If the relaxation has no feasible solution, this implies that there is no feasible integral solution since the feasible region of the relaxation is a superset of the feasible region of the corresponding integer linear program.*
2. *If the optimal solution of a relaxation is a feasible solution of the entire problem, no further decomposition is required.*
3. *The objective function value z^i_R of RP^i always represents a lower bound on the objective function value of IP^i. If a feasible solution of IP has already been found whose objective function value $\overline{z_{IP}}$ is less than or equal to z^i_R, IP^i cannot provide a better feasible solution. Therefore it is not necessary to divide IP^i; the enumeration tree can be pruned at the node for IP^i.*
4. *If the dual has no upper bound, the corresponding primal problem is infeasible. In consequence there is no feasible solution, especially no feasible integral solution.*
5. *Because of the weak duality, each feasible solution of the dual problem represents a lower bound for all feasible solutions of the primal problem. Therefore the best solution found so far cannot be improved by solving IP^i.*

The general branch-and-bound algorithm is presented in pseudocode on page 41 (Algorithm 4.4). Let $L = \{IP^i\}$ be a set of integer linear programs of the form $z^i_{IP} = \min\{c^T x \mid x \in S^i\}$ where $S^i \subset S$. Each problem from L is associated with a lower bound $\underline{z^i} \leq z^i_{IP}$.

$L = \{IP\};\ S^0 = S;\ \underline{z^0} = -\infty;\ \overline{z_{IP}} = \infty;$

while $(L \neq \emptyset)$ {

  /* Node selection and relaxation */

  Choose a problem $IP^i \in L$ and remove it from L;

  Solve the relaxation RP^i;

  Let z^i_R be the optimal objective function value of the relaxation and let x^i_R be an optimal solution.

  /* Pruning */

  if $(z^i_R \geq \overline{z_{IP}})$ {

    Start the next loop iteration. If the relaxation is solved by using a dual algorithm, this can be done as soon as the dual objective function value reaches or exceeds $\overline{z_{IP}}$.

  }

  if $(x^i_R \in S^i)$ {

    if $(c^T x^i_R < \overline{z_{IP}})$ {

      Set $\overline{z_{IP}} = c^T x^i_R$;

      $x^0 = x^i_R$;

    }

    Remove from L all problems with $\underline{z^i} \geq \overline{z_{IP}}$;

    if $(c^T x^i_R = \overline{z_{IP}})$ {

      Start the next loop iteration.

    }

  }

  /* Partitioning */

  Let $\{S^{ij}\}_{j=1}^k$ be a partition of S^i.

  Add to L the problems $\{IP^{ij}\}_{j=1}^k$ where $\underline{z^{ij}} = z^i_R$ for $j = 1, \ldots, k$. For each of the new subproblems the objective function value of their parent's relaxation, i. e. of the current relaxation, is used as lower bound.

}

if $(\overline{z_{IP}} < \infty)$ {

  The solution x^0, whose objective function value is $\overline{z_{IP}} = c^T x^0$ is optimal.

  $x^* = x^0;\ z^* = \overline{z_{IP}};$

else {

  There is no feasible solution: $S = \emptyset$;

}

Figure 4.4.: The Branch-and-bound Algorithm.

There are several algorithms that are available to perform the partitioning. One of them is the *Dakin method* [Dak65]. First the algorithm selects a decision variable x_n and then generates the subproblems IP^{i_1} and IP^{i_2} from IP. Let x^R denote an optimal solution of the relaxation of IP. Then the subproblem IP^{i_1} is built from IP by adding the inequality $x_n \leq \lfloor x_n^R \rfloor$; adding $x_n \geq \lceil x_n^R \rceil$ yields the subproblem IP^{i_2}. An advantage of this method is that only upper and lower bound constraints are added to the relaxation. Therefore the computation effort required to solve the relaxations does not increase with increasing depth of the branch-and-bound tree in contrast to some other methods as the *Land-Doig* method [LD60].

Proposition 4.5 *Let S^t be the feasible region of node t in the enumeration tree. If $\min\{c^T x \mid x \in S_R^t\} < \overline{z_{IP}}$ holds, the node t cannot be pruned.*

This proposition indicates that independently from the shape of the tree, the bounds, i. e. the quality of the relaxations, are the primary factor for the efficiency of a branch-and-bound algorithm. This is the reason why it is essential that the feasible region of the relaxation be close to the feasible region of the integral problem. The solution efficiency is also strongly influenced by the selection of the next node to be examined and the selection of the variable that defines the division. For variable selection no robust methods have been established yet [NW89]. Therefore it is common practice to rely on user-defined *branching priorities*. As an example a binary variable indicating whether a project is to be started or not should be assigned a higher priority than a variable corresponding to detailed decisions within the project.

The input of the *node selection* is a list L of the active subproblems, i. e. the set of nodes of the decision tree that have not been pruned yet. The decision that has to be made is which of those nodes is to be examined next. It can be distinguished between *a priori rules* that determine the development of the tree in advance and *adaptive rules* that use information about the status of the active nodes. The most important a priori rules are *depth first search with backtracking* and *breadth first search*. If the current node is not pruned, the depth first search selects one of its two children to visit next. Backtracking means that if a node is pruned the path from that node to the root of the enumeration tree is traversed until the first node is reached that has a still unvisited child. In breadth first search all nodes of a given depth are traversed before any nodes of the next depth are visited. Due to the large amount of memory required to store the list of the active nodes, this strategy often is not practicable for large problems. Moreover practical experience indicates that feasible solutions can be expected to be located deep in the tree and not near the root [NW88]. Common adaptive rules are the *best bound* and the *best estimate* rules. The best bound rule chooses the node with the smallest lower bound, i. e. whose relaxation has the lowest objective function value. The best estimate rule chooses the node that has the largest probability of representing an optimal solution. If $\hat{z}^i \geq \underline{z}^i$ is an estimate for z_{IP}^i, then that $i \in L$ is selected that minimises $\hat{z}^i$.

Contemporary mathematical programming libraries as, e. g., CPLEX [ILO99] usually provide an extension of branch-and-bound algorithms, the so-called branch-and-cut algorithms. The *branch-and-cut algorithm* is a synthesis of the branch-and-bound algorithm and cutting-plane methods [JNS97]. A *cut* is a valid inequality of the original integer problem *IP* that is not part of the current formulation and that is not satisfied by all feasible points of the current LP relaxation. If a solution of the LP relaxation of *IP* does not satisfy all of the constraints the separation problem is to find a violated inequality. Since the convex hull of the feasible points is a polyhedron such an inequality must exist. A linear time algorithm to detect such violated inequalities has been proposed by Gomory [JNS97]. In pure cutting plane algorithms cuts are added at the root node until an optimal MIP solution has been found [Gom63]. While in practice these algorithms turned out to be time and memory consuming, better results can be achieved with branch-and-cut algorithms [CJP83, PR87]. At each node of the enumeration tree that is not pruned a separation problem is solved. If one or more violated inequalities are found they are added to the formulation and the LP is solved again. If none are found the algorithm branches. There are specialised branch-and-cut algorithms for a variety of combinatorial optimisation problems [JNS97]. For more comprehensive surveys the reader may refer to [NW88, NW89, Sch86, PS82].

5. ILP-Models for the Code Generation Problem

Integer linear programming has a long tradition as a method for investigating general scheduling problems. There is a large amount of literature where ILP formulations for special classes of scheduling problems together with their polyhedral characteristics are presented. A detailed survey would go beyond the scope of this thesis; comprehensive surveys are given in [QS94, BEP+96, Sch96a, Hal97, KW99]. The scheduling problems addressed in those approaches mostly represent operations research problems; well-known problems are the flow shop or the job shop scheduling problems [BEP+96]. It is only during the last years that integer linear programming has gained increased attention in the area of code generation. ILP formulations for instruction scheduling of sequential code have been presented in [Ary85, Leu97, HLW00]. In [GW96, KW98] an ILP model for register allocation is proposed. In the area of software pipelining, ILP models have been developed that aim at computing an optimal unrolling factor and at overlapping different loop iterations in order to get to an optimal schedule [AJLA95, RGSL96, GAG96]. In most of the software pipelining approaches homogeneous VLIW-like architectures without significant resource restrictions are considered and no phase coupling with other code generation phases is attempted.

The use of integer linear programming for phase-coupled code generation however is still rare. In traditional code generation systems, calculation speed plays an important role. Usually fast graph-based heuristics are applied that lead to the generation of suboptimal code. For homogeneous architectures the code quality of these techniques is satisfactory. Further improvements of the code quality are considered less important than any increase of compilation time. Traditional code generation techniques however fail to achieve a satisfactory code quality for irregular architectures, as detailed in Chap. 2. In the area of embedded systems where mostly processors with irregular architectures are used the code quality plays an important role. Therefore in this area higher calculation times are acceptable if they allow for increased code quality. Similar preconditions are given in the area of architectural synthesis. The goal of architectural synthesis is to design the fastest architecture for a given input algorithm that does not exceed a fixed cost maximum, or to design the cheapest architecture for the input algorithm that meets a fixed performance criterion. In order to evaluate the performance capacities of a hardware design it is important to determine the optimal code sequence for the given input algorithm. In [GE92, GE93] and [Zha96] two well-structured ILP models for architectural synthesis have been presented. Both models can be used to perform phase-coupled instruction scheduling, register assignment and functional unit binding. An important characteristic of these approaches is that the findings of polyhedral theory have been applied in order to arrive at well-structured formulations that allow for efficient problem solving.

In previous publications [Käs97, KL98, KL99] we have investigated the applicability of the phase-coupled ILP formulations [GE92, GE93] and [Zha96] for the code generation problem for irregular architectures. Based on the results of these studies the ILP models have been extended to form the basis of the retargetable phase-coupled code optimisation framework presented in this thesis. The use of integer linear programming offers several advantages. First integer linear programming allows for a concise problem description in which irregular hardware characteristics can be easily incorporated. Second the theory of integer linear programming has lead to sophisticated solution techniques [NW88, Sch86, JNS97, Wil93b] so that powerful tools are available for computing the solution even of larger problem instances [ILO99, SN98]. Moreover as will be shown in this thesis, ILP-based approximations can be used to reduce calculation time and yet compute high-quality solutions such that a scalable optimisation quality is achieved.

Depending on the choice of the main decision variables, integer linear programming models for the code generation problem can be classified as time-indexed formulations or order-indexed formulations [KW99]. In time-indexed approaches, the decision variables are based on the discrete point in time the modelled events are assigned to. The ordering of the events is derived from the assignment to points of time. In order-indexed approaches, the semantics of the decision variables reflect the ordering of the modelled events. Here the assignment of the events to points of time is derived from the computed ordering. The SILP-model (Sec. 5.2) is an order-indexed formulation, while the OASIC formulation described in Sec. 5.3 is an example of a time-indexed formulation. As detailed in Sec. 5.5, depending on the architectural design and the kind of optimisation to be performed, both formulation styles exhibit different performance characteristics. Our results indicate that some architectural parameters can be better described by a time-indexed formulation while for others an order-indexed formulation is better suited.

This chapter is organised as follows: after some fundamental definitions, the ILP models of the SILP and the OASIC formulations are presented for sequential program flow in Sec. 5.2 respectively Sec. 5.3. In Sec. 5.4 a mechanism to incorporate the control flow structure of the input procedure in integer linear programming formulations is presented. This modelling is the presupposition of the superblock-based code optimisation presented in Chap. 7. ILP constraints to incorporate irregular hardware characteristics are presented in Sec. 8.4. In Sec. 5.5, the connection between the ILP modelling style (time-indexed vs. order-indexed) and the architectural characteristics of the target processor are detailed.

5.1. Basic Definitions

The goal of the ILP models presented in this chapter is to compute an optimal solution to the problems of instruction scheduling, register assignment and resource allocation for a given input procedure p. Let N_I be a set of nodes where each operation i of p is mapped to an individual node $n_i \in N_I$. As detailed in Chap. 2 the data dependences of the input program define a precedence relation $\preceq \subseteq N_I \times N_I$ where $i \preceq j \Leftrightarrow i \overset{*}{\Rightarrow}_{E_D} j$.

Each operation of the target processor can be executed by dedicated functional units. In general, several alternative functional units are available for the execution of an operation. If the properties of those functional units are identical, they can be considered as instances of the same resource type. Then it is not necessary to distinguish between them in the generated integer linear programs. If the properties of the functional units differ, it may be necessary to differentiate between them when generating the ILP formulations. It is assumed that operations mapped to different functional unit types can be executed in parallel if this is not prevented by data dependences. The number of operations that can be simultaneously executed by the same type of functional units is given by the number of instances of this resource type. This corresponds to the VLIW execution model where an *instruction* can be composed of several *microoperations* that are executed in parallel. In order to determine the feasible assignment of operations to functional unit types the resource graph G_R [Zha96] is used. The resource graph is derived from the TDL-specification of the target architecture (see Chap. 8). The set N_R^F contains a node for each functional unit type of the target processor.

Definition 5.1 (Resource Graph) *The resource graph $G_R = (N_R, E_R)$ is a bipartite directed graph. Its node set $N_R = N_I \cup N_R^F$ contains a node for each operation of the input procedure and a node for each functional unit type of the target processor. The set of edges $E_R \subseteq N_I \times N_R^F$ describes all possible assignments of operations to functional unit types. If $(j, k) \in E_R$, operation $j \in N_I$ can be executed by an instance of the functional unit type $k \in N_R^F$.*

In architectures with heterogeneous register files the set of destination registers available for storing the result of an operation is usually restricted. It may be necessary to differentiate between registers of the same register file, or between different resource aliases. In order to describe the registers available for storing the result of an operation, the register graph $G_A = (N_A, E_A)$ is used. Registers that can be considered equivalent during code generation are grouped to register groups. Since those groups do not necessarily correspond to the register files of the target architecture they are called *abstract register files* (see Sec. 10.1.1). The set N_R^A of abstract register files of a given target architecture is determined from the TDL-description of the target processor (see Chap. 8).

Definition 5.2 (Register Graph) *The register graph $G_A = (N_A, E_A)$ is a bipartite directed graph. Its node set $N_A = N_I^A \cup N_R^A$ contains a node for each operation* *of the input procedure that performs a write access to a register and a node for each abstract register file of the target processor. The set of edges $E_A \subseteq N_I^A \times N_R^A$ describes which registers can be used to store the result of each operation in N_I^A. If $(j, r) \in E_A$, operation $j \in N_I^A$ can store its result in a register of the abstract register file $r \in N_R^A$.*

Before generating the integer linear programs, an interval is calculated for each operation i containing all control steps in which the execution of i can be started in any feasible schedule. This interval is defined as $N(i) = \{asap(i), \dots alap(i)\}$ where $asap(i)$ denotes the *as soon as possible*, $alap(i)$ the *as late as possible* control step for the starting time of operation i [Fou81]. The *asap* control step is the earliest control step in which i can be started without violating any data dependences; it is calculated as the longest path to operation i in the data dependence graph. The computation of the *alap* control step requires an upper bound U on the execution time of the input program to be given. The longest path to operation i is calculated in the inverse data dependence graph and its length is subtracted form U, yielding $asap(i)$. It denotes the latest control step in which the execution of i can be started such that the upper bound U of the execution time is not exceeded and no data dependences are violated. The *asap* and *alap* values are refined by taking into account the number of available hardware resources. The algorithm for computing the *asap* and *alap* values is summarised in Sec. 9.4.

All decision variables introduced in the remainder of this chapter are assumed to be non-negative. The non-negativity constraints are not explicitly listed in the description of the ILP formulations. Throughout this chapter the following terminology is used:

- $asap(i)$ denotes the as soon as possible, $alap(i)$ the as late as possible control step for the starting time of operation i.
- $N(i) = \{asap(i), \dots, alap(i)\}$ is a superset of all control steps in which i can be started in any feasible schedule.
- $t_i \in N(i)$ denotes the starting time for the execution of an operation i.
- $Q_i^k \in \mathbb{N}$ represents the amount of time required to execute operation i by an instance of functional unit type k.
- $L_j^k \in \mathbb{N}$ represents the latency of the functional unit type $k \in N_R^F$ executing operation j, i. e. the minimal time interval between two successive data inputs to the same instance of k.
- The number of available instances of a functional unit type $k \in N_R^F$ is denoted R_k.
- The length of the life range of a variable defined by operation i is denoted $\tau_i \in \mathbb{N}$.

5.2. The SILP Model

The SILP formulation (*Scheduling and Allocation with Integer Linear Programming*) has been developed by Li Zhang in 1996 [Zha96] as an approach for phase-coupled scheduling, allocation of functional units and register assignment. In [Käs97, KL98, KL99] it has been modified to incorporate irregular hardware characteristics and refined to meet the requirements of phase-coupled postpass optimisations; this extended formulation is presented in the remainder of this section.

The SILP model is an order-indexed formulation. The main decision variables describe the flow of the hardware resources of the target architecture through the operations of the input procedure. Each flow variable $x_{ij}^k \in \{0, 1\}$ indicates whether operation i passes an instance of the functional unit type k to operation j. The starting time for the execution of each operation is determined from the calculated resource flow.

Most ILP formulations used in the area of code generation are time-indexed models. For classical scheduling problems order-indexed models have been proposed where the decision variables specify whether one job transitively precedes another one [Pot80, Pet88, Fis92, NS92]. The difference of the SILP formulation is that the decision variables indicate whether one operation is the immediate predecessor of another operation on a specified resource type. To our knowledge no order-indexed formulations other than SILP have been developed for the code generation problem. Our results however indicate that order-indexed formulations can be superior to time-indexed formulations depending on the problem dimension and the architecture of the target processor.

5.2.1. Basic Formulation

In the SILP formulation, the integer linear programs are generated from the *resource flow graph* $G_F = (N_F, E_F)$. This graph describes the execution of the input procedure as a flow of the available execution units through the operations of the procedure. For each functional unit type a separated flow network is generated. Each functional unit type $k \in N_R^F$ is represented by two nodes $k_Q, k_S \in N_F$ where the node k_Q represents the source and k_S the sink of the corresponding flow network. The first operation to be executed on resource type k gets an instance k_r of this type from the source node k_Q; after the execution has been completed, it passes k_r to the next operation using the same resource type. The last operation using a certain instance of the functional unit type k returns it to k_S. The number of simultaneously used instances of each resource type must never exceed the number R_k of available instances.

Definition 5.3 (Resource Flow Graph) *The resource flow graph is a directed graph* $G_F = (N_F, E_F)$ *with a set* $N_F = N_I \cup \bigcup_{k \in N_R^F} \{k_Q, k_S\}$ *of nodes and a set* $E_F = \bigcup_{k \in N_R^F} E_F^k \subseteq N_F \times N_F \times N_R^F$ *of labelled edges. The edges of* E_F^k *describe all* *possible resource flows with respect to the functional unit type k.*

$$\begin{aligned} E_F^k &= \{(i,j,k) \mid i,j \in N_I \;\wedge\; (i,k) \in E_R \;\wedge\; (j,k) \in E_R \;\wedge\; j \not\preceq i \;\wedge\; i \neq j\} \\ &\cup \;\{(k_Q, j, k) \mid (j,k) \in E_R\} \\ &\cup \;\{(j, k_S, k) \mid (j,k) \in E_R\} \end{aligned}$$

Each edge $(i,j,k) \in E_F$ is associated with a binary flow variable $x_{ij}^k \in \{0,1\}$. An instance of the functional unit type k is moved along the edge (i,j,k) from i to j if and only if $x_{ij}^k = 1$.

Example 5.1 Fig. 5.1 shows an example resource flow graph for two functional unit types ALU and MUL. It is assumed that the operations that have to be executed by the ALU are independent, such that there is no restriction of the operation ordering. Assume further that operation n is data dependent on m such that in the subgraph of the functional unit type MUL there is only one feasible operation ordering. If the target architecture disposes of two ALUs, i. e. if there are two instances of the functional unit type ALU, the operations i and j can be executed in parallel; otherwise both have to share the same instance of ALU and have to be sequentialised.

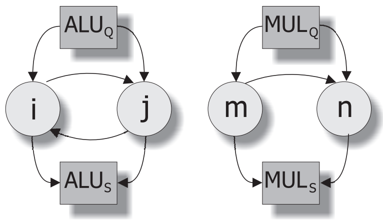


Figure 5.1.: Example of a resource flow graph.

□

In the following both the source and the sink node of a resource type k are denoted as k in order to simplify the notation of the equations; from the context it is always clear whether a source or a sink node is meant.

An algebraic representation of the resource flow graph can be given with the following flow equations. For each node the value of the flow entering the node must be equal to the value of the flow leaving this node; this is guaranteed by the flow conservation constraints (5.1). Additionally it has to be ensured that each operation is executed by exactly one resource instance; for this purpose the execution constraints are used (5.2).

$$\Phi_j^k - \Psi_j^k = 0 \qquad \forall\, j \in N_I \;\forall\, k \in N_R^F : \;(j,k) \in E_R \tag{5.1}$$

$$\sum_{\substack{k \in N_R^F: \\ (j,k) \in E_R}} \Phi_j^k = 1 \qquad \forall\, j \in N_I \tag{5.2}$$

The variable Φ_j^k denotes the flow entering a node $j \in N_I$, and Ψ_j^k denotes the flow leaving it. The exact definitions read as follows:

$$\Phi_j^k = \sum_{(i,j,k) \in E_F} x_{ij}^k \quad , \qquad \Psi_j^k = \sum_{(j,i,k) \in E_F} x_{ji}^k \tag{5.3}$$

Example 5.2 The flow conservation constraints generated for the resource flow graph of Fig. 5.1 read as follows:

$$\begin{aligned}
x_{ALU,i}^{ALU} + x_{j,i}^{ALU} - x_{i,ALU}^{ALU} - x_{i,j}^{ALU} &= 0 \\
x_{ALU,j}^{ALU} + x_{i,j}^{ALU} - x_{j,ALU}^{ALU} - x_{j,i}^{ALU} &= 0 \\
x_{MUL,m}^{MUL} - x_{m,MUL}^{MUL} - x_{m,n}^{MUL} &= 0 \\
x_{MUL,n}^{MUL} + x_{m,n}^{MUL} - x_{n,MUL}^{MUL} &= 0 \\
x_{ALU,i}^{ALU} + x_{j,i}^{ALU} &= 1 \\
x_{ALU,j}^{ALU} + x_{i,j}^{ALU} &= 1 \\
x_{MUL,m}^{MUL} &= 1 \\
x_{MUL,n}^{MUL} + x_{m,n}^{MUL} &= 1
\end{aligned}$$

□

The number of resource flow edges and thus the number of flow variables x_{ij}^k can be reduced by the *asap*/*alap* analysis as presented in Chap. 9.4. If for two operations i and j the inequality $L_i^k + asap(i) > alap(j)$ holds for all functional unit types i can be assigned to, no resource flow edge (i, j, k) is needed for any k. The reason is that operation i is always executed after operation j such that i can never pass a resource instance to j.

If each operation j can only be executed by exactly one resource type k, the constraints 5.1 and 5.2 are replaced by

$$\Phi_j^k = 1 \qquad \forall\, j \in N_I \qquad \forall\, k \in V_R^F : (j,k) \in E_R \tag{5.4}$$

$$\Psi_j^k = 1 \qquad \forall\, j \in N_I \qquad \forall\, k \in V_R^F : (j,k) \in E_R \tag{5.5}$$

The execution time of an operation and the corresponding functional unit latency may vary with different functional unit types the operation can be mapped to. If an operation is mapped to a functional unit type k, one of the instances of k must reach that operation by an incoming flow edge. Thus the execution time w_j of an operation j can be defined as

$$w_j = \sum_{k \in N_R^F} \sum_{(i,j,k) \in E_F} Q_j^k \cdot x_{ij}^k \qquad \forall\, j \in N_I \tag{5.6}$$

Similarly the functional unit latency associated with the execution of an operation j can be calculated as

$$z_j = \sum_{k \in N_R^F} \sum_{(i,j,k) \in E_F} L_j^k \cdot x_{ij}^k \qquad \forall\ j \in N_I \tag{5.7}$$

If the execution time (latency) of an operation does not vary depending on the chosen functional unit type, w_j (z_j) are considered as constants when solving the integer linear programs.

The number of operations simultaneously executed by a functional unit type k must never exceed the number of available instances of that resource. This can be achieved by restricting the maximal value of the flow leaving the node of the functional unit type k in the resource flow graph.

$$\sum_{(k,j,k) \in E_F} x_{kj}^k \leq R_k \qquad \forall\ k \in N_R^F \tag{5.8}$$

Example 5.3 Let $R_{ALU} = 2$ and $R_{MUL} = 1$. Then the resource constraints generated for the resource flow graph of Fig. 5.1 read as follows:

$$\begin{aligned} x_{ALU,i}^{ALU} + x_{ALU,j}^{ALU} &\leq 2 \\ x_{MUL,m}^{MUL} + x_{MUL,n}^{MUL} &\leq 1 \end{aligned}$$

□

Instruction scheduling requires a modelling of the data dependences among the operations of the input procedure. The constraints to be generated depend on the type of the dependences. In case of a true dependence $(i, j, r, t) \in E_D^t$, j may only be started after the execution of i has been finished and the result is available:

$$t_j - t_i \geq w_i \quad \forall\ (i, j, r, t) \in E_D^t \tag{5.9}$$

If there is an output dependence between two operations i and j ($(i, j, r, o) \in E_D^o$), j must not be allowed to write its result before the result of i has been written:

$$t_j - t_i \geq w_i - w_j + 1 \quad \forall\ (i, j, r, o) \in E_D^o \tag{5.10}$$

In case of anti dependences $(i, j, r, a) \in E_D^a$ it has to be ensured that j writes its result after the execution of i has been started.

$$t_i \leq t_j + w_j - 1 \quad \forall\ (i, j, r, a) \in E_D^a \tag{5.11}$$

Finally dedicated constraints are required in order to synchronise the starting time of the operations with the values of the flow variables. If two operations i and

j are both assigned to the same functional unit type k, j must await the execution of i if an instance of k is passed along the edge $(i, j) \in E_F^k$, i.e. if $x_{ij}^k = 1$.

$$t_j - t_i \geq z_i + \left(\sum_{\substack{k \in N_R^F: \\ (i,j,k) \in E_F}} x_{ij}^k - 1 \right) \cdot \alpha_{ij} \qquad \forall\ (i, j) \in E_F^k \tag{5.12}$$

The constants α_{ij} have to be chosen large enough such that if i and j are not mapped to the same resource instance the inequality is always valid. The better the feasible region of the relaxation P_F approximates the feasible region of the integer linear program P_I the more efficient the integer linear program can be solved. The larger the value of α_{ij} is chosen, the larger grows the distance between P_F and P_I. Since this can strongly affect the solution efficiency, the smallest possible value is chosen for α_{ij}. The tightest solution polytope for a serial constraint $t_j - t_i \geq z_i + (x_{ij} - 1)\alpha_{ij}$ is realized by $\alpha_{ij} = z_i - asap(j) + alap(i)$. The proof is given in [Zha96]. A serial constraint only has to be generated if $z_i + alap(i) \geq asap(j)$. Otherwise operation j is always executed after operation i such that the sequentialisation has not to be explicitly enforced.

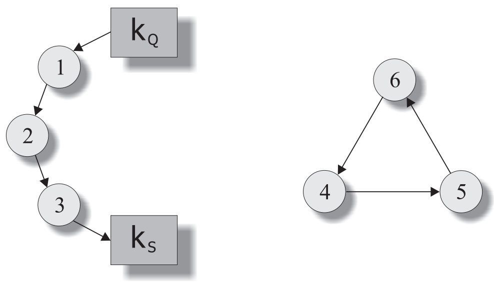


Figure 5.2.: Infeasible resource flow.

Apart from the sequentialisation of operations, the serial constraints are required in order to ensure the well-definedness of the flow modelling. The infeasible resource flows shown in Fig. 5.2 do not violate any of the constraints (5.1, 5.2). The serial constraints however imply that $t_4 < t_5 < t_6 < t_4$ such that the flows of Fig. 5.2 cannot represent a feasible solution.

The goal of the ILP-formulation is to minimise the execution time of the input procedure. The execution time is measured in control steps (clock cycles). So the integer linear program for the problem of instruction scheduling and resource allocation can be summarised as follows:

- Objective Function

$$\min \quad M_{steps} \tag{5.13}$$

- ILP Constraints

1. Time Constraints
   The maximal number of control steps M_{steps}, i. e. the value of the objective function is defined as the starting time of the last operation to be executed.
   $$t_j \leq M_{steps} \qquad \forall\ j \in N_I \tag{5.14}$$
2. Precedence Constraints
   The data dependences of the input program have to be respected.
   $$\begin{array}{rcll} t_j - t_i & \geq & w_i & \forall\ (i,j,r,t) \in E_D^t \\ t_j - t_i & \geq & w_i - w_j + 1 & \forall\ (i,j,r,o) \in E_D^o \\ t_i & \leq & t_j + w_j - 1 & \forall\ (i,j,r,a) \in E_D^a \end{array}$$
3. Flow Conservation Constraints
   The flow entering a node must have the same value as the flow leaving that node.
   $$\Phi_j^k - \Psi_j^k \quad = \quad 0 \qquad \forall\ j \in N_D,\ \forall\ k \in N_R^F:\ (j,k) \in E_R$$
4. Execution Constraints
   Each operation must be executed exactly once by exactly one hardware component.
   $$\sum_{\substack{k \in N_R^F: \\ (j,k) \in E_R}} \Phi_j^k \quad = \quad 1 \qquad \forall\ j \in N_D$$
5. Resource Constraints
   The number of instances of all functional unit types must not be exceeded.
   $$\sum_{(k,j,k) \in E_F} x_{kj}^k \leq R_k \qquad \forall\ k \in N_R^F$$
6. Serial Constraints
   If two operations i and j are both assigned to the same functional unit type k, j must await the execution of i if an instance of k is passed along the edge $(i,j,k) \in E_F$, i. e. if $x_{ij}^k = 1$. Define $\alpha_{ij} = z_i - asap(j) + alap(i)$, then the constraints are defined as follows:
   $$t_j - t_i \geq z_i + \left(\sum_{\substack{k \in N_R^F: \\ (i,j,k) \in E_F}} x_{ij}^k - 1 \right) \cdot \alpha_{ij} \qquad \forall\ i,j : \exists k : (i,j,k) \in E_F$$

The resource flow concept presented so far is capable of modelling hardware architectures where the pipeline behaviour can be described by the two parameters execution time and latency. Each operation is assigned to exactly one functional unit. If additional restrictions for the usage of the system buses have to be taken into account, the modelling is extended. Assume, e. g., that the number of results that can be written to the system bus per clock cycle is restricted. The usage of the result bus can be modelled as a dedicated resource flow that has to be considered in addition to the resource flows of the functional units. Then each operation must be assigned to a functional unit and additionally to the result bus. The access restrictions with respect to the bus are represented by resource constraints, the flow constraints (5.1) and the execution constraints (5.2) guarantee the correctness of the flow modelling. The synchronisation of the operations writing to the result bus is enforced by dedicated serial constraints

$$t_j + w_j - t_i - w_i \geq z_B + \left(\sum_{j \in N_I} x_{ij}^B - 1 \right) \alpha_{ij}$$

The serial bus constraints have to be generated for each pair of operations that may write their result in the same clock cycle. The latency z_B corresponds to the number of clock cycles required to transfer a result from the functional unit to the destination location across the bus. The difference to the serial constraints (5.12) is that the completion time of the operations $(t_i + w_i)$ is at the base of the serial constraints and not the starting times. A separated modelling of the result bus is not necessary for the Analog Devices ADSP-2106X SHARC [Ana95], the Infineon C16x [Sie96], the Texas Instruments TMS320C6x [Tex98b] and the Infineon TriCore [Inf00]; however it is required for the Philips TriMedia TM1000 [Phi97] (see Sec. 10.2).

5.2.2. Integration of Register Assignment

In this section, the basic SILP formulation is extended to incorporate the register assignment problem. The register assignment problem is formulated as a register distribution problem based on the *register flow graph*. This graph is similar to the compatibility graph of [ST94] where each lifetime is mapped to a node and edges connect all nodes whose life ranges are disjoint. Let N_I^A be a set of nodes where each node represents an operation of the input program that performs a write access to a register. Each node $j \in N_I^A$ is connected with the register resource nodes representing the abstract register files r where $(j, r) \in E_A$.

Definition 5.4 (Register Flow Graph) *The register flow graph $G_Z = (N_Z, E_Z)$ is a directed graph with a set $N_Z = N_I^A \cup \bigcup_{r \in N_R^A} \{r_Q, r_S\}$ of nodes and a set $E_Z = \bigcup_{r \in N_R^A} E_Z^r \subseteq N_Z \times N_Z \times N_R^A$ of directed marked edges. Each node $j \in N_I^A$ represents an operation performing a write access to a register generating a variable with life* *range τ_j. Each edge $(i,j,r) \in E_Z$ represents a possible flow of an element register of the abstract register file $r \in N_R^A$ from i to j.*

$$\begin{aligned} E_Z^r \;=\; & \{(i,j,r) \mid i,j \in N_I^A \;\wedge\; (i,r) \in E_A \;\wedge\; (j,r) \in E_A \;\wedge j \not\preceq i \;\wedge \\ & \; i \neq j \wedge nso(i,j) = true\} \;\cup\; \{(r_Q, j, r) \mid (j,r) \in E_A\;\} \\ & \cup \{(j, r_S, r) \mid (j,r) \in E_A\;\} \end{aligned}$$

Each edge $(i,j,r) \in E_Z$ is associated with a binary register flow variable $x_{ij}^r \in \{0,1\}$. If $x_{ij}^r = 1$, the same register is used to store the variables generated by operations i and j. No register flow edge is required between two operations i and j, if it is statically known that the life ranges of the variables defined by i and j overlap ($nso(i,j) = true$). This is, e. g., the case if both define an input operand of another operation l.

Life ranges of variables are defined by true dependences. If an operation i performs a write access to a register used by the operations $j_1, \ldots, j_m$, the variable defined by i is alive until the last use of that variable. The modelling of the life ranges requires the constraints for the true dependences to be modified. The distance between the control step where the destination operand defined by an operation i becomes available and a use j is measured by an integer variable $b_{ij} \geq 0$. The constraints (5.9) for true dependences are replaced by the following equivalent constraints:

$$t_j - t_i - b_{ij} = w_i \qquad \forall\ (i,j,t,r) \in E_D^t \tag{5.15}$$

For the life range of the register defined by operation i the following inequality holds:

$$\tau_i \geq b_{ij} + w_i \qquad \forall\ (i,j,t,r) \in E_D^t \tag{5.16}$$

An operation j is only allowed to define the same register as a preceding operation i, if the execution of j starts after the life range τ_i of the variable defined by i has expired. This is ensured by the register serial constraints

$$t_j - t_i \geq \tau_i - w_j + \left(\sum_{\substack{r \in N_R^A: \\ (i,j,r) \in E_Z}} x_{ij}^r - 1 \right) \cdot U \quad \forall\ i,j : \exists\ k\ (i,j,k) \in E_Z \tag{5.17}$$

The value U is an upper bound of the execution time of the input program. If i and j do not use the same register, i. e. if $\sum_{\substack{r \in N_R^A: \\ (i,j,r) \in E_Z}} x_{ij}^r = 0$, constraint (5.17) is always satisfied.

Similarly to Sec. 5.2.1 the modelling of the register flow graph requires flow conservation constraints, resource constraints and execution constraints. Then, the the mixed integer linear program for integrated instruction scheduling, resource allocation and register assignment can be given as follows:

$$\min \quad M_{steps} \tag{5.18}$$

subject to

$$
\begin{aligned}
t_j &\leq M_{steps} \qquad \forall\ j \in N_I \\
t_j - t_i - b_{ij} &= w_i \qquad \forall\ (i,j,t,r) \in E_D^t \\
t_j - t_i &\geq w_i - w_j + 1 \qquad \forall\ (i,j,o,r) \in E_D^o \\
t_i &\leq t_j + w_j - 1 \qquad \forall\ (i,j,a,r) \in E_D^a \\
\Phi_j^k - \Psi_j^k &= 0 \qquad \forall\ j \in N_I,\ \forall\ k \in N_R^F : (j,k) \in E_R \\
\sum_{\substack{k \in N_R^F: \\ (j,k) \in E_R}} \Phi_j^k &= 1 \qquad \forall\ j \in N_I \\
\sum_{(k,j,k) \in E_F} x_{kj}^k &\leq R_k \qquad \forall\ k \in N_R^F \\
t_j - t_i &\geq z_i + \left(\sum_{\substack{k \in N_R^F: \\ (i,j,k) \in E_F}} x_{ij}^k - 1 \right) \cdot \alpha_{ij} \qquad \begin{array}{l} \forall\ i,j \in N_I : \\ \exists\ k : (i,j,k) \in E_F \end{array} \\
\tau_i &\geq b_{ij} + w_i \qquad \forall\ (i,j,t,r) \in E_D^t \\
\Phi_j^r - \Psi_j^r &= 0 \qquad \forall\ j \in N_I^A,\ \forall\ r \in N_R^A :\ (j,r) \in E_A \qquad (5.19) \\
\sum_{\substack{r \in N_R^A: \\ (j,r) \in E_A}} \sum_{(i,j,r) \in E_Z} x_{ij}^r &= 1 \qquad \forall\ j \in N_I^A \qquad (5.20) \\
\sum_{(r,j,r) \in E_Z} x_{rj}^r &\leq R_r \qquad \forall\ r \in N_R^A \qquad (5.21) \\
t_j - t_i &\geq \tau_i - w_j + \left(\sum_{\substack{r \in N_R^A: \\ (i,j,r) \in E_Z}} x_{ij}^r - 1 \right) \cdot U \qquad \begin{array}{l} \forall\ i,j : \\ \exists\ r :\ (i,j,r) \in E_Z \end{array}
\end{aligned}
$$

From the solution of the integer linear program the physical register assignment for sequential code can directly be derived; the necessary extensions for programs with complex control flow are described in Sec. 7.2.

5.2.3. The Structure of the SILP Polytope

In [Zha96] it has been shown that the polytope of the precedence constraints is integral if the execution time of the machine operations does not depend on the functional unit binding. If each operation can be mapped to exactly one functional unit type, also the polytope of the constraints (5.4) and (5.5) representing the flow conservation and execution constraints is integral. In the following we will prove that the polytopes of the original flow conservation constraints (5.1), execution constraints (5.2) and the resource constraints (5.8) are integral. The combined polytope of the flow conservation constraints and the execution constraints is not integral in the general case; this is shown by a counterexample. The combined polytope of the flow conservation and the resource constraints however is integral as well as the combined polytope of the execution and the resource constraints. The same integrality properties also hold for the polytopes of the corresponding register flow constraints (5.19),(5.20), (5.21).

In the coefficient matrix of the SILP polytope each row corresponds to one constraint and each column to a decision variable x_{ij}^k. The strategy of all following proofs is to show that the coefficient matrices of the respective constraints are totally unimodular. Thus it follows from Theorem 4.2 that the corresponding polytopes are integral.

Lemma 5.1 *The polytope of the flow conservation constraints (5.1) is integral.*

Proof: The flow conservation constraints have been defined as follows:

$$\Phi_j^k - \Psi_j^k = 0 \qquad \forall\ j \in N_D\ \forall\ k \in N_k :\ (j,k) \in E_R$$

Let $C \in \{0,1,-1\}^{m_1 \times n_1}$ be the coefficient matrix of the flow conservation constraints. Consider the variable x_{ij}^k. This variable is used to describe the flow of resource type k entering operation j and the flow of resource type k leaving operation i. In the flow conservation constraint for operation i and resource type k x_{ij}^k is used with coefficient -1 and in the flow conservation constraint for operation j and resource type k its coefficient is 1. In all other constraints it does not appear, i. e. its coefficient is equal to 0.

Thus in each column there are exactly two nonzero entries whose sum is equal to zero. Then it follows from Theorem 4.4 that C is totally unimodular. □

Lemma 5.2 *The polytope of the execution constraints (5.2) is integral.*

Proof: The execution constraints have been defined as follows:

$$\sum_{\substack{k \in N_A: \\ (j,k) \in E_R}} \Phi_j^k = 1 \qquad \forall\ j \in N_D$$

Let $E \in \{0,1\}^{m_2 \times n_2}$ be the coefficient matrix of the execution constraints. Only in the constraint generated for operation j the coefficient of a variable x_{ij}^k is equal to 1; in all other constraints it is equal to zero. Thus each column contains exactly one nonzero entry. Then it follows from Theorem 4.4 that E is totally unimodular. □

The coefficient matrix of the combined polytope of the flow conservation constraints (5.1) and the execution constraints (5.2) however is not totally unimodular. This can be shown by a simple example. Let three operations $N_I = \{1,2,3\}$ and two resource types $N_R^F = \{A,B\}$ and the resource flow graph of Fig 5.3 be given. The dotted edges represent the flow edges of resource type A, the dashed ones the

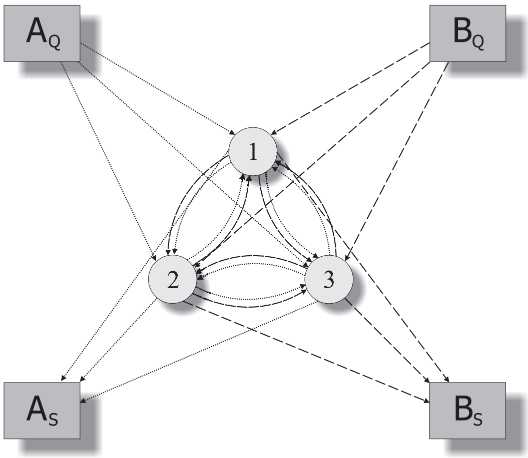


Figure 5.3.: Example of a resource flow graph.

flow edges of B. Then the following flow conservation and the execution constraints are generated:

$$
\begin{array}{rcll}
x^A_{A1} + x^A_{21} + x^A_{31} - x^A_{12} - x^A_{13} - x^A_{1A} & = & 0 & (i) \\
x^A_{A2} + x^A_{12} + x^A_{32} - x^A_{21} - x^A_{23} - x^A_{2A} & = & 0 & \\
x^A_{A3} + x^A_{13} + x^A_{23} - x^A_{31} - x^A_{32} - x^A_{3A} & = & 0 & \\
x^B_{B1} + x^B_{21} + x^B_{31} - x^B_{12} - x^B_{13} - x^B_{1B} & = & 0 & \\
x^B_{B2} + x^B_{12} + x^B_{32} - x^B_{21} - x^B_{23} - x^B_{2B} & = & 0 & (ii) \\
x^B_{B3} + x^B_{13} + x^B_{23} - x^B_{31} - x^B_{32} - x^B_{3B} & = & 0 & \\
x^A_{A1} + x^A_{21} + x^A_{31} + x^B_{B1} + x^B_{21} + x^B_{31} & = & 1 & \\
x^A_{A2} + x^A_{12} + x^A_{32} + x^B_{B2} + x^B_{12} + x^B_{32} & = & 1 & (iii) \\
x^A_{A3} + x^A_{13} + x^A_{23} + x^B_{B3} + x^B_{13} + x^B_{23} & = & 1 & (iv)
\end{array}
$$

Now consider some entries of the rows $(i) - (iv)$ in the coefficient matrix.

$$
\begin{array}{rrrrr}
 & x^A_{12} & x^A_{13} & x^B_{12} & x^B_{23} \\
(i) & -1 & -1 & 0 & 0 \\
(ii) & 0 & 0 & 1 & -1 \\
(iii) & 1 & 0 & 1 & 0 \\
(iv) & 0 & 1 & 0 & 1
\end{array}
$$

Since a matrix A is totally unimodular if and only if its transpose is totally unimodular (Theorem 4.5), Theorem 4.6 can equivalently be formulated as follows: A matrix $A \in \{0, 1, -1\}^{m \times n}$ is totally unimodular if and only if for every $Q \subseteq \mathcal{M} = \{1, \ldots, m\}$ there exists a partition Q_1, Q_2 of Q such that

$$
\left| \sum_{i \in Q_1} a_{ij} - \sum_{i \in Q_2} a_{ij} \right| \leq 1 \text{ for } j = 1, \ldots, n. \tag{5.22}
$$

However in the example there is no partitioning of $Q = \{(i), (ii), (iii), (iv)\}$ that satisfies condition (5.22). Because of the variables x_{12}^A and x_{13}^A, $(iii), (iv)$ must always be in the same partition as (i). Because of x_{12}^B, (ii) must not be in the same partition as (iii). Thus the partitioning $Q_1 = \{(i), (iii), (iv)\}$, $Q_2 = \{(ii)\}$ results. But this partitioning violates condition (5.22) in the column of variable x_{23}^B. Thus it follows from Hoffman and Kruskal's theorem that the coefficient matrix of the flow conservation and the execution constraints does not imply the integrality of the polytope.

Lemma 5.3 *The polytope of the resource constraints (5.8) is integral.*

Proof: The resource constraints have been defined as follows:

$$\sum_{(k,j,k) \in E_F} x_{kj}^k \leq R_k \qquad \forall\, k \in N_R^F \tag{5.23}$$

Let $R \in \{0,1\}^{m_3 \times n_3}$ be the coefficient matrix of the resource constraints. The coefficient of a variable x_{ij}^k is 1 only in the constraint generated for resource type k; in all other rows of R it is equal to zero. Thus each column contains exactly one nonzero entry. Hence R is totally unimodular. □

Lemma 5.4 *The polytope of the flow conservation and the resource constraints is integral.*

Proof: Consider the matrix $F = \begin{pmatrix} C \\ R \end{pmatrix} \in \{0, 1, -1\}^{m_4 \times n_4}$ where $m_4 = m_1 + m_3$. The only variables whose coefficients are 1 in the resource constraints have the form x_{rj}^r for a resource type r and an operation j. Those variables are used exactly once in the specification of the flow conservation constraints, when specifying the resource flow entering the node of an operation j. Thus the columns representing variables of the form x_{rj}^r contain exactly one nonzero coefficient in C and in R; in both matrices the values of the coefficients are equal to 1. Therefore for each $Q \subseteq \{1, \ldots, m_1 + m_3\}$ there exists a partitioning satisfying condition (5.22); the rows from C are inserted in Q_1, the rows from R in Q_2. Thus the matrix F is totally unimodular. □

Lemma 5.5 *The polytope of the execution and the resource constraints is integral.*

Proof: Define the matrix $F_2 = \begin{pmatrix} E \\ R \end{pmatrix} \in \{0,1\}^{m_5 \times n_5}$ where $m_5 = m_2 + m_3$. Since each column of E and each column of R contains at most one nonzero entry, for each $Q \subseteq \{1, \ldots, m_2 + m_3\}$ there exists a partitioning satisfying condition (5.22). The partitioning can be constructed by inserting the rows from E into Q_1 and the rows from R in Q_2. Hence F_2 is totally unimodular. □

In [Zha96] it has been shown that if the execution time of the operations does not depend on the functional unit assignment, the polytope of the precedence constraints is integral. Thus, deviations of the feasible region of the SILP model from the integral polytope are only due to the serial constraints and to the combination of flow conservation and execution constraints.

5.2.4. Valid Inequalities

In this section, some valid inequalities are presented that can be used to increase the efficiency of the ILP formulation. By adding the valid inequalities, the feasible area P_F of the relaxation is approximated more closely to the solution polytope of the integral problem (cf. Chap4) [Zha96].

- The following constraint must be satisfied:

$$x_{ij}^k + x_{ji}^k \leq 1 \qquad \forall\ i, j \in N_I, (i, j, k) \in E_F^k$$

  If this inequality is violated, $x_{ij}^k = x_{ji}^k = 1$ holds. Thus the nodes of i and j cannot be reached from a resource node. The resulting flow cannot represent a feasible solution. An illustration is given in Fig. 5.4.

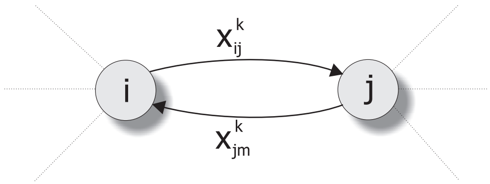


Figure 5.4.: Valid inequality: $x_{ij}^k + x_{ji}^k \leq 1$.

- Another valid inequality reads as follows:

$$x_{jm}^k + x_{nj}^k \leq 1 \qquad \forall\ (m, n) \in E_D\ \wedge\ (j, m), (n, j) \in E_F^k$$

  An illustration is given in Fig. 5.5. If $x_{jm}^k = 1$ and $x_{nj}^k = 1$, operation n must be executed before operation m. This is a contradiction to the data dependence $(m, n) \in E_D^t$, represented by the bold edge in the figure.

5.2.5. Complexity

In the following the number of constraints and the number of integer variables are given as a measure for the complexity of the ILP formulation [NW88].

For each operation $j \in N_I$ one time constraint (inequality 5.14) has to be generated, such that there are $|N_I|$ time constraints altogether. The number of precedence constraints (inequalities 5.9 - 5.11) can be determined as $|E_D| = \mathcal{O}(|N_D|^2) = \mathcal{O}(|N_I|^2)$. There are at most $|N_I| \cdot |N_R^F|$ flow conservation constraints (equation

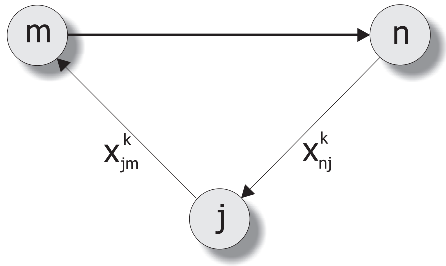


Figure 5.5.: Valid inequality: $x^k_{jm} + x^k_{nj} \leq 1$.

5.1), and at most $|N_I|$ execution constraints (equation 5.2). The number of resource constraints is determined by the number $|N_R^F|$ of functional unit types of the target architecture and is independent of the input procedure. The number of serial constraints (inequality 5.12) can be given as

$$\begin{aligned} N_s &= |\{(i,j,k) \in E_F | i,j \in N_I \wedge (i,k) \in E_R \wedge (j,k) \in E_R \wedge j \not\prec i \wedge i \neq j\}| \\ &= \mathcal{O}(|N_I|^2) \end{aligned}$$

Thus an upper bound for the total number of constraints is $\mathcal{O}(|N_I|^2)$; this bound is also valid for the extended model that incorporates the register assignment problem.

Now let us turn to the number of decision variables. In order to model the starting time for the execution of the operations $|N_I|$ variables are required. The number of flow variables is bounded by $\mathcal{O}(|N_R^F| \cdot |N_I|^2 + |N_R^A| \cdot |N_I^A|^2) = \mathcal{O}((|N_R^F| + |N_R^A|)|N_I^2|)$. In [Zha96] it has been shown that if the problem of functional unit assignment can be neglected only the flow variables used in the serial constraints have to be specified as integers. Thus the bound $\mathcal{O}(|N_I|^2)$ is a pessimistic upper bound for the number of the variables explicitly specified as integers.

5.3. The OASIC Model

The OASIC formulation (*Optimal Architectural Synthesis with Interface Constraints*) has been developed by Gebotys/Elmasry [GE92, GE93] for simultaneously performing scheduling, functional unit allocation and register assignment. In [KL98, KL99] it has been modified to incorporate irregular hardware characteristics and refined to meet the requirements of phase-coupled postpass optimisations; this extended formulation is presented in the remainder of this section. The OASIC model is a time-indexed formulation. The main decision variables are called $x^k_{jn} \in \{0,1\}$, where $x^k_{jn} = 1$ means, that microoperation j is assigned to the nth control step ($n \geq 1$) in the generated schedule and is executed by an instance of the functional unit type k.

5.3.1. Basic Formulation

In the OASIC model, polyhedral theory is used to formulate constraints that identify integral facets of the solution polytope and incorporate them in the generated integer linear programs. A preliminary ILP model is transformed into the node packing problem from which some integral facets are extracted [GE92, GE93]. Those facets are taken into account in the final model. The preliminary model for instruction scheduling and functional unit allocation consists of three types of general constraints: the assignment constraints, the resource and the precedence constraints.

The operation assignment constraints ensure that the execution of each operation is started in exactly one control step and that it is assigned to exactly one functional unit.

$$\sum_{\substack{k \in N_R^F: \\ (j,k) \in E_R}} \sum_{n \in N(j)} x_{jn}^k = 1 \qquad \forall\, j \in N_D \tag{5.24}$$

Example 5.4 Assume that the target architecture contains one ALU and one multiplier $N_R^F = \{ALU, MUL\}$, both with a latency of 1 clock cycle. Let three operations be given $N_I = \{i, j, k\}$ where $N(i) = \{1, 2, 3\}$, $N(j) = \{2, 3, 4\}$, and $N(k) = \{1, 2, 3, 4\}$, $(i, ALU) \in E_R$, $(j, ALU) \in E_R$, and $(k, MUL) \in E_R$ such that each operation can be executed in one clock cycle. Then the following assignment constraints are generated:

$$\begin{aligned}
x_{i1}^{ALU} + x_{i2}^{ALU} + x_{i3}^{ALU} &= 1 \\
x_{j2}^{ALU} + x_{j3}^{ALU} + x_{j4}^{ALU} &= 1 \\
x_{k1}^{MUL} + x_{k2}^{MUL} + x_{k3}^{MUL} + x_{k4}^{MUL} &= 1
\end{aligned}$$

□

The resource constraints prevent more than R_k operations from being assigned to each functional unit type k at the same control step.

$$\sum_{\substack{i \in N_I: \\ (i,k) \in E_R}} \sum_{\substack{n_i = n \\ n_i \in N(i)}}^{n + L_i^k - 1} x_{in_i}^k \leq R_k \qquad \forall\, k \in N_R^F \;\wedge\; 0 \leq n \leq M_{steps} \tag{5.25}$$

Example 5.5 The resource constraints generated for the scenario of Example 5.4

read as follows:

$$
\begin{aligned}
x_{i1}^{ALU} &\leq 1 \\
x_{k1}^{MUL} &\leq 1 \\
x_{i2}^{ALU} + x_{j2}^{ALU} &\leq 1 \\
x_{k2}^{MUL} &\leq 1 \\
x_{i3}^{ALU} + x_{j3}^{ALU} &\leq 1 \\
x_{k3}^{MUL} &\leq 1 \\
x_{j4}^{ALU} &\leq 1 \\
x_{k4}^{MUL} &\leq 1
\end{aligned}
$$

□

The data dependences of the input program are modelled by the precedence constraints. In [GE92, GE93] precedence constraints have only been presented for true dependences; in the following also the constraints for output and anti dependences are given. Moreover the formulation has been extended to take into account interdependencies between the values of latency and execution time of an operation and its functional unit assignment.

- In case of a true dependence $(i, j, r, t) \in E_D^t$ the execution of operation j must not be started before the execution of i has been finished. For each dependence one constraint has to be generated; variations of the execution time of i in dependence of the functional unit used to execute i are considered.

$$
\sum_{\substack{k:(i,k)\in E_R: \\ Q_i^k \geq n_j - n_i + 1}} x_{in_i}^k + \sum_{k:(j,k)\in E_R} x_{jn_j}^k \leq 1 \qquad \forall\ (i,j,r,t) \in E_D^t \tag{5.26}
$$
$$
\forall \quad n_i \in N(i)\ \forall\ n_j \in N(j), n_j \leq n_i + \max_k Q_i^k - 1
$$

- In case of an output dependence $(i, j, r, o) \in E_D^o$ it has to be ensured that the execution of i is finished before the execution of j has been finished. If the execution times of the operations depend on the functional unit assignment, for each combination of execution time values a dedicated constraint has to be generated. Let $W_i = \{Q_i^k \mid (i,k) \in E_R\}$ for each operation i; then the constraints are defined as follows:

$$
\sum_{\substack{k:(i,k)\in E_R: \\ Q_i^k = w_i}} x_{in_i}^k + \sum_{\substack{k:(j,k)\in E_R: \\ Q_j^k = w_j}} x_{jn_j}^k \leq 1 \qquad \forall\ (i,j,r,o) \in E_D^o \tag{5.27}
$$
$$
\begin{aligned}
\forall \quad & n_i \in N(i)\ \forall\ n_j \in N(j), n_j \leq n_i + w_i - w_j, \\
\forall \quad & w_i \in W_i\ \forall\ w_j \in W_j
\end{aligned}
$$

- In case of an anti dependence $(i,j,r,a) \in E_D^a$ the execution of i must have been started before the execution of j has been finished. Again it is sufficient to generate one constraint per dependence; variations of the execution time of i are taken into account.

$$\sum_{k:(i,k)\in E_R} x_{in_i}^k \quad + \sum_{\substack{k:(j,k)\in E_R \\ n_j+Q_j^k \leq n_i}} x_{jn_j}^k \leq 1 \qquad \forall\ (i,j,r,a) \in E_D^a \tag{5.28}$$
$$\forall \quad n_i \in N(i)\ \forall\ n_j \in N(j),\ n_j \leq n_i - \min_k Q_j^k$$

Example 5.6 Consider again the scenario of example 5.4. Assume that each operation can be executed in one clock cycle. Let the following data dependences be given: $(i,j,t,r_1) \in E_D^t$, $(i,j,o,r_1) \in E_D^o$, and $(i,k,a,r_2) \in E_D^a$. A visualisation of the corresponding data dependence graph is shown in Fig. 5.6. Then the following precedence constraints are generated:

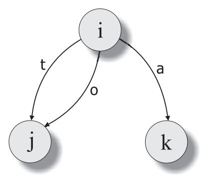


Figure 5.6.: Data dependence graph of example 5.6

$$\begin{aligned}
x_{i2}^{ALU} + x_{j2}^{ALU} &\leq 1 \\
x_{i3}^{ALU} + x_{j2}^{ALU} &\leq 1 \\
x_{i3}^{ALU} + x_{j3}^{ALU} &\leq 1 \\
\\
x_{i2}^{ALU} + x_{j2}^{ALU} &\leq 1 \\
x_{i3}^{ALU} + x_{j2}^{ALU} &\leq 1 \\
x_{i3}^{ALU} + x_{j3}^{ALU} &\leq 1 \\
\\
x_{i2}^{ALU} + x_{k1}^{MUL} &\leq 1 \\
x_{i3}^{ALU} + x_{k1}^{MUL} &\leq 1 \\
x_{i3}^{ALU} + x_{k2}^{MUL} &\leq 1
\end{aligned}$$

□

In [GE92, GE93] an improved modelling of the precedence constraints has been developed that takes into account integral facets derived from a transformation to the node packing problem. Let an undirected graph $G_N = (N_N, E_N)$ and a cost vector c be given. Then the weighted node packing problem is defined as follows:

$$
\begin{array}{lrcll}
max & \sum_u c_u x_u & & & \\
 & x_u + x_v & \leq & 1 & \forall\ (u,v) \in E_N \\
 & x_u & \in & \{0,1\} &
\end{array}
$$

From constraints (5.24),(5.26), (5.27), and (5.28) a system of linear inequalities of the form $Ax \leq b$ can be formed, where A is a matrix all whose entries are 0 or 1, and b is a unit vector. From this system a node packing graph $G_N = (N_N, E_N)$ is generated. Each variable x_{in}^k is mapped to a node $u \in N_N$. The edges E_N of the graph are defined as follows: for each row of A (each constraint), the columns (variables) with coefficient 1 define a complete subgraph (clique) in G. The resulting graph represents the node packing problem. An *odd cycle* C is a cycle consisting of an odd number of nodes. It is called *chordless* if no two nodes of C share an edge that does not belong to the cycle. Let $\mathcal{K}$ be the set of all cliques of G and $\mathcal{C}$ be the set of all chordless odd cycles in G. Then the following inequalities represent facets of the node packing polytope:

$$
\sum_{u \in K} x_u \leq 1 \qquad \forall\ K \in \mathcal{K} \tag{5.30}
$$

$$
\sum_{u \in C} x_u \leq \frac{1}{2}(|C| - 1) \quad \forall\ C \in \mathcal{C} \text{ where } |C| \geq 5 \tag{5.31}
$$

The precedence constraints (5.26), (5.27), and (5.28) are replaced by the constraints (5.32), (5.33), and (5.34) generating these clique facets. Details about that transformation can be found in [GE92]. If the execution times of the operations depend on the functional unit assignment, for each combination of execution time values a dedicated constraint has to be generated. For each operation i define the sets $W_i = \{Q_i^k \mid (i,k) \in E_R\}$ and $R_i(w_i) = \{k \in N_R^F \mid (i,k) \in E_R \ \wedge\ Q_i^k = w_i\}$; then the constraints are defined as follows:

$$\sum_{k\in R_j(w_j)} \sum_{\substack{n_j \le n \\ n_j\in N(j)}} x^k_{jn_j} + \sum_{k\in R_i(w_i)} \sum_{\substack{n_i \ge n-w_i+1 \\ n_i\in N(i)}} x^k_{in_i} \le 1 \tag{5.32}$$

$$\forall\ (i,j,r,t) \in E^t_D, w_i \in W_i, w_j \in W_j$$

$$n \in N^t(i,j) = \{n' + w_i - 1 \mid n' \in N(i)\} \cap N(j)$$

$$\sum_{k\in R_j(w_j)} \sum_{n_j \le n} x^k_{jn_j} + \sum_{k\in R_i(w_i)} \sum_{\substack{n_i \ge n-w_i+w_j \\ n_i\in N(i)}} x^k_{in_i} \le 1 \tag{5.33}$$

$$\forall\ (i,j,r,o) \in E^o_D, w_i \in W_i, w_j \in W_j,$$

$$n \in N^o(i,j) = \{n' + w_i - w_j \mid n' \in N(i), (i,k) \in E_R\} \cap N(j)$$

$$\sum_{k\in R_j(w_j)} \sum_{\substack{n_j \le n-w_j+1 \\ n_j\in N(j)}} x^k_{jn_j} + \sum_{k\in R_i(w_i)} \sum_{\substack{n_i > n \\ n_i\in N(i)}} x^k_{in_i} \le 1 \tag{5.34}$$

$$\forall\ (i,j,r,a) \in E^a_D, w_i \in W_i, w_j \in W_j,$$

$$n \in N^a(i,j) = \{n' + w_j - 1 \mid n' \in N(j)\} \cap N(i)$$

Since in [GE92, GE93] only the formulation of the constraints for the true dependences is presented and no proof of correctness is given, in the following a correctness proof for all types of data dependence constraints is given.

Proof: It has to be shown that the inequalities (5.32–5.34) prevent violations of the data dependences and that they do not exclude any feasible ordering.

1. True dependences

   a) Assume that a true dependence (i,j,r,t) is violated. Then there must be an assignment of operations to control steps where $x^{k_i}_{im_i} = 1 \wedge x^{k_j}_{jm_j} = 1$ and $m_j < m_i + w_i$. We will show that in this case the precedence constraints (5.32) must have been generated for $x^{k_i}_{im_i}$ and $x^{k_j}_{jm_j}$ which is a contradiction to the assumption.

      Define $n := m_i + w_i - 1$. We know that $m_j \in N(j) \le m_i + w_i - 1 = n \le alap(i) + w_i - 1 \le alap(j) \in N(j)$ such that $n \in N^t(i,j)$.

      But then an inequality of the type (5.32) is generated where both $x^{k_i}_{im_i}$ and $x^{k_j}_{jm_j}$ appear on the left hand side.

      $$\sum_k \sum_{n_j \le m_i+w_i-1} x^k_{jn_j} + \sum_k \sum_{n_i \ge m_i+w_i-1-w_i+1} x^k_{in_i} \le 1$$

      This is a contradiction to $x^{k_i}_{im_i} = x^{k_j}_{jm_j} = 1$.

b) We have to show that no feasible ordering is excluded. For all pairs of variables $(x^{k_i}_{in_i}, x^{k_j}_{jn_j})$ that appear on the left hand side of an inequality of type (5.32) the following condition holds:

$$\begin{aligned} & n_i \geq n - w_i + 1 \geq n_j - w_i + 1 \\ \Leftrightarrow \quad & n_i + w_i \geq n_j + 1 \\ \Leftrightarrow \quad & n_i + w_i > n_j \end{aligned}$$

Since the constraints are only generated if there is a true dependence from i to j ($(i, j, r, t) \in E^t_D$) only assignments of i and j to control steps that violate this dependence are excluded.

2. Output dependences

a) Assume that an output dependence (i, j, r, o) is violated. Then there must be an assignment of operations to control steps where $x^{k_i}_{im_i} = 1 \wedge x^{k_j}_{jm_j} = 1$ and $m_j < m_i + w_i - w_j + 1$.
Define $n := m_i + w_i - w_j$. We know that $m_j \in N(j) \leq m_i + w_i - w_j = n \leq alap(i) + w_i - w_j \leq alap(j) \in N(j)$ such that $n \in N^o(i, j)$.
But then an inequality of the type (5.33) must have been generated where both $x^{k_i}_{im_i}$ and $x^{k_j}_{jm_j}$ appear on the left hand side.

$$\sum_k \sum_{n_j \leq m_i + w_i - w_j} x^k_{jn_j} + \sum_k \sum_{n_i \geq m_i + w_i - w_j - w_i + w_j} x^k_{in_i} \quad \leq \quad 1$$

This is a contradiction to $x^{k_i}_{im_i} = x^{k_j}_{jm_j} = 1$.

b) We have to show that no feasible ordering is excluded. For all pairs of variables $(x^{k_i}_{in_i}, x^{k_j}_{jn_j})$ that appear on the left hand side of an inequality of type (5.33) the following condition holds:

$$\begin{aligned} & n_i \geq n - w_i + w_j \geq n_j - w_i + w_j > n_j - w_i + w_j - 1 \\ \Leftrightarrow \quad & n_j < n_i + w_i - w_j + 1 \end{aligned}$$

Since the constraints are only generated if there is an output dependence from i to j ($(i, j, r, o) \in E^o_D$) only assignments of i and j to control steps that violate this dependence are excluded.

3. Anti dependences

a) Assume that an anti dependence (i, j, r, a) is violated. Then there must be an assignment of operations to control steps where $x^{k_i}_{im_i} = 1 \wedge x^{k_j}_{jm_j} = 1$ and $m_i > m_j + w_j - 1$.

Choose $n := m_j + w_j - 1$. We know that $m_i \in N(i) > m_j + w_j - 1 = n \geq asap(j) + w_j - 1 \geq asap(i) \in N(i)$ such that $n \in N^a(i, j)$.

But then an inequality of the type (5.34) must have been generated where both $x^{k_i}_{im_i}$ and $x^{k_j}_{jm_j}$ appear on the left hand side.

$$\sum_k \sum_{n_j \leq m_j + w_j - 1 - w_j + 1} x^k_{jn_j} + \sum_k \sum_{n_i > m_j + w_j - 1} x^k_{in_i} \quad \leq \quad 1$$

This is a contradiction to $x^{k_i}_{im_i} = x^{k_j}_{jm_j} = 1$.

b) We have to show that no feasible ordering is excluded. For all pairs of variables $(x^{k_i}_{in_i}, x^{k_j}_{jn_j})$ that appear on the left hand side of an inequality of type (5.34) the following condition holds:

$$n_i > n \geq n_j + w_j - 1$$

Since the constraints are only generated if there is an anti dependence from i to j ($(i, j, r, a) \in E^a_D$) only assignments of i and j to control steps that violate this dependence are excluded.

□

The constraints presented so far allow the modelling of hardware architectures where the pipeline behaviour can be completely described by the two parameters execution time and latency. Each operation is assigned to exactly one functional unit. If additional restrictions for the usage of the system buses have to be taken into account, the modelling is extended. Assume again that the number of results that can be written to the system bus per clock cycle is restricted. This means that the completion time of the operations has to be synchronised with respect to the bus. It has to be ensured that if the execution of an operation i is started in control step c, an access to the result bus is registered in cycle $c + w_i$. Let B denote the result bus, then the following constraint has to be generated:

$$x^B_{j,n+Q^k_j} \geq x^k_{jn} \qquad \forall\ j \in N_I,\ \forall\ k : (j, k) \in E_R, \forall\ n \in N(j) \tag{5.35}$$

Additionally, resource and operation assignment constraints have to be generated for the result buses. For the assignment constraints, all control steps where the completion of the operation can take place have to be considered.

Similarly to the SILP formulation, for each operation i the variable t_i denotes the starting time for the execution of i. However the value of the t_i variables is computed directly from the main decision variables x_{ij}^k by the following equation:

$$t_j = \sum_{k:(j,k)\in E_R} \sum_{n\in N(j)} n \cdot x_{jn}^k$$

Then the integer linear program for instruction scheduling and functional unit allocation in the OASIC-based formulation can be summarised as follows:

- Objective function

$$\min \ M_{steps} \tag{5.36}$$

- Constraints

1. Time Constraints

   The maximal number of control steps M_{steps}, i. e. the value of the objective function is defined as the starting time of the last operation to be executed.

$$t_j \leq M_{steps} \qquad \forall\, j \in N_I \tag{5.37}$$

2. Precedence Constraints

   The data dependences of the input program have to be respected. Depending on the type of data dependences there are three types of constraints:

$$\begin{array}{rcl}
\displaystyle\sum_{k\in R_j(w_j)} \sum_{\substack{n_j\leq n\\ n_j\in N(j)}} x_{jn_j}^k + \sum_{k\in R_i(w_i)} \sum_{\substack{n_i\geq n-w_i+1\\ n_i\in N(i)}} x_{in_i}^k & \leq & 1 \\
\forall\ (i,j,r,t) & \in & E_D^t, w_i \in W_i, w_j \in W_j \\
n \in N^t(i,j) = \{n' + w_i - 1 & | & n' \in N(i)\} \ \cap \ N(j)
\end{array}$$

$$\begin{array}{rcl}
\displaystyle\sum_{k\in R_j(w_j)} \sum_{n_j\leq n} x_{jn_j}^k + \sum_{k\in R_i(w_i)} \sum_{\substack{n_i\geq n-w_i+w_j\\ n_i\in N(i)}} x_{in_i}^k & \leq & 1 \\
\forall\ (i,j,r,o) & \in & E_D^o, w_i \in W_i, w_j \in W_j, \\
n \in N^o(i,j) = \{n' + w_i - w_j \mid n' \in N(i) & , & (i,k) \in E_R\} \ \cap \ N(j) \\
\displaystyle\sum_{k\in R_j(w_j)} \sum_{\substack{n_j\leq n-w_j+1\\ n_j\in N(j)}} x_{jn_j}^k + \sum_{k\in R_i(w_i)} \sum_{\substack{n_i> n\\ n_i\in N(i)}} x_{in_i}^k & \leq & 1 \\
\forall\ (i,j,r,a) & \in & E_D^a, w_i \in W_i, w_j \in W_j, \\
n \in N^a(i,j) = \{n' + w_j - 1 & | & n' \in N(j)\} \cap \ N(i)
\end{array}$$

3. Execution Constraints

   The execution of each operation must start in exactly one control step and is performed by exactly one resource instance.

$$\sum_{\substack{k\in N_R^F:\\ (j,k)\in E_R}} \sum_{n\in N(j)} x_{jn}^k \;=\; 1 \qquad \forall\, j \in N_I \tag{5.38}$$

4. Resource Constraints

   In no control step there must be more than R_k operations being executed by resource type k, i. e. the number of available instances of each functional unit type must not be exceeded.

$$\sum_{\substack{i\in N_I:\\ (i,k)\in E_R}} \sum_{\substack{n_i=n\\ n_i\in N(i)}}^{n+L_i^k-1} x_{in_i}^k \leq R_k \qquad \forall\, k \in N_R^F \;\wedge\; 0 \leq n \leq M_{steps} \tag{5.39}$$

5.3.2. Integrating Register Assignment

If the register assignment problem is incorporated in the ILP formulation, it has to be ensured that in no control step more than R_r registers are used for each abstract register file r of capacity R_r. Then at most R_r variable life ranges stored in this register file are overlapping. If the ordering of the operations is fixed, the life range of a variable can be represented by a lifetime-defining edge (i, j) between the operation i defining the value and the operation j that is the last to use that value. When addressing instruction scheduling and register assignment simultaneously, the ordering of the operations is not fixed such that, in general, a lifetime-defining edge cannot be statically determined. A naive approach is to assume a lifetime-defining edge between a definition and all uses. The number of edges can be reduced by transitivity analysis and by *asap*/*alap* analysis [GE92, Käs97].

Definition 5.5 *Let U be an upper bound for the execution time of the input program. An edge $i \prec j$ crosses control step n, if $N(i) \cap \{0, 1, \ldots, n-(w_i-1)\} \neq \emptyset$ and $N(j) \cap \{n+1, n+2, \ldots, U\} \neq \emptyset$.*

The number of edges with head i that cross control step n is denoted $e_n(i)$; formally $e_n(i) \;=\; |\{i \prec j_k \mid k = 1, \ldots, l;\; (i, j_k) \text{ crosses control step } n\}|$. $M(n)$ denotes the set of all maximal sets $M'(n)$ of edges crossing control step n with pairwise different heads.

In order to ensure that the number of available registers is not exceeded, the register crossing constraints (5.40) are introduced. The number of constraints to be generated for control step n can be given as $\prod_i e_n(i)$.

$$
\begin{aligned}
\sum_{j_a \prec j_b \in M'(n)} \Big(\sum_{k \in N_A} \sum_{\substack{n_1 \leq n \\ n_1 \in N(j_a)}} x^k_{j_a n_1} + \sum_{k \in N_A} \sum_{\substack{n_2 > n \\ n_2 \in N(j_b)}} x^k_{j_b n_2} & - \\
\sum_{k \in N_A} \sum_{\substack{n_3 \leq n \\ n_3 \in N(j_b)}} x^k_{j_b n_3} - \sum_{k \in N_A} \sum_{\substack{n_4 > n \\ n_4 \in N(j_a)}} x^k_{j_a n_4} \Big) & \leq 2 \cdot R \\
\forall\, n \,\wedge\, \forall\, M'(n) \in M(n) &
\end{aligned}
\tag{5.40}
$$

Example 5.7 Let two operations i and j be given with $w_i = w_j = 1$ and $N(i) = \{1, \ldots, 4\}$, $N(j) = \{2, \ldots, 5\}$. Then for control step $n = 2$ and $M'(2) = \{(i, j)\}$ the following constraint is generated:

$$
\begin{aligned}
& \sum_{1 \leq n_1 \leq 2} x^k_{in_1} + \sum_{3 \leq n_2 \leq 5} x^k_{jn_2} - \sum_{2 \leq n_3 \leq 2} x^k_{jn_3} - \sum_{3 \leq n_4 \leq 4} x^k_{in_4} && \leq 2R \\
\Leftrightarrow \quad & x^k_{i1} + x^k_{i2} - x^k_{i3} - x^k_{i4} - x^k_{j2} + x^k_{j3} + x^k_{j4} + x^k_{j5} && \leq 2R
\end{aligned}
$$

Graphically visualised the following ordering results:

	+	−
1	x^k_{i1}	
2	x^k_{i2}	x^k_{j2}
3	x^k_{i3}	x^k_{j3}
4	x^k_{i4}	x^k_{j4}
5		x^k_{j5}
	−	+

For exactly one $r \in \{1, \ldots, 4\}$ the variable x_{ir^k} must take the value 1, and for exactly one $s \in \{2, \ldots, 5\}$, $x^k_{js} = 1$ must hold. If there is a crossing edge, the left hand side of inequality (5.40) evaluates to 2, otherwise it evaluates to 0. It is excluded that the left hand side takes a negative value since otherwise the precedence constraints would be violated.

□

5.3.3. The Structure of the OASIC Polytope

The polytope of the OASIC formulation is defined by three types of constraints: the precedence constraints, the execution and the resource constraints. In [GE92, GE93] it has been shown that the polytope of the precedence constraints is not integral. The formulation of the constraints (5.32) – (5.34) however takes into account integral facets derived from a transformation to the node packing polytope. Thus while the polytope cannot be guaranteed to be integral it represents a good approximation to the integral polytope [GE92, GE93]. In [CWM93, CWM94] the integrality of some subpolytopes of the OASIC polytope has been shown when the task of functional unit binding is not addressed. In the following we will prove for the problem formulation addressing the functional unit binding that the polytopes of the execution constraints, of the resource constraints and the combined polytope of the execution and the resource constraints are integral. Thus any deviations of the feasible region from the integral polytope are due to the precedence constraints.

Lemma 5.6 *The polytope of the execution constraints is integral.*

Proof: The execution constraints have been defined as follows:

$$\sum_{\substack{k \in N_R^F: \\ (j,k) \in E_R}} \sum_{n \in N(j)} x_{jn}^k \quad = \quad 1 \qquad \forall\, j \in N_I$$

Let $E \in \{0,1\}^{m_1 \times n_1}$ be the coefficient matrix of the execution constraints. Only in the constraint generated for operation j the coefficient of a variable x_{in}^k is equal to 1; in all other constraints it is equal to zero. Thus each column contains exactly one nonzero entry. Then it follows from Theorem 4.4 that E is totally unimodular. Hence the polytope of the execution constraints is integral (cf. Theorem 4.2). $\square$

Lemma 5.7 *The polytope of the resource constraints is integral.*

Proof: The resource constraints have been defined as follows:

$$\sum_{\substack{i \in N_I: \\ (i,k) \in E_R}} \sum_{\substack{n_i = n \\ n_i \in N(i)}}^{n+L_i^k-1} x_{in_i}^k \leq R_k \qquad \forall\, k \in N_R^F \ \wedge\ 0 \leq n \leq M_{steps}$$

Let $R \in \{0,1\}^{m_2 \times n_2}$ be the coefficient matrix of the resource constraints. Assume that the constraints have been generated by iterating through the available resource types in an outer loop and traversing all feasible control steps in the inner loop. Then it follows from the definition of the constraints that if the coefficient of a variable x_{in}^k in a row α is equal to 1, its coefficient in row $\alpha + 1$ is either 1, or 0. If it is equal to zero, it will be equal to zero in all rows $\beta > \alpha$. Thus R is an interval matrix and it follows from Theorem 4.7 that R is totally unimodular. Since the property of total unimodularity is maintained when interchanging rows and columns (see Theorem 4.5) R remains totally unimodular even if the constraints are generated in another order. Hence the polytope of the resource constraints is integral. $\square$

Lemma 5.8 *The combined polytope of the execution and the resource constraints is integral.*

Proof: Consider the matrix $M = \begin{pmatrix} R \\ E \end{pmatrix} \in \{0,1\}^{m_3 \times n_3}$ where $m_3 = m_1 + m_2$ i. e. where the rows of the resource constraints precede the rows of the execution constraints. Assume, e. g., that operations 1 and 2 can be executed on two resource types k_1 and k_2 then the matrix M has the following shape:

$$\begin{bmatrix}
1\cdots 1 & 0\cdots 0 & 0\cdots 0 & 1\cdots 1 & 0\cdots 0 & 0\cdots 0 \\
0\cdots 0 & 1\cdots 1 & 0\cdots 0 & 0\cdots 0 & 1\cdots 1 & 0\cdots 0 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
A_1^{k_1} & 0\cdots 0 & 0\cdots 0 & 0\cdots 0 & 0\cdots 0 & 0\cdots 0 \\
0\cdots 0 & 0\cdots 0 & 0\cdots 0 & A_1^{k_2} & 0\cdots 0 & 0\cdots 0 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0\cdots 0 & A_2^{k_1} & 0\cdots 0 & 0\cdots 0 & 0\cdots 0 & 0\cdots 0 \\
0\cdots 0 & 0\cdots 0 & 0\cdots 0 & 0\cdots 0 & A_2^{k_2} & 0\cdots 0 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots
\end{bmatrix}$$

We will show that the matrix M is totally unimodular with the help of Theorem 4.6 applied to row partitionings. It is obvious that it is sufficient to show that feasible partitionings can be found for the matrices $B_i^k = \begin{bmatrix} 1\cdots 1 \\ A_i^k \end{bmatrix}$ since a partitioning of M can be constructed directly from those partial partitionings. Each of the matrices B_i^k has the following shape (only the nonzero entries are shown):

$$B_i^k = \begin{bmatrix}
1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\
1 & \cdot & \cdot & \cdot & 1 & & & & \\
 & 1 & \cdot & \cdot & \cdot & 1 & & & \\
 & & \ddots & & & & \ddots & & \\
 & & & 1 & \cdot & \cdot & \cdot & \cdot & 1 \\
 & & & & 1 & \cdot & \cdot & \cdot & 1 \\
 & & & & & \ddots & & & \vdots \\
 & & & & & & & & 1
\end{bmatrix}$$

As an example assume that there is an operation i where $|N(i)| = 5$ that is executed on a functional unit type k with latency 2. Then the matrix B_i^k is defined as follows:

$$B_i^k = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix}$$

It is obvious that the transpose of B_i^k is an interval matrix and thus is totally unimodular. From Theorem 4.5 follows directly that B_i^k is totally unimodular. Thus for each B_i^k and every $Q \subseteq \mathcal{M} = \{1, \ldots, m_i^k + 1\}$ there exists a partition Q_1, Q_2 of Q such that

$$\left| \sum_{p \in Q_1} b_{po} - \sum_{l \in Q_2} b_{lo} \right| \leq 1 \text{ for } o = 1, \ldots, n_i^k. \tag{5.41}$$

This concludes the proof. □

5.3.4. Complexity

Similarly to Sec. 5.2.5 we will use the number of integer variables and the number of constraints as a measure of complexity. First we will neglect the register assignment problem and only present the bounds for the modelling of instruction scheduling and resource allocation.

For each operation $i \in N_I$ one variable x_{in}^k is generated per control step $n \in N(i)$ and per functional unit k the operation can be assigned to. With $u_x := \max\{alap(j) - asap(j) \mid j \in N_I\}$ the number of the integer variables can be bounded by $u_x \cdot |N_I| \cdot |N_R^F| = \mathcal{O}(|N_R^F| \cdot |N_I|^2)$.

Now let us consider the required number of constraints. It is necessary to generate $|N_I|$ time constraints (5.37) and $|N_I|$ execution constraints (5.24); the number of resource constraints (5.25) can be given as $|N_R^F| \cdot |N_I| \cdot \max_{i \in N_I} w_i$. The number of precedence constraints (5.32 - 5.34) is bounded by $\mathcal{O}(u_r \cdot |E_D|)$ with $u_r = \max\{alap(j) \mid j \in N_D\} - \min\{asap(i) \mid i \in N_D\} + 1$. Since $\mathcal{O}(u_r \cdot |E_D|) \subseteq \mathcal{O}(|N_D| \cdot |E_D|) \subseteq \mathcal{O}(|N_D|^3)$ the number of precedence constraints is bounded by $\mathcal{O}(|N_D|^3)$ which also represents an upper bound on the total number of constraints.

If the register assignment problem is incorporated into the ILP formulation, the complexity increases significantly. For each control step n there are $\Pi_i e_n(i)$ crossing constraints. If there are two uses for each definition such that the corresponding lifetime-defining edges cross control step n, the number of crossing constraints for n evaluates to $\Pi_i 2 = 2^i$. Thus the number of register crossing constraints can grow exponentially in the number of operations; this can lead to an exponential space consumption. Since the experimental analysis of [Käs97] indicate that the complexity is too high for practical use, the incorporation of the register assignment in the OASIC formulation has not been implemented in the PROPAN-framework and will not be considered during the remainder of this thesis.

5.4. Control Flow Modelling

In order to preserve the semantics of input procedures containing conditional code or loops, the modelling of the data dependences is not sufficient. For example an operation from the *then* block of a conditional statement might be moved into the *else* part without any data dependences being violated. This transformation, in general, is not semantic preserving and must be prevented. A common approach is to insert virtual nodes at the beginning and the end of each basic block and prevent reordering across basic block boundaries by adding data dependences to those nodes [Ell86, GE92]. In the remainder of this section, an ILP-based mechanism is presented that allows to perform instruction scheduling across basic block boundaries [Käs97, KL98]. The set of basic blocks in the input program is assumed to be fixed. From the control dependence graph for each operation the set of all basic blocks it can be assigned to is determined. The corresponding disjunctions are inserted into the integer linear programs with the help of additional binary variables. In the following the general mechanism for modelling disjunctive constraints is presented; then the application of this modelling for representing the control flow structure is explained.

5.4.1. Modelling Disjunctive Constraints

Disjunctive constraints are used to model decision problems where feasible solutions must be part of at least l of m alternative sets. Let $X_1, \ldots, X_m \subset \mathbb{R}^n_+$ denote the alternative sets where each set is characterised as follows:

$$X_j = \{x \in \mathbb{R}^n \mid A^j x \leq b^j\}$$

Furthermore let an upper bound U be given such that

$$A^j x \leq b^j + U \quad \forall\, j = 1, \ldots, m$$

If a constraint $a_i^j x \leq b_i^j$ is replaced by $a_i^j x \leq b_i^j + U_i$, it becomes irrelevant.

In order to force a variable x to be contained in at least l of the sets X_j, dedicated binary variables $y_1, \ldots, y_m$ are introduced where $y_j = 0 \Leftrightarrow\ x \in X_j$ and $y_j = 1 \Leftrightarrow\ x \notin X_j$. The binary variables y_j are incorporated into the constraints such that the constraints of each X_j must be satisfied if $y_j = 0$ and are redundant if $y_j = 1$. This is achieved by the following system of linear inequalities:

$$A^j x \leq b^j + U y_j \quad \forall\, j = 1, \ldots, m \tag{5.42}$$

$$\sum_{j=1}^{m} y_j \leq m - l \tag{5.43}$$

$$y_j \in \{0, 1\} \quad \forall\, j = 1, \ldots, m \tag{5.44}$$

This modelling is correct since $y_i = 0$ yields the constraints $A^j x \leq b^j$ while $y_j = 1$ yields the trivial constraint $A^j x \leq b^j + U$ [NW88, DK96]. If each solution has to be contained in exactly l of the m alternatives, inequality (5.43) has to be replaced by

$$\sum_{j=1}^{m} y_j = m - l$$

5.4.2. Representing the Control Flow Structure

In order to represent the control flow structure of a program by a set of disjunctive constraints each basic block b_k is associated with a starting time t_k^A and a completion time t_k^E. An operation j is contained in a basic block b_k if and only if $t_k^A \leq t_j \leq t_k^E$. The set X_k of all operations that may be contained in basic block b_k is defined as follows:

$$X_k = \{j \in N_I \mid t_k^A - t_j \leq 0 \ \wedge \ t_j - t_k^E \leq 0\}$$

The fact that an operation j can be assigned to exactly one of m possible basic blocks $b_1, \ldots, b_m$ can be represented by a system of linear inequalities using dedicated binary variables $y_j^1, \ldots, y_j^m$ where $y_j^k = 0$ if $j \in b_k$ and $y_j^k = 1$, if $j \notin b_k$.

Let U denote an upper bound of the execution time of the input program; then in order to ensure that each operation j is contained in exactly one of the sets $X_1, \ldots, X_m$ the following constraints are generated:

$$t_k^A - t_j - U y_j^k \leq 0 \quad \forall\, k = 1, \ldots, m \tag{5.45}$$
$$t_j - t_k^E - U y_j^k \leq 0 \quad \forall\, k = 1, \ldots, m \tag{5.46}$$
$$\sum_{k=1}^{m} y_j^k = m - 1 \tag{5.47}$$
$$y_j^k \in \{0, 1\} \quad \forall\, k = 1, \ldots, m \tag{5.48}$$

If $m = 2$, only one binary variable is required. Then the following constraints are generated:

$$t_1^A - t_j - U y_j \leq 0 \tag{5.49}$$
$$t_j - t_1^E - U y_j \leq 0 \tag{5.50}$$
$$t_1^A - t_j - U(1 - y_j) \leq 0 \tag{5.51}$$
$$t_j - t_1^E - U(1 - y_j) \leq 0 \tag{5.52}$$
$$y_j \in \{0, 1\} \tag{5.53}$$

Example 5.8 Assume operation j can be contained in the three basic blocks b_1, b_2, and b_3. Then the following constraints are generated:

$$\begin{array}{ccccccc}
t_1^A & - & t_j & - & U y_j^1 & \leq & 0 \\
t_j & - & t_1^E & - & U y_j^1 & \leq & 0 \\
t_2^A & - & t_j & - & U y_j^2 & \leq & 0 \\
t_j & - & t_2^E & - & U y_j^2 & \leq & 0 \\
t_3^A & - & t_j & - & U y_j^3 & \leq & 0 \\
t_j & - & t_3^E & - & U y_j^3 & \leq & 0 \\
y_j^1 & + & y_j^2 & + & y_j^3 & = & 2
\end{array}$$

□

The set of basic blocks where an operation j can be located is determined from the control dependence graph. Code movements are only allowed between basic blocks that are control equivalent. In order for the assignment of operations to basic blocks to be well defined all control flow operations like branches, or loop operations must remain in their original basic block. If a basic block b begins with a loop operation i_1 and ends with a branch i_2, assigning another operation j to basic block b means scheduling j between i_1 and i_2. The basic blocks themselves must not be reordered, i. e. the basic block order of the input program is kept in the output program. This way, the ordering of the control flow operations is fixed; all remaining operations can be moved between control equivalent basic blocks. The ordering of the basic blocks is included in the integer linear programs by generating the following inequalities for each pair of basic blocks b_i, b_j where $(b_i, b_j) \in E_{cf}^{+}$.

$$t_i^A - t_i^E \leq 0 \tag{5.54}$$
$$t_j^A - t_j^E \leq 0 \tag{5.55}$$
$$t_i^E < t_j^A \tag{5.56}$$

Additional constraints are required to take into account the completion time of the basic blocks when calculating the value of the objective function:

$$t_b^E - M_{steps} \leq 0 \qquad \forall\ b \in N_B \tag{5.57}$$

The extensions to allow restricted code movements between basic blocks that are not control equivalent, e. g. in loop-invariant code motion, assignment sinking or assignment hoisting [Rüt98] are straightforward. These movements however lead to the insertion of compensation code [WM95] and have not been incorporated in the current implementation.

With the ILP formulations presented above it is not possible to incorporate disjoint control flow paths in the same integer linear program. Consider, e. g., the *then* and the *else* block of a conditional statement. Then there can be program executions where some of the operations of the *then* block have the same starting time as some of the operations of the *else* block in another program execution. In the ILP formulations for the code generation problem however assigning the same starting time to two operation means that they are part of the same machine instruction and are executed in parallel—which, of course, is not true for the operations of the disjoint control flow paths.

The representation however is powerful enough to allow (possibly nested) loops to be represented by a homogeneous integer linear program; details about this are presented in Chap. 7.

5.5. ILP Models and Hardware Architectures

The complexity of the ILP models for the code optimisation problem is influenced by the design of the target architecture. The effects of architectural characteristics on the generated integer linear programs however differ between the time-indexed OASIC model and the order-indexed SILP formulation. In the OASIC formulation the incorporation of the register assignment problem cannot be considered promising since the required number of constraints grows exponentially in the number of input operations. In the SILP formulation the number of additional constraints and binary variables is in the order of $\mathcal{O}(|N_I|^2)$. Thus if an integration of instruction scheduling and register assignment is required, the SILP formulation outperforms the OASIC model.

In the following we will investigate the performance of both models for the problems of instruction scheduling and resource allocation. If the execution times of the machine operations depend on the functional unit binding, the efficiency of both models decreases. Compared to the case where the execution time of the operations is independent of the functional unit assignment, a higher number of precedence constraints is required in the OASIC formulation (cf. page 64). This leads to a performance degradation since the polytope of the precedence constraints is not integral. In the SILP formulation the polytope of the precedence constraints is integral if the execution time of each operation is uniquely defined, but it is not integral if the execution time varies depending on the functional unit assignment (cf. page 52) [Zha96]. With increasing execution times of the machine operations the number of constraints in the OASIC formulation increases whereas this does not influence the number of constraints of the SILP formulation.

The number of resource flow variables in the SILP formulation is bounded by $\mathcal{O}(|N_R^F| \cdot |N_I|^2)$, the number of decision variables in the OASIC formulation by $\mathcal{O}(u_x \cdot |N_R^F| \cdot |N_I|)$ where $u_x = \max_{i \in N_I}\{alap(i) - asap(i)\}$. Thus in both formulations the number of binary variables in the worst case depends linearly on the number of resource types. Since the number of decision variables however is only influenced by the number of alternative resource types each operation can be mapped to, it is in fact independent from the number of resource types if the assignment of operations to resources is uniquely defined. In a postpass framework it is a reasonable approach to derive an upper bound for the execution time from the schedule of the input program. For architectures with a high degree of instruction-level parallelism this can lead to small intervals $N(i)$ for the starting time of each operation i. In consequence the value u_x can be significantly smaller than its worst-case value, $u_x = \mathcal{O}(|N_I|)$, such that in the best case the number of decision variables of the OASIC model only grows slightly more than linearly with the number of operations. The number of flow variables in the SILP formulation however cannot be reduced this way; there is a binary variable x_{ij}^k for each pair (i, j) of operations such that both can be executed by resource type k and there is no data dependence from j to i. Thus for architectures with a high degree of instruction-level parallelism the number of variables in the SILP formulation grows nearly proportionally to

$|N_I|^2$ and in the OASIC model almost proportionally to $|N_I|$. If there is a large number of different resource types each operation can be mapped to, this can lead to a large difference in the variable number since the number of resource types is a factor in the term for the number of decision variables $(\mathcal{O}(|N_R^F| \cdot |N_I|^2))$. As the optimisers generated for the TriMedia TM1000 show, this difference can be significant (see Chap. 10).

Irregular restrictions of instruction-level parallelism and interdependencies between instruction scheduling and resource usage also increase the complexity of the code generation problem. Such restrictions are modelled by integer linear constraints generated automatically from logical conditions specified in the machine description (see Sec. 8.4). Often the same constraints are generated for the SILP and the OASIC model such that a similar performance degradation can be expected in both formulations. Explicitly preventing two operations from being executed in parallel however can be modelled more efficiently in the OASIC model than in the SILP model. In the SILP formulation it is necessary to generate disjunctive constraints which leads to the insertion of additional binary variables. In the OASIC formulation no disjunctive constraints and no additional binary variables are required (see Sec. 8.4).

To summarise we can conclude that the SILP model allows an efficient integration of the register assignment problem and is well suited for irregular architectures where the resource competition is high. For architectures with a high degree of instruction-level parallelism and a large number of alternative resource types each operation can be mapped to, the time-based OASIC formulation is better suited.

6. ILP-Based Approximation Techniques

Since register assignment and instruction scheduling are NP-complete problems even when addressed in isolation there is an upper bound on the size of the code sequences for which a provably optimal solution can be computed in practicable time. This bound depends on several factors. One important factor is the problem dimension, i. e. the number of tasks among instruction scheduling, functional unit binding and register assignment that are addressed simultaneously. Other important factors are the architectural restrictions of the target machine and the characteristics of the input programs, e. g. the number of precedence constraints or overlapping life ranges.

In order to be able to address code sequences exceeding this bound we have developed approximative methods that cannot guarantee an optimal solution but lead to a significant reduction of computation time while still obtaining a high solution quality. The basic idea of the approximations is to iteratively solve partial relaxations of the original problem. After the solution of one partial relaxation has been computed certain parts of the solution that are guaranteed to be integral are extracted. The corresponding variables are fixed to their current value by additional equality constraints. In subsequent iterations they are treated as constants and do not contribute to the problem complexity any more. Then the next partial relaxation is addressed. In the end an integral solution is obtained. There is no analytically determined performance guarantee that holds for all possible problem instances. Our experimental results however show that in most cases optimal solutions can be obtained.

The approximation techniques proposed in this thesis require solving integer linear programs that are partial relaxations of the original problem in each iteration step. While there is no polynomial time bound for the approximations they allow a significant reduction of computation time in comparison with the original problem. The obtained code quality is very high. The approximations can take advantage of sophisticated solution techniques developed for integer linear programming. Moreover they can be applied to the integrated code generation problems even in the presence of additional constraints modelling irregular architectural restrictions.

A naive approximation method from this class is the *approximation by rounding* [Zha96, Käs97, KL98] that can be applied to both the time-indexed OASIC model and the order-indexed SILP formulation. First, a partial relaxation of the problem is solved where the x-variables are allowed to take non-integral values. From the solution of the relaxation a relaxed variable with a non-integer value and a minimal distance to the nearest integer is selected. A new constraint is added to the mixed integer linear program (MILP) where the variable is fixed to that integer and the resulting MILP is solved. This is repeated until an integral solution is obtained. The basic idea of this approximation is that a close distance from an integer might be interpreted as a hint that this integer will be the optimal value of the variable. However this is a blind guess since no knowledge about the target architecture and about the semantics of the decision variables is exploited. In consequence it cannot be excluded that the fixations make the MILP infeasible. Therefore backtracking mechanisms have to be implemented that allow undoing variable fixations. Previous studies [Käs97, KL98] have shown that indeed the guesses are often wrong. The resulting code quality is not satisfactory and due to the need for backtracking the computation can take longer than the calculation of an exact solution. Thus more elaborate approaches are required.

In the following, approximation algorithms are presented for the SILP and the OASIC models. The order-indexed SILP formulation allows more efficient approximations than the time-indexed OASIC formulation. The reason is that in the order-indexed formulation the problem can be better decomposed to allow a gradual solution refinement until a feasible solution is obtained. After a flow variable of the SILP formulation x_{ij}^k has been fixed, the ordering of i and j and the resource type i and j are assigned to is fixed. The starting times of the operations however remain variable; if necessary they can change in any subsequent approximation step. Since the decision variables in the OASIC formulation specify the resource assignment and the starting time of the operations, no flexibility remains after a variable x_{in}^k has been fixed. Then operation i must start at control step n and must be executed by an instance of resource type k.

In Sec. 6.1 a short overview of search-based approximation techniques is given. Sec. 6.2 deals with the ILP-based approximations for the SILP formulation while the approximations for the OASIC model are presented in Sec. 6.3. In the pseudocode representation of the algorithms it will be assumed that the information about the mixed integer linear programs to be solved is globally available such that it has not to be passed as a parameter. The modifications of the mixed integer linear programs are described in an informal way.

6.1. Related Work

For several classes of scheduling problems approximation algorithms have been developed that have polynomial time complexity and whose result is guaranteed to be at most ϵ times optimal (ϵ-approximations) [SUW97, Goe97, Sch96b, Sch96a]. The basic idea of many of these algorithms is to use results of LP relaxations as criteria for making scheduling decisions. Mostly however they are restricted to "simple" classes of scheduling problems [SUW97, Sch96b, Sch96a]. In [Goe97] a

2-approximation algorithm is presented for the problem of scheduling jobs with release dates on one machine such as to minimise the weighted sum of the completion times of the jobs scheduled. In [Sch96a] a 3-approximation algorithm has been presented that additionally takes precedence constraints into account and a 7-approximation algorithm that can deal with several parallel identical machines. However, for some combinatorial problems it can be proved that there is no hope of finding an approximation algorithm of specified accuracy unless P=NP [GJ79]. To our knowledge no ϵ-approximation for the instruction scheduling problem as presented in Chap. 2.2.3 has been developed yet. Further complications are that additionally the control flow structure of the programs and the phase-coupling between instruction scheduling, functional unit assignment and register assignment have to be taken into account. Moreover the instruction scheduling problem itself depends on the target architecture with respect to important parameters as the number of functional unit types (machines), the number of instances of each functional unit type, and the execution times of operations. Since the ϵ-approximations are tailored to special problem classes more general approaches are required.

In [BEP+96] an overview of more general search-based heuristic algorithms is given; important methods are simulated annealing, tabu search, and evolutionary algorithms. *Simulated annealing* has originally been inspired by the cooling of solids after they have been heated to their melting point. The algorithm starts with a feasible solution and changes non-deterministically to another feasible solution depending on the value of a cost function. The probability for accepting a new solution increases with the cost reduction but in order to avoid getting trapped in a local optimum a deterioration of the objective function can also be acceptable. The procedure stops if the objective function remains constant in a certain number of consecutive iterations, or if the number of iterations becomes too large. Experimental results in [Mic94] indicate that the results of simulated annealing for instruction scheduling problems is comparable to the result of list scheduling, however at the cost of a possibly much higher computation time.

One of the central ideas of *tabu search* is to guide deterministically the search process out of local optima. The algorithm changes from one feasible solution to another if this change leads to the smallest deterioration of the objective function. In order to prevent oscillating, a dynamic list containing forbidden transitions is maintained, called the tabu list. Tabu search can be extended by intensification and diversification. Regional intensification restricts the search to the subset of feasible solutions sharing some salient characteristics of the best solutions found in some phase of the procedure. The opposite idea is at the base of diversification: if all solutions discovered in some phase of the algorithm share some common features this may indicate that other regions of the search space have not been sufficiently explored.

Another class of search-based algorithms are the *evolutionary algorithms* that have recently also been used for code generation tasks. Evolutionary algorithms, as e. g. genetic programming are probabilistic search algorithms applying the principles of natural evolution (selection and random variation) to a random set of points in the search space [Bli96, ZT99]. The application of evolutionary algorithms to system synthesis has been investigated in [Bli96]; [ZT99] gives an overview (cf. Chap. 11). While evolutionary algorithms are well suited for multicriteria optimisation problems and support parallel algorithms, they typically require high computational effort and do not guarantee to find an optimal solution. They are very sensitive to the adjustment of the internal parameters and the definition of the fitness function [Bli96].

6.2. Approximations for the SILP Formulation

The approximations developed for the SILP formulation start with an initial relaxation where the integrality restrictions of all flow variables associated with functional units and abstract register resources are removed. The set X^{rel} denotes the set of all relaxed flow variables, i. e. of all flow variables that are explicitly specified as integers in the original ILP formulation. It is necessary to distinguish between resources representing functional units and and result buses. If the target architecture requires a synchronisation of the result bus in addition to the synchronisation of functional units the flow variables associated with the result bus are not relaxed. The values of the flow variables of functional units determine the value of the flow variables associated with the result bus. Thus the synchronisation of the result bus must be fully taken into account when synchronisating the functional units.

6.2.1. Stepwise Approximation

The algorithm of the *stepwise approximation* is based on an approximative method sketched in [Zha96]; in Fig. 6.1 the algorithm of the stepwise approximation is shown in pseudocode notation. It consists of two phases; first the flow variables of the execution units are addressed and then the register flow variables. Each phase starts with computing an optimal solution of the current relaxed mixed integer linear program (M_1); in case of the resource flow variables this is the initial relaxation. The core of each phase is a loop that iterates over all control steps c until an upper bound for the execution time of the input program is reached. Separately addressing the resource and register flow variables can cause the number of available registers to be exceeded in the result of the approximations such that additional data moves have to be inserted (see Sec. 7.4.3). In our experimental evaluation however this has never been necessary.

Let X_R be the set of all flow variables $x_{ij}^k \in X^{rel}$ such that k is a functional unit type $k \in N_R^F$, $t_i + z_i - 1 \geq c$, $t_i \leq c$, $t_j + z_j - 1 \geq c$, and $t_j \leq c$ in the solution of M_1. Then in the current schedule both operations i and j use functional units of type k such that those are not ready to accept new data inputs in control step c. All variables contained in X_R are specified as binary and the resulting mixed integer linear program M_2 is solved. The effect of this strategy is that flow variables are only specified as binary where this is inevitable in order to prevent violations of

```
procedure ApproxAstep(X^rel)
{
   AstepIterate( {x_ij^k ∈ X^rel | k ∈ N_R^F});
   AstepIterate( {x_ij^k ∈ X^rel | k ∈ N_R^A});
}
procedure AstepIterate(L^rel)
{
   Solve the relaxed MILP (M_1);
   for c = 1 to M_steps{
      X_R  =  {x_ij^k |  (t_i + z_i − 1 ≥ c ∧ t_i ≤ c)∧
                         (t_j + z_j − 1 ≥ c ∧ t_j ≤ c)∧
                         x_ij^k ∈ L^rel ∧ fixed(x_ij^k) = false}
     Declare all x_ij^k ∈ X_R as binary;
     Solve the modified MILP (M_2);
     forall x_ij^k ∈ X_R {
       if ((x_ij^k = 1) ∧ ((t_i = c) ∨ (t_j = c)) ∧ (fixed(x_ij^k) = false)) {
         Add the constraint x_ij^k = 1;
         fixed(x_ij^k) := true;
       }
     }
   }
   forall x_ij^k ∈ L^rel{
     if (fixed(x_ij^k) = false)
       Specify x_ij^k as binary;
   }
   Solve the modified MILP;
}
```

Figure 6.1.: Stepwise Approximation.

the serial constraints with respect to the current control step c.

Then, if several conditions are met, the variables $x_{ij}^k \in X_R$ are fixed to the value they have in the solution of M_2. In subsequent optimisation steps they can be considered as constants and do not contribute to the problem complexity any more. One necessary condition is that they have the value 1 in the solution of M_2. Fixing a variable x_{ij}^k then means adding the equation $x_{ij}^k = 1$ to the mixed integer linear program. Fixing variables $x_{ij}^k \in X_R$ where $x_{ij}^k = 0$ can lead to infeasibility since non-integral values of other variables might shadow violations of the flow constraints produced by this setting. If in a later step those variables are specified as binary, the MILP becomes infeasible. When fixing only the variables with value 1, this problem cannot occur due to the structure of the flow constraints. The other presupposition is that only variables x_{ij}^k can be fixed that are associated with an operation scheduled to control step c in the solution of M_2. If both operations i and j are scheduled to later control steps, additional serial constraints might be affected such that they have to be addressed in a later step. Since the definition of X_R implies that the algorithm can only fix flow variables between independent operations the fixations do not lead to violations of the precedence constraints. Thus the feasibility of the modified MILP is ensured.

The proceeding of the algorithm is exactly the same for functional resources and abstract register resources. In both cases the function *AstepIterate* is used; the appropriate set L^{rel} of relaxed variables is passed as a parameter. After traversing all control steps, it is still possible for some variables $x_{ij}^k \in L^{rel}$ to have non-integral values. Therefore all flow variables of L^{rel} that have not been fixed to an integer up to now are specified as binary and the resulting MILP is solved. This solution is guaranteed to be integral since all previously relaxed variables have been fixed to an integral value, or have been respecified as binary.

In the main loop of the algorithm only operations whose current scheduling violates the serial constraints are specified as binary. This way the number of variables simultaneously specified as integers is reduced. Then an optimal solution of the resulting mixed integer linear program is computed and the ordering among the colliding operations is fixed. In addition to further reducing the number of variables simultaneously specified as binary the fixations have the effect that serial constraints are transformed into precedence constraints. As detailed in Chap. 5, deviations of the SILP polytope from the integral polytope are mostly caused by the serial constraints. Thus with each fixation one serial constraint is effectively eliminated such that the resulting polytope approximates the integral polytope more closely. In consequence a significant reduction of computation time can be achieved. In each iteration an optimal solution with respect to the collisions occurring at the current control step is computed. Therefore it can be assumed that while significantly reducing the computation time the stepwise approximation still leads to a good overall solution. This is confirmed by the experimental results (see Chap. 10).

```
procedure ApproxAIFlow(X^rel)
{
  Solve the relaxed MILP (M_1);
  ResourceList = SortResources(N_R^F, N_R^A);
  forall k ∈ ResourceList {
    forall x_ij^κ ∈ X^rel {
      if (κ = k)
        Specify x_ij^κ as binary;
    }
    Solve the modified MILP (M_2);
    forall x_ij^κ ∈ X^rel {
      if (κ = k) {
        Fix x_ij^κ to its current value;
        Relax x_ij^κ;
      }
    }
  }
}
```

Figure 6.2.: Approximation of Isolated Flows.

6.2.2. Isolated Flow Analysis

In the SILP formulation a separated flow network is generated for each resource type. The concept of resource flows offers a natural method of problem decomposition that is at the basis of the *approximation of isolated flows*: The resource types of the target processor are addressed iteratively; only the flow variables associated with the current resource type are specified as binary. This way the number of variables that are simultaneously specified as binary can be significantly reduced. The algorithm for the isolated flow analysis is shown in Fig. 6.2.

In the initial relaxation, all flow variables $x_{ij}^k \in X^{rel}$ are relaxed. In the main loop of the algorithm the hardware resources of the target processor are traversed. In each iteration, the flow variables associated with the current resource type k are specified as binary and the modified MILP is solved. Then the flow variables x_{ij}^k of the current resource type k are fixed to the value they have in the solution of M_2 by additional equality constraints and the next resource type is addressed. This is repeated for all resource types, so a feasible integral solution is obtained in the end.

In each iteration only the flow variables of the current resource type k are specified as binary. The variables of the other resource types either have been addressed in a previous iteration and can be considered as constants or they are still relaxed (see Fig. 6.3).

$$
\begin{array}{lll}
0 \le x_{ij}^{\kappa} \le 1 & \forall\ x_{ij}^{\kappa}\ \in L^{rel} \text{ where } \kappa \neq k & \wedge\, \mathit{fixed}(x_{ij}) = \mathit{false} \\
x_{ij}^{\kappa} \in \{0,1\} & \forall\ x_{ij}^{\kappa}\ \in L^{rel} \text{ where } \kappa = k & \wedge\, \mathit{fixed}(x_{ij}) = \mathit{false}
\end{array}
$$

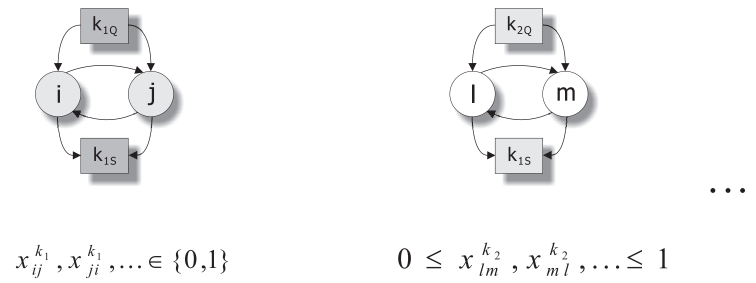


Figure 6.3.: Isolated flow analysis.

The order in which the resource types of the target processor are traversed is based on the number of flow variables x_{ij}^k associated with each resource type k. First the resource types with the lowest number of flow variables are addressed. This way the search space for the largest subproblems (in terms of associated flow variables) is reduced due to the fixations of the preceding iterations. There are other possible heuristics that could be applied; however this is subject of future research.

In each iteration of the algorithm an optimal solution with respect to the current resource type is computed. Since no constraints are omitted the complete problem information is used such that the solution is also influenced by the effects on other resource flows. While no optimal solution can be guaranteed a high solution quality can be expected since the overall solution is composed of the individually optimal solutions of all subproblems. The experimental results indicate that in fact an optimal solution is obtained in most cases. The computation time is reduced significantly since only the flow variables associated to one resource type are considered as binary at a time and the search space is tightened after each approximation step.

6.2.3. Stepwise Approximation of Isolated Flows

The *stepwise approximation of isolated flows* is a combination of the two previously presented algorithms. Again all resource types of the target processor are traversed, but instead of exactly solving the decomposed problem as in the approximation of isolated flows, in each iteration the stepwise approximation is used for the variables of the current resource type. Thus for each resource type, all control steps are traversed in an inner loop until an upper bound for the execution time of the input program is reached. In each iteration of the inner loop the variables of the current resource type are specified as binary if they cause violations of serial constraints with respect to the current control step c. Fixations of variables induce a partial ordering among the operations and reduce the number of active serial constraints. After all control steps have been traversed, feasible resource flows have been computed for the current resource type. After all resource types have been traversed, a feasible integer solution results. The algorithm of the stepwise

```
procedure ApproxASIF(X^rel)
{
  Solve the relaxed MILP;
  ResourceList = SortResources(N_R^F, N_R^A);
  forall k ∈ ResourceList {
    L^rel := ∅;
    forall x_ij^κ ∈ X^rel{
      if (κ = k)
        Insert x_ij^κ into L^rel;
    }
    AstepIterate(L^rel);
    forall x_ij^κ ∈ X^rel{
      if (κ = k) {
        Fix x_ij^κ to its current value;
        Relax x_ij^κ;
      }
    }
  }
}
```

Figure 6.4.: Stepwise Approximation of Isolated Flows.

approximation of isolated flows is shown in pseudocode in Fig. 6.4.

The setup overhead of this approximation is larger than that of the previously presented approximations so that the computation times for small input programs are higher. For larger input programs however experimental results show that in most cases the stepwise approximation of isolated flows leads to the shortest computation time while producing solutions of very high quality (see Chap. 10).

6.2.4. Approximation of Isolated Operations

The *approximation of isolated operations* has been developed for architectures where the set of alternative functional units an operation can be mapped to is large. In this case, a reduction of complexity can be achieved when in a first pass only the assignment of operations to resources is determined. This assignment can be used to eliminate the variables that describe possible assignments of each operation to other resources. The resulting MILP can be solved either exactly or by using any of the previously presented approximations.

The algorithm of the approximation of isolated operations is shown in Fig. 6.5. First for each operation of the input procedure the number of functional units it can be mapped to is determined. Then the operations are traversed where those with the smallest number of alternative functional units are addressed first. All flow variables associated with the current operation are specified as binary and the resulting mixed integer linear program M_1 is solved. Then the functional unit k is determined to which the variable is assigned in the solution of M_1. The mixed

```
procedure ApproxAIOP(X^rel)
{
  Insert all operations of the input procedure in a list Oplist;
  Sort Oplist such that i ⪯ j ⇔ |{k|(i,k) ∈ E_R}| ≤ |{k|(j,k) ∈ E_R}|;
  forall o ∈ Oplist {
    forall x_ij^k ∈ X^rel{
      if (i = o ∨ j = o)
        Specify x_ij^k as binary;
    }
    Solve the modified MILP (M_1);
    Determine κ ∈ N_R^F such that ∃ x_ij^κ = 1 where (i = o ∨ j = o);
    Add the constraint Φ_o^κ = 1 to the MILP;
    Add the constraint Ψ_o^κ = 1 to the MILP;
    forall x_ij^k ∈ X^rel{
      if (k ≠ κ ∧ (i = o ∨ j = o)) {
        Add the constraint x_ij^k = 0 to the MILP;
    }
    forall x_ij^k ∈ X^rel{
      if (i = o ∨ j = o) {
        Relax x_ij^k;
    }
  }
  Solve the modified MILP exactly or by calling
    ApproxAStep, ApproxAIFlow, or ApproxASIF (M_2);
}
```

Figure 6.5.: Stepwise Approximation of Isolated Operations.

integer linear program is extended by additional constraints that force the incoming and outgoing resource flow of the current operation with respect to k to be equal to one. All variables associated with the current operation and another functional unit type than k are set to 0. Then all variables are relaxed again and the next operation is addressed.

After all operations have been traversed, the assignment of operations to functional units has been determined. The resulting mixed integer linear program M_2 then can be solved either exactly, or by using the approximative algorithms of Sec. 6.2.1 – Sec. 6.2.3. The effect of the approximation of isolated operations is that the task of functional unit allocation is partially decoupled from instruction scheduling and register assignment. However phase-coupling effects are still taken into account since the complete problem information is used and no constraints are omitted when computing the functional unit assignment of each operation.

6.3. Approximations for the OASIC Formulation

As mentioned at the beginning of this chapter, problem decomposition is more difficult in the OASIC formulation than in the SILP formulation. This is due to the choice of the decision variables; the consequence is that the development of efficient approximations is impeded. In the OASIC model each decision variable x_{in}^k describes the starting time of an operation and the functional unit it is assigned to. After fixing a variable x_{in}^k, the starting time of the operation, its completion time and the functional unit it is assigned to become invariant.

If a flow variable x_{ij}^k of the SILP formulation is fixed, the assignment to functional units and the relative ordering of the operations i and j are fixed; the starting time however may vary. If the resource flows are addressed iteratively, after each iteration the flow variables associated with one resource type are fixed. This induces a partial ordering of the operations but still the distance between them can be changed, if this is necessary due to the sequentialisation of operations of other resource types.

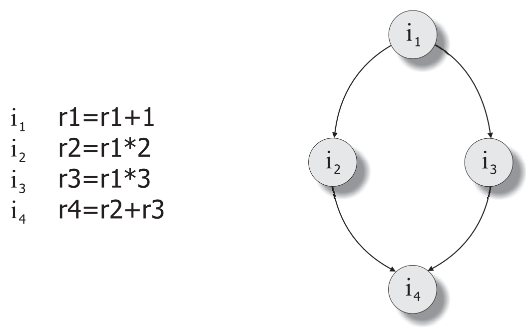


Figure 6.6.: Fixation problem for OASIC models.

If a variable x_{in}^k of the time-based OASIC formulation is fixed, its starting

```
procedure ApproxOAstep(X^rel, l)
{
   Solve the relaxed MILP (M_1);
   for c = 1 to M_steps{
     X = {x^k_in | x^k_in ∈ X^rel ∧ n = c ∧ fixed(x^k_in) = false} ∪ L^A(l, c);
     Declare all x^k_ic ∈ X as binary;
     Solve the resulting MILP (M_2);
     forall x^k_ic ∈ X{
       if ((x^k_ic = 1) ∧ fixed(x^k_ic) = false){
         Add the constraint x^k_ic = 1;
         k*(i) := k;
         fixed(x^k_ic) := true;
       }
     forall x^k_ic ∈ X{
       if (k ≠ k*(i) ∧ n ≠ c) {
         Add the constraint x^k_ic = 0;
         fixed(x^k_ic) = true;
       };
       Relax x^k_ic;
     }
   }
}
```

Figure 6.7.: Stepwise Approximation in OASIC.

time cannot be changed later on. The immediate consequence is that a problem decomposition based on resource types is not possible. This can be illustrated by a simple example. Consider the situation of Fig. 6.6. Assume that the target architecture disposes of one ALU and one multiplier. If only variables associated with the ALU are specified as binary, it would be possible to set $t_{i_1} = c$ and $t_{i_2} = c + 2$. The necessary sequentialisation of i_2 and i_3 will only become apparent in the next iteration when the multiplier variables are specified as binary. But then there would be no feasible solution any more while in the SILP formulation the modification of the starting time would still be possible.

Thus among the approximations presented in Sec. 6.2.1 – Sec. 6.2.3 only the stepwise approximation can be applied to the OASIC formulation. The algorithm of the stepwise approximation of the OASIC model is shown in pseudocode in Fig. 6.7. Similarly to the SILP approximations, if the target architecture requires a synchronisation of the result bus the decision variables associated with the result bus are not relaxed. Fixing the starting time of an operation to control step c implicitly determines the time at which the operation accesses the result bus and vice versa. Thus the synchronisation of the result bus must be fully taken into account when synchronising the functional units.

A difference to the stepwise approximation of the SILP formulation is the usage of the lookahead set $L^A(l, c)$ where l is a lookahead value and c denotes the current control step. The lookahead can be specified as a command line parameter in the PROPAN-system. For a lookahead of l cycles the set $L^A(l,c)$ is defined as follows:

$$L^A(l,c) = \{x_{in}^k \mid x_{in}^k \in X^{rel} : alap(i) \leq c + l\}$$

It contains the relaxed decision variables associated with all operations whose latest possible starting time is reached before l cycles from the current control step. Those variables are specified as binary.

In the time-indexed OASIC formulation for each control step n where the execution of an operation i may be started a dedicated variable x_{in}^k is introduced. The set of feasible control steps is determined by *asap*/*alap* analysis with the help of an upper bound of the execution time of the input program. No variable will be generated that allows an operation to be scheduled after its *alap* control step. However it is a reasonable approach to use the execution time of the schedule of the input program as an upper bound and enforce this upper bound by an additional constraint in the generated integer linear programs. But then suboptimal decisions during the approximation could force an operation to be scheduled after its *alap* control step which leads to infeasibility. The lookahead can be used to prevent that situation. In the experimental results a lookahead of two cycles has been sufficient.

The algorithm works similarly as the algorithm of Fig. 6.1. First the initial relaxation is solved where all variables x_{in}^k associated with functional units of type $k \in N_R^F$ are relaxed. Then all control steps are traversed until an upper bound for the execution time of the input program has been reached. In each iteration a set X is computed that contains all relaxed variables associated with the current control step c and the variables of the lookahead set $L^A(l,c)$. The variables $x_{in}^k \in X$ are specified as binary and the modified mixed integer linear program M_2 is solved. Then for all variables $x_{in}^k \in X$ where $x_{in}^k = 1$ in the solution of M_2 a constraint is added to the MILP fixing that variable to 1. Additionally all other variables from the set X that are associated with operation i can be set to 0. Finally all variables $x_{in}^k \in X$ are relaxed again and the next control step is addressed. The fixed variables are treated as constants in the next iteration and do not contribute to the problem complexity any more.

Again in each iteration variables are only specified as binary where this is inevitable in order to obtain a feasible solution. The number of variables that are simultaneously specified as binary is reduced such that the computation is significantly accelerated. As the experimental results show in most cases an optimal result is obtained.

7. Superblock-Based Code Optimisation

A number of studies have established that the typical size of single basic blocks is about 5 to 20 instructions on the average [RF93]. In consequence the available parallelism inside of basic blocks is limited. For this reason, global scheduling algorithms have been developed that can jointly schedule multiple basic blocks. These algorithms can be classified as cyclic or acyclic scheduling strategies. Acyclic scheduling methods do not allow operations to be moved across loop back edges. Popular algorithms of this class are trace scheduling [Fis81, Ell86], superblock scheduling [HMC+93], region scheduling [GS90], or mutation scheduling [Nic85, NN94]. Cyclic global scheduling methods attempt to directly optimise the schedule across loop back edges as well; they are concerned with unrolling and overlapping different loop iterations. The most prominent algorithms of this category are loop unrolling and software pipelining algorithms [Lam88, NN92, EM92, BGS94, HHR97].

While most of the global approaches are based on graph-based heuristic decisions, virtually all exact scheduling methods are limited to single basic blocks [Leu97, HD98, WGHB95, Zha96]. In order to overcome this restriction, we have developed a method to incorporate the control flow structure of a code sequence into an integer linear program. The resulting scheduling strategy can be categorised as global acyclic scheduling in terms of the classification mentioned above. Since an important goal of this thesis is the exact coupling of different code generation phases, a further extension of the problem dimension by incorporating cyclic scheduling strategies does not seem to be promising for reasons of complexity [RGSL96].

The optimisation phase of Propan works on superblocks that are constructed by appropriately grouping the basic blocks of the input program. The basic blocks contained in each superblock are optimised jointly. The construction of the superblocks is performed by a modified version of the trace construction algorithm of Fisher [Fis81]. In contrast to trace scheduling code movements are only allowed between control equivalent basic blocks that are contained in the same superblock. If code movements between basic blocks that are not control equivalent are to be supported, it becomes necessary to insert compensation code in order to preserve program semantics. This can be done in a dedicated bookkeeping phase after the optimisation proper. The necessary extensions are straightforward, but are not part of the current implementation. An extension with respect to the trace scheduling algorithm is the transgression of loop boundaries that is motivated by the phase integration of instruction scheduling, register assignment and resource allocation. Apart from increasing the available parallelism the superblock mechanism has the function of permitting global allocation decisions. The register assignment and resource allocation performed in one basic block can have significant consequences for the optimisation of other basic blocks. By generating one integer linear program per superblock the effects of the allocation decisions on all contained basic blocks are precisely modelled. Since the superblocks can be extended across loop boundaries it becomes also possible to consider interdependencies of allocation decisions between different loops. Since the loops usually represent the most important parts of an algorithm, this can have a significant impact on the resulting code quality, especially in case of nested loops.

Similar to the traces of [Fis81] it is not allowed to incorporate disjoint control flow paths in the same superblock. Moreover, in order to limit the complexity of the individual integer linear programs it may be advisable to stop the superblock construction process if a certain code size threshold is reached. Thus the input program can be represented by several superblocks. The allocation and scheduling decisions of one superblock then have to be respected during the subsequently addressed superblocks. In order to restrict the negative consequences of those interdependencies to less important program parts, the superblocks of the input routine are optimised in the order of decreasing execution frequency, similarly to the trace scheduling algorithm.

The remainder of this chapter is organised as follows: the basic concepts of the superblock mechanism and the algorithms for computing the superblocks are presented in Sec. 7.1. Sec. 7.2 details the extensions of the ILP formulations that are necessary for modelling the global register assignment problem, i. e., if operations from different loops have to be represented by the same integer linear program. Further extensions to the ILP formulations that are required for correctly performing instruction scheduling across loop boundaries are presented in Sec. 7.3. Sec. 7.4 deals with the constraints that make the scheduling and allocation decisions of previously addressed superblocks of the input program visible. Those constraints have to be respected in order to obtain a globally feasible schedule. In Sec. 7.5 approaches for completing the register assignment in the presence of multiple superblocks is presented.

7.1. The Superblock Graph

In this section the concepts of superblock and superblock graph are defined and the algorithm for computing the superblocks is presented. The algorithm consists of two parts: first it determines an initial covering of the basic blocks in the input routine by superblocks where the superblock construction stops at loop boundaries. After that a merging algorithm is performed that aims at extending the superblocks across loop boundaries. By suppressing the merging step, any enlargement of superblocks across loop boundaries can be prevented such that each loop is separately optimised.

Definition 7.1 (Superblock) *Let a basic block graph G_B be given. A superblock s is a subgraph of G_B, $s = (N_s, E_s, b_A, b_\Omega)$ such that $N_s \subseteq N_B$, $E_s \subseteq E_B$, and $E_s \subseteq N_s \times N_s$. The block b_A is the uniquely determined entry block b of s such that $\nexists b' \in N_s : (b', b) \in E_B^+$. b_Ω is the uniquely determined exit block b of s such that $\nexists b' \in N_s : (b, b') \in E_B^+$.*

Definition 7.2 (Superblock Graph) *Let a basic block graph G_B be given. The superblock graph $G_S = (N_S, E_S, s_A, s_\Omega)$ is generated from G_B by grouping the nodes for individual basic blocks and the edges between them into superblocks. An edge $e = (s_i, s_j) \in E_S$ connects two superblocks s_i and s_j if there is a block $b_n \in N_{s_i}$ and a block $b_m \in N_{s_j}$ such that $(b_n, b_m) \in E_B$. The set N_S represents a partition of N_B, i. e. the node sets of the superblocks are pairwise disjoint ($\forall s_i, s_j \in N_S : N_{s_i} \cap N_{s_j} = \emptyset$) and the union of their node sets is the node set of the basic block graph $\bigcup_{s \in N_S} N_s = N_B$.*

7.1.1. Superblock Covering

In the superblock covering phase, the basic block graph is partitioned into a set of disjoint control flow paths. The central idea is to group basic blocks that represent a frequently executed program path to one superblock. The information about how often the basic blocks are executed can be derived from profiling information, or from heuristic estimates. In the covering phase it is not allowed to extend superblocks across loop boundaries.

The algorithm starts with selecting the most frequently executed basic block of the input routine as the seed of the first superblock. For each loop header block an individual superblock is created; the enlargement of those superblocks is addressed in the merging stage. If the seed block is no loop header, the algorithm tries to enlarge the superblock by adding additional basic blocks. The basic blocks that are inserted into a superblock s must always form a connected path p in the basic block graph G_B. Candidates for being inserted into s are all predecessor blocks of the entry block $b_A(s)$ of s and all successor blocks of its exit block $b_\Omega(s)$ in the reduced transitive hull G_B^+ of the basic block graph. Additional conditions are that they have not been assigned to a superblock yet and that by the enlargement, no loop boundaries are crossed. In consequence there are never disjoint control flow paths in the same superblock since extensions along side entrances or side exits of s are prevented. It is, e. g., excluded that both the *then*- and the *else*-part of a conditional statement are contained in the same superblock. The enlargement is stopped if the candidate list is empty, or if an optional code size threshold is exceeded. Then the most frequently executed basic block that has not been assigned to a superblock yet is selected as the seed of a new superblock that will subsequently be enlarged as much as possible. This process is repeated until all basic blocks of the routine have been covered by superblocks.

An example basic block graph, and the superblocks that will have been constructed at the end of the covering stage are shown in Fig. 7.1.

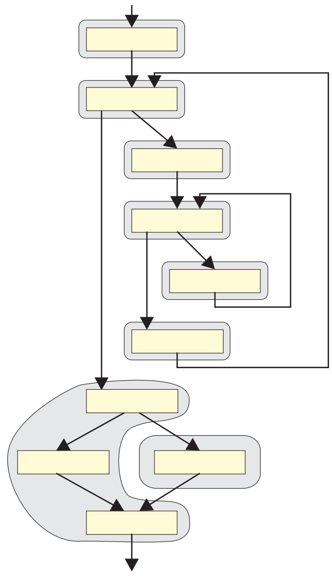


Figure 7.1.: Superblock graph after covering phase.

7.1.2. Superblock Merging

After the basic block graph has been covered by superblocks, a merging algorithm is performed with the goal of extending the superblocks across loop boundaries. Before the algorithm is described in more detail, some definitions have to be given.

The *priority* $\pi(s)$ of a superblock s is defined as the maximum of the execution frequencies of all basic blocks contained in s: $\pi(s) = max_{b \in N_s} \zeta(b)$. The *directly enclosing loop* of a superblock s is defined as the loop l with the largest nesting depth such that all basic blocks $b \in N_s$ are contained in the body of l. A superblock that only contains sequential code, i. e. that does not contain a loop header block, is called *sequential superblock*. Let a superblock s be given that contains a set $B \subseteq N_s$ of basic blocks from the body of a loop $l = (N_l, E_l, h_l)$, i. e., $B \subseteq N_l$. Then s is called a *primary trace* of l if the blocks in B represent a complete path through one loop iteration of l. If the blocks in B do not represent a complete path through l, s is called *fragmentary trace* of l.

The algorithm does not allow the merging of sequential superblocks or primary loop traces with fragmentary traces. Fragmentary traces of a loop l can only be merged with other fragmentary traces of l. This restriction prevents basic blocks from outside a loop from being contained in the same superblock as the basic blocks of the loop body unless those represent the most frequently executed

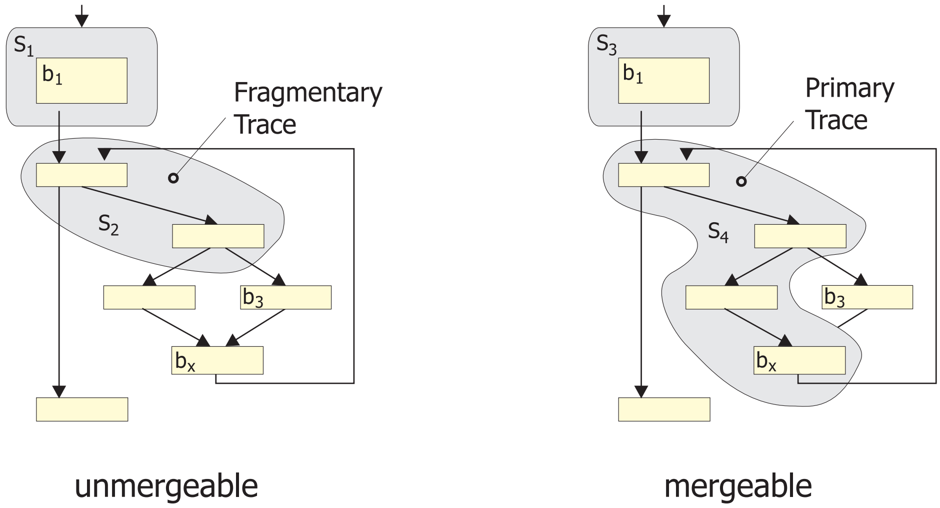


Figure 7.2.: Examples for unmergeable/mergeable superblocks.

path through the loop. An illustration is given in Fig. 7.2. If the superblocks s_1 and s_2 were merged the scheduling and allocation decisions for block b_1 could prevent an efficient scheduling of b_x since one integer linear program is generated per superblock. This way the optimisation of the combined superblock of s_1 and s_2 would impose restrictions to more important program parts since the scheduling of b_x affects each path through the loop body. By merging the superblocks s_3 and s_4 however a higher code quality can be achieved since the interdependencies of scheduling and allocation decisions for all operations in one superblock are precisely taken into account. The block b_1 is scheduled in the best way that allows an optimal scheduling of the most frequently executed path through the loop. A performance degradation can only result for the rarely executed block b_3. If it is explicitly required that no decisions from outside a loop are allowed to affect the optimisation of the loop body, the merging stage can be suppressed. This way, any enlargement of superblocks across loop boundaries is prevented so that each loop is separately optimised.

The algorithm for superblock merging is illustrated in Fig. 7.3 in pseudocode notation. It starts with selecting the superblock s with the highest priority $\pi(s)$ from the current superblock graph G_S and attempts to enlarge s by performing merge operations with preceding or succeeding superblocks. Candidates for merging are the superblocks sp whose exit block $b_\Omega(sp)$ is a predecessor of the entry block of $b_A(s)$ of s and the superblocks ss whose entry block $b_A(ss)$ is a successor of the exit block $b_\Omega(s)$ of s. Two superblocks s' and s'' can only be merged if several conditions are satisfied:

- s' contains only the header h_l of a loop l and by inserting h_l into s'', s'' becomes a primary trace of l, or
- s' and s'' are primary loop traces, or

```
procedure MergeSuperblock(G_S)
{
  repeat {
    foreach s ∈ N_S in priority order {
      repeat {
        CList := SelectMergingCandidates(s);
        Sort CList in priority order;
        foreach s' ∈ CList until (merging_done = true) {
          if (MergingFeasible(s, s') = true) {
            Merge(s, s');
            merging_done = true;
          }
        }
      } until s cannot be further enlarged (L1);
    } (L2)
  } until G_S is stable (L3);
}
```

Figure 7.3.: Superblock Merging.

- s' and s'' are fragmentary loop traces of the same loop l, and
- the new superblock obtained by merging s' and s'' does not exceed the code size threshold.

These restrictions are checked by the function *MergingFeasible* and ensure that no superblock contains disjoint control flow paths and that the less important program parts do not impose constraints to more important parts.

The algorithm terminates if there are no superblocks left that can be enlarged by merge operations. If the input routine contains n basic blocks at most n merge operations can be performed. A new iteration of the loop L_3 is only started if there has been a merging operation in the previous iteration. Since each merging step reduces the number of superblocks at least by one there are at most $\frac{n(n-1)}{2}$ iterations of the loop L_1. Due to the sorting algorithm the worst case time complexity for an iteration of L_1 is $\mathcal{O}(n \log n)$ yielding a worst case time complexity of $\mathcal{O}(n^3 \log n)$ for the complete algorithm. Since most basic blocks of structured programs have one or two successors respectively predecessors, it can be expected that in each iteration of L_2 several merging operations are performed so that the average case complexity can be assumed to approach linearity. In Fig. 7.4 the proceeding of the merging algorithm is visualised for the initial superblock graph of Fig. 7.1.

7.1.3. Partitioning

In the general case it cannot be excluded that the input routine contains basic blocks that are too large for an ILP solution to be calculated in acceptable time. This may, e. g., occur if the input program has been generated using techniques like

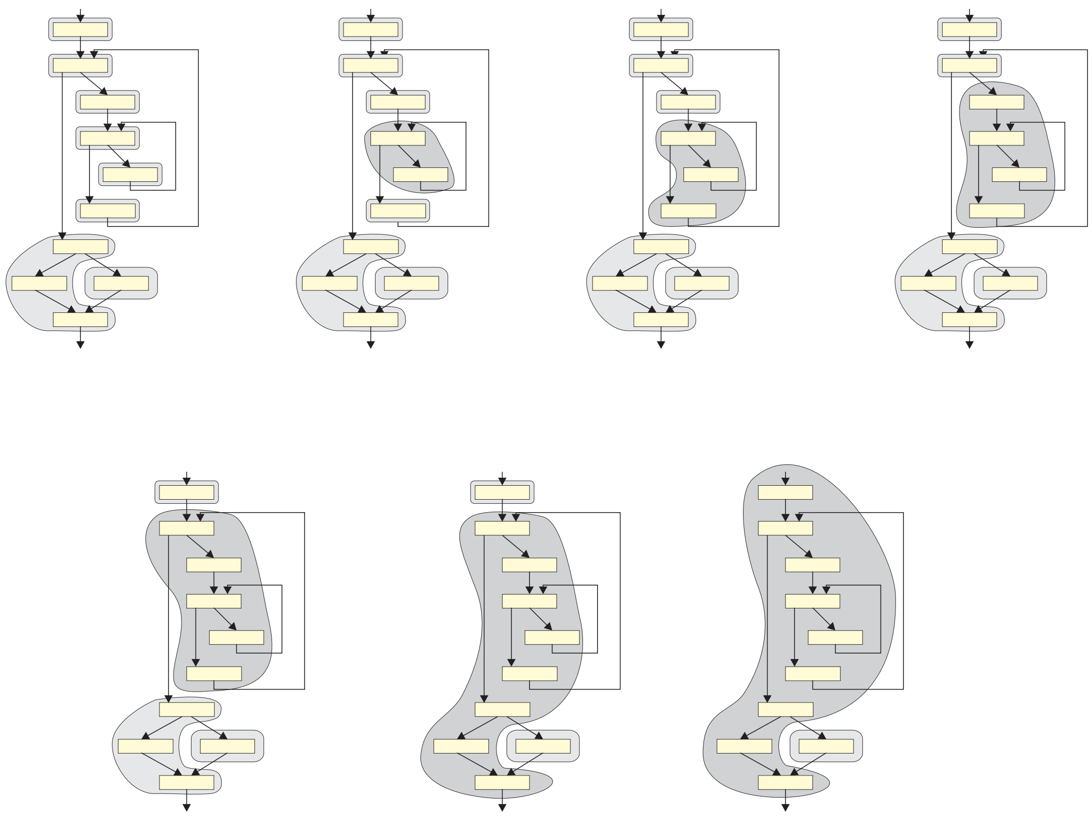


Figure 7.4.: Proceeding of the merging algorithm.

if-conversion [PS91, DT93] or extensive loop unrolling. In order to allow an efficient ILP-based optimisation those blocks have to be partitioned. In the following two simple partitioning algorithms are presented. Alternative approaches have been proposed in the literature [MP97]; an integration of those algorithms might be a promising topic for future research.

Order-based Partitioning

Order-based partitioning takes advantage of the ordering of the operations in the input routine which is a reasonable approach in a postpass framework. Each basic block exceeding the code size threshold is subdivided into a minimal set of subblocks of equal size such that no subblock exceeds the threshold. When constructing the subblocks it has to be ensured that all machine operations that have been scheduled to one long instruction in the input program are contained in the same subblock. Dedicated edges are added to the basic block graph that connect consecutive subblocks. We will denote the transformed graph as *weak basic block graph* since the blocks are not necessarily operation sequences of maximal length any more as required by Def. 2.3. The weak basic block graph is used as the input of the superblock construction algorithm; each of the newly created blocks will be contained in a separate superblock.

Dependence-based Partitioning

Another possibility is to use the data dependence graph E_D of the input program as the basis for partitioning.

Let a basic block b be given that has to be partitioned. First, an empty block b_s is allocated. For each operation i of the original basic block b, a priority is calculated as the maximal length of any path in G_D from i to another operation from the same basic block; this corresponds to the highest level first criterion often used in list scheduling algorithms [LDS80]. Then the operations of b are traversed in the order of decreasing priority. For each operation i the algorithm collects all transitive predecessors of i in G_D that have not yet been inserted into a subblock of b and inserts them into b_s together with i. If the number of operations in b_s has reached the code size threshold, a new subblock is allocated. This is repeated until all operations of b have been transferred into smaller subblocks. Again, successive subblocks are connected by edges in the resulting weak basic block graph that is used as the input of the superblock construction algorithm and each of the newly created blocks will be contained in a separate superblock.

The effect of the dependence-based partitioning is that the instruction scheduling of large basic blocks exceeding the code size threshold can be considered as an enhanced version of list scheduling. The highest-level-first heuristic is used to redistribute the operations of the input program in a set of scheduling windows. Each window corresponds to a superblock that contains one part of the original basic block. The operations of each scheduling window can be scheduled optimally by integer linear programming.

7.2. The Global Register Assignment Problem

In Chap. 5 it was described how the basic SILP and OASIC models can be extended to incorporate the register assignment problem on basic block level. In this section the modelling extensions are presented that are necessary to cope with optimisation scopes comprising multiple basic blocks and possibly crossing loop boundaries. Since our previous studies [Käs97, KL98, KL99] indicate that the incorporation of the register assignment in the OASIC modelling cannot be considered promising for complexity reasons we will exclusively focus on the SILP formulation during the remainder of this section.

This section is structured as follows: first a global register renaming algorithm is presented that undoes the register assignment of the input program and can cope with heterogeneous register files. Subsequently the modelling of variable life ranges in the presence of multiple definitions reaching individual uses is explained. If superblocks are extended across loop boundaries this modelling plays an essential role for preserving the semantics of the input program.

7.2.1. Global Heterogeneous Register Renaming

Since Propan has been designed as a framework for postpass optimisations the input consists mainly of assembly programs that have been generated by traditional compilers or by hand. In consequence the task of register assignment has already been addressed when generating the assembly files that are used as the input of Propan. The register assignment of the input program induces *false dependences* that do not reflect data dependences imposed by the program semantics but are caused by the reuse of physical registers. Most output and anti dependences are false dependences that can be removed without changing the program semantics. In consequence they artificially limit the available parallelism of the input program. The impact of false dependences is illustrated in Fig. 7.5. There is an anti dependence from operation y to z that is caused by the reuse of register `r1`. Assume that each operation is executed in one clock cycle and that an addition and a multiplication can be parallelised. Because of the anti dependence, the third operation z is transitively dependent on x and cannot precede y. Now assume that register `r6` is available; then using `r6` as the destination of z allows x and z to be executed in parallel such that one clock cycle is saved.

It is obvious that the register assignment of the input program has to be undone before the optimisation is started in order to not artificially restrict the optimisation opportunities. The algorithm used for this purpose is called *register renaming*. In the literature, the term *register renaming* has been used in different contexts: register renaming on source level and register renaming on hardware level. In the scope of this thesis we will refer to register renaming as the task of replacing

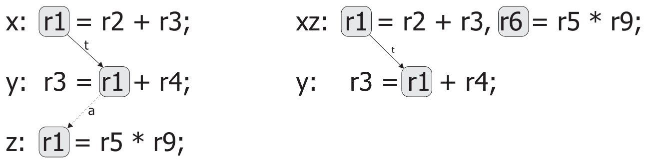


Figure 7.5.: Effects of false dependences.

references to physical registers by references to virtual registers while preserving the program semantics. Since the number of virtual registers can be considered infinite, registers are reused only if the reuse is dictated by the program semantics. This approach is similar to the usage of virtual registers in [Bra91].

Our work differs from previous register allocation and assignment algorithms in two aspects. Since the optimisation scope is extended across basic block boundaries there may be more than one definition that reaches the same use. It has to be ensured that in the resulting machine code all those definitions write to the same physical register. In high-level code generation systems, the programs are usually converted into *Static Single Assignment* form (SSA-form) in order to deal with this problem [CFRW91]. Intuitively, in SSA-form each variable definition is assigned a unique destination; those destinations can be considered as virtual registers. If there are uses which are reached by several definitions, all reaching definitions are merged into a new location by the so-called Φ-nodes. Different register allocation algorithms propose different ways to handle Φ-nodes. The Yorktown Register Allocator [Cha82] and improvements of it (cf. [Bri92, BCT94]) unify all life ranges reaching a given use thus coalescing life ranges at control flow joins. Since all definitions are represented by the same node, it is guaranteed that they are assigned to the same physical register. The probabilistic register allocation of [PF92] is an example of an approach where the Φ-nodes are first translated to a set of register moves or load/store operations. Subsequently the algorithm tries to detect unnecessary data transfer operations and remove them in a way that allows for the greatest increase in efficiency. However both approaches cannot directly be applied in the setting of this thesis. Since instruction scheduling and register assignment are addressed jointly, all definitions reaching the same use must be modelled as separate operations, i. e. they cannot be coalesced. The insertion of additional operations has to be avoided, since the goal of the optimisation phase is to compact the code sequence for reducing code size and speeding up program execution. Moreover if the ILP model was extended by an additional code selection to insert new data transfer operations or remove data transfers that become superfluous due to a skilful register assignment the complexity of the modelling and the required computation time would rise significantly (cf. [WGB94]). Thus the integer linear programs are formulated with the goal of determining an optimal solution for the given set of machine operations that is considered to be invariant. The register renaming algorithm has to use the same virtual register for all definitions of a variable that reach the same use. In the ILP modelling, additional synchronisation constraints have to be generated to ensure that the sharing of the result register is respected among all concurrent definitions. Those constraints are detailed in Sec. 7.2.4.

The second extension is due to the fact that embedded processors often dispose of heterogeneous register files. The renaming algorithm must be aware of that heterogenicity in order to ensure that always registers of an appropriate register file are selected and in order to allow restricting the renaming to registers of individual register files. Comparable extensions to graph colouring algorithms have been suggested in [BCT94]. The effects of heterogenicity can be illustrated with the example of the Analog Devices ADSP-2106x SHARC. The SHARC disposes of several special purpose register files, e. g. an index and an offset register file used for memory addressing. However the effect of the special purpose register assignment on the available parallelism can be neglected in the ILP-based optimisations; only the assignment of general purpose registers has a significant impact. Therefore the special purpose register assignment of the input program can be kept while the general purpose register assignment should be undone. This way, the complexity of the optimisation phase is reduced. If the register reassignment was computed for all register files, the ILP formulation would be significantly more complex without holding a prospect of improving the solution quality.

In the following, the register renaming algorithm shown in Fig. 7.6 is described in detail. For each physical register file subject to renaming, a dedicated virtual register file of unconstrained capacity is introduced. The information about which registers should be renamed is specified by dedicated attributes in the TDL-description of the target processor so that the renaming process can be fully automated. The algorithm traverses the operations of each basic block in backward direction. For each use of a physical register that is to be renamed all reaching definitions are collected and inserted in a set D. If none of these operations have already been assigned to a virtual register a new register of the appropriate virtual register file is allocated and is marked as the destination for all operations in D. If there is exactly one virtual register among the destinations of the operations in D, this register is used as the destination of all operations in D. However if there is more than one virtual register among the destinations of D a new virtual register v is allocated. Then the algorithm replaces all references to the virtual registers used as destinations for operations in D by v. This is necessary to ensure that all concurrent definitions write to the same virtual register.

Subsequently the register renaming algorithm addresses the procedure calling conventions. The registers used for procedure parameters and return values must be preserved by the register renaming which is ensured by the function *AddressCallingConventions*. In the virtual registers allocated for parameter and return values the corresponding mandatory physical registers are annotated. This information is taken into account when generating the integer linear programs and computing the modified register assignment. Memory access operations used for saving the values of caller-saved registers before a call operation and restoring them after the call has returned are excluded from the register renaming process. Those

procedure *RenameRegisters*(G_C)
{
  GenerateVirtualRegisterFiles(ResourceTable);
  foreach basic block $b \in N_C${
    foreach operation $i \in b$ in reverse order {
      foreach use of a register r subject to renaming by operation i {
        $D =$ *CollectReachingDefinitions(i,r)*;
        $V =$ *CollectVirtualDestinations(D)*;
        if $(|V| = 1)$ {
          $v =$ *head(V)*;
          AdaptDestinations(D,v);
        }
        else {
          $v =$ *AllocateVreg(r)*;
          AdaptDestinations(D,v);
        }
      }
    }
  }
  AddressCallingConventions(G_C);
  foreach basic block $b \in N_C$ {
    foreach operation $i \in b$ {
      AdaptUses(i);
    }
  }
}

Figure 7.6.: Register Renaming Algorithm.

operations have to be considered as a part of the call and their original physical register assignment is kept. Callee-saved registers are saved at procedure entry and restored at procedure exit. Since those operations depend on the set of registers used within the procedure it is assumed in the current implementation that they are identified before the optimisation phase and not passed to the ILP-based optimisations. After the optimisation phase the necessary operations are inserted by user-supplied functions. An alternative solution is to except callee-saved registers whose value is not saved in the input code from the register renaming process and keep the original save and restore operations. While this is not part of the current implementation the necessary extensions are straightforward.

Finally, the function *AdaptUses* performs a last pass over the control flow graph to rename all uses to the corresponding virtual registers. Since the information about reaching definitions is available when the algorithm is started (see Sec. 9.1), the worst case time complexity of the algorithm is $\mathcal{O}(n^2)$ where n is the number of operations in the input program.

7.2.2. Virtual Registers and Abstract Resources

In the SILP formulation, the problem of register assignment is modelled as a register flow problem. In the register flow graph the node of a variable definition is connected to the resource nodes that represent the abstract register files the value can be stored in. The programmer is responsible for the mapping of physical register files to abstract register files. This mapping is propagated to the register renaming phase where the references to physical registers of the input routine are replaced by references to virtual registers. The register flow graph is constructed after the register renaming phase. Thus for each virtual register the set of abstract register files it can be mapped to has to be known. The register renaming algorithm has to ensure that this information is propagated to the optimisation phases. In the following, the relationship between physical, virtual and abstract register files is detailed in a mathematical formulation.

Let $\mathcal{R}$ be the set of all physical register files subject to renaming, let $\mathcal{V}$ be the set of all virtual register files, and let $\mathcal{A}$ be the set of all abstract register files. Then the following functions are derived from the TDL-description of the target processor:

- The assignment of abstract resources to physical register files is described by the function $f_A : \mathcal{R} \longrightarrow \mathcal{P}(\mathcal{A})$. Each operation that defines a physical register is associated with a set of abstract register files that represent the storage locations that are available for the destination operand.

- The mapping of physical register files to virtual register files is expressed by the function $f_V : \mathcal{R} \longrightarrow \mathcal{V}$. For each physical register file subject to renaming exactly one virtual register file is allocated, such that f_V is a bijective function.

- Each register file can be considered as an ordered set of element registers; each register is contained in exactly one register file. The assignment of virtual registers to abstract resources is described by the function $f_I : \mathcal{V} \longrightarrow \mathcal{P}(\mathcal{A})$ where $f_I(V) = f_A(f_V^{-1}(V))$. For each $r \in f_V^{-1}(V) = R$ a virtual register $v \in V = f_V(R)$ is allocated during register renaming.

The virtual register files are assumed to have infinite capacity. In the solution of the integer linear program to be generated, each definition of a virtual register $v \in V$ is mapped to exactly one abstract register file $A \in f_I(V)$ and is finally associated with a physical register $r \in R$ where $A \in f_A(R)$. As a summary, the interaction of those functions can be described in the following diagram:

$$\begin{tikzcd}
\mathcal{R} \arrow[rr, "f_A"'] \arrow[dr, "f_V"'] & & \mathcal{P}(\mathcal{A}) \\
& \mathcal{V} \arrow[ur, "f_I"'] &
\end{tikzcd}$$

In order to illustrate these definitions, the resource functions f_A, f_V, and f_I of the Analog Devices ADSP-2106x SHARC will be shown. The ADSP-2106x SHARC has one general purpose register file that serves as a fixed-point and as a floating-point register file. The assembly representation of the general purpose registers depends on whether they are used as fixed or as floating point registers: fixed point registers are denoted rx, floating point registers are denoted fx. In consequence, both views have to be separately incorporated into the hardware description. For each of these views a separate virtual register file is created, V_i, resp. V_f, that is assumed to have infinite capacity in order to allow the removal of the false dependences of the input program. In the ILP modelling both views have to be mapped to the same set of abstract register files since they physically refer to the same register file. The general purpose register file is no homogeneous register file but consists of 4 groups that have to be distinguished during instruction scheduling. ALU and multiplier can only be executed in parallel if all operands are located in uniquely defined groups within the register file as shown in Fig. 7.7.

For each of these groups one dedicated abstract register file is introduced such that their different properties can be fully taken into account in the generated integer linear programs. During ILP optimisation each variable definition is assigned to one of the abstract resources representing the register groups. Let R_i denote the integer view of the general purpose register file, R_f the floating-point view, V_i the virtual register file generated for R_i and V_f the virtual register file generated for R_f. The abstract register files representing the four register groups are denoted A_a, A_b, A_c, A_d. Then the resource functions are defined as follows:

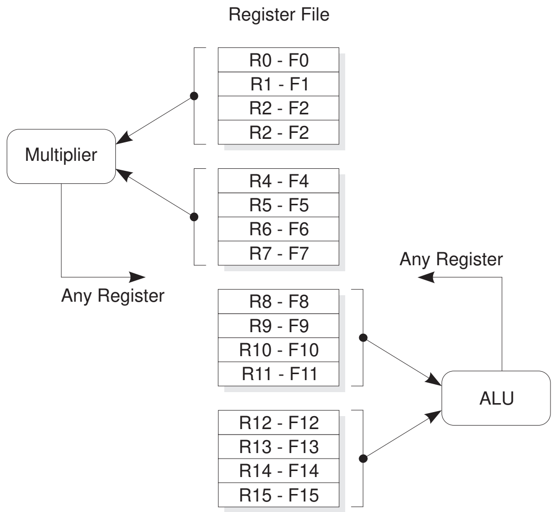


Figure 7.7.: Register groups of the ADSP-2106X SHARC.

$$\begin{array}{rcl} f_A : \{R_i, R_f\} & \longrightarrow & \mathcal{P}(\{A_a, A_b, A_c, A_d\}) \\ f_A(R_i) & = & \{A_a, A_b, A_c, A_d\} \\ f_A(R_f) & = & \{A_a, A_b, A_c, A_d\} \end{array}$$

$$\begin{array}{rcl} f_V : \{R_i, R_f\} & \longrightarrow & \{V_i, V_f\} \\ f_V(R_i) & = & V_i \\ f_V(R_f) & = & V_f \end{array}$$

$$\begin{array}{rcl} f_I = f_A \circ f_V^{-1} : \{V_i, V_f\} & \longrightarrow & \mathcal{P}(\{A_a, A_b, A_c, A_d\}) \\ f_V(V_i) & = & \{A_a, A_b, A_c, A_d\} \\ f_V(V_f) & = & \{A_a, A_b, A_c, A_d\} \end{array}$$

7.2.3. Virtual Definitions and Virtual Uses

For each superblock of the input routine an individual integer linear program is generated. Usually the input routine cannot be represented by a single superblock. Since computing the register assignment requires global liveness information, information about variable lifetimes that do not begin or end within the superblock currently being optimised have to be available.

Consider the situation of Fig. 7.8. There the life range of variable v_b begins with the definition in instruction d_v and ends with the use in instruction u_v. Thus variable v_b is alive across superblock s without any definitions or uses of v appearing inside of s. In the register flow graph as defined in Sec. 5.2.2, the modelling of variable lifetimes is based on the defining operations. Since no definition of the variable v_b is contained in superblock s, it is not represented in the generated register flow graph. In consequence, the number of variables simultaneously alive could be underestimated leading to an infeasible register assignment.

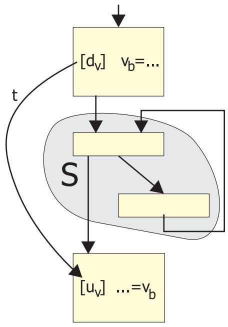


Figure 7.8.: Superblock-external life ranges.

In order to prevent this situation each basic block b inside a superblock s is associated with a set $V_D(b)$ of virtual definitions and a set $V_U(b)$ of virtual uses. All variables living at the exit of a predecessor block of b that is not contained in s are represented by a virtual definition. Let p_0^b denote the program point corresponding to the entry of block b and p_x^b be the program point immediately after the last operation inside of b.

$$V_D(b) = \{d_v \mid \exists (b', b) \in E_B : b' \notin N_s \wedge v \text{ alive at } p_x^{b'}\}$$

Similarly, if a variable is alive at the end of a block b but not at the entry of its successor block b' inside of s, it is represented by a virtual use.

$$V_U(b) = \{u_v \mid v \text{ alive at } p_x^b \wedge \nexists (b, b') \in E_B : b' \in N_s \wedge v \text{ alive at } p_0^{b'}\}$$

In the ILP formulation the virtual definitions are treated like normal definitions with the exception that their starting time is not subject to the ILP optimisation. Instead it is fixed to a time earlier than the starting time of all regular operations of the superblock. The virtual definitions ensure that the life ranges of externally defined variables are visible inside the superblock. The virtual uses ensure that externally used variables are alive along each path to an external use inside the superblock. The virtual definitions also play an important role in synchronising the solutions of the individual superblocks (see Chap.7.4). In this context, they contribute to preventing inconsistent register group assignments.

7.2.4. Global Lifetime Modelling

Extensions of the modelling presented so far are required if there is more than one definition in the same superblock that reaches the same use of a register variable. This situation can be caused by loops and by conditional statements. One problem is the representation of variable life ranges. Consider the situation of Fig. 7.9. If the loop is not executed the life range of the variable v defined by d_1 reaches the use in u_2 but if there is at least one loop iteration, it is the definition d_2 of v that reaches u_2. The life range constraints (5.16) however would prevent the reuse of the physical register assigned to d_1 if its life range would extend until u_2. Therefore for the representation of the life ranges we will assume that each loop is executed at least once and dedicated synchronisation constraints are generated that ensure a consistent modelling that is also correct if the loops are not traversed.

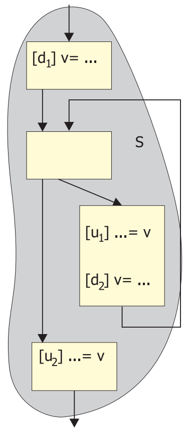


Figure 7.9.: Multiple reaching definitions.

Definition 7.3 (Concurrent Definitions) *Two operations i and j performing a write access to a register are called* concurrent definitions*, if both operations define the same destination register and there is a use of that register that is reached by both definitions.*

The operations d_1 and d_2 of Fig. 7.9 are examples of concurrent definitions; both define the virtual register v and reach the use in u_1. The first definition d_1 supplies the initial value of the variable while d_2 defines its value in all subsequent loop iterations. Then it must be ensured that both definitions d_1 and d_2 write to the same physical register.

If there is only one register in the corresponding register group, this can be done in a simple way: the assignment to the abstract resource is equivalent to an assignment to the physical register itself. Thus it is sufficient to enforce both operations to write to a register of the same abstract register file. This can always be achieved by introducing one dedicated resource for each individual register. When considering the example of the ADSP-2106x SHARC, instead of declaring four abstract resources A_a, A_b, A_c, A_d, it would be necessary to introduce 16 abstract resources $A_{r_0}, \ldots, A_{r_{15}}$. However then the number of binary variables required for modelling the register assignment problem would be 4 times larger and the number of constraints would increase. In consequence, the complexity of the formulation increases and one important advantage of the flow modelling, the unified modelling of equivalent resources, is given away.

In the remainder of this section, the *resource path constraints* are presented that allow an exact modelling of concurrent definitions. Since the resource path constraints can significantly increase the complexity of the overall ILP formulation, we have developed two mechanisms to reduce the complexity. In the first one, the superblock construction is stopped at loop boundaries; then an exact modelling of concurrent definitions is possible without significant increase of complexity. In the second one, the user can restrict the maximal length of the resource paths to be considered. This way, the additional complexity is reduced. The experimental results show that usually a low bound of the path length can be chosen without preventing the calculation of an optimal solution. Thus only a moderate increase in complexity is required.

Note that no explicit modelling is required if the definitions are located in different superblocks. Since different superblocks are optimised one after another the abstract resource assignment calculated for the superblock that is optimised first is mandatory for the second superblock (see Sec. 7.4). Only if the concurrent definitions are contained in the same superblock, explicit constraints are required in the integer linear program to force them to write to the same register. The reason of this is that in the flow-based SILP formulation the sharing of individual physical registers is a dynamic property: An operation j writes its result to the same register of an abstract register file r as a preceding operation i if and only if there is an *active path* from i to j in the register flow graph, i. e.

$$i = k_1 \to \cdots \to k_m = j, \text{ where } x^r_{k_l,k_{l+1}} = 1 \quad \forall\, 1 \le l \le m-1.$$

In general, it cannot be decided statically which path in the register flow graph from i to j will be chosen, if any. Therefore it is necessary to explicitly enforce that in each feasible solution of the integer linear program there is an active path from i to j in the register flow graph. This means that the problem of selecting one path from the set of all possible paths between i and j has to be incorporated into the generated integer linear programs. If the target architecture does not support predicated execution[1] ([PS91, DT93]), the worst case number of paths to be considered can be bounded by the following lemma.

Lemma 7.1 *If two operations i and j form a pair of definitions reaching the same use k inside a superblock s then the ordering of i and j can be statically determined.*

Proof: Let b_i denote the basic block containing i and b_j be the basic block containing j. If i or j is a virtual definition the proof is trivial since the starting time of virtual definitions is fixed to the start of the surrounding basic block. So in the following we assume that neither i nor j are virtual definitions. The proof can be done in three steps.

[1] In predicated code, control dependences have been converted to data dependences so that operations from originally disjoint control flow paths can be contained in the same basic block.

(i) First we will show that i and j must belong to different basic blocks, i. e. $b_i \neq b_j$.

Assume that $b_i = b_j$. Since both operations i and j define the same resource, they cannot be contained in the same instruction; otherwise the result of the instruction would not be well-defined. All instructions of a basic block are executed consecutively, so there must be a data dependence between i and j. In consequence their relative ordering inside the basic block is fixed. But this means that one of the operations will precede the other one on every path to the use in operation k, so they cannot both reach k.

(ii) The basic blocks b_i and b_j are not control equivalent.

Assume that b_i and b_j are control equivalent. Then every program path through b_i will also traverse b_j. But then again there must be a data dependence between i and j; again one of the operations will precede the other one on every path to k, so they cannot both reach the use in k.

(iii) In consequence, i and j must belong to basic blocks that are not control equivalent. Yet, since a superblock does not contain mutually exclusive control flow paths there must be an edge $(b_i, b_j) \in E_B^+$. But then there must also be a path from i to j in the control flow graph. Since i and j both define the same resource, j is transitively dependent on i. Furthermore we know that this dependence is not classified as a loop-carried dependence since $i \overset{*}{\underset{E_D}{\rightarrow}} j$ and $(b_i, b_j) \in E_B^+$. Then, by construction, an edge from i to j is introduced in the register flow graph and there is no path from j to i. In consequence, the ordering of i and j can be statically determined.

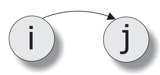
□

Using this lemma we can derive an upper bound of the number of alternative paths to be considered between i and j.

Lemma 7.2 *Let i, j be a pair of concurrent definitions reaching a given use k and let n be the number of register flow nodes which are transitive successors of i and transitive predecessors of j different from i and j in the register flow graph G_Z, $n = \left|\{m \in N_Z \mid i \overset{+}{\underset{G_Z}{\longrightarrow}} m \overset{+}{\underset{G_Z}{\longrightarrow}} j \wedge m \neq i \wedge m \neq j\}\right|$. Then the number of paths from i to j in the register flow graph is bounded by $B_{RFG}(n) = \sum_{k=0}^{n} \frac{n!}{(n-k)!}$.*

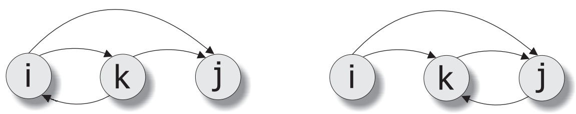
Proof: Induction over the maximal number n of nodes that are successors of i and predecessors of j in G_Z.

$n = 0$: j is the only immediate successor of i in the register flow graph.



$$B_{RFG}(0) = \frac{0!}{0!} = 1$$

$n = 1:$ There is one possible intermediate node.



$$B_{RFG}(1) = \frac{1!}{1!} + \frac{1!}{0!} = 1 + 1 = 2$$

$n \to n+1:$ Assume that the number of intermediate paths for n nodes is $B_{RFG}(n) = \sum_{k=0}^{n} \frac{n!}{(n-k)!}$. Now we are considering an additional intermediate node and have to prove that $B_{RFG}(n+1) = \sum_{k=0}^{n+1} \frac{(n+1)!}{(n+1-k)!}$.

Consider the set of additional paths that have to be considered due to the insertion of the additional intermediate node i_{n+1}. This is exactly the set of paths containing the newly added node. In order to construct a path of length k which contains i_{n+1} we select $k-1$ nodes from the previous node set and consider all $(k-1)$-permutations of those n elements. Since the number of those permutations is $\frac{n!}{(n-k+1)!}$ and there are k positions where we can insert node i_{n+1}, the number of newly added paths of length k is $k \cdot \frac{n!}{(n-k+1)!}$.

Thus we get

$$\begin{aligned}
B_{RFG}(n+1) &= B_{RFG}(n) + \sum_{k=0}^{n+1} k \cdot \frac{n!}{(n-k+1)!} \\
&\overset{\text{ass}}{=} \sum_{k=0}^{n} \frac{n!}{(n-k)!} + \sum_{k=0}^{n+1} \frac{k \cdot n!}{(n-k+1)!} \\
&= \sum_{k=0}^{n} \frac{(n-k+1) \cdot n! + k \cdot n!}{(n-k+1)!} + \frac{(n+1) \cdot n!}{(n+1-(n+1))!} \\
&= \sum_{k=0}^{n+1} \frac{(n+1)!}{(n+1-k)!}
\end{aligned}$$

$\square$

Since $\sum_{k=0}^{n} \frac{n!}{(n-k)!} = \sum_{k=0}^{n} k! \cdot \binom{n}{k} \geq \sum_{k=0}^{n} \binom{n}{k} = 2^n$ there are $\Omega(2^n)$ possible paths. However while this is a feasible upper bound, it is nevertheless very pessimistic. It can only be reached if all intermediate nodes on the path from i to j are independent of one another since any ordering among those nodes is considered. In real-life input programs there will be dependences among intermediate operations

Program	D	n	P
waveletk	1	2	2
fir	4	1	4
dft	5	2	8
waveleti	4	2	12
dmatrix1	1	5	20
dmatrix2	1	10	1728
cascade	4	5	1967
histo	5	8	4636
dfir2dim	2	14	20000

Table 7.1.: Statistics on Path Constraints for the ADSP-2106x SHARC.

and life ranges that can be statically determined to overlap, so that the number of path alternatives will be significantly smaller. If we assume that there are dependences between all potential intermediate nodes, each set of k nodes has to be counted exactly once since there is only one possible ordering of them. In that case we get $B_{RFG}(n) = \sum_{k=0}^{n} \binom{n}{k} = 2^n$, such that the complexity is bounded by $\mathcal{O}(2^n)$. While the complexity seems prohibitive, usually the number of possible intermediate nodes is relatively small since these nodes represent operations that can be scheduled between i and j without any overlapping of the corresponding life ranges. Among a total of 23 input programs evaluated for the ADSP-2106x SHARC processor, path constraints have only been required for 9 programs. In order to give an impression of the average case complexity, Tab. 7.1 shows the number D of pairs of concurrent definitions, the maximal path length n, and the total number P of alternative paths for those input programs.

For each pair of concurrent definitions a set of ILP constraints has to be generated that force exactly one path from i to j to be taken. Similarly to the modelling of the control flow structure (see Sec. 5.4) a set of disjunctive constraints is generated among which exactly one constraint has to be satisfied. Let k paths between two operations i and j be given and let P be the set of those paths, $|P| = k$. Furthermore, let d be a unique number identifying the operation pair (i, j) and let $R = \{r \mid (i, r) \in E_A \wedge (j, r) \in E_A\}$. Then for each path $p_m \in P$, $1 \leq m \leq k$ of length l (where the length is defined as the number of edges on this path) the following constraint is generated:

$$\sum_{r \in R} x^r_{k_1 k_2} + \sum_{r \in R} x^r_{k_2 k_3} + \cdots + \sum_{r \in R} x_{k_{l-1} k_l}{}^r + l c^d_m \geq l \qquad (7.1)$$

The c-variables are binary integers, $c^d_m \in \{0, 1\}$, which are used in the following constraint to force exactly one of those paths to be taken:

$$\sum_{m=1}^{k} c^d_m = k - 1 \qquad (7.2)$$

The flow conservation and assignment constraints ensure that the same abstract register file is used consistently along each active path. This way, a correct modelling is ensured.

The intermediate nodes which can appear on a path from i to j in the register flow graph correspond to the nodes of variable definitions that can be scheduled between i and j without any intersection of the corresponding life ranges. Although this number will usually be small compared to the total number of variable definitions in a program an exact modelling cannot always be guaranteed. Calculating all feasible paths is a problem of exponential time complexity that has to be performed at the generation time of the integer linear programs. Generating the path disjunctions can lead to an exponential space consumption and finally, since for each alternative path a dedicated binary variable has to be introduced the computation time required for solving the generated integer linear programs can increase exponentially, too. Therefore it is necessary to develop methods that allow computing a solution even if no exact modelling is possible.

Complexity Reduction by Loop Barriers

The first approach to reduce complexity is straightforward: if, similar to [Fis81, HMC$^+$93] the superblocks are not allowed to be extended across loop boundaries, the modelling is simplified. Since each superblock contains at most one loop, the length of the paths to be considered for path constraints is reduced. Since the most frequently executed program parts are optimised first, negative effects can occur, but are restricted to less important program parts. This situation is illustrated in Fig. 7.10; if there are no redefinitions of v_2 and v_3 path constraints have only to be generated for the virtual definition representing d_{v_1} and i_k.

Complexity Reduction by Bounding Path Lengths

The second approach does not prevent the extension of superblocks across loop boundaries. Instead it takes a user-specified upper bound on the length of any path in the register flow graph between concurrent definitions. The resource path constraints force two operations to write into the same register by formulating a disjunction over all feasible paths between the two operations in the register flow graph. When specifying an upper bound on the path length, only the shortest paths are considered in the resource path constraints; in consequence, the resource path constraints discard all paths exceeding the given bound from the solution space. With increased path length, the register reuse increases that limits the available parallelism. So by specifying an upper bound on the path length, only the paths leading to the highest degree of instruction level parallelism are considered. However since this can prevent a feasible solution from being found, this parameter has to be chosen carefully. In our experimental analyses for the ADSP-2106X SHARC a maximal path length of 3 has always been sufficient for an optimal solution to be found. Thus the required increase in complexity is only moderate and the

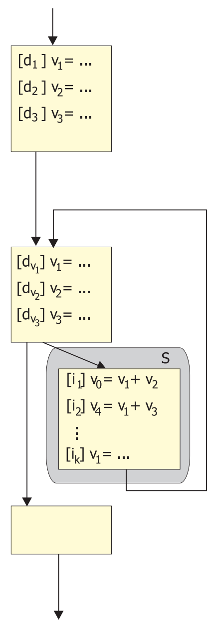


Figure 7.10.: Concurrent definitions in restricted superblocks.

path constraint modelling can be considered as practicable. The total number of path constraints with an upper bound of 3 for the programs of Tab. 7.1 is shown in Tab. 7.2.

7.3. Global Timing Constraints

Since superblocks can comprise complete loop nests, loop-carried data dependences and loop-carried timing constraints have to be respected during optimisation. The incorporation of such cyclic scheduling constraints is the subject of this section.

7.3.1. Inter-Iteration Data Dependences

Consider the situation of Fig.7.11. The operation d_l defines an operand of that instance of operation u_l that is executed during the next loop iteration. It has to be ensured that in any loop iteration all operands used by u_l are available when the execution of u_l is started. This can only be done by incorporating loop-carried data dependences into the ILP model.

Let s be the superblock currently being optimised. For each loop-carried data dependence (i, j, r, t) where i is contained in basic block $b_i \in N_s$ and j is contained in $b_j \in N_s$, the set P of all acyclic paths from b_i to b_j is calculated. For each path $p \in P$, a lower bound l_p of the required execution time is determined. The bound l_p can be determined to have a positive value, if p traverses basic blocks that have already been scheduled due to preceding superblock optimisations. The path p^{min} with the smallest lower bound l_p^{min} is selected. If that lower bound is greater than

Program	D	P
waveletk	1	2
fir	4	4
dft	5	8
waveleti	4	12
dmatrix1	1	12
dmatrix2	1	76
cascade	4	203
histo	5	76
dfir2dim	2	165

Table 7.2.: Statistics on Path Constraints for the Adsp-2106x Sharc with maximal path length of 3.

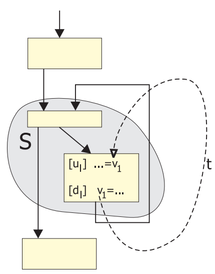


Figure 7.11.: Loop-carried dependences.

the execution time of i, i. e. $l_p^{min} > w_i$, the destination operand of i is definitively available when operation j is reached in the next loop iteration and no additional constraints are required. Otherwise, two basic blocks b_e and b_x are selected from p. The block b_e is the first block of p that is a predecessor of b_j in G_B^+ and that is contained in superblock s or b_j itself if there is no predecessor in s. Similarly, b_x is the last block on p that is a successor of b_i in G_B^+ and that is contained in s or b_i itself if there is no successor in s. Let t_A^e denote the starting time of the first operation in block b_e and t_E^x the starting time of the last operation in block b_x. Then the following precedence constraint is generated:

$$t_j - t_A^e + t_E^x - t_i \geq w_i - l_p^{min} \tag{7.3}$$

Intuitively, this constraint splits the path from i to j in two parts: the path from i to the end of the loop, and the path from the beginning of the loop to operation j. The sum of the execution times of both parts must be large enough for the execution of i to finish when j is reached.

7.3.2. Inter-Iteration Latency Constraints

Loop-carried timing constraints are required to prevent latency violations in subsequent loop iterations. Consider the situation of Fig. 7.12. Assume that there is only one instance of resource r used by two operations i and j; then j must be scheduled in a way that the time between the start of i and the start of j in the next loop iteration is at least the latency of r.

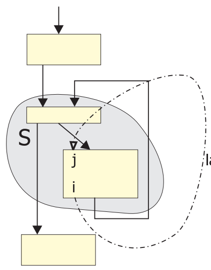


Figure 7.12.: Inter-iteration latency constraints.

In the general case this is a flow problem that is closely related to the path synchronisation problem of Sec. 7.2.4. The minimal distance has to be enforced only when the same instance of a functional unit is used by both operations. Thus, in the SILP formulation additional constraints are required that become active only if there is an active path from i to j in the resource flow graph. The complexity of the modelling is comparable to Sec. 7.2.4. Since the concept of resource flows is not represented in the OASIC formulation a mechanism similar to the register crossing constraints (equation 5.40) is required for an exact modelling.

Inter-iteration latency constraints only have to be generated for functional unit types that have a latency of several machine cycles and keeping track of the resource flow is only necessary for functional units of which several instances exist. In contemporary hardware architectures, there is a trend to fully pipelined functional units with single-cycle latency. When functional units with multi-cycle latency are employed, it can be expected that there is only one instance of them. Therefore we have implemented a simplified modelling that is analogous to the modelling of loop-carried data dependences and that is identical in the SILP and the OASIC formulation. For each resource that has a multiple-cycle latency, the set L of all operations that may be the last to be executed by any instance of r during a single loop iteration is calculated. Similarly all operations that may be the first to be executed by any instance of r during a single loop iteration are collected in a set F. Similarly to the previous section, for each pair $(i, j) \in L \times F$, the set P of all acyclic paths from i to j is calculated. For each path $p \in P$, a lower bound l_p of the required execution time is determined and the path p_{min} with the smallest lower bound l_p^{min} is selected. If the latency L_r of the resource r is greater than l_p^{min}, two blocks b_e and b_x are determined where b_e is the first block of p that is a predecessor of b_j in G_B^+ and that is contained in superblock s or b_j if there is no predecessor in s. The block b_x is the last block on p that is a successor of b_i in G_B^+ and that is contained in s, or b_i if there is no successor in s. Then the following constraint is generated:

$$t_j - t_A^e + t_E^x - t_i \geq L_r - l_p^{min} \tag{7.4}$$

This is a pessimistic modelling since it may enforce a minimal distance between two operations that are executed independently by different instances of the same functional unit. As long as there is only one instance of each multi-cycle latency resource, the modelling is exact. For the target architectures modelled so far, this assumption holds so that no feasible solution is excluded from the solution space.

7.4. Superblock Synchronisation

The control flow graph of the input program is partitioned into a set of superblocks that are optimised one after another. In order to obtain a feasible global solution, it is necessary to synchronise the optimisation results of the individual superblocks. In this context, timing synchronisation, lifetime synchronisation and resource synchronisation can be distinguished. After the schedules of all superblocks have been computed, they are composed to form the final result schedule. Timing synchronisation is required in order to ensure that the latencies of the functional units are also respected in the composed schedule. Lifetime synchronisation is required to guarantee that the minimal distance between definitions and all corresponding uses is respected in the composed schedule. Additionally, resource synchronisation is required since the allocation decisions made during the optimisation of one superblock have to be respected during subsequent superblock optimisations.

7.4.1. Timing Synchronisation

Consider the situation of Fig. 7.13. Let s' be a superblock that has already been scheduled and let s be the superblock currently being optimised. If there is an operation i in block $b' \in N_{s'}$ that is executed by a functional unit r with a multi-cycle latency L_r it must be ensured that the distance to operations from superblock s that use the same resource is larger than L_r. This must be taken into account when calculating the schedule of s.

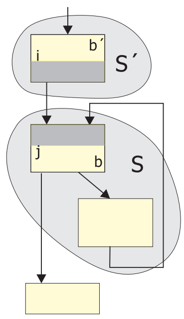


Figure 7.13.: Timing synchronisation.

For each basic block $b \in N_s$, the sets $Pred_b$ and $Succ_b$ of all immediate predecessor and successor blocks in the control flow graph that have already been scheduled is determined. If there is a basic block $b \in N_s$ where $Pred_b \neq \emptyset$, the set of all acyclic paths from the program entry point through one of the blocks in $Pred_b$ to b is determined. For each instance r_k of r, all previously scheduled operations that have been assigned to r_k and that may be the last to be executed by r_k before the program flow reaches block b are collected. Based on the execution time that has been determined for the previously scheduled blocks, a lower bound of the minimal distance of those operations to b is computed. If this lower l_b bound is larger than the latency of r, we know that the latency will always be respected and no dedicated constraints are required. Otherwise, a virtual operation is inserted into b whose starting time is fixed to the starting time of b and that is associated with a latency of $L_r - l_b$. In the SILP formulation the virtual operation is taken into account when generating the register flow graph and causes additional serial constraints to be generated. In the OASIC formulation, the virtual operation leads to the generation of additional resource constraints. This way, an exact modelling of the inter-superblock latency dependences is achieved. Analogous constraints are generated for the control flow successors s_b of b.

As an alternative to the exact modelling we have developed a simplified approach to address the timing synchronisation. All virtual operations that have been inserted into a block b in order to provide a synchronisation with the blocks in $Pred_b$ with respect to a resource r are traversed and the maximal remaining latency l_r^{max} among them is determined. The virtual operations are not considered when generating the serial constraints respectively the assignment constraints. Instead for each operation i of s that can be executed by r the following constraint is generated:

$$t_i - t_b^A \geq l_r^{max} \quad \forall (i, r) \in E_R \tag{7.5}$$

Analogous constraints are generated for the control flow successors $succ_b$ of b. The effect of these constraints is that no operation that can be executed by r may be scheduled at a control step where any instance of r may still be busy due to the execution of an operation from a previously scheduled superblock. If there is only one instance of r, the approach is equivalent to the proceeding described above. If there are multiple instances of r, none of those instances can be used in the first l_r control steps of block b such that feasible solutions may be discarded from the solution space. This simplification is motivated by the fact that there is a tendency to fully pipelined functional units with single-cycle latency. If there are functional units with multi-cycle latency, it can be expected that there is only one instance of them. In the current implementation (see Chap. 9) the simplified method is applied. For the target architectures modelled so far, the assumption holds, so that no feasible solution is discarded from the solution space and the modelling is exact.

7.4.2. Lifetime Synchronisation

Let s be the superblock currently being optimised. Then, dedicated lifetime synchronisation constraints are required for operations that use a value defined by an operation of a previously scheduled superblock; an illustration is given in Fig. 7.14.

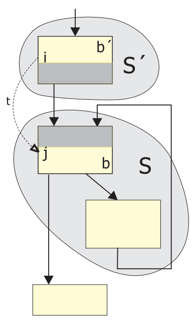


Figure 7.14.: Lifetime synchronisation.

As described in Sec. 7.2.3 the external definitions are represented by virtual definitions inside of s. Let an operation j of a block $b \in N_s$ be given such that there is a data dependence (i, j, r, t) where i is an operation from a previously scheduled block $b' \in N_{s'}$. Then the set of all acyclic paths from b_i to b_j is determined. Based on the execution time that has been determined for the previously scheduled blocks, a lower bound of the minimal distance between i and j is computed. If this lower bound l_{ij} is larger than the execution time of i, no additional constraints are required. Otherwise, a dedicated precedence constraint between the virtual definitions introduced for i and j is generated:

$$t_j - t_i \geq l_{ij} \tag{7.6}$$

Similarly, the definitions from basic blocks of s that have previously scheduled external uses must be accounted for. For those operations, dedicated precedence constraints to the virtual uses at the end of the appropriate exit block of s are inserted.

7.4.3. Resource Synchronisation

The decisions made in one superblock can have effects on the subsequently optimised superblocks. Those effects concern not only the timing of the operation execution but also the assignment of abstract resources to virtual registers, if instruction scheduling and register assignment are addressed jointly. Since this integration is not addressed for the OASIC formulation, we will only refer to the SILP-based formulation in the remainder of this section. If during the optimisation of a superblock s_1 a virtual register v_1 has been mapped to an element register of an abstract register resource g_1, this mapping has to be respected in all subsequently optimised superblocks where v_1 is alive. In all those superblocks s_k a virtual definition $d_{v_1}^k$ of v_1 has been introduced. In order to prevent $d_{v_1}^k$ to be assigned to another abstract register resource than g_1 during the optimisation of s_k the following constraint is added to the ILP representation of s_k:

$$\Phi_{d_{v_1}^k}^{g_1} = 1 \tag{7.7}$$

If the constraint was omitted, $d_{v_1}^k$ could be mapped to any other register resource during the optimisation of s_k so that there would not exist a feasible global schedule. The resource path constraint mechanism of Sec. 7.2.4 guarantees that the concurrent definitions share the same physical register. Thus the previously determined assignments to abstract resources are incorporated into the integer linear program of the current superblock so that inconsistencies can be prevented.

Unfortunately there are situations where the constraints (7.7) lead to the generation of infeasible integer linear programs. This can happen, if in different superblocks some independent decisions are made that are locally correct but whose common effect on superblocks to be optimised subsequently leads to contradictions. To give an example, consider the situation of Fig. 7.15.

Let a set G of abstract register resources be given and assume that $g_1 \in G$ represents a register group with two element registers, $|g_1| = 2$. Further assume that the superblocks are optimised in the order s_1, s_2, s_3. Let the virtual registers v_1 and v_3 be not alive in superblock s_1, and v_4 be not alive in s_2. The virtual register v_4 will be assigned to an abstract register resource during the optimisation of s_1; assume that the register group g_1 is chosen. Similarly, let the virtual registers v_1 and v_3 also be mapped to register group g_1 during the optimisation of superblock s_2. All these assignments are feasible, since no life ranges are violated: v_4 only lives in s_1, and v_1, v_3 only in s_2. However at the entry of superblock s_3 all three virtual registers are alive and the number of available registers of register group g_1 is exceeded. So while no conflicts arise during the isolated optimisation of s_1 and

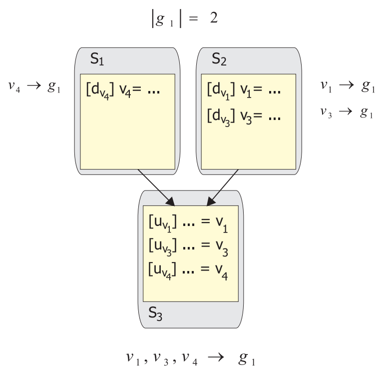


Figure 7.15.: Resource synchronisation.

s_2, the aggregated effects of the scheduling and allocation decisions of s_1 and s_2 prevent a feasible solution to be calculated for superblock s_3.

Before presenting our approach to deal with this problem, we will analyse the conditions for this situation to occur. Evidently there must be a superblock that combines data flow information of other superblocks, i. e. the error situation can only occur in superblocks that contain a join node of the control flow graph. Furthermore the superblocks of the control flow paths adjacent to the join block must be optimised before the superblock of the join block. Therefore the situation shown in Fig. 7.16 is not critical, since the superblocks are always optimised in the order of decreasing execution frequency. In sequential code, the execution count of a

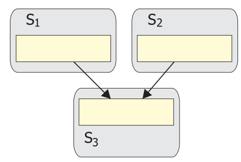


Figure 7.16.: Non-critical situation.

join block is always higher than the execution frequencies of the incident, mutually exclusive blocks. So another necessary condition is that the execution frequency of the incident superblocks must be higher than that of the superblock containing the join block. This is the case in the situation of Fig. 7.17. But also this situation is not critical, since inconsistent resource assignments can be prevented by simply forbidding this situation during superblock construction: If there are two superblocks s_1 and s_2 representing mutually exclusive control flow paths that are both incident to a join block b_j, this join block b_j is forced to be added either to s_1 or to s_2. The only critical situation occurs when there are more than two incoming edges to a join node in the control flow graph, e. g. due to switch statements (see Fig. 7.18).

It can be assumed that the probability for this situation to occur in real input programs is very low. Nevertheless it is necessary to develop a method that can deal

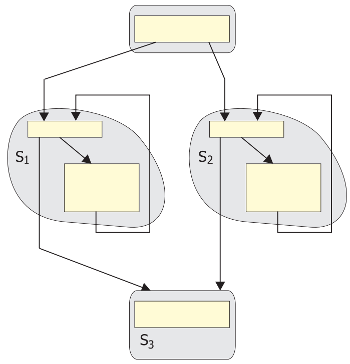


Figure 7.17.: Non-critical situation.

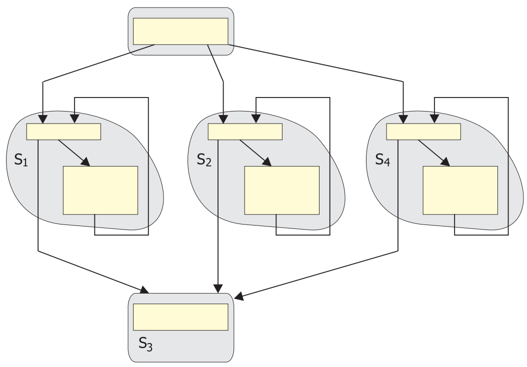


Figure 7.18.: Critical situation.

with inconsistent resource assignments and allows to calculate a feasible solution. The key idea is not to completely forbid inconsistent resource assignments, but to prevent them whenever possible. In consequence the objective function is not scalar any more; instead we have to deal with a vectorial objective function. This extension is also necessary since the separated optimisation of resource and register flows by the ILP-based approximations of the SILP model can increase the number of overlapping life ranges, as mentioned in Chap. 6. Although in our experimental evaluation this situation did never occur, it has to be properly modelled. Let a vectorial decision problem be given:

$$max\{z(x)|x \in X\} \tag{7.8}$$

$$z(x) = \begin{pmatrix} z_1(x) \\ \dots \\ z_K(x) \end{pmatrix}$$

The component functions $z_1, \dots z_K$ are independent, possibly even contradicting goals that are simultaneously pursued. In order to solve vectorial decision problems, several *compromise models* have been developed [DK96]. A well-known model is the *goal weighting model*; there the objective function of the compromise model is composed from the weighted sum of the individual objective functions.

$$max\{ \quad \Psi(z(x)) \quad |x \in X\} \tag{7.9}$$

$$\begin{array}{ccc} \Psi & : & \mathbb{R}^K \to \mathbb{R} \\ z(x) & \longmapsto & \Psi(z(x)) = \sum_{k=1}^{K} \sigma_k z_k(x) \end{array}$$

where $\sigma_k > 0 \quad \forall\ k = 1, \dots, K$. In order to prevent violations of register resource bounds, multiple goals have to be formulated: the required execution time is to be minimised and the number of instances should not be exceeded for any abstract resource. When using the compromise model, this can be considered as declaring a penalty for exceeding register resource restrictions. The new objective function is defined as follows:

$$min\ \Psi \begin{pmatrix} S \\ s_{g_1} \\ \vdots \\ s_{g_K} \end{pmatrix} = S + \sum_{l=1}^{K} B \cdot s_{g_l} \tag{7.10}$$

For each abstract register resource $g \in N_R^A = \{g_1, \dots, g_K\}$, a dedicated integer variable $s_g \in \mathbb{N}$ is introduced that indicates the amount by which the capacity of register group g is exceeded. The register resource constraints (equation 5.21) are reformulated:

$$\sum_{(g,j,g) \in E_Z} x_{gj}^g \leq R_g + s_g \qquad \forall g \in N_R^A \tag{7.11}$$

The value B is a large constant that must be an upper bound of the execution time of the complete input program. This way, overflows of register groups can be detected without infeasibility of the generated integer linear program. However the question remains how a feasible global register assignment can be found when an overflow has occurred. This is the subject of the remainder of this section.

Repairing Overflows of Abstract Register Files

A straightforward approach of dealing with resource overflows is to introduce additional register-to-register moves such that diverging register file assignments in different superblocks can be arranged to fit together. Since the most frequent superblocks are optimised first, the additional operations will only be inserted into less important program parts. Nevertheless this is not a satisfactory solution; if possible the insertion of additional operations should be avoided. In order to achieve this goal, we have developed two heuristic methods, *collision-based repairing* and *exclusion-based repairing*. First the collision-based approach is employed; if that does not lead to a feasible solution without resource overflows, exclusion-based repairing is attempted. If still no feasible solution without resource overflows can be determined within the given time frame, the insertion of register moves takes place as a fallback solution. Both heuristic methods lead to reoptimisations of the complete program, so that the increase in calculation time can be significant. Therefore the number of reoptimisations induced by those methods can be restricted by a command line parameter in our implementation. In the following the two repairing heuristics are presented and then the insertion of register-move operations is outlined.

Definition 7.4 (Register Flow Chain) *Let $x \in \mathbb{Z}^M$ denote the solution of the integer linear problem generated for a superblock s in the SILP formulation. Let G_Z be the register flow graph generated for s and let N_R^A be the set of abstract register files. Then a* register flow chain *c of an abstract register file $g \in N_R^A$ is defined as a path $i_0 \to \cdots \to i_K$ in G_Z such that $i_0, \ldots, i_K$ represent operations and $x^g_{i,i+1} = 1 \quad \forall\, i \in 0, \ldots, K$. The value K, i. e. the number of edges on this path, is called the length of c, $l(c) = K$.*

Collision-Based Repairing. If the number of available element registers of an abstract register file is exceeded in the solution of an ILP there must be previous assignments of virtual registers to abstract resources that affect the optimisation of the current superblock. Let g be an abstract register file with k element registers ($|g| = k$) whose capacity is exceeded in the solution of the ILP generated for a superblock s. The collision-based algorithm searches for two colliding definitions (d_1, d_2) of virtual registers that both have been assigned to g during preceding superblock optimisations and that appear in different register flow chains of g.

This situation is illustrated in Fig. 7.19. Belonging to different flow chains of the same abstract register file means that the operations write their result to

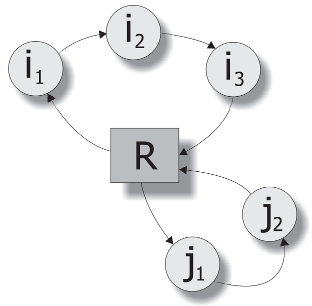


Figure 7.19.: Resource flow chains.

different registers of the same abstract register file. The basic idea of the collision-based repairing is that if d_1 and d_2 had been prevented from both being fixed to the same abstract resource, there would have been a better chance of finding a global solution without resource overflows. The algorithm traverses the register flow chains of the current solution that are associated with g in the order of increasing length. All operations in the same register flow chain use the same destination register. Therefore the smaller the length of an individual chain is, the better is the chance of getting a feasible solution when some of its operations are transferred to other abstract register files. All operations associated to g are analysed and a pair (d_1, d_2) of colliding definitions is determined where the lengths of the flow chains containing d_1 and d_2 are minimal.

If a suitable pair of colliding definitions has been found, a new constraint is generated that prohibits d_1 and d_2 from being assigned to the same abstract register file in any feasible solution

$$\Phi^g_{d_1} + \Phi^g_{d_2} \leq 1 \tag{7.12}$$

This constraint is registered in all superblocks where the virtual registers defined by d_1 or d_2 are alive. Then all previously determined solution information except for the constraint (7.12) is discarded and the optimisation is restarted for all superblocks of the program. The probability of suffering a violation of abstract register constraints is reduced since it is excluded that during the optimisation of any superblock d_1 and d_2 are mapped to the same abstract register file.

In the worst case, this heuristic can lead to n^2 complete optimisation passes, where n is the number of superblock-crossing register lifetimes. The method terminates, if a feasible solution has been found, or if no suitable pair of external resource fixations can be found.

Exclusion-based Repairing. If there is no pair of external resource fixations satisfying the conditions described above, the algorithm for exclusion-based repairing is invoked. Again the flow chains of the abstract register file whose capacity has been exceeded are traversed in the order of increasing length. The algorithm searches for an operation whose destination register has been fixed to g by a preceding superblock optimisation, and that could also be assigned to other abstract register resources. If such an operation d is found, a constraint is generated that prevents d from being assigned to the abstract resource g and the traversal is ended.

$$\Phi_d^g = 0 \tag{7.13}$$

Then all previously determined solution information except from the constraints (7.12) and (7.13) is reset and the program optimisation is restarted. The effect of constraint (7.13) is that the operation d will be mapped to another register resource than g during the next program optimisation so that it cannot contribute to an overflow of g. The probability of getting an overflow of g is reduced. On the other hand the exclusion-based repairing can have feasible solutions being excluded from the solution space.

The worst case number of recomputations induced by this method is in the order of $\mathcal{O}(n)$ where n is the number of virtual registers alive in more than one superblock. The method terminates if a feasible solution has been found, the problem is infeasible, or the iteration limit is reached. If the method terminates without providing an overflow-free solution, the spill code insertion is done as fallback solution. Certainly more elaborate heuristics can be applied here; however this is subject of future research.

Inserting Register Moves. If the collision-based and the exclusion-based heuristics cannot determine a feasible overflow-free solution, the partial solution of the ILP generated for superblock s before invoking the collision-based repairing is restored and is used as the basis for inserting spill code. Each routine of the input program is associated with an individual virtual register table; each superblock of that routine is associated with a local view of the virtual register table. The assignment of virtual registers to abstract registers can differ in local and global views. For each differing assignment, additional operations have to be inserted into the superblocks whose local views differ from the global one. After the optimisation of each individual superblock, the virtual registers that previously had not been assigned to an abstract register file are collected and the assignment information of the local view of the virtual register file is copied to the global view.

In the following an algorithm for inserting additional register moves is proposed. An implicit assumption is that the number of available registers is always sufficient for transferring the values to the required locations. If no free registers are available as temporary storage locations, it could be necessary to spill some values to memory by inserting additional load/store operations.

The result of the algorithm for the situation of Fig. 7.15 is shown in Fig. 7.20. Let g_1 be the abstract resource whose instance number has been exceeded in the solution of the ILP generated for superblock s_3. The resource overflow is caused by the resource assignments determined by the previously optimised superblocks s_1 and s_2. The flow chains of the abstract register file g_1 are traversed in the order of increasing length and an operation d is determined whose destination register has been fixed to g_1 by a preceding superblock optimisation. Assume this destination

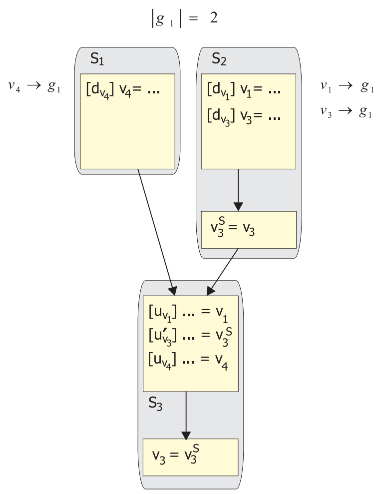


Figure 7.20.: Insertion of register moves.

register to be the virtual register v_3 in Fig. 7.20. Then a new virtual register v_3^s is introduced locally for superblock s_3. Along the edge from superblock s_2 to s_3 a new empty basic block is created and a register move $v_3^s = v_3$ is inserted. Similar operations have to be inserted along control flow edges leaving superblock s_3 where the value from v_3^s is transferred to v_3 again. All references to v_3 inside of superblock s_3 are changed to v_3^s. The fixation of v_3 to the abstract register file g_3 does not hold for the local copy v_3^s so that now an assignment to other abstract register files becomes feasible. The data dependence graph and the flow graphs are modified to incorporate the new operations and a new integer linear program for superblock s is generated. This step is repeated until a feasible solution has been found; the existence of a feasible solution is guaranteed due to the feasibility of the input program.

The additional operations that have been inserted can affect the overall solution quality. However since the most important superblocks are optimised first, the negative effects of the newly inserted operations will mainly concern the less important superblocks in the program. After the phase-coupled optimisation of all superblocks has been finished, an additional instruction scheduling phase is required that preserves the previously determined register assignment and searches an efficient global schedule taking into account the additional register moves. However in our experimental evaluation the insertion of additional data transfer operations has never been required.

7.5. Completing the Register Assignment

In the previous section it has been described how the assignment of abstract resources can be synchronised among the solutions of different superblocks. If all abstract register resources correspond to exactly one physical register the solution of the ILP also determines the final physical register assignment. For abstract resources that represent several physical registers with the same architectural properties, the physical register assignment is represented implicitly in the calculated register flow. The solution of the ILP determines which operations write into the same register, i. e. which operations are part of the same register flow chain, and which ones write to different registers. The decision to which physical register a given register flow chain is mapped still has to be taken. This is a reasonable approach since all elementary registers of one abstract register file share the same properties. In consequence, the exploitation of architectural irregularities does not require them to be distinguished. Due to interdependencies between allocation decisions of different superblocks however a greedy assignment of the register flow chains to physical registers is not sufficient and a more elaborate approach is required. This will be detailed in the following.

Consider the situation of Fig. 7.21 that visualises the result of an ILP solution. The operations i_1, i_2, i_3, j_1, j_2 have all been assigned to the same abstract resource g_1; i_1, i_2, i_3 share one register of g_1 and j_1, j_2 share another one. If there are two

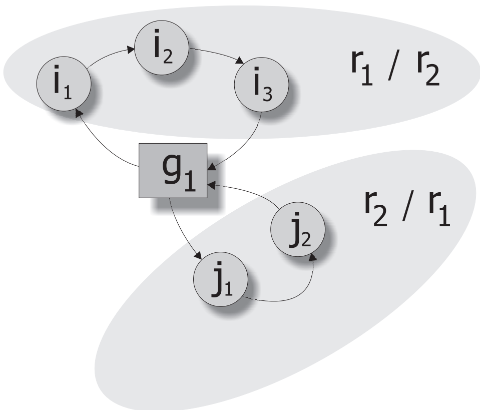


Figure 7.21.: Mapping of register flow chains.

element registers r_1 and r_2 of g_1 there are two possible physical register assignments: either i_1, i_2, i_3 are mapped to r_1 and j_1, j_2 to r_2 or i_1, i_2, i_3 are mapped to r_2 and j_1, j_2 to r_1. This detailed mapping is not part of the ILP formulation.

In order to determine the mapping of register flow chains to physical registers, an interference graph is built from the solution of the integer linear program generated for a given superblock s. Each node represents a register flow chain i. e. a sequence of definitions writing into the same physical register. Edges connect nodes that are associated with the same abstract register resource but have to be mapped to different physical registers.

Definition 7.5 (Interference Graph) *Let a superblock s be given and let D be the set of operations in s that perform write accesses to virtual registers. Let $C \subseteq \mathcal{P}(D)$ be a partition of D, i. e.$c_1 \cap c_2 = \emptyset \quad \forall\, c_1, c_2 \in C$, $\bigcup_{c \in C} c = D$. Finally let $f_c : C \to N_R^A$ be a function that assigns each $c \in C$ uniquely to one abstract* *register file $r \in N_R^A$. Then the interference graph $G_{\mathcal{I}}(s)$ of the superblock s is defined as $G_{\mathcal{I}}(s) = (N_{\mathcal{I}}, E_{\mathcal{I}})$ where each $c \in C$ is represented by a node $n_c \in N_{\mathcal{I}}$ and there is an edge $(c_1, c_2) \in E_{\mathcal{I}}$, iff $c_1 \neq c_2 \wedge f_c(c_1) = f_c(c_2)$.*

The partition C and the function f_C are derived from the solution of the ILP generated for superblock s that models the instruction scheduling and register assignment problem. C is the set of all register flow chains computed for s, i. e., for each $c \in C$ with $f_c(c) = r$ the following condition holds: if d_1, d_2 are elements of c there is a path in G_Z from d_1 to d_2 such that $d_1 = k_0 \rightarrow \cdots \rightarrow k_m = d_2$ and $x_{k_i,k_i+1} = 1 \quad \forall\, i = 0, \ldots, m-1$ in the solution of the integer linear program for superblock s. A set $c = \{d_1, \ldots, d_m\} \in C$ denotes a set of definitions sharing the same physical register. In the interference graph there is an edge between two nodes n_{c_1} and n_{c_2} if c_1 and c_2 are different register flow chains associated with the same abstract register file. Fig. 7.22 shows the interference graph for the example of Fig. 7.21.

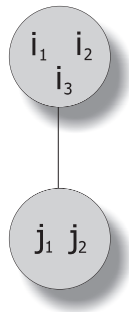


Figure 7.22.: Interference graph for Fig. 7.21.

In order to compute the physical register assignment we have developed two alternative algorithms. In the first approach, the *aggregated register mapping*, the interference graphs of all superblocks are merged after the ILP optimisation of all superblocks has been finished. Then a graph colouring algorithm is used to determine a feasible colouring, i. e., a feasible assignment of physical registers to register flow chains. No extension of the ILP formulations is required; however it cannot be guaranteed that the resulting graph is colourable if the input program consists of more than one superblock. Therefore it may be necessary to insert additional register moves in order to obtain a feasible colouring. In the second approach, the *incremental register mapping*, the individual interference graphs of each superblock are coloured separately after its ILP-based optimisation has been completed. Additional constraints are generated in subsequently optimised superblocks in order to take into account the previously determined register mappings. This way the resulting graphs will always be colourable.

Aggregated Register Mapping. After the individual interference graphs of all superblocks have been computed, they are unified to an aggregated interference graph $G_{\mathcal{I}}^{\mathcal{S}} = (\bigcup_{s\in\mathcal{S}} N_{\mathcal{I}}^{s}, \bigcup_{s\in\mathcal{S}} E_{\mathcal{I}}^{s})$. This graph consists of disjoint subgraphs for each superblock and for each abstract register resource. The number of nodes is bounded by $|N_{\mathcal{I}}^{\mathcal{S}}| = |\mathcal{S}| \cdot \sum_{g\in N_R^A} |g|$, the number of edges by $|E_{\mathcal{I}}^{\mathcal{S}}| = |\mathcal{S}| \cdot \sum_{g\in N_R^A}(|g|^2)$. Since it is possible for the same variable definition to be visible in different superblocks, it is necessary to merge some nodes of $G_{\mathcal{I}}^{\mathcal{S}}$. All nodes containing a definition of the same virtual register are merged and all edges adjacent to the original node are associated with the newly created node.

The synchronisation constraints of Sec. 7.4.3 only guarantee that the definitions are mapped to the same abstract resource. Thus two virtual definitions may be contained in the same register flow chain in the solution of one superblock but in different flow chains in the solution of another superblock. Merging the corresponding interference nodes will produce a node with an edge to itself in the interference graph, i. e., $(c_i, c_i) \in E_I^{\mathcal{S}}$. Since this prevents a feasible colouring, no merging operation is admitted that would lead to a zero-length cycle in the interference graph; the original nodes are kept, but are explicitly marked. During graph colouring, these nodes can be mapped to different physical registers. In order to maintain global consistency for these nodes, data transfer operations have to be inserted similarly to Sec. 7.4.3.

There are also other situations where merging operations can lead to an uncolourable graph. Consider Fig. 7.23 for an example. The graph is colourable with two colours, but it is not possible to merge the two nodes n_1 and n_2 and colour the resulting graph with two colours. In those cases again data transfer operations have to be inserted. Fortunately it can be expected that such cases will not often occur; in our experimental analysis the insertion of additional register moves has never been required. In the incremental register mapping algorithm described in the next paragraph the colourability of the interference graphs is guaranteed at the cost of higher calculation times of the individual integer linear programs.

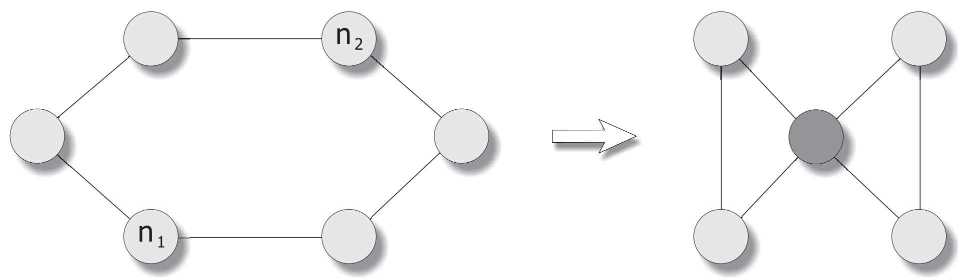


Figure 7.23.: Not colourable with 2 colours after merging n_1 and n_2.

Incremental Register Mapping. In the incremental approach there is no merging of interference nodes. After the solution of the ILP generated for one superblock has been determined, its individual interference graph is generated and a colouring, i. e. a mapping of register flow chains to physical registers is determined. In order to guarantee consistency, the results of previously computed physical register assignments have to be incorporated into the generated ILPs.

When generating the integer linear program for a superblock s, it is checked for each abstract register resource $r \in N_R^A$ whether there are two virtual definitions associated with r whose virtual destination register has already been assigned to a physical register. Let i and j be such a definition pair. If both operations have been mapped to the same physical register, resource path constraints are generated between i and j as described in Sec. 7.2.4. If they have been mapped to different physical registers, they must be prevented from being assigned to the same register flow chain when solving the ILP generated for s. Similarly to the resource path constraints, all paths between i and j must be considered. Let $i = k_1 \to \cdots \to k_m = j$ be a path from i to j in the register flow graph G_Z. Then at least one edge must be inactive, i. e., $\exists o \in \{1, \cdots, m-1\} : x^r_{o,o+1} = 0$. Let $l(p)$ denote the length of path p, then the following constraint is generated for each path p from i to j in G_Z:

$$\sum_{(i,j)\in p} x^g_{ij} < l(p) \qquad \forall\, g : (i,g) \in E_A \;\wedge (j,g) \in E_A \tag{7.14}$$

These constraints guarantee that the results of all superblock optimisations are consistent: if two virtual registers are mapped to the same physical register in the solution of one superblock, this mapping is respected in the solution of all subsequently optimised superblocks. As a consequence, the interference graphs of all optimised superblocks are colourable.

Proof: In the incremental register mapping no merging of interference nodes is performed. Edges connect nodes that represent different flow chains of the same abstract resource. The number of colours available for each abstract resource corresponds to the number of element registers of that resource.

The interference graph is based on the register assignment determined by the optimisation of the integer linear program for the current superblock. Assume that there are n colours. Then each node can only be adjacent to $n-1$ nodes, since otherwise the number of available registers would have been exceeded by the solution of the ILP — but this would be no feasible solution. (Exceedings of abstract resources are temporarily allowed, but lead to the insertion of additional operations. The final solution of the ILP optimisation phase is guaranteed not to exceed the resource constraints of any resource.) In consequence there will be a free register, i. e. a free colour, available for each node of the interference graph. □

The assignment of the register flow chains to physical registers is done by a colouring algorithm that is performed separately for the interference graphs of each individual superblock. Previously determined register mappings are respected by the colouring. The constraints (7.14) in connection with the resource synchronisation constraints (7.7) guarantee that a feasible solution can be found.

The Colouring Algorithm. The problem of computing a feasible register assignment for a given input routine is modelled as the problem of finding a k-colouring of its interference graph.

Definition 7.6 (k-Colourability) *Let $G = (N, E)$ be a graph with a set N of nodes and a set $E \subseteq N \times N$ of edges. Furthermore, let there be a set of k colours. G is colourable if there is a function $f : N \rightarrow \{1, \ldots, k\}$ such that $f(u) \neq f(v)$ for all $(u, v) \in E$.*

Computing a k-colouring is known to be an NP-complete problem for $k > 2$ [GJ79]. However when analysing the definition of the interference graph (Def. 7.5) it becomes apparent that, in contrast to the interference graphs for register allocation [Cha82, BCT94], the size of the graph only depends on the number of superblocks generated for the input routine and the hardware architecture. The number of operations does not influence the size of the interference graph. If there is more than one abstract resource, each of them is represented by disjoint subgraphs of the interference graph further limiting the complexity of the colouring. Therefore the size of the interference graph can be expected to be sufficiently small to justify using an algorithm to compute an exact solution of this problem (see, e. g., Fig. 7.21 and Fig. 7.22).

The exact graph colouring algorithm works in a straightforward way: the nodes of the interference graph are traversed and to each node a colour is assigned that is different from the colours of all its neighbours. If no free colour is available, backtracking is performed so that in the worst case all combinations of nodes and colours are traversed. The node selection is based on the number of adjacent edges: the nodes with a high degree are coloured first.

If the size of the interference graph grows too large, a modified version of the optimistic graph colouring of [Bri92, BCT94] is performed. All nodes whose degree is smaller than the number of element registers of the corresponding abstract file are removed from the graph in arbitrary order and are pushed to a stack. If all of the remaining nodes have a degree larger than k, one of them is selected. The algorithm assumes optimistically that the node will be colourable in spite of its degree because of some neighbours sharing the same colour. So the node is marked as a spill candidate, and pushed on the stack. After all nodes have been removed from the graph, they are successively popped from the stack and a colour is selected that is different from the colours of all its neighbours. If there is a node for which no colour is available, the nodes that have been generated by merging operations are traversed. First those that have been marked as spill candidates are considered. If such a node can be found, its merging is undone and the colouring algorithm is restarted for the resulting interference graph. If no such node can be found, the colour that occurs the most rarely among the neighbours is selected and is used for the current node. In this case, the insertion of data transfer operations is necessary. In the experimental evaluation the size of the interference graphs always allowed the colouring to be computed by using the exact algorithm.

8. The Target Description Language TDL

TDL (Target Description Language) is a descriptive language that allows to concisely specify the hardware resources and the assembly language of the processor to be modelled. A TDL specification provides all information about the target architecture that can influence program analyses and optimisations. This includes the properties of the relevant hardware resources, the syntax and semantics of the machine operations and additional information as, e. g., timing characteristics. TDL is easily extendible and flexible in use which is a presupposition for modelling a wide range of target architectures and for supporting different kinds of optimisations and analyses.

TDL can be classified as a mixed structural/behavioural description formalism [MG95]. The resources are declared in a structural style while the machine operations are mainly described from the view of the operation behaviour, i. e. their semantics. A TDL description has a modular structure. It can comprise a specification of the hardware resources, a description of the instruction set, a constraint section, and an assembly section. Specification parts that are not needed for the target applications can be omitted. In the resource section the relevant hardware resources are introduced and their properties are specified by an extendible attribute mechanism. The instruction set section contains a definition of the instruction set in the form of an attribute grammar. Each operation is represented by a rule of this grammar. A flexible attribute mechanism allows the identification of important operation properties including the assembly representation, the set of source and destination operands, timing information, and the operation semantics. Information stored in attributes can be easily accessed by target applications; thus retrieving architecture-specific information does not implicitly require interpreting the specification of the operation semantics. The semantics of the operations is declared in a dedicated register transfer language (RTL). By functional abstraction the semantics specification can be modularised and different views of the instruction set can be expressed. The constraint section is composed of a set of logical conditions that have to be respected in order to preserve correctness during code transformations. The conditions specify interactions between different operations, interdependencies of operand locations and the parallelisation of operations, etc. An automatic translation of the logical conditions into integer linear constraints allows them to be precisely incorporated into ILP-based code optimisations. The assembly section deals with instruction and operation delimiters, assembly directives and comments. Each TDL description is checked for semantical consistency so that input errors and inconsistencies in the machine description are detected early.

From a TDL specification several ANSI-C files are generated containing data structures representing the specified information and various access and manipulation functions. The generated files can be compiled and linked with any target application. Second, a parser for the assembly language is automatically generated. The parser reads the input programs and calculates their control flow graphs that are represented in the generic format CRL (*Control Flow Representation Language*) [Lan99]. The CRL-representation of the control flow graph constitutes the interface to all optimisation and analysis algorithms and allows the flexible incorporation of additional user-supplied program analyses and optimisations in the PROPAN system (see Sec. 9.1).

This chapter is organised as follows: first an overview of related work in the field of hardware description languages is given. Then the different sections of a TDL description are presented in detail in the sections 8.2-8.5. Sec. 8.4.1 addresses the transformation of the logical expressions of the constraint section into integer linear constraints. The detailed language definition of TDL is given in [Käs99b].

8.1. Related Work

Hardware description languages are used for a variety of application areas: for architectural synthesis, hardware simulation, code generation and program analysis. In consequence, a large number of different hardware description formalisms has been developed. In the area of processor modelling and simulating, widely used languages are VHDL [LSU93] and Verilog [TM95]; well-known approaches used in code generation are ISPS [Bar81], MARIL [BHE91, Bra91], the MIMOLA language [Now87], the SALTO language [BCRS97], SLED [RF97], and nML [FVPF95]. Languages currently under development are LISA [PHZM99] aiming at generating cycle-accurate simulators for architectures with complex pipelines, ISDL [Had98], EXPRESSION [HGG$^+$99], and $\lambda-$RTL [DR98].

Hardware description languages can be categorised as behavioural, structural or mixed behavioural/structural languages [MG95]. A behavioural description specifies the instruction set of the target processor and focuses on the semantics, i. e. the behaviour of the machine operations. Structural specifications typically are close to the gate-level and describe the hardware modules of the processor with their interconnections. Many hardware description languages used for code generation incorporate aspects of both views and are classified as mixed-level approaches.

MARIL is a [BHE91, Bra91] machine description language designed for RISC architectures. Each MARIL-description consists of three sections. In the first section, hardware resources like registers, pipeline stages, and memories are declared. In the second section the runtime model is specified indicating which registers are used as stack pointer or frame pointer, and which registers are callee-saved or caller-saved registers. The third section describes the instruction set of the target processor. Each instruction is listed with its mnemonic, the operands, a C-expression describing the mapping to a node of the intermediate representation, a reservation table specifying the resource usage, and timing parameters. There is no extendible attribute mechanism and no mechanism to specify complex architectural irregularities. MARIL does not support a detailed specification of the semantics of machine operations and there is no semantical analysis of the machine description itself. A detailed specification of the assembly language including assembly directives or macros is not possible.

The SALTO system [BCRS97] has been developed to provide the user with an environment that allows to implement tools for analyses and transformations of low-level code. From the description of the target machine and the assembly language a parser is generated that reads assembly programs and calculates their control flow graph. Only a restricted set of assembly languages is supported; languages, e. g., with infix notation as the assembly language of the ADSP-2106X SHARC [Ana95] are not accepted. The user is offered an object-oriented interface to access and manipulate the data structures representing the control flow graph. In the first part of the machine description the hardware resources of the target processor are introduced and their properties are specified by a restricted set of predefined attributes. The instructions are specified by defining their assembly representation and their resource access sequence. Apart from a coarse classification of control flow operations no specification of operation semantics is supported. An attribute mechanism for specifying additional properties of machine operations is not available. It is not possible to specify constraints for modelling complex interactions among operations or interdependencies between scheduling and allocation decisions. Finally there is no semantical analysis of the machine description file, such that it is not possible during the parsing of the machine specification to inform the user of errors and inconsistencies.

SLED [RF97] is a specification language for encoding and decoding that is used in the New Jersey Machine Toolkit generating bit-manipulating code for applications that process machine code. It is intended to support, e. g., the development of debuggers, linkers, execution time analysers, or run-time code generators. Since the focus is the specification of instruction representations, it is not possible to declare hardware resources, there is no attribute mechanism and no mechanism to specify the semantics of the machine instructions. The SLED language is part of the CSDL (*Computer Systems Description Languages*) language family [DR98]. In order to specify the semantics of machine operations a dedicated language called λ-RTL is currently being developed within the CSDL family [DR98]. λ-RTL is a higher-order functional language, based on SML. The effect of each machine operation is specified as a register-transfer list (RTL) that describes the change to the machine state induced by the operation. Specification writers can introduce new RTL functions. However due to the functional specification style it is difficult to support generating cycle-accurate instruction-set simulators.

ISDL [Had98] has been designed to support a wide range of tools, from code generators, disassemblers up to instruction set simulators. Since assemblers and disassemblers are automatically generated, ISDL can make simplifying assumptions on the structure of the assembly language. It is not possible to specify assembly directives or assembly comments. Only storage resources from a set of predefined resource types can be declared; there is no possibility for introducing user-defined resource types. As an example declaring functional units is not possible such that the problem of functional unit allocation cannot be addressed on the basis of an ISDL description. ISDL offers a constraint specification mechanism to model restrictions of the available parallelism; the constraints are based on the operation syntax. ISDL does not provide an attribute mechanism for specifying the properties of hardware resources or machine operations. Thus retrieving hardware-specific information in optimisation and analysis algorithms mostly requires interpreting the complete semantical specification of each operation. There is no semantical analysis of the machine description such that input errors and inconsistencies are not detected when parsing the machine description.

The machine description language nML [FVPF95] is used in the retargetable compiler CHESS [LVPK+95]. The instruction set is modelled as an attribute grammar; each terminal of the grammar corresponds to a valid instruction. In addition to the instruction set grammar, an nML-description can contain declarations for storage objects, data types, constants and macros. Resource declarations are only possible for storage objects; declarations of functional units or caches are not supported and there is no possibility of introducing additional, user-defined resource types. Each instruction of the target processor is defined by specifying its semantics, the assembly representation and its binary representation. There is no generic way of describing the timing behaviour of operations. Moreover the VLIW machine model, i. e., the composition of long instructions from independent operations is not explicitly supported. Attributes are declared implicitly in nML without specifying a domain for the feasible attribute values. It is not possible to check the feasibility of attribute settings; there is no semantical analysis of the machine description itself. No constraint mechanism is available to model complex dependences between operations. A specification of the syntax of assembly expressions, comments, or directives is not supported.

EXPRESSION [HGG+99] is a mixed-level language for supporting architectural design space exploration for embedded systems-on-chip and automatic generation of a compiler/simulator toolkit. Structural information is given by specifying the hardware resources of the architecture, the pipeline mechanism, and the data transfer paths. Each operation is described by declaring its opcode, the operands and the mapping to generic operations of an intermediate representation. The composition of operations to long instructions is specified by instruction templates. A fixed set of attributes is available to declare the properties of storage resources; an extendible attribute mechanism however is not provided. Reservation tables to specify the resource usage of operations are extracted automatically from the structural specification on a per-operation basis. This allows resource constraints to be respected in the code generation process but limits the flexibility of the specification language. In [HGG+99, GHK+98] no information about the specification of the detailed semantics of machine operations is given.

8.2. The Resource Specification

Hardware-sensitive program analyses and optimisations require knowledge about the hardware resources of the target processor. This includes, e. g., its functional units, and the available register sets, memories and caches. As a rule of thumb, all hardware components that are important for the target application have to be declared as resources in the TDL description. TDL offers a set of predefined resource types whose properties can be described by a predefined set of attributes. Each of these attributes is associated with a domain and a scope of its own. The domain represents the feasible attribute values; its scope denotes the resource types the attribute can be associated with. The predefined resource types comprise functional units, register sets, memories and caches. Attributes are available to describe the bit width of registers, their default data type, the size of a memory, its access width, alignment restrictions, etc.

An implicit assumption of the machine model of TDL is that all resources of different types can work in parallel. If there are functional unit types with multiple instances it is assumed that those can work in parallel, too. This way, the VLIW machine model is supported. Architectures without instruction-level parallelism can be considered as a special case of VLIW architectures where only one (possibly virtual) execution unit is available. Most architectures do not have a fully orthogonal instruction set, i. e. the parallelism of functional units is restricted. Such restrictions of the parallelism can be modelled in the constraint section. Currently TDL cannot be used to completely specify architectures with superscalar pipelines. However there is ongoing work to develop a specification mechanism for complex superscalar pipelines and integrate it into TDL.

The designer can extend the domain of the predefined attributes and declare user-defined attributes if additional properties have to be taken into account. Similarly to the predefined attributes, each user-defined attribute must be associated with an explicitly declared domain and scope. Those declarations are used during the semantical analysis of the machine description. The attribute domains are restricted to simple numerical types (integer and floating-point), character strings and references to previously declared resource types.

It is important to support different views of physical hardware resources. As an example the same register file could be used as an integer and a floating-point register file where one view is associated with fixed-point operations, the other with floating-point operations and the assembly representation of both views differs. Different views of hardware resources are supported in TDL by a dedicated alias mechanism. The attribute settings of aliased resources can differ.

Finally the designer can introduce additional resource types and describe their

```
Resources-Section
...
FuncUnit ALU replication=2;
Register gpr "r%d" [0:31] size=32, type=signed<32>;
SetProperties gpr[30] usage=SP;
RegisterAlias dreg "d%d" gpr mapping=[2:1], type=float<56,8>;
Memory DM type=data, align=16, access=32;
DefineAttribute Replacement {"LRU","FIFO"} associated to Cache;
Cache InstrCache assoc=2, size=256, linesize=32, type=instr,
    Replacement=LRU;
```

Figure 8.1.: Example of a Resource Section.

properties by dedicated attributes. This allows for maximum flexibility with respect to the range of supported hardware architectures and target applications.

An example of a resource section is shown in Fig. 8.1. Keywords are printed in typewriter font. The keyword `Resources-Section` marks the beginning of the resource section. Each resource type is assigned an unambiguous name that is used to identify the resource type in the target applications. First a functional unit type with unambiguous name *ALU* is declared of which two instances exist. This means that the target architecture disposes of two functionally equivalent ALUs that do not have to be distinguished during analyses and optimisations. Subsequently a register file named *gpr* is declared that consists of thirty-two 32-bit registers. The width of the element registers is given by the predefined attribute `size`; the value of the predefined attribute `type` indicates that by default the registers are used to store 32-bit two's complement numbers. The assembly representation of each element register is declared in the notation of C format strings; in the example the element registers of *gpr* are represented by r0,..., r31. The `SetProperties` statement allows to modify or extend the attribute setting of previously declared resources. In the example, the register *gpr*[30] is declared as the stack pointer. Subsequently the alias mechanism is used to declare another view of the register file *gpr*. Two successive integer registers are combined to form one 64-bit floating-point register. The assembly representation of the combined registers is declared as d0, ..., d15. The following declaration introduces a data memory named *DM* that supports 32-bit accesses that have to be aligned on 16-bit boundaries. The example concludes with declaring a two-way set-associative instruction cache with 256 lines of 32 byte length. The replacement strategy is an example of a user-defined attribute: its domain consists of the two strings *LRU* and *FIFO* and its scope is the resource type `Cache`.

8.3. The Specification of the Instruction Set

The central part of a TDL specification is the description of the instruction set of the target processor. The assembly representation of all machine operations, their timing behaviour and their semantics have to be known and the specification of additional information must be supported. As already mentioned the execution model of TDL corresponds to that of a VLIW architecture. Each instruction can be composed of several machine operations that are executed in parallel.

The definition of the instruction set is given in the form of an attribute grammar. Each operation is represented by a rule of this grammar; orthogonal operation parts are encapsulated in rules of their own. The terminals of the grammar represent the feasible machine operations. In the following, some fundamental concepts of the theory of attribute grammars presented in [WM95] are summarised; in [WM95] a comprehensive survey can be found.

Definition 8.1 (Attribute Grammar) *Let $G = (N_N, N_T, P, S)$ be a context-free grammar whose pth production in P is written as $p : X_0 \rightarrow X_1 \ldots X_{n_p}$, $X_i \in N_N \cup N_T$, $0 \leq i \leq n_p$. An attribute grammar AG over G consists of*

- *an association of two disjoint sets $Inh(X)$, the set of inherited attributes and $Syn(X)$ the set of synthesised attributes, with each symbol of $N_N \cup N_T$. We let $Attr(X) = Inh(X) \cup Syn(X)$ denote the set of all attributes of X. If $a \in Attr(X_i)$ then a has an occurrence in production p at the occurrence of X_i, which we write as a_i. Let $V(p)$ be the set of all attribute occurrences in production p.*

$$Inh = \bigcup_{X \in N_N \cup N_T} Inh(X); \quad Syn = \bigcup_{X \in N_N \cup N_T} Syn(X); \quad Attr = Inh \cup Syn$$

- *the specification of a domain D_a for each attribute $a \in Attr$ containing all potential values;*

- *a semantic rule*

$$a_i = f_{p,a,i}(b^1_{j_1}, \ldots, b^k_{j_k}) \ (0 \leq j_l \leq n_p)(1 \leq l \leq k)$$

  for each attribute $a \in Inh(X_i)$ $(1 \leq i \leq n_p)$ and each $a \in Syn(X_0)$ in every production p, where $b^l_{j_l} \in Attr(X_{j_l})(0 \leq j_l \leq n_p)(1 \leq l \leq k)$. Thus, $f_{p,a,i}$ is a function from $D_{b^1} \times \cdots \times D_{b^k}$ to D_a.

An attribute is always viewed as an attribute of one nonterminal or terminal; that is, the assignments Inh and Syn can be viewed as injective functions from the set $N_N \cup N_T$ into the set of attributes. This does not mean that attributes for different nonterminals and terminals cannot have the same name. This is sensible when they are carriers of the same type of information. The semantic rules of the attribute grammar represent the functional dependences between the values of *occurrences of attributes* in the productions of the grammar. Such a functional dependence can be viewed as a computational prescription specifying how the value of the occurrence of an attribute is calculated from the values of other occurrences of attributes of the same production.

Let a syntax tree t of the underlying context-free grammar be given and let n be a node of t. Let $symb(n) \in N_N \cup N_T$ be the symbol labelling n. If $symb(n) \in N_N$, then let $prod(n)$ be the production applied at n. For every attribute $a \in Attr(symb(n))$ there exists an *attribute instance* a_n at n. This instance should be assigned a value from its domain D_a. Let $val(a_m)$ be the value of the instance of a for the node m. If $a_i = f_{p,a,i}(b^1_{j_1}, \ldots, b^k_{j_k})$ is a semantic rule of $prod(n) = p$, then it induces the following relation between values of attribute instances:

$$val(a_i) = f_{p,a,i}(val(b^1_{j_1}), \ldots, val(b^k_{j_k}))$$

The structure of an attribute grammar representing the instruction set of a target processor is simple. There are no inherited attributes; all attributes declared to specify the properties of machine operations are synthesised attributes. The domain of each user-defined attribute is given in the attribute declaration of TDL. Their domains are restricted to numerical types, character strings and references to previously declared resources. All attributes used for specifying properties of machine operations are associated with each symbol of the attribute grammar.

The designer can provide placeless semantic rules that represent *external rules* for computing the values of instances of attributes for the symbols of the attribute grammar. Apart from those placeless rules there is no mechanism for introducing user-defined semantic rules. The semantics of the attribute grammar of the instruction set is the representation of all relevant information about the machine operations. It is sufficient to have two generic semantic rules f_A and f_S for evaluating the values of attribute occurrences. The generation of user-defined attribute evaluators is not supported in the current implementation.

Assume that the values of all attribute instances are initialised to a special value $\perp$ representing an undefined value. Let $p : X_0 \to X_1 \ldots X_{n_p}$ be a production of the underlying context-free grammar, $X_i \in N_N \cup N_T$ $(1 \leq i \leq n_p)$, $x_0 \in N_N$. Then there is one semantic rule $a_0 = f_A(a_1, \ldots, a_{n_p})$ for each attribute $a \in Attr = Attr(X_0) = Syn(X_0)$ that induces the function

$$val(a_o) = f_A(val(a_1), \ldots, val(a_{n_p})) = \begin{cases} val(a_j), & if\, val(a_j) \neq \perp \wedge \\ & \quad val(a_k) = \perp\ \forall k \neq j \\ \perp, & if\, val(a_j) = \perp\ \forall k \\ \top, & else \end{cases}$$

The user has to ensure that the value of each attribute instance is "computed" at most once on any path from the root of a syntax tree for an operation to a leaf. The value $\top$ is treated as an error symbol; if this value is encountered, the TDL parser reports an error and does not accept the specified attribute grammar. If the value of an instance of an attribute a is defined at a node m of a syntax tree, the function f_A propagates the value of a_m to the root of the tree for this operation and ensures that there is no contradictory setting of a.

Let $\int$ denote the `semantics`-attribute of an operation which represents the specification of the operation semantics. The semantic rule used to determine the value of $\int$ is essentially a string substitution function. The specification of the operation semantics is treated as a character string. Let $p : X_0 \to X_1 \dots X_{n_p}$ be a production of the attribute grammar, $X_i \in N_N \cup N_T$ $(1 \leq i \leq n_p)$, $x_0 \in N_N$. Let γ be a string provided by the user as external definition of the defining occurrence of the attribute $\int$ in rule p. Then there is one semantic rule $\int_0 = f_S(\int_1, \dots, \int_{n_p})$ for this attribute $\int_0 \in Attr = Attr(X_0) = Syn(X_0)$ that induces the following function:

$$val(\int_0) = f_S(val(\int_1), \dots, val(\int_m)) = \gamma[X_1.\int/val(\int_1), \dots, X_{n_p}.\int/val(\int_{n_p})]$$

The operation semantics is specified in a register transfer language (RTL) that is presented in Sec. 8.3.1 in more detail. It has to be ensured that the composition of the values of the applied occurrences $\int_1, \dots \int_{n_p}$ yields a feasible RTL code sequence. This is checked by the TDL parser; if this condition is not fulfilled, an error is reported. It is obvious that the system of equations induced by the attribute grammar for any syntax tree is non-cyclic. Thus it is ensured that the attribute grammar is well-formed.

TDL assumes the following execution model: each operation is executed by one resource of the target processor and the timing behaviour can be described by the two parameters *execution time* and *latency*. The execution time denotes the number of control steps the operation execution takes and the latency describes the minimal number of clock cycles between two successive inputs of the appropriate functional unit. The completion time of an operation can optionally be associated with a dedicated resource type that models a write-back bus of the target architecture. This way the number of simultaneous accesses on the result bus can be restricted independently from the starting time of the operations.

With each operation, at least two attributes are associated: its timing behaviour and the operation semantics. The timing behaviour of an operation is specified in TDL by a simple reservation table mechanism: all hardware resources that can be used to execute an operation are specified together with the appropriate execution time and latency. Additionally a dedicated resource for result synchronisation can be given. The semantics of the operations is described by a dedicated register transfer language; this is described in detail in Sec. 8.3.1.

A dedicated set of attributes is available for specifying the operands of the machine operations. The attributes $src_1, src_2, \dots$ represent the set of source operands and indicate the storage locations that are feasible for each operand. Likewise, the attributes $dst_1, dst_2, \dots$ represent the set of destination operands and indicate which storage locations are feasible for each of them. Additional predefined attributes are available for describing frequent memory access modes, jump targets, etc. The designer can also introduce user-defined attributes with explicitly declared domains for specifying further operation properties that are important for the target application.

An example for an operation declaration is shown in Fig. 8.2. In the example a machine operation with unambiguous name `IAdd` is declared. This name is followed by the definition of the assembly representation of the operation. The

```
DefineAttribute guarded {"true","false"} associated to Operation;
DefineOp IAdd "%!(optguard) %s = %s + %s"
  {dst1="$2" in {gpr}, src2="$3" in {gpr}, src3="$4" in {gpr}},
  {ALU(exectime=1, latency=1);},
  {unsigned<1> gval;
   optguard.semantics;
   if ((guarded = true)&&(gval<0>=1)) { dst1:=src2+src3;}};
OpNT optguard "if %s" {src1="$1" in {gpr}, guarded=true},{;},
   {gval:=src1<0>;}
  | "if !%s" {src1="$1" in {gpr}, guarded=true},{;},
   {gval:=!src1<0>;}
  | "" {guarded=false},{;},{};
```

Figure 8.2.: Example Operation Declaration

assembly representation is specified in a syntax similar to C format strings. The expression `%!(optguard)` represents an occurrence of a non-terminal "optguard". The productions for non-terminals are introduced by a dedicated keyword *OpNT* in order to distinguish them from complete operations. In the definition of the assembly representation, the `%s` directive represents a sequence of characters. The expression `dst1 = "$2" in {gpr}` represents an external rule describing how the value of the occurrence of the attribute `dst1` at the rule for operation `IAdd` is computed. Its meaning can be described as follows: the second placeholder in the format string corresponds to the assembly representation of a register that is modified by the current operation. The set of feasible storage locations for this operand is restricted to the registers of the register file `gpr`. This way the information of the resource section is reused in the instruction set description. The resource section can be viewed as a special part of the attribute grammar of the instruction set that contains dedicated rules for representing the resources of the target processor. As a consequence the description is shortened and, more importantly, a semantical analysis of the operation definitions becomes possible. It is checked that all referenced resources have been declared and that the semantical specification is type-correct.

The other operands are declared as source operands by using the attributes `src1`, `src2`, `src3`. The attribute `guarded` of Fig. 8.2 is an example of a user-defined attribute. First the unique attribute name is introduced, followed by the specification of the domain of all potential values. Finally the scope of the attribute is declared to be the set of all machine operations. The attribute `guarded` is used as a flag indicating whether the operation is guarded by an additional source operand (predicated execution). The operation part corresponding to the optional guard is represented by the non-terminal `optguard`. Its assembly representation can be an empty string, if the operation is not guarded (see last line of Fig. 8.2). In this case, the `guard`-attribute of the operation is set to `false`. Otherwise, the operation is introduced by the string `if` followed by a reference to a register of `gpr`, possibly with a leading exclamation mark. Then, the `guard`-attribute is set to `true` and the register used as a guard represents an additional source operand. If a condition register is specified (with a leading exclamation mark) the operation is executed only if the least significant bit of that register has the value 1 (0).

The subsequent two blocks represent the timing and the semantics of the operation. The reservation table specification of Fig. 8.2 indicates that the operation `IAdd` is executed by an instance of the resource type `ALU` and that both execution time and functional unit latency take one clock cycle.

It must be distinguished between the declaration of operations in the attribute grammar of the instruction set and the representation of the operation instances of an input procedure. For each operation the attribute grammar defines a template denoting all feasible operation instances. Assume that the destination operand of an operation is defined by the expression `dst1="$1" in {gpr}`. Each operation instance of the input program will have concrete register operands, i. e. the value of the attribute instance of `dst1` for this operation instance is the physical register that is used. The value of the attribute instance of `dst1` of the operation template defined by the attribute grammar denotes all feasible storage locations. This distinction is important in order to be able to efficiently perform register renaming or to change the assignment of operations to functional units.

The attribute mechanism plays an important role in supporting different views of the instruction set. When computing the data dependences of an input procedure, it must be known for each operation which storage locations are read and which ones are modified. It is not necessary to know how storage locations are modified, the information that they are modified is sufficient. If the machine description is used as the basis of a value analysis, or constant propagation, the information about how the result of an operation is computed is essential. Finally, if an instruction set simulator is to be generated from the machine description, it must additionally be known at which clock cycles the effects of an operation take place. All those views are supported by TDL in an efficient way. As an example, the calculation of the generic data dependence graph is based on the values of the attribute instances of the operation instances of the input program as, e. g., dst_i and src_i. It is not necessary to interpret the detailed specification of the operation semantics.

8.3.1. The Specification of the Semantics

The semantics of an operation describes how the execution of the operation affects the machine state. Components of the machine state are the storage resources of the processor, i. e. memory and register cells, condition codes, and so on. In [Die95] the semantics of the operations are specified in C. This is a flexible approach suited for the generation of instruction set simulators. If, however, analysis and optimisation algorithms have to be supported this approach is not feasible due to the difficulty of analysing the semantical specification itself. In most approaches, e. g. nML [FVPF95], ISDL [Had98], λ-RTL [DR98], register transfer lists are used to specify the operation semantics. A register transfer represents the transfer of

```
RTLPrg      →  (RTLStat)*
RTLStat     →  RTLAsgn | RTLIfStat | RTLForStat | RTLWhileStat
               | RTLSwitchStat | RTLVStorageDecl | RTLPreFunc
               | RTLCanFunc | RTLBlock
RTLBlock    →  { RTLStat }
RTLAsgn     →  AsgnLHS := RTLExpr ;
IfStat      →  if ( RTLExpr ) RTLBlock [ else RTLBlock ] ;
ForStat     →  for ( RTLStat ; RTLExpr ; RTLStat ) RTLBlock ;
WhileStat   →  while ( RTLExpr ) RTLBlock ;
SwitchStat  →  switch ( RTLExpr ) { SwitchCaseList SwitchOptDef };
RTLExpr     →  RTLExpr BinOp RTLExpr | UnOp RTLExpr
               | RTLCanFunc | RTLPreFunc | IntConst
               | FloatConst | StorageRef | AttribRef;
BinOp       →  || | && | | | ^ | % | == | != | < | > | <= | >=
               | << | >> | + | - | * | /
UnOp        →  - | ~ | !
RTLCanFunc  →  ID ( RTLParalist )
```

Figure 8.3.: Skeleton of the RTL grammar in BNF form.

a value into a storage location and thus a change of the machine state. Register transfers contained in the same list are assumed to take simultaneously effect.

In TDL the semantics is specified by a dedicated register transfer language (RTL) that is similar to the mechanism of register transfer lists. Similarly to λ-RTL the language has been designed with the goal of supporting type-checking of the machine description. The register transfer language of TDL is statement-oriented. The main reason is to retain the flexibility for generating cycle-accurate instruction set simulators. Statements can be grouped; each group of statements is assumed to take simultaneously effect. In contrast to the functional approach of λ-RTL the execution of an operation can be viewed as a sequence of effects spanning several control steps. All effects can be explicitly assigned to the appropriate control step. A skeleton of the RTL grammar is shown in Fig. 8.3 in BNF form; the complete definition is given in [Käs99b].

The semantics of each machine operation is specified by an RTL program consisting of a sequence of statements. The RTL program may contain declarations of virtual storage locations that are helpful in describing the semantics of complex operations. The scope of virtual storage declarations is the complete RTL program, i. e. the complete specification of the semantics of one machine operation. Feasible statements are assignments, conditional statements, loops, and function calls. Statements can be grouped to blocks. Values are computed by expressions without side effects. This means that there are no implicit modifications of storage locations; all changes have to be explicitly given. Expressions may be numerical or string constants, attribute values, references to storage locations, or applications of operators or RTL functions to expressions. Attribute references in the RTL language always represent references to the attribute instances of operation instances. All storage resources that are referenced in RTL statements must have been declared in the resource section or by preceding virtual storage declarations in the same RTL program. The operators include the set of arithmetic and logical operators known from the C language. Those are assumed to produce no side effects. Additionally there is a set of predefined RTL-functions for common functions that offer the possibility of explicitly specifying side effects, as e. g. the setting of condition codes, overflow flags, etc.

The designer has the possibility of declaring user-defined "canonical" functions. This way the specification can be modularised and shortened. Additionally, the canonical functions can be used to express different views of the instruction set. If the function body is not of interest for a target application, the function can be considered as a black box. All parameters passed to functions can be thought of bit sequences with optionally a given type constructor. In the declaration of the formal parameters it must be annotated whether the parameters are read-only, or can be modified within the body of the function.

RTL expressions are typed. This facilitates analysing properties of machine operations and is helpful in finding bugs in the machine description. Since register transfers represent operations on machine level, the basic data types of RTL are bit sequences of a given length, denoted by the keyword `storage`. When specifying the semantics of an operation it must be clear how the contents of a storage location are to be evaluated. Often the type of a storage location depends on special conditions or mode settings. As an example the contents of the same register might be interpreted as signed integer in two's complement representation, as unsigned integer, as a fractional value or even a floating-point number. In order to allow for maximum flexibility, RTL provides the possibility of annotating a type to each storage location. These annotations do not represent conversion functions but can be seen as explicit type constructors. The basic data types of RTL are the following:

- `signed<n>` denotes an n-bit signed integer in two's complement representation.
- `unsigned<n>` denotes an n-bit unsigned integer.
- `float<m,n>` denotes an IEEE floating point value with an m-bit mantissa and an n-bit exponent.
- `fract<n>` denotes fractional values. Let $a = (a_0, a_1, \ldots, a_{n-1})$ be an n-bit number of type `fract<n>`. Then its value is defined as $\langle a \rangle = -2^0 + \sum_{i=1}^{n-1} a_i \cdot 2^{-i}$
- `storage<n>` denotes a storage location of length n whose type can be any of the previously mentioned.

TDL uses a polymorphic type inference algorithm; if an explicit type constructor is used, the inferred type is overridden. An example is the expression `_signed(dst1, 32)` that requires the destination to be interpreted as a 32-bit two's complement value. The polymorphism of the RTL language is restricted; it is similar to the subtype rule in object-oriented languages. If the formal parameter of a function has been declared to be of type `storage`, it is also feasible to pass storage locations whose contents have been inferred to be of `signed` or `unsigned` type. In most cases the polymorphism is concerned with the width of storage locations. As an example consider the predefined function `_iabs` that has the following prototype:

```
signed<T> _iabs (signed<T>, unsigned<1>, unsigned<1>*).
```

The first parameter can be a signed integer of any length; then the result is a signed integer of the same length. If the bit passed as second parameter has the value 1, saturation is performed; then in the case of positive (negative) overflow the largest positive (smallest negative) number of the appropriate bit width is returned. If an overflow occurs, the bit passed as third parameter is set to 1. All parameters except the last are read-only parameters; the symbol `*` indicates that this parameter is modified by the function.

The RTL operators as, e. g., `*`, `+`, `-`, `&` are provided as default operators; they are overloaded. If the types of the operands do not match, the result type is determined by the conversion rules shown in Tab. 8.1. The following type variables are used: $t_1 \in \{\texttt{signed, unsigned, float}\}$, $t_2 \in \{\texttt{signed, unsigned}\}$, and $t \in \{\texttt{signed, unsigned, float, storage}\}$.

Operand 1	Operand 2	Result
$\texttt{storage}<m>$	$t_1<n>$	$\texttt{storage}<max(m, n)>$
$\texttt{float}<m,n>$	$t_2<u>$	$\texttt{float}<m, n>$
$\texttt{signed}<m>$	$\texttt{unsigned}<n>$	$\texttt{signed}<max(m, n+1)>$
$\texttt{float}<m_1,n_1>$	$\texttt{float}<m_2,n_2>$	$\texttt{float}<max(m_1, m_2), max(n_1, n_2)>$

Table 8.1.: Automatic Type Conversion.

Special attention has to be paid for constants. Integer constants are always treated as 32-bit two's complement values, floating-point constants as 32-bit numbers with 24-bit mantissa and 8-bit exponent. If another representation is required, this must be explicitly annotated by an appropriate type constructor. Binary and hexadecimal values are always exactly represented.

8.4. The Constraint Section

The basic execution model of TDL assumes that all functional units declared in the resource section work in parallel. However especially in irregular architectures there are restrictions of instruction-level parallelism. Such restrictions can be caused by resource constraints or by encoding constraints. The restricted parallelism of ALU and multiplier in the ADSP-2106x SHARC (see Sec. 10.1.1) is due to encoding restrictions: ALU and multiplier can operate in parallel but the instruction word is too short to allow each operand to be located in any of the 16 general purpose registers. By restricting each operand to a set of 4 registers all four source operands can be specified with 8 bits.

There are only few behavioural or mixed-level hardware description languages used for code generation that allow irregular hardware constraints to be specified. In the EXPRESSION [HGG+99] language hardware resources are modelled in a structural way. While this allows to model resource constraints, the incorporation of encoding restrictions is not directly possible. In ISDL [Had98] restrictions of instruction-level parallelism and resource usage are modelled by specifying boolean expressions over the assembly representation of the machine operations.

In the constraint section of TDL, restrictions of instruction-level parallelism and resource usage as well as interdependencies between scheduling and allocation are modelled by specifying a set of *rules*. The rules are composed from boolean expressions that refer to the properties of the hardware resources and machine operations specified in the preceding sections of a TDL description. This way the rules are independent from the assembly syntax and allow a semantical analysis of the machine description. The resulting representation is more concise, more flexible and less error prone than in the ISDL language and it allows modelling resource and encoding restrictions in a uniform way. The most important property of the rule-based approach of TDL is that the specified rules can be transformed into integer linear constraints. This way, irregular hardware properties can be modelled in a generic way and can be incorporated into a homogeneous problem description.

The constraint section of TDL can be viewed as a constraint store containing the specified rules. The rules represent conditions that have to be respected to preserve correctness during program transformations. Each rule is composed of a *premise* and a *consequent*. The premise is a boolean expression that can be statically evaluated, i. e. when the TDL description is processed. It represents the condition under which the boolean expression of the consequent must be satisfied. The condition specified in the premise must describe operation properties that are invariant with respect to scheduling and allocation decisions. An example is a test whether an operation belongs to a certain operation class. The consequent represents a condition that must be dynamically evaluated, i. e. during the runtime of the optimisation phases. Since the focus is on instruction scheduling, register assignment and functional unit allocation, this includes conditions over the usage of storage resources or execution units, parallel execution of operations, and operation sequencing. In order to support additional optimisations to be based on TDL, it is also possible to incorporate conditions that are static with respect to instruction scheduling, register assignment and functional unit allocation. During these phases the conditions are treated as static information. The TDL parser checks whether the premise can be statically evaluated; if this is not the case an error is reported.

From the constraint section, C functions are generated that can be invoked by the target application in a generic way in order to take into account the speci-

```
CRule         →  CExpr : CExpr ;
CExpr         →  CExpr | CTerm | CTerm
CTerm         →  CTerm & CFactor | CFactor
CFactor       →  | !(CExpr) | ( CExpr )
                 | ResAttribref in ResRefs
                 | id in OpOrOpClassList
                 | id && id | id ->(n) id | id ->>(n) id
                 | COpnd CRel COpnd
CRel          →  == | != | > | < | <= | >=
COpnd         →  id.AttribName | Const
ResAttribRef  →  id .src1 | id .src2 | ...
                 | id .dst1 | id .dst2 | ... | id . exec
```

Figure 8.4.: Excerpt of the grammar of the constraint language in BNF form.

fied architectural restrictions. For each rule, two functions are generated, one for the use in ILP-based optimisation, the other for the use in a generic list scheduling algorithm. The first function traverses the operations of each input program in a nested loop and for each set of operations that satisfy the condition of the premise, ILP constraints that are equivalent to the consequent of the rule are generated. These constraints can be integrated into integer linear programs in the SILP or OASIC formulation generated for the input program. This way, additional hardware-specific information can be flexibly incorporated into the integer linear programs. The second function is a support function for generic list scheduling algorithms that is called in order to decide whether the scheduling of an operation to a given control step does not violate irregular hardware properties. Additional user-supplied optimisation phases have to individually analyse the representation of the rules and extract the information relevant for them.

The syntax of the constraint language is shown in Fig. 8.4 in BNF form. The constraint section is introduced by the keyword `Constraint-Section` and consists of a set of rules. Each rule consists of two boolean expressions separated by a colon. These expressions represent the premise and the consequent of the rule. In the premise unbound variables for operations can be introduced. These variables are bound to operation instances of each input program during the runtime of the generated optimisers. The boolean expressions are constructed by the boolean operators disjunction, conjunction, and negation from a predefined set of atomic expressions. The atomic expressions describe relations between the resource usage and scheduling properties of machine operations. Atomic expressions are available to check whether an operand is located in a given set of storage locations, whether an operation belongs to a given operation class, whether two operations are executed in parallel, and whether one operation is executed exactly (at least) n cycles after another. Additionally the values of operation attributes can be compared; the feasible comparisons depend on the type of the attribute values (character strings

```
Constraints-Section

/* Parallelisation Restrictions */
op in {C0}: op.dst1 = op.src1;
op1 in {C1} & op2 in {C2}: !(op1 && op2);
op1 in {CAluFixed} & op2 in {CMulFixed}:
    !(op1 && op2) | op1.src1 in {iregC} & op1.src2 in {iregD}
    & op2.src1 in {iregA} & op2.src2 in {iregB};
```

Figure 8.5.: Exemplary constraint section.

and resource references can only compared for (in)equality, numerical values also for the relations $<, >, \geq, \leq$).

Some exemplary rules are shown in Fig. 8.5. The first one enforces the first source operand to be identical to the destination operand for all operations of the operation class *C0*. The second rule prevents any operation of operation class $C1$ to be executed in parallel with an operation of operation class $C2$. The last rule models the restricted parallelism of ALU and multiplier of the Analog Devices ADSP-2106X SHARC. There some operations can only be executed in parallel if all operands reside in uniquely defined register groups within the general purpose register file. If the operands use other registers, the resulting operations are valid but cannot be grouped into one instruction (see Sec. 10.1.1). So the rule states that an operation of the operation class `AluOps` and an operation of `MulOps` can only be executed in parallel if all operands are located in the appropriate register groups.

8.4.1. Generating Integer Linear Constraints

The premise of each rule is mapped to C code checking for a given set of operations of the input program if the condition of the premise is met (see Fig. 8.6). If this is the case, a set of integer linear constraints is generated that is equivalent to the boolean expression of the consequent. The effect of these constraints is that a point $x \in \mathbb{Z}^p$ is only feasible for the generated integer linear program if the assembly program represented by x satisfies the logical condition. Before the integer linear constraints are generated, the consequent of each rule is transformed into disjunctive normal form [Bal98].

Definition 8.2 *Let an alphabet $\mathcal{A} = \{X_1, \ldots, X_n\}$ be given and let $\mathcal{B}(\mathcal{A})$ be the set of boolean expressions over $\mathcal{A}$. A boolean expression $b \in \mathcal{B}(\mathcal{A})$ is in disjunctive normal form if it has the form*

$$b = \bigvee_{i=1}^{m} (\bigwedge_{j=1}^{n_i} x_{ij})$$

where $x_{ij} \in \{X_1, \ldots, X_n, \neg X_1, \ldots, \neg X_n\}$.

```
op1 in OpClass1 & op2 in OpClass2:  !(op1 && op2)
```

$$\Downarrow$$

```
int ConstrFun (/* InputOps is an array with n entries containing
                  representations of all operations of the input
                  program */ )
{
  /* Declarations */
  for (i=0; i<n; i++) {
    op1 = InputOp[i];
    for (j=0; j<n; j++) {
      op2 = InputOp[j];
      if (OpContainedInOpClass(op1, "OpClass1") &&
          OpContainedInOpClass(op2, "OpClass2")) {
        /* Generate ILP-Constraint */
      }
    }
  }
}
```

Figure 8.6.: Example of a rule and the generated C-code for generating ILP-constraints.

The goal of this transformation is to achieve a high efficiency of the generated integer linear constraints. A boolean expression in disjunctive normal form is a disjunction of monoms where each monom is composed from negated or non-negated atomic expressions. For the generation of ILP-constraints each negated atomic expression is considered as an atomic expression of its own with a transformation rule of its own. The generation of integer linear constraints for atomic expressions can take advantage of the complete information about the target architecture and the underlying domains. If we allowed composed boolean expressions to be negated this would have to be modelled only by exploiting equivalence transformations of boolean expressions leading to more complex ILP constraints. In [Bal74b, Bal74a] the problem of obtaining valid cutting planes from arbitrary logical conditions brought to disjunctive normal form has been addressed. In [Bal98] the properties of the convex hull of the feasible points of a disjunctive program are studied and the facets of the convex hull are characterised. These results can be exploited in order to efficiently solve integer linear programs with disjunctive constraints.

In the following an inductive proof is given that the boolean expressions in the consequent of a rule can be transformed into an equivalent set of ILP constraints. The proof is constructive, i. e., it represents the algorithm that is used to generate the ILP constraints. In the following we assume that the consequent is composed from atomic expressions representing conditions that are dynamic with respect to the instruction scheduling, the register assignment, or the functional unit assignment. The following atomic expressions and their negations are considered:

- $op.attribref\ in\ \ \{r_1, \ldots, r_k\}$
- op_1 && op_2
- op_1 -> (n) op_2
- op_1 ->> (n) op_2
- $op_1.attribref == op_2.attribref$

Let $\mathcal{L}_C$ be the language containing all expressions of those types that can be generated by the grammar of Fig. 8.4.

Theorem 8.1 *Let $\mathcal{B}_{DNF}(\mathcal{C})$ be the set of boolean expressions in disjunctive normal form over the alphabet $\mathcal{L}_C$ as defined above. Then for each $b \in \mathcal{B}_{DNF}(\mathcal{L}_C)$ there is a set of integer linear constraints that is equivalent to b.*

Proof: First we have to show that the atomic expressions as well as their negated form can be translated to integer linear constraints. The induction step shows how the representation of boolean expressions formed from those literals is obtained.

Induction Basis

(i) *op.attribref in* $\{r_1, \ldots, r_k\}$

The following attribute references are feasible: src_i ($i \in \mathbb{N}$) denoting a source operand, dst_i ($i \in \mathbb{N}$) denoting a destination operand, and *exec* denoting the functional unit executing the operation. The TDL-parser checks that operand references are always associated with a set of storage locations and the *exec*-attribute with a set of execution units. First we consider the cases that the attribute reference denotes a source or destination operand. If no register assignment is performed no allocation decisions have to be formulated as integer linear constraints. In this case it is sufficient to check whether the condition is satisfied by the current allocation. If so, the constraint $1 = 1$ is generated, otherwise the constraint $1 = 0$. Generating these constraints allows a uniform implementation; they are removed in the preprocessing phase of the ILP solver [ILO99].

If the task of register assignment is addressed, the SILP formulation must be used. It is necessary to distinguish between references to source and destination operands.

(i.1) The expression $op.src_\nu$ *in* $\{r_1, \ldots, r_k\}$ means that the source operand src_ν must be located in one of the abstract register resources $r_1, \ldots, r_k$. Let j be the operation currently bound to *op*. Since the register assignment is formulated as a register distribution problem in the SILP formulation it is necessary to determine a definition of the source operand src_ν. This can be done using the data dependence graph. If there is more than one definition reaching the use in operation j, one of these definitions is selected to generate the ILP constraint. The synchronisation of several definitions reaching the same use is described in Sec. 7.2.4.

Let d be a definition of the source operand src_ν of operation j. Then the following constraint is generated:

$$\sum_{r \in \{r_1, \ldots, r_k\}} \sum_{(m,d,r) \in E_Z} x^r_{md} = 1$$

This constraint guarantees that d uses a register from the set $\{r_1, \ldots, r_k\}$ as its destination. In consequence the source operand src_ν of operation j is located in a register from the set $\{r_1, \ldots, r_k\}$, too.

(i.2) The expression $\neg$ ($op.src_\nu$ *in* $\{r_1, \ldots, r_k\}$) means that the source operand src_ν must not be located in any of the registers represented by $r_1, \ldots, r_k$. Again let j be the operation currently bound to *op* and let d be a definition of the source operand src_ν of j. The constraint to be generated is analogous to (i.1):

$$\sum_{r \in \{r_1, \ldots, r_k\}} \sum_{(m,d,r) \in E_Z} x^r_{md} = 0$$

The destination of d must not be located in any of the registers represented by $r_1, \dots, r_k$.

(i.3) The expression $op.dst_\nu$ *in* $\{r_1, \dots, r_k\}$ means that the destination operand dst_ν must be located in one of the storage resources $r_1, \dots, r_k$. Let j be the operation currently bound to *op*. Then the following constraint is generated:

$$\sum_{r \in \{r_1,\dots,r_k\}} \sum_{(m,j,r) \in E_Z} x^r_{mj} = 1$$

(i.4) The expression $\neg$ ($op.dst_\nu$ *in* $\{r_1, \dots, r_k\}$) means that the destination operand dst_ν must not be located in any of the storage resources $r_1, \dots, r_k$. Let j be the operation currently bound to *op*. Then the following constraint is generated:

$$\sum_{r \in \{r_1,\dots,r_k\}} \sum_{(m,j,r) \in E_Z} x^r_{mj} = 0$$

If the attribute reference denotes the functional unit used to execute operation j currently bound to *op*, it is necessary to distinguish between the SILP and the OASIC formulation.

(i.5) The expression *op.exec in* $\{r_1, \dots, r_k\}$ forces one of the resources $r_1, \dots,$ r_k to be used as the execution unit for *op*. The following constraints are generated:

- SILP: There must be exactly one operation that passes an instance of one of the resource types $\{r_1, \dots, r_k\}$ to operation j.

$$\sum_{r \in \{r_1,\dots,r_k\}} \sum_{(m,j,r) \in E_F} x^r_{mj} = 1$$

- OASIC: The execution of operation j must be started by exactly one instance of one of the resource types $\{r_1, \dots, r_k\}$.

$$\sum_{r \in \{r_1,\dots,r_k\}} \sum_{n \in N(j)} x^r_{jn} = 1$$

(i.6) The expression $\neg$ (*op.exec in* $\{r_1, \dots, r_k\}$) prevents any of the resources $r_1, \dots, r_k$ to be used as the execution unit for *op*. The following constraints are generated:

- SILP: There must be no operation that passes an instance of any of the resource types $\{r_1, \dots, r_k\}$ to operation j.

$$\sum_{r \in \{r_1,\dots,r_k\}} \sum_{(m,j,r) \in E_F} x^r_{mj} = 0$$

- OASIC: The execution of operation j must not be started by an instance of any of the resource types $\{r_1, \ldots, r_k\}$.

$$\sum_{r \in \{r_1,\ldots,r_k\}} \sum_{n \in N(j)} x_{jn}^r = 0$$

(ii) op_1 && op_2

Let op_1 be bound to operation i and op_2 be bound to operation j in the current iteration.

$(ii.1)$ The expression op_1 && op_2 forces the operations i and j to be executed in parallel. This is modelled by the following constraint:

$$t_i - t_j = 0$$

The same constraint is valid for the SILP and the OASIC formulation since in both cases the starting point for the execution of an operation is denoted as t_i.

$(ii.2)$ The expression $\neg$ (op_1 && op_2) prevents the operations i and j from being executed in parallel. In the SILP model an integer linear constraint has to be generated that is equivalent to

$$t_i \neq t_j \;\equiv\; (t_i < t_j) \vee (t_i > t_j)$$

Disjunctive constraints are formulated with the help of binary variables. This mechanism is described in 5.4.1 on page 76.

In the OASIC formulation no disjunctive constraints are necessary; instead the following constraints are generated:

$$\sum_{k:(k,i) \in E_R} x_{in}^k + \sum_{k:(k,j) \in E_R} x_{jn}^k \leq 1 \;\forall\; n \in N(i) \cap N(j)$$

(iii) op_1 −> (n) op_2

Let op_1 be bound to operation i and op_2 be bound to operation j in the current iteration.

$(iii.1)$ The expression op_2 −> (n) op_1 forces the the execution of j to be started exactly n control steps after the starting of operation i. The following constraint is generated:

$$t_j = t_i + n$$

The same constraint is valid for the SILP and the OASIC formulation since in both cases the starting point for the execution of an operation i is denoted as t_i.

($iii.2$) The expression $\neg\ (op_2 \text{ -> } (n)\ op_1)$ prevents the execution of j to be started n cycles after starting the execution of i. Here an integer linear constraint has to be generated that is equivalent to

$$t_j \neq t_i + n \ \equiv \ (t_j < t_i + n) \vee (t_j > t_i + n)$$

Disjunctive constraints can be formulated with the help of binary variables. This mechanism is described in Sec. 5.4.1 on page 76.

(iv) $op_1 \text{ -}\gg (n)\ op_2$

Let op_1 be bound to operation i and op_2 be bound to operation j in the current iteration.

($iv.1$) The expression $op_2 \text{ -}\gg (n)\ op_1$ forces the the execution of operation j to be started at least n control steps after the starting of operation i. The following constraint is generated:

$$t_j \geq t_i + n$$

The same constraint is valid for the SILP and the OASIC formulation.

($iii.2$) The expression $\neg(op_2 \text{ -}\gg (n)\ op_1)$ forces the execution of j to be started earlier than n cycles after starting the execution of i. The following constraint is generated in the SILP and the OASIC formulation:

$$t_j \leq t_i + n - 1$$

(v) $op_1.attribref_1 == op_2.attribref_2$

The atomic expressions $op_1.attribref_1 == op_2.attribref_2$ and $\neg\ (op_1.attribref_1\ != op_2.attribref_2)$ are equivalent so that they can be handled in the same way. Let i denote the operation bound to op_1 in the current iteration and let j denote the operation bound to op_2. If the attribute references denote operands of the instructions, i. e., $attribref_1, attribref_2 \in \{src_\nu, dst_\nu | \nu \in \mathbb{N}_0\}$ and no register assignment is performed, constant integer linear constraints are generated similarly to (i). Again it has to be distinguished between operand references and references to execution units.

First we consider the case that the attribute reference denotes a source or destination operand. If the task of register assignment is addressed, the SILP formulation must be used. If the attribute denotes a source operand, a defining operation has to be determined similarly to (i). Define $d_i = i$ if $attribref_1 = dst_\nu$, and $d_i = d$ if $attribref_1 = src_\nu$ where d is a definition reaching the use of $attribref_1$ in i. Let d_j be defined analogously for j and $attribref_2$. Then it is necessary to guarantee that d_i and d_j both refer to the same physical register. In the SILP formulation this can only be done by generating path constraints as explained in Sec. 7.2.4. Define $R = \{r | (d_i, r) \in$

$E_A \wedge (d_j, r) \in E_A\}$. Let k paths between i and j be given and let P be the set of these paths, $|P| = k$. Furthermore, let o be a unique number identifying the operation pair (i, j). Then for each path $p_m \in P, 1 \leq m \leq k$ of length l the following constraint is generated:

$$\sum_{r \in R} x^r_{k_1 k_2} + \sum_{r \in R} x^r_{k_2 k_3} + \cdots + \sum_{r \in R} x_{k_{l-1} k_l}{}^r + l c^o_m \geq l$$

Additionally a constraint is required which forces exactly one of these paths to be taken:

$$\sum_{m=1}^{k} c^o_m = k - 1$$

The c-variables have to be binary integers, i. e. $c^o_m \in \{0, 1\} \;\; \forall m \; \forall o$.

If no upper bound on the length of path constraints has been specified this can lead to an exponential number of additional constraints.

Now let the attribute references denote the functional unit type used to execute the operations i and j. The expression $op_1.exec == op_2.exec$ ($\neg$ ($op_1.exec$ $!= op_2.exec$)) means that the functional unit type used to execute op_1 and op_2 must be the same[1]. Let R be the set of all functional units r that can be used to execute i and j. Different constraints have to be generated for the SILP and the OASIC formulation:

- SILP: Both operations must receive an instance of the same resource type. This is guaranteed by the following constraint:

$$\Phi^r_i - \Phi^r_j = 0 \quad \forall \; r \in R$$

- OASIC: The execution of operations i and j must be started by an instance of the same resource type. This is guaranteed by the following constraint:

$$\sum_{n \in N(i)} x^r_{ni} - \sum_{n \in N(j)} x^r_{nj} = 0 \quad \forall \; r \in R$$

(vi) $op_1.attribref != op_2.attribref$

Again let i denote the operation bound to op_1 in the current iteration and let j denote the operation bound to op_2. If the attribute references denote operands of the instructions, i. e., $attribref_1, attribref_2 \in \{src_\nu, dst_\nu | \nu \in \mathbb{N}_0\}$ and no register assignment is performed, constant integer linear constraints are generated similarly to (v).

In the following we assume that the task of register assignment is addressed and that the attribute references denote a source or destination operand. Let

[1] Note that (in contrast to the usage for registers) this does not necessarily mean that the same instance of a resource type is used to execute i and j.

R, d_i and d_j be defined as in (v). Then it is necessary to prevent d_i and d_j from referring to the same physical register. In the SILP formulation this can be done as explained in Sec. 7.5. Let $p = (i = k_1 \rightarrow \cdots \rightarrow k_m = j)$ be a path from i to j in the register flow graph. Then at least one edge must be inactive, i.e. $\exists\, o \in \{1, \cdots, m-1\} : x^r_{o,o+1} = 0$. Let $l(p)$ denote the length of path p, then the following constraint is generated for each path p:

$$\sum_{(i,j,r)\in p} x^r_{ij} < l(p) \quad \forall r \in N^A_R$$

Again the number of constraints to be generated may be exponential, but can be bounded by the methods presented in Sec. 7.2.4.

Now let the attribute references denote the functional unit type used to execute the operations i and j. The expression $op_1.exec != op_2.exec$ ($\neg\ (op_1.exec == op_2.exec)$) means that the functional unit type used to execute op_1 and op_2 must not be the same. Let R be the set of all functional units r that can be used to execute i and j. Different constraints have to be generated for the SILP and the OASIC formulation:

- SILP: It is excluded that both operations receive an instance of the same resource type. This is guaranteed by the following constraint:

$$\Phi^r_i + \Phi^r_j \leq 1 \quad \forall\ r \in R$$

- OASIC: The execution of operations i and j must not be started by instances of the same resource type. This is guaranteed by the following constraint:

$$\sum_{n\in N(i)} x^r_{in} + \sum_{n\in N(j)} x^r_{jn} \leq 1 \quad \forall\ r \in R$$

Induction Step Since the boolean expressions are given in disjunctive normal form, it is sufficient to analyse the boolean operators $\vee$ and $\wedge$. Assume that for each of the boolean expressions $e_1, \ldots, e_k$ a set $\mathcal{C}(e_1)$ of equivalent integer linear constraints has already been generated.

(i) Consider the boolean expression $e = e_1 \wedge \cdots \wedge e_k$. Then a set of integer linear constraints equivalent to e is

$$\bigcup_{i=1}^{k} \mathcal{C}(e_i)$$

since all constraints must be satisfied.

(ii) Consider the boolean expression $e = e_1 \vee \cdots \vee e_k$. In that case, the constraints in $\mathcal{C}(e_1), \ldots \mathcal{C}(e_k)$ have to be modified. The mechanism to represent disjunctive constraints as integer linear constraints has been presented in Sec. 5.4.1. Let

$c_1(e_m), \ldots, c_{n_m}(e_m)$ be the constraints contained in $\mathcal{C}(e_m)$, let $c^l_j(e_m)$ denote the left hand side of constraint $c_j(e_m)$, and $c^r_j(e_m)$ the right hand side of $c_j(e_m)$. Then, the resulting constraints read as follows:

$$
\begin{array}{rcl}
c^l_1(e_1) & \leq & c^r_1(e_1) + My_{\alpha_1} \\
 & \vdots & \\
c^l_{n_1}(e_1) & \leq & c^r_{n_1}(e_1) + My_{\alpha_1} \\
c^l_1(e_2) & \leq & c^r_1(e_2) + My_{\alpha_2} \\
 & \vdots & \\
c^l_{n_2}(e_2) & \leq & c^r_{n_2}(e_2) + My_{\alpha_2} \\
 & \vdots & \\
c^l_1(e_k) & \leq & c^r_1(e_k) + My_{\alpha_k} \\
 & \vdots & \\
c^l_{n_k}(e_k) & \leq & c^r_{n_k}(e_k) + My_{\alpha_k} \\
\sum_{i=1}^{k} y_{\alpha_i} & \leq & k - 1
\end{array}
$$

At least one of the constraint sets $\mathcal{C}(e_1), \ldots \mathcal{C}(e_k)$ has to be satisfied.

$\square$

8.4.2. Generating Support Functions for List Scheduling

An overview of the list scheduling algorithm has been given in Sec. 2.2.3. The PROPAN framework offers a generic implementation of the list scheduling with highest-level first heuristic.

The generation of the list scheduling support functions from the TDL description of the target processor is straightforward. The boolean expressions are directly translated into C code. In the generic list scheduling algorithm an operation from the data ready list is tentatively scheduled in the current instruction. Subsequently the generated support functions are invoked to check whether this leads to a violation of a condition specified in the constraint section of the TDL description. If the condition of the premise of a rule is fulfilled, the condition of the consequent is checked. In order for the operation to be schedulable to the control step corresponding to the current instruction, no constraints must be violated, i. e., all generated functions must return `false`. If the scheduling is not correct, the tentative scheduling of the current operation is undone. This allows the specified resource and encoding restrictions to be incorporated into the list scheduling algorithm in a generic way. The generated functions have been designed for simplicity; more efficient implementations are possible, but as the experimental results show, the computation times required for the list scheduling in the current implementation are always very low. An example of a generated support function is shown in Fig. 8.7.

```
op1 in OpClass1 & op2 in OpClass2:  !(op1 && op2)
```

$\Downarrow$

```
int ConstrFun (/*InputOps is an array with n entries containing
                 representations of all operations of the input
                 program; cstep is the current control step. */)
{
  /* Declarations */
  for (i=0; i<n; i++) {
    op1 = InputOp[i];
    for (j=0; j<n; j++) {
      op2 = InputOp[j];
      if (OpContainedInOpClass(op1, "OpClass1") &&
          OpContainedInOpClass(op2, "OpClass2")) {
        if ((op1.slot == cstep)&&(op2.slot == cstep))
          return 1;
      }
    }
  }
  return 0;
}
```

Figure 8.7.: Example of a rule and the generated list scheduling support function.

8.5. The Assembly Section

The assembly section deals with syntactic details of the assembly language such as instruction or operation delimiters, assembly directives, etc. In order to perform semantic-preserving transformations of an assembly program it must be possible to distinguish between machine operations and assembly directives. Among the existing machine specification languages, the only approach that supports this distinction is the description language of the SALTO system [BCRS97]. While the specification of the syntax of assembly directives would also be possible in other languages as, e. g., nML [FVPF95] those could not be distinguished from regular machine operations. In the SALTO system, the user has to provide dedicated C++-classes for parsing and interpreting the assembly directives. In TDL, similarly to the specification of the instruction set, the language of the assembly directives is given in the form of an attribute grammar. Again there is a set of predefined attributes, each with a dedicated domain. The predefined domains can be extended by the user, and new attributes with explicitly specified domains can be declared. The assembly parser generated from the TDL description reads the assembly directives and builds an internal representation that contains all specified information. The user is responsible for the correct interpretation of the assembly directives via a dedicated interface to include user-defined C-functions. An example of a simple directive declaration is shown in the following; more details can be found in [Käs99b].

```
DefineDirective DirGlobVar ".global %s;"
  {type = GlobalDecl, name="$1"};
```

9. The Implementation of the PROPAN Framework

In this chapter an overview of the implementation of PROPAN is given. First the structure of the PROPAN system is explained, then the generic algorithms for computing the program representations are presented. The generic list scheduling and resource allocation algorithm is outlined in Sec. 9.3 and Sec. 9.4 presents the algorithm for computing the *asap* and *alap* control steps. In Sec. 9.5 the most important command-line parameters of the generated optimisers are summarised.

9.1. The Structure of PROPAN

An overview of the implementation of the PROPAN system is given in Fig. 9.1. For each target architecture a TDL specification has to be developed. From this machine description, the TDL parser generates the assembly parser for the specified target architecture and the architectural database that consists of ANSI-C files providing the hardware-specific information to the PROPAN core system. The PROPAN core system comprises the generic functions for computing the program representations, generating the integer linear programs and solving the specified optimisation problem. For each target architecture the architectural database is linked with the PROPAN core system yielding a dedicated hardware-sensitive optimiser. The core system of PROPAN can be extended by additional user-supplied optimisation and analysis algorithms. These can be invoked before or after the phase-coupled optimisations using the CRL language as central interface.

The assembly parser reads the input programs and calculates their control flow graph that is represented in the generic CRL language [Lan99]. The input format is not restricted to assembly files; it is also possible to specify the output format of disassemblers reading executable files, or textual representations of compiler-specific intermediate formats. From the control flow graph, the data and control dependence graphs are computed by generic algorithms. If required, register renaming is performed and the data dependence information is updated. Then the input routines are covered by superblocks and the *asap* and *alap* control steps which are needed to generate the integer linear programs are computed for each operation. For each superblock, an individual integer linear program is generated that models the code generation problems to be considered. Either the SILP, or

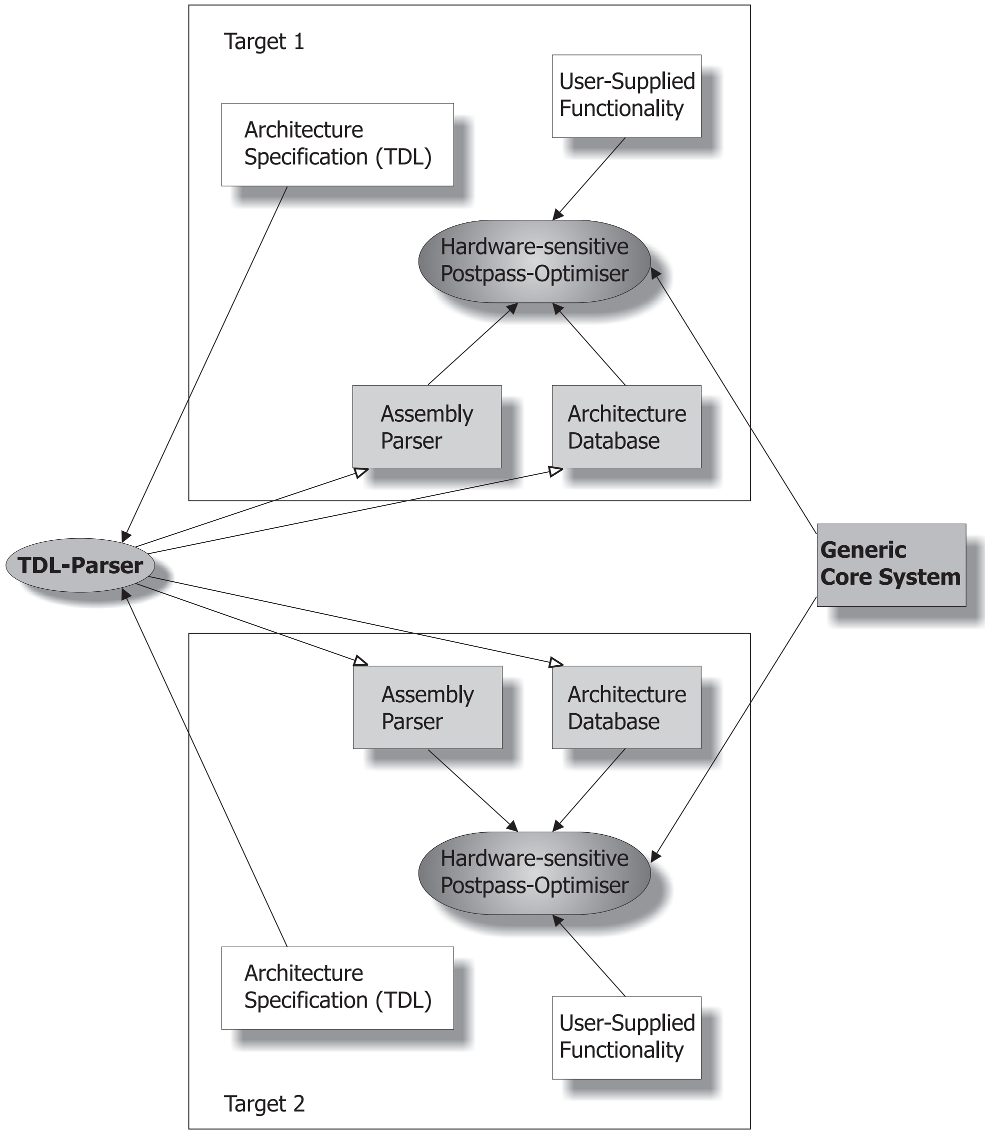


Figure 9.1.: The implementation structure.

the OASIC modelling is used, depending on command-line parameters of the postpass optimiser. If register reassignment is part of the optimisation problem, the SILP model has to be used. As an alternative to the ILP-based methods, a generic list scheduling algorithm that performs resource allocation on the fly is available. If ILP-based methods are chosen, the solutions of the integer linear programs are interpreted to build the optimised assembly code. The output of the postpass optimiser is the transformed assembly file.

9.2. Computing the Program Representations

All required program representations are calculated in a generic way: the control flow graph, the data dependence graph, the control dependence graph and the superblock graph. The necessary information about the target architecture is provided by the data structures and functions generated from the TDL-description. Each input program can consist of several routines or functions; those are detected during the reconstruction of the control flow graph. The program representations are calculated individually for each routine of the input program. PROPAN offers an interface to the graphical visualisation system `aiSee` [San94, San96, San99, Abs00b]. Visualisations are available for all program representations, the instruction set of the target architecture, and the resource and register flows computed by the ILP optimisation phases. While the algorithms involved in the computation of the superblock graph have been detailed in Chap. 7, the algorithms used to generate the control flow graph, the data dependence graph and the control dependence graph are presented in the remainder of this section.

9.2.1. The Control Flow Graph

The control flow graph (see Sec. 2) is at the base of most code generation and optimisation phases and constitutes the starting point of the flow-sensitive program analyses [Mar99]. The process of calculating the control flow graph in a postpass framework is different from the situation in a high-level language compiler. When generating code for a high-level language program, the control flow usually is represented explicitly in the input program. In assembly programs there are no high-level statements; as an example loops are mostly represented as a sequence of branch instructions. Therefore dedicated algorithms are required in order to reconstruct the control flow from the sequence of machine operations.

The reconstruction of the control flow graph requires information about the instruction set of the target processor. The parser generated from the TDL-description calculates a generic representation of the input program in a dedicated control flow representation language CRL (*Control Flow Representation Language*) [Lan99]. First the machine operations of the input program are recognised and attribute values that depend on each individual operation instance as, e. g., the storage locations of source and destination operands are listed in the form of key-value pairs. The output of this phase is a sequential list containing the generic representation of each operation in the input program. This operation list is the input of the control flow reconstruction algorithm that represents the second phase of the assembly parser.

A detailed description of the control flow reconstruction process is given in [The00]; here only a short overview is presented. The operation list produced by the parser can be considered as a single basic block. This basic block is stepwise split at control flow operations and the new basic blocks are connected by control flow edges according to the type of the operations. In the TDL specification of the target machine the type of each operation is declared; there are dedicated types for explicit or computed branches, calls, returns, etc. The control flow reconstruction algorithm currently is based on the values of this type classification; where this is not sufficient to compute the control flow, user-supplied target-dependent functions can be invoked by a dedicated interface. However there is ongoing work to develop a fully generic control flow reconstruction mechanism where the effect of each operation on the control flow is reconstructed from the specification of the operation semantics. The last stage of the control flow reconstruction is the detection of loops that is based on the algorithm presented in [LT79].

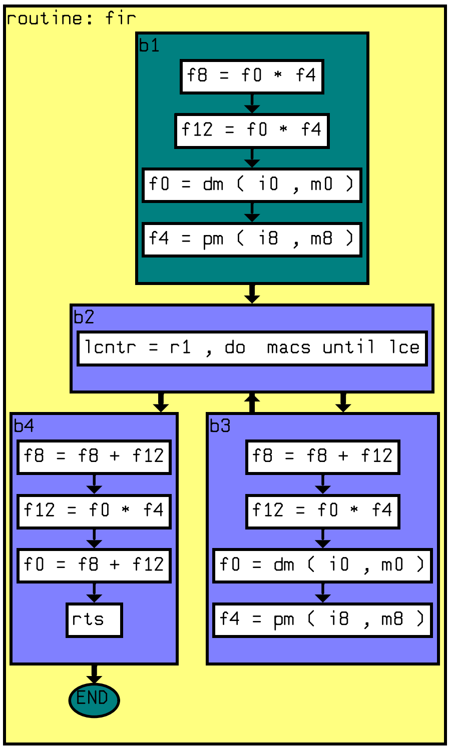


Figure 9.2.: Visualisation of a control flow graph.

9.2.2. The Data Dependence Graph

Since the scope of the optimisations of the PROPAN system is not restricted to basic block level, it is necessary to compute a global data dependence graph. The data dependence information is computed against the control flow sense by a two-stage algorithm. In the first stage for each basic block b the resource accesses from operations of other basic blocks are registered that can induce data dependences with operations from b. In the second stage the operations of each basic block are traversed and the data dependences are registered. The first stage is based on two backward data flow analyses: the *analysis of exposed definitions* (def-use chains) and the *analysis of active uses* [ASU86]. The two-stage approach allows the data flow analyses to be performed on basic block rather than microoperation level which leads to a reduction of computation time. In the following we will use the term of symbolic variables to denote the storage locations accessed by the operations of the input program. For each symbolic variable v the analysis of exposed definitions computes the set of all definitions of v that can be reached on a program path starting from program point p without other intermediate definitions of v. Similarly the analysis of active uses computes for each symbolic variable v the set of all uses of v that are upwards exposed, i. e. that can be reached on a path from program point p without intermediate definitions of v. The basic idea of data flow algorithms is to merge information about different control flow paths at control flow joins; this way the exponential computation time for analysing each path separately can be avoided (see also the more general framework of abstract interpretation) [ASU86, WM95, NNH99].

In order to compute the data dependences of an assembly program it is necessary to distinguish between microoperations and instructions. We assume that each instruction J is composed of a set of microoperations $j_1, \ldots, j_k \in J$ whose execution is started in parallel. If there is no instruction-level parallelism, each instruction contains exactly one microoperation. Let $Def(j)$ be the set of all storage resources modified by operation j, $Use(j)$ the set of all storage resources whose contents are read by operation j and let N_I be the set of all operations of the input program. Similarly we define for an instruction J $Def_I(J)$ as the set of all storage resources modified by any operation contained in J, and $Use_I(J)$ as the set of all storage resources whose contents are read by operations contained in J. The ordering of the instructions of each basic block b induces a relation $\prec_b \subseteq N_I \times N_I$ where $i \prec_b j \Leftrightarrow i, j \in N_I^b \wedge I \xrightarrow[G_C]{+} J$ where I is the instruction containing operation i and J the instruction containing j.

For a program point ξ the dataflow analyses maintain two sets $D(\xi)$ and $U(\xi)$ that are defined as follows:

- $D(\xi)$ contains all definitions exposed to program point ξ. It is the set of all pairs $(i, r) \in N_I \times \mathcal{R}$ such that there is a path p starting from program point ξ on which i is the first operation to define resource r. Let J_1 be the instruction immediately following program point ξ; then $D(\xi)$ can be formally defined as

$D(\xi) = \{(i,r) \mid \exists p = (J_1, \ldots J_k, I) : i \in I \wedge r \in \mathit{Def}(i) \wedge r \notin \mathit{Def}_I(J_l), l = 1, \ldots, k\}$.

- $U(\xi)$ contains all active uses visible from program point ξ. It is the set of all pairs $(i,r) \in N_I \times \mathcal{R}$ such that i uses resource r and there is a path p from ξ to i that does not contain a definition of r. Let J_1 be the instruction immediately following program point ξ; then $U(\xi)$ can be formally defined as

  $U(\xi) = \{(i,r) \mid \exists p = (J_1, \ldots J_k, I) : i \in I \wedge r \in \mathit{Use}(i) \wedge r \notin \mathit{Def}_I(J_l), l = 1, \ldots, k\}$.

Let a basic block b with n instructions be given and let N_I^b be the set of micro-operations of b. The entry point of b is denoted p_0^b, the program point between two instructions i and $i+1$ is denoted p_i^b and the exit point is denoted p_n^b. The dataflow analyses of exposed definitions and active uses compute the sets $D(p_n^b)$ and $U(p_n^b)$ for the exit points p_n^b of all basic blocks b in the program. As described in [ASU86], dataflow information can be calculated by formulating and solving a system of recursive equations which correlate information about different program points. For the analyses of exposed definitions and active uses, the dataflow equations have the following form:

$$\begin{aligned} in(b) &= gen(b) \cup (out(b) - kill(b)) \\ out(b) &= \bigcup_{b \xrightarrow[G_B]{*} b'} in(b') \end{aligned}$$

The meaning of these equations can be described as follows: The information at the entry of a basic block b is either generated inside of b, or it is available at the exit of b and has not been destroyed inside of b. The information at the exit of b corresponds to the union of the information at the entries of all successors b' of b in the basic block graph G_B.

The *gen* and *kill* sets for the analyses of exposed definitions ($gen_D, kill_D$) and active uses ($gen_U, kill_U$) are defined as follows:

- $gen_D(b)$ is the set of all pairs (i,r) such that i is the first operation of basic block b to define resource r.

$$gen_D(b) = \{(i,r) \in N_I^b \times \mathcal{R} \mid r \in \mathit{Def}(i) \wedge \nexists j \in N_I^b : j \prec_b i \wedge r \in \mathit{Def}(j)\}$$

- $gen_U(b)$ is the set of all pairs (i,r) such that i uses resource r and there is no definition of r in b which precedes i.

$$gen_U(b) = \{(i,r) \in N_I^b \times \mathcal{R} \mid r \in Use(i) \wedge \nexists j \in N_I^b : (j \prec_b i \wedge r \in Def(j))\}$$

- $kill_D(b)$ is the set of all pairs $(i,r) \in N_I \times \mathcal{R}$ such that i is an operation that defines resource r and does not belong to b and there is a definition j of r in b.

$$kill_D(b) = \{(i,r) \in (N_I - N_I^b) \times \mathcal{R} \mid r \in Def(i) \wedge \exists j \in N_I^b : r \in Def(j)\}$$

- $kill_U(b)$ is the set of all pairs $(i,r) \in N_I \times \mathcal{R}$ such that i is an operation that uses resource r and does not belong to b and there is a definition j of r in b.

$$kill_U(b) = \{(i,r) \in (N_I - N_I^b) \times \mathcal{R} \mid r \in Use(i) \wedge \exists j \in N_I^b : r \in Def(j)\}$$

The solution of this system of recursive equations is determined by a fixed point computation based on a workset algorithm [Mar99]. The exit point p_n^b of a basic block b is represented by a node n_b. A node n_b of a basic block b will be contained in the workset if for one of its successors b' the value $out(n_{b'})$ has changed since the last computation of $out(n_b)$. In each iteration a node n_b is selected and removed from the workset and a new value for n_b is computed. Then it is checked whether this value has changed since the last computation. If it has changed, the value is propagated again to all predecessors of b by entering them into the workset. The iteration ends when no more nodes are contained in the workset. The ordering of the nodes in each workset is based on a topological ordering of the strongly connected components, i. e., of the loops of the basic block graph. The nodes within each strongly connected component are sorted in the order of the inverse reduced transitive hull of the basic block graph.

After the fixed point computation has terminated the data dependences are computed by an extended version of the algorithm of [WM95]. The algorithm is shown in Fig. 9.3 in pseudocode. The input of the algorithm consists of the basic blocks of the input program that are associated with the information about exposed definitions and active uses at their exit point. Each instruction of the input program can be composed of several microoperations whose execution is started from the same machine state. The instructions of each basic block are traversed in backward direction. The algorithm traverses all microoperations of the current instruction and registers the data dependences to the operations of previously processed instructions and other basic blocks. After all operations of the current instruction have been traversed, the data dependences among them are registered and only then the machine state is updated. Then the next instruction is addressed and the algorithm iterates until all instructions of the basic block have been traversed.

The function $RegisterInterInstructionDeps(i, r, D(p_i), U(p_i))$ registers all dependences of an operation i with respect to a resource r according to the definition

```
procedure CalculateBBDDG (Basic block b)
{
    ExpDefs := D(p_n^b);
    ActUses := U(p_n^b);
    foreach instruction ins of b in backward direction {
        ED_ins := ∅;
        AU_ins := ∅;
        foreach operation i of ins {
            ED_ins := ED_ins ∪ {(i,r) ∈ N_I × R | (i,r) ∈ Def(i)};
            AU_ins := AU_ins ∪ {(i,r) ∈ N_I × R | (i,r) ∈ Use(i)};
            foreach r ∈ Def(i) ∪ Use(i)
                RegisterInterInstructionDeps (i, r, ExpDefs, ActUses);
            }
        }
        RegisterIntraInstructionDeps (ins, ED_ins, AU_ins);
        ExpDefs := ExpDefs − {(j,r) ∈ N_I × R | (j,r) ∈ ExpDefs ∧
                    ∃(i,r) ∈ ED_ins} ∪ ED_ins;
        ActUses := ActUses − {(j,r) ∈ N_I × R | (j,r) ∈ ActUses ∧
                    ∃(i,r) ∈ ED_ins} ∪ AU_ins;
    }
}
```

Figure 9.3.: Computation of the Data Dependences.

of the data dependence graph in Sec. 2.1. The parameters $D(p_i)$ and $U(p_i)$ represent the sets of last definitions and exposed uses at the program point immediately following the instruction containing i. The following cases are distinguished:

- There is a true dependence (i, j, r, t) from i to j with respect to r if i defines r and j is an active use of r $((j, r) \in U(p_i))$.
- There is an anti dependence (i, j, r, a) from i to j with respect to r, if i uses r and j is an exposed definition of r $((j, r) \in D(p_i))$.
- There is an output dependence (i, j, r, o) from i to j with respect to r, if i defines r, j is an exposed definition of r $((j, r) \in D(p_i))$ and there is no active use of r $(\nexists(k, r) \in U(p_i))$ between the instructions containing i and j.

We assume that between operations of the same instruction there are no true and no output dependences. There is an intra-instruction anti dependence (i, j, r, a) from i to j with respect to r, if i uses r, j defines r and both operations are contained in the same instruction.

The worst-case time complexity of the algorithm of Fig. 9.3 is $\mathcal{O}(n_B \cdot n_m^2)$ where n_B is the number of basic blocks in the input program and n_m is the maximal number of operations per basic block. The algorithm does not differentiate between data dependences in control flow sense and loop-carried data dependences. The detection of loop-carried dependences is addressed in a subsequent pass over the operations of the input program. Each dependence $(i, j, r, \tau) \in E_D$ where the instruction containing i is no predecessor of the instruction containing j in the reduced transitive hull of the control flow graph ($(I, J) \notin E_C^+$) is a loop-carried dependence. In a last step the transitive closure of the data dependence graph is computed [Käs97].

The modelling of data dependences with respect to memory accesses is conservative; each memory access is assumed to access the same memory cell, so there are dependences between each memory access. The set of dependences can be reduced by analysing the memory accesses in the program; if two accesses can be guaranteed to access different memory cells, no dependences are registered [ZABT00]. There is ongoing work to integrate this refinement of the data dependence analysis into the PROPAN-framework. The basic idea is to perform a value analysis for the address registers of the target machine in order to statically compute the address of a memory reference. This way it is possible to disambiguate memory accesses on assembly level. Other extensions are required in the context of predicated execution [PS91, DT93]. In some architectures, the execution of the operations depends on the values of explicitly specified registers. Then it is possible for operations that are not control equivalent to be contained in the same instruction, and operations contained in consecutive instructions are not necessarily executed consecutively. For those architectures the data dependence analysis presented above and the life range modelling of Sec. 7.2.4 have to be extended. A value analysis of the predicate registers can be used to detect operations from the same basic block that are not control equivalent. Based on the result of this analysis the control flow graph is refined and the additional control flow information is considered when calculating the data dependences. This is also subject of ongoing work.

9.2.3. The Control Dependence Graph

In order to decide whether code movements between basic blocks are feasible without the necessity of inserting compensation code the control dependences of the input program have to be known. In [FOW87] an algorithm for computing the control dependence graph is presented that is summarised in the following.

Let a basic block graph $G_B = (N_B, E_B, b_A, b_\Omega)$ be given. First the set $P(b)$ of the postdominators for each node $b \in N_B$ is calculated by the algorithm shown in Fig. 9.4 [ASU86]. Since the set $P(b)$ of line (*) is a subset of the set $P(b)$ from the previous loop iteration the algorithm is guaranteed to terminate.

Let F be the set of all edges (b_i, b_j) in the basic block graph such that b_j does not postdominate b_i. The algorithm for computing the control dependences traverses all edges $(b_i, b_j) \in F$. Let l be the earliest common predecessor of b_i and b_j in the postdominator tree. Then two cases can be distinguished:

- l is a predecessor of b_i in the postdominator tree. Then all nodes in the postdominator tree on the path from l to b_j are control dependent on b_i except for l.

procedure *PostDominatorTree* (G_C)
{
    $P(b_\Omega) := \{b_\Omega\};$
    foreach $b \in N_B - \{b_\Omega\}$ {
        $P(b) := N_B;$
    }
    while there are changes in any $P(b)${
        foreach $b \in N_B - \{b_\Omega\}$ {
            Let S be the set of all successors of b.
            $P(b) := \bigcap_{s \in S} P(s) \cup \{b\};$ ($*$)
        }
    }
    foreach $b \in N_B$ {
        $P(b) := P(b) - \{b\};$
    }
}

Figure 9.4.: Computing the Postdominator Tree.

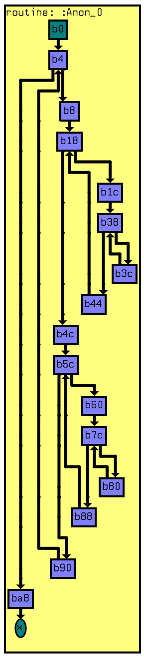


Figure 9.5.: Visualisation of a basic block graph.

- $l = b_i$: All nodes in the postdominator tree on the path from b_i to b_j including b_i and b_j are control dependent on b_i. This way loop dependences are taken into account.

For each edge (b_i, b_j) the algorithm traverses the postdominator tree starting from b_j backwards until a predecessor p of b_i is reached. All nodes visited before reaching p are marked as control dependent on b_i. If l is a predecessor of b_i in the postdominator tree, b_i is not contained in the path to l and is not marked. The time complexity of this algorithm is $\mathcal{O}(n^2)$ where n is the number of nodes of the basic block graph [FOW87].

A basic block graph is shown in Fig. 9.5 and Fig. 9.6 shows the corresponding control dependence graph.

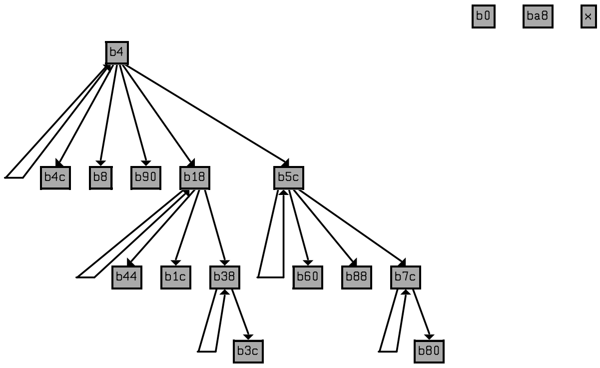


Figure 9.6.: Visualisation of a control dependence graph.

9.3. Generic List Scheduling and Resource Allocation

In order to be able to compare the ILP-based optimisations to a conventional graph-based approach, a generic list scheduling algorithm has been implemented in the PROPAN system. The list scheduling uses the highest level first heuristic and has been extended to perform resource allocation on the fly. In consequence the algorithm allows a heuristic coupling of scheduling and allocation decisions. The algorithm is shown in pseudocode in Fig. 9.7.

The basic blocks of the input program are traversed in the order of decreasing execution frequency; the operations of each block are scheduled by the function *ListScheduleBlock*. First a priority is calculated for each operation of the current basic block using the highest-level-first heuristic [Bea91]. It is assumed that one instruction is issued per control step. The data ready set *drs* contains all operations whose operands are available in the current control step; it is maintained by the function *UpdateDRS*. The algorithm starts with an empty list of microoperations and iterates until all operations of the block have been scheduled. Each

```
procedure ListScheduleBlock(Basic block b, CFG c)
{
    CalculateOperationPriorities(b);
    insnlist := ∅;
    drs := ∅;
    UpdateDRS(b, drs);
    while (not all operations of b have been scheduled) {
        cinsn := AllocateInstruction();
        foreach operation i ∈ drs {
            r := SelectAvailRes(i, cinsn, b);
            if (r ≠ null) {
                Assign i to hardware resource r;
                Append i to cinsn;
            }
        }
        if (cinsn = ∅)
            Append nop to cinsn;
        UpdateDRS(b, drs);
        Append cinsn to insnlist;
    }
    AdjustExit(b, c);
    FillDelaySlots(b);
}
```

Figure 9.7.: Computation of the Data Dependences.

iteration starts with an empty instruction. The operations of the data ready set are traversed in highest-level-first order and for each operation i the function *SelectAvailRes* is called. This function checks whether there is an available execution resource to which i can be assigned. The latencies of the functional units and the resource constraints of write-back buses have to be respected. *SelectAvailRes* also checks whether appending operation i to the current instruction leads to violations of the conditions specified in the constraint section of the TDL description. For this purpose the generated support functions described in Sec. 8.4.2 are used. If there is an available hardware resource r to which i can be assigned such that the insertion of i into the current instruction is possible, *SelectAvailRes* returns r, otherwise it returns `null`. If the insertion is possible, the selected resource r is registered for i, and i is appended to the current instruction *cinsn*. Then the data ready set is updated, the current instruction is appended to the instruction list and a new instruction is allocated. After all operations have been scheduled it is checked whether additional *nops* have to be executed before leaving the basic block. This can be caused by inter-iteration data dependences and inter-iteration resource conflicts. Moreover data dependences and resource conflicts with previously scheduled successor blocks of b have to be addressed (cf. Sec. 7.3). The basic algorithm always schedules control flow operations to the end of the basic blocks and fills delay slots with *nops*. The function *FillDelaySlots* attempts to move instructions of the blocks into the delay slots of the branch operations. The worst case complexity of the algorithm is $\mathcal{O}(n^2)$ where n is the number of operations in the input program.

9.4. Computing the ASAP and ALAP Control Steps

Generating the integer linear programs in the SILP and the OASIC formulations requires an interval to be calculated for each operation i that contains all control steps in which the execution of i can be started in any feasible schedule. This interval is defined as $N(i) = \{asap(i), \dots alap(i)\}$ [Fou81]. The computation of the *asap* and *alap* control steps is partitioned into two phases. First for each operation i a preliminary value $s(i)$ of $asap(i)$ and a preliminary value $l(i)$ of $alap(i)$ are determined based on computing longest paths in the data dependence graph. The second phase tries to improve these bounds by taking the number of available hardware resources into account. In our implementation the longest paths are computed by the Bellman Ford Algorithm [CLR90]; it is shown in Fig. 9.8 in pseudocode.

Let a weighted graph $G = (N, E)$ with a set $S \subseteq N$ of start nodes be given. The algorithm computes the longest paths from the nodes in S to all nodes $n \in N$. The weight of an edge $(u, v) \in E$ is denoted $w(u, v)$. At the end of the algorithm, $d(n)$ represents the length of the longest path from any node in S to n and $\Pi(n)$ is the immediate predecessor of n on this path. The function *Relax* checks for each edge (u, v) if the largest distance $d(v)$ found so far can be improved by considering

```
procedure LongestPaths(G = (N, E), S)
{
  foreach (n ∈ N) {
    d(n) := −∞;
    Π(n) := null;
  }
  foreach (n ∈ S)
    d(n) := 0;
  repeat |N| − 1 times {
    foreach (u, v) ∈ E
      Relax(u, v);
  }
  foreach (u, v) ∈ E
    if (d(v) < d(u) + w(u, v))
      exit(Failure);
}

procedure Relax(u, v)
{
  if (d(v) < d(u) + w(u, v)) {
    d(v) := d(u) + w(u, v);
    Π(v) := u;
  }
}
```

Figure 9.8.: The Bellman Ford Algorithm.

the path consisting of the longest path to u and the edge (u, v). If there are cycles in the input graph, they are detected and the algorithm reports an error. The worst case time complexity is $\mathcal{O}(|N| \cdot |E|)$.

The value $s(i)$ for an operation i is calculated as the length of the longest path to i in the data dependence graph that starts from an operation with no dependence predecessors. The value $l(i)$ is computed by inverting the edges of the data dependence graph and determining the longest path in the inverse graph starting from an operation without data dependence successors. For each operation i the computed distance $d(i)$ is subtracted from an upper bound U of the execution time of the input program yielding the value $l(i) = U - d(i)$.

The weights of the dependence edges depend on the types of the data dependences. Let $(i, j, r, \tau) \in E_D$, then the weight $w(i, j)$ is defined as follows:

$$w(i,j) = \begin{cases} w_i, & \text{if } \tau = t \\ w_i - w_j + 1, & \text{if } \tau = o \\ 0, & \text{if } \tau = a \end{cases} \tag{9.1}$$

In [CCK97] bounds are presented that take the available hardware resources of a regular r-issue processor into account. In the following these bounds are extended to consider functional unit latencies of multiple clock cycles and operations that can be mapped to different resource types. Let $R(i)$ denote the set of resource types, an operation i can be mapped to, i. e. $R(i) = \{k \in N_R^F \mid (i,k) \in E_R\}$. Let an operation i be given and let p_i denote the number of predecessors in the data dependence graph that can be executed by exactly the same set of resource types as operation i, i. e. $p_i = |\{j \mid j \xrightarrow[E_D]{*} i \;\wedge\; R(i) = R(j)\}|$. Similarly define f_i as the number of successors in the data dependence graph that can be executed by exactly the same set of resource types as operation i, i. e. $f_i = |\{j \mid i \xrightarrow[E_D]{*} j \wedge R(i) = R(j)\}|$. Define r as the number of all instances of the resource types i can be mapped to and L_i as their minimal latency. Then the $asap(i)$ and the $alap(i)$ control steps are computed as follows:

$$asap(i) \;=\; \max\left\{s(i), \left(\left\lfloor \frac{p_i}{r} \right\rfloor\right) \cdot L_i + 1\right\} \tag{9.2}$$

$$alap(i) \;=\; \min\left\{l(i), U - \left(\left\lfloor \frac{f_i}{r} \right\rfloor\right) \cdot L_i - 1\right\} \tag{9.3}$$

This way, resource restrictions are conservatively taken into account when computing the superset of possible control steps for each operation.

9.5. The Optimisation Interface

For each input program the generated optimisers compute the control flow graph, the control dependence graph and the data dependence graph. If the task of register reassignment is addressed the register renaming algorithm of Sec. 7.2.1 is invoked to replace references to physical registers by references to virtual registers. The register renaming algorithm is followed by a recomputation of the data dependence graph that is based on the virtual registers and does not contain spurious data dependences any more. Figures 9.9 and 9.10 show examples of a data dependence graph before and after renaming. It is obvious that a large number of dependence edges is removed by the renaming process.

Subsequently the superblock graph is computed. The algorithms for constructing the superblocks have been described in Sec. 7.1. The information about the execution frequencies of the basic blocks is assumed to be given by annotations in the CRL representation of the input program. The developer can influence the superblock construction and optimisation process by several parameters of the generated optimisers. Dedicated command line parameters are available for

- enabling/disabling superblock enlargement beyond loop boundaries (see Sec. 7.1.1).
- providing a code size threshold that stops the superblock enlargement if the threshold is reached and causes basic blocks of the input program exceeding the threshold to be split (see Sec. 7.1.3).

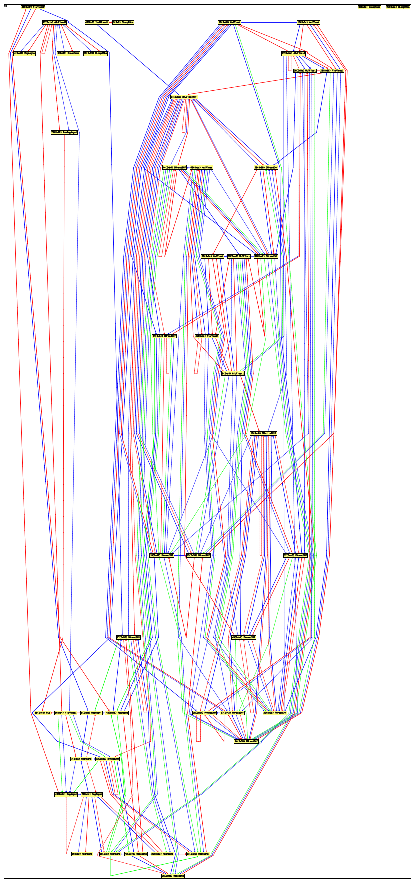


Figure 9.9.: Visualisation of a data dependence graph before renaming.

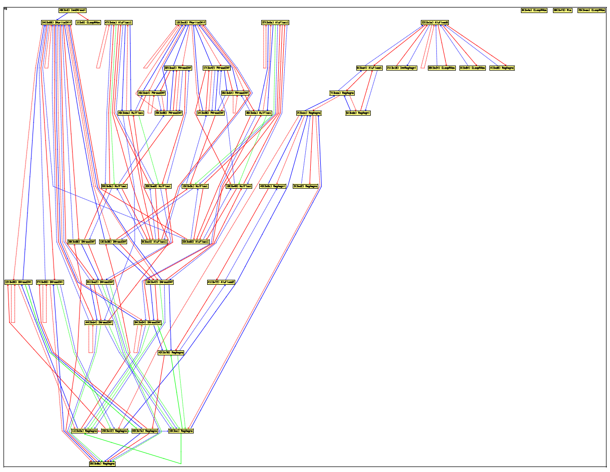


Figure 9.10.: Visualisation of the data dependence graph from Fig. 9.9 after renaming.

- providing an upper bound for the maximal length of paths to be considered in the resource path constraints of Sec. 7.2.4.
- enabling/disabling the integration of the register assignment problem.
- choosing between SILP modelling, OASIC modelling and generic list scheduling.
- choosing between exactly solving the generated integer linear programs or using any of the approximations presented in Chap. 6.
- specifying an explicit upper bound for the execution time of the input program to be optimised. This bound is used in the computation of the *alap* control step and can increase the efficiency of the ILP solution process.
- specifying an upper bound for the solution time of the generated integer linear programs. If the bound is exceeded the ILP solution process returns the best feasible solution found so far.
- selecting the repairing mode to cope with capacity violations of abstract register files. Either collision-based or exclusion-based repairing can be chosen (see Sec. 7.4.3).
- choosing between exact or heuristic graph colouring for constructing the physical register assignment from the solution of the generated integer linear programs (see paragraph 7.5).

Finally the CRL representation of the optimised program is generated from which the final assembly file is constructed.

10. Experimental Results

The PROPAN framework has been retargeted to several different target architectures. It has been used to generate ILP-based postpass optimisers for two widely used contemporary digital signal processors with considerably different hardware characteristics, the Analog Devices ADSP-2106X SHARC [Ana95] and the Philips TriMedia TM1000 [Phi97]. Additionally the PROPAN system is part of a retargetable framework for calculating worst-case execution time guarantees for real-time systems [FKL+99]. One hardware architecture investigated in this framework is the Infineon TriCore μC/DSP [Inf00]; the relevant target-specific information is provided by a TDL specification. Furthermore PROPAN has been used as a platform to implement hardware-specific postpass optimisations for the Infineon C16x [Sie96] microprocessor family that are part of a commercial postpass optimiser [Abs00a]. TDL descriptions for the TI320C6x [Tex97] and for the Intel Embedded Pentium are currently under construction. In the following sections the performance of the ILP-based optimisers generated for the Analog Devices ADSP-2106X SHARC and the Philips TriMedia TM1000 is evaluated. Each section is introduced by a brief overview of the hardware architecture, followed by the presentation of the experimental results; a summary of the experimental evaluation is given in Sec. 10.3.

In [ZVSM94] the `dspstone` benchmark suite has been presented as a basis for comparing the performance of digital signal processors for typical applications. Part of the suite is a collection of C-programs representing typical DSP kernels, i. e. functions often used in digital signal processing algorithms. The collection comprises computations of digital filters, matrix multiplications, convolutions, vector products, complex arithmetic, etc. The computation routines of this kernel benchmark are used as input programs for the generated ILP-based optimisers of both the ADSP-2106X SHARC and the TriMedia TM1000. Additionally hand-crafted assembly programs are investigated that also represent typical applications of digital signal processing. The experiments have been performed under SunOS 5.7 on a SPARC Ultra-Enterprise 10000; the generated mixed integer linear programs are solved by the CPLEX library [ILO99].

10.1. Analog Devices ADSP-2106x SHARC

10.1.1. Architecture

The ADSP-2106x SHARC is a 32-bit digital signal processor for speech, sound graphics and imaging applications. Its cycle time is 25 ns, the clock frequency 40 MHz, and its instruction rate is 40 MIPS. As a floating-point DSP the ADSP-2106x SHARC supports the 32-bit IEEE floating point format, 32-bit integer and fractional formats, and extended-precision 40-bit IEEE floating-point format. An overview of the architecture is given in Fig. 10.1. The core processor of the ADSP-2106x SHARC consists of the data register file, three computation units, the control unit, two address generators DAG1 and DAG2, the timer and the instruction cache.

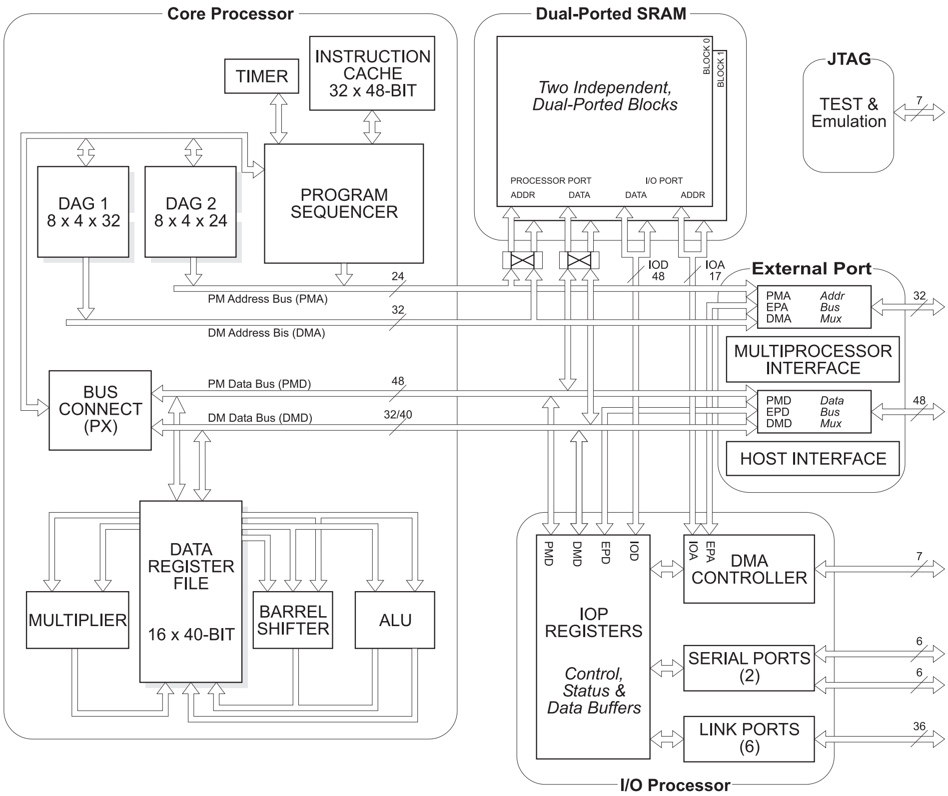


Figure 10.1.: ADSP-2106x SHARC block diagram.

The ADSP-21060 contains 4 MBit of on-chip SRAM consisting of two 2 Mbit blocks. The blocks are dual-ported to support independent accesses from core processor and from I/O processor or DMA controller in one cycle. Each block can be configured for different combinations of code and data. Instructions are fetched over the 48-bit PM bus or from the instruction cache. Data can be accessed over both the 40-bit DM bus (using DAG1) and the PM bus (using DAG2). DAG1 supplies 32-bit addresses over the DM bus, DAG2 supplies 24-bit addresses for PM bus data accesses. Both blocks of memory can be accessed simultaneously if one access uses the DM bus and the other one the PM bus. Accesses to memory can be made for 16-bit, 32-bit, or 48-bit words.

The general purpose register file consists of sixteen 40-bit registers that can be used for fixed-point and floating-point computations. Additionally there are four types of registers used for data addressing: index registers (I), modify registers (M), base registers (B) and length registers (L). Each data address generator contains 8 registers of each of those types where the registers of DAG1 are 32-bit wide and those of DAG2 are 24-bit wide. The following addressing modes are supported: direct addressing, indirect addressing with pre- and post-modify and circular addressing. The I registers are used as pointers to memory, the M registers contain the increment value for advancing the pointers. B and L registers are only used for circular data buffers. Each B register holds the base address of a circular buffer, and the corresponding L register the length of the buffer.

There are three independent computation units: an ALU, a multiplier with fixed-point accumulator and a shifter. ALU and multiplier can perform fixed-point and floating-point operations. The shifter executes logical and arithmetical shifts, as well as special bit manipulation operations on 32-bit operands. The ADSP-2106X SHARC has a load/store architecture; memory accesses are dedicated to special load/store operations. All instructions have a fixed length of 48 bit. There are instructions for computing the arithmetic mean, the minimum and maximum of two operands, for clipping, multiply/accumulate, bit extraction and insertion, shifts and rotates. Most operations can optionally be guarded; then their execution depends on the value of certain bits in control and status registers that have been set by preceding operations.

The ADSP-2106X SHARC has a three-stage instruction pipeline consisting of fetch, decode and execute cycle. In the fetch cycle the instruction is read from the program memory PM or from the instruction cache, in the decode cycle it is decoded and in the execute cycle the instruction is executed. In sequential program flow while one instruction is fetched the instruction fetched in the previous cycle is decoded and the instruction fetched two cycles before is executed. Thus the throughput is one instruction per clock cycle. Jumps, calls and returns can be delayed or non-delayed. In a delayed branch the two instructions immediately after the branch instruction are executed before the branch takes effect; in a nondelayed branch, the program sequencer suppresses the execution of these two instructions and performs nops instead. Furthermore there is a zero-overhead loop instruction that can be used to execute counter-based loops.

The instruction-level parallelism of the ADSP-2106X SHARC is restricted. Data accesses to program memory and data memory can be executed in parallel if the address of the DM access is specified by DAG1 registers and the address of the PM access by DAG2 registers. Many arithmetic operations can be executed in parallel to memory accesses and some control flow operations can be executed in parallel to arithmetic operations. Additionally there is a restricted parallelism between ALU and multiplier. Some ALU and multiplier operations can be combined to a multifunctional instruction and can then be executed in parallel. This however requires that each of the four input operands is located in a dedicated group of four register locations within the general-purpose register file (see Fig. 10.2). If any operand is located in another register the operation is feasible but cannot be part of a multifunctional instruction.

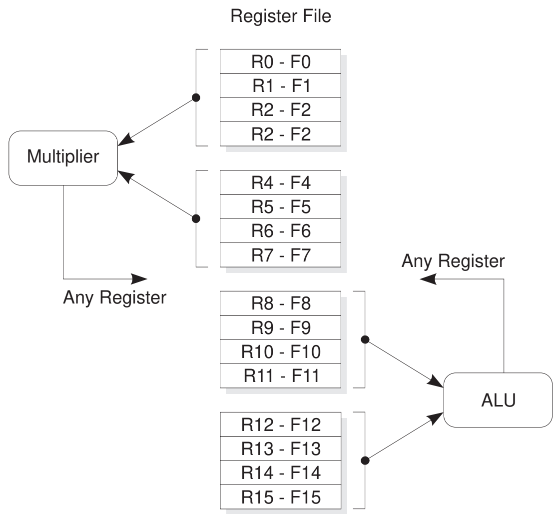


Figure 10.2.: Input registers for parallel execution of ALU and multiplier.

10.1.2. Performance of the Optimiser

For each machine operation of the ADSP-2106x SHARC the functional unit assignment is uniquely determined so that the resource allocation problem has not to be considered. Thus the optimiser generated for the ADSP-2106x SHARC performs integrated instruction scheduling and register reassignment. All target-specific information is provided by the TDL specification. The restrictions of instruction-level parallelism and the interdependencies between register assignment and operation parallelisation are incorporated into the ILP models by the integer linear constraints derived from the constraint section. These constraints model the restricted parallelism of ALU and multiplier as well as explicit parallelisation prohibitions between microoperations. Since previous studies [Käs97, KL98] have shown that the integration of the register assignment in the OASIC formulation is not promising for complexity reasons, in the following we will concentrate on the SILP formulation.

The computation routines of the `dspstone` benchmark are compiled to assembly code by the `gcc`-based `g21k` compiler. Tab. 10.1 lists the number of machine operations and the number of compacted instructions of the input programs. The number of basic blocks and the number of loops of each program are given in the last two columns. In addition to the `dspstone` kernels, another matrix multiplication (`mamu2`) and a routine from the whetstone benchmark (`whetp3`) are compiled to assembly code by the `g21k` and used as input programs. The `g21k`-generated input programs contain between 14 and 55 microoperations and usually consist of several basic blocks representing (possibly nested) loops. Apart from the `g21k`-generated input programs, some hand-crafted assembly programs are evaluated whose characteristics are shown in the lower part of Tab. 10.1. They comprise a discrete fourier transformation (`dft`), a finite impulse response filter (`fir`), a histogram (`histo`), a cascaded infinite impulse response filter (`cascade`), and two routines from a wavelet transformation (`waveletk`, `waveleti`). The hand-crafted assembly programs contain between 18 and 49 microoperations. Because of an efficient code selection, they offer a high degree of parallelism. They are written as sequential code, so that the available instruction-level parallelism is not exploited. Performing the scheduling and register assignment to exploit the parallelism is left to the generated optimiser. The *alap* control steps of each operation are computed using the execution time of the input program as an upper bound for the length of any feasible schedule.

Name	Operations	Instructions	Blocks	Loops
`dbiquadn`	50	44	4	1
`dbiquado`	30	30	1	0
`dcompmul`	20	20	1	0
`dcompupd`	38	36	1	0
`dconvolu`	17	17	4	1
`ddotprod`	22	21	4	1
`dfir2dim`	55	52	14	5
`dmatrix2`	38	33	10	3
`dfir`	22	18	4	1
`dlms`	36	35	7	2
`dmat1x3`	24	23	7	2
`dmatrix1`	38	37	10	3
`dncompup`	42	38	4	1
`dnrealup`	25	24	4	1
`drealup`	14	13	1	0
`mamu2`	51	48	10	3
`dft`	26	26	7	2
`fir`	18	18	4	1
`histo`	44	44	8	2
`cascade`	23	23	4	1
`waveleti`	49	49	16	5
`waveletk`	39	39	10	3
`whetp3`	26	22	1	0

Table 10.1.: Statistics about the input programs of the ADSP-2106x SHARC.

Optimisations restricted to Loop Boundaries

The experimental evaluation is partitioned into two phases. In the first phase, the superblock construction is stopped at loop boundaries. The generated superblocks correspond to the traces of the trace scheduling algorithm [Fis81]. Most programs are represented by several superblocks each of which is modelled by an individual integer linear program. The result of the optimisations are summarised in Tab. 10.2 and Tab. 10.3. Column m denotes the computation method; a_1 corresponds to the approximation of isolated flows, a_2 to the stepwise approximation of isolated flows, a_3 to the stepwise approximation and e to the exact, i. e. provably optimal solution. Since the assignment of operations to functional units is uniquely defined, the approximation of isolated operations is not investigated. Column S shows the number of superblocks generated for each input program. The total measured CPU-time is listed in column t, column I shows the number of compacted instructions in the result of each method.

The programs `dbiquado` and `dcompupd` consist of a single basic block which is too large for an ILP-based solution to be obtained in less than 8 hours. Thus a threshold value of 25 operations for the maximal superblock size has been used to split the basic blocks into two parts each of which is represented by a dedicated superblock. For all optimisations a time limit of 8 hours has been specified as an upper bound for any computation of the ILP solver. After 8 hours, the best feasible solution found so far is returned; the optimality of that solution cannot be guaranteed. For the approximative methods the total computation time can exceed 8 hours since during the iterative process the ILP solver is called several times. In the following, the computations where this time limit leads to premature terminations of the ILP solver are marked by '(p)'. Among the 23 investigated programs a premature termination of approximative methods only occurs for `dfir2dim`; here only the stepwise approximation with a computation time of 4 hours and 48 minutes stays under the time limit. All programs with the exception of `cascade` are optimised with a maximal length of 3 for the paths to be modelled by path constraints (see Sec. 7.2.4). Because of the large number of path constraints required for `cascade` an upper bound of 2 has been chosen to limit the computation time.

The ILP-based solutions are compared to the result of the generic list scheduling algorithm with highest-level first heuristic (Fig. 9.7). The support functions of the list scheduling modelling the restrictions of instruction level parallelism and interactions between instruction scheduling and register assignment are generated from the TDL description. In order to allow a conservative comparison, the register assignment in the input programs of the list scheduling algorithm is already optimal while it is explicitly recomputed in the ILP-based methods. Tab. 10.4 shows the computation time of the list scheduling algorithm and the generated number of compacted instructions for each input program. Column I_o lists the optimal number of instructions (verified by hand), in column Δ_o the percentage deviation of the result of list scheduling from the optimal solution is listed and the number

Name	m	S	t	I	Name	m	S	t	I
`dbiquadn`	a_1	3	1' 15.56"	43	`dfir`	a_1	3	1.35"	17
	a_2		4' 11.3"	43		a_2		3.9"	17
	a_3		3' 11.64"	43		a_3		2.11"	18
	e		3h 53' 12.36"	43		e		1.84"	17
`dbiquado`	a_1, t25	2	1' 21.58"	24	`dlms`	a_1	5	44.19"	32
	a_2, t25		1' 31.57"	24		a_2		1' 26.67"	32
	a_3, t25		1' 11.87"	24		a_3		4.59"	33
	e, t25		24' 34.17"	24		e		6' 41.6"	32
`dcompmul`	a_1	1	35.34"	17	`dmat1x3`	a_1	5	2.28"	22
	a_2		19.11"	17		a_2		4.06"	22
	a_3		10' 7.03"	17		a_3		1.17"	22
	e		14' 22.17"	17		e		2.35"	22
`dcompupd`	a_1, t25	2	1h 2' 4.01"	34	`dmatrix1`	a_1	7	26' 50.9"	36
	a_2, t25		1h 4' 8.17"	34		a_2		1' 20.44"	36
	a_3, t25		8h 31.84"	34		a_3		1' 25.45"	36
	e, t25		8h 34' 58.7"(p)	34		e		6h 15' 26.6"	36
`dconvolu`	a_1	3	0.82"	14	`dncompup`	a_1	3	3' 18.33"	37
	a_2		1.22"	14		a_2		4' 14.85"	37
	a_3		0.42"	14		a_3		13.01"	38
	e		0.53"	14		e		47' 34.0"	37
`ddotprod`	a_1	3	6.93"	20	`dnrealup`	a_1	3	2' 43.26"	23
	a_2		18.65"	20		a_2		4' 15.83"	23
	a_3		1.57"	20		a_3		4.33"	23
	e		19.02"	20		e		48' 13.69"	23
`dfir2dim`	a_1	9	8h 6' 1.07" (p)	51	`drealup`	a_1	1	2.46"	12
	a_2		8h 9' 18.28" (p)	51		a_2		4.88"	12
	a_3		4h 47' 41.36"	51		a_3		3.67"	12
	e		8h 8' 36.91" (p)	51		e		5.94"	12
`dmatrix2`	a_1	7	3.8"	32	`mamu2`	a_1	7	8' 31.42"	46
	a_2		3.98"	32		a_2		8' 58.41"	46
	a_3		1.27"	32		a_3		32.57"	46
	e		2.35"	32		e		2h 5' 25.7"	46

Table 10.2.: Performance of the SILP-based optimisations for the ADSP-2106X SHARC not exceeding loop boundaries (1).

Name	m	S	t	I	Name	m	S	t	I
dft	a_1	5	1.32”	14	waveleti	a_1	9	1.63”	35
	a_2		3.01”	14		a_2		2.83”	35
	a_3		1.54”	14		a_3		1.69”	35
	e		0.54”	14		e		1.28”	35
fir	a_1	3	0.45”	8	waveletk	a_1	7	1.73”	29
	a_2		0.7”	8		a_2		3.83”	29
	a_3		0.4”	8		a_3		1.57”	29
	e		0.36”	8		e		1.01”	29
histo	a_1	5	7.6”	31	whetp3	a_1	1	15’ 2.88”	19
	a_2		16.52”	31		a_2		22’ 22.71”	19
	a_3		31.1”	33		a_3		12h 37’ 58.47”	19
	e		14.66”	31		e		8h 14’ 1.43”(p)	21
cascade	a_1	3	44.74”	9					
	a_2		59.09”	9					
	a_3		10.61”	9					
	e		1.2”	9					

Table 10.3.: Performance of the SILP-based optimisations for the ADSP-2106X SHARC not exceeding loop boundaries (2).

of instructions of the input program is listed in column I_{in}. The hand-crafted assembly programs that have not been compacted before using them as input to the optimisations are marked by an asterisk (*).

In Fig. 10.3 – Fig. 10.6, a visualisation of the experimental results is given. In Fig. 10.3 and Fig. 10.4 the number of compacted instructions in the result of the ILP-based methods is compared to the schedule of the `g21k`-generated input programs and to the result of the list scheduling algorithm. The vertical axes show the number of compacted instructions, the horizontal axes the computation methods with the abbreviations introduced above. Fig. 10.5 and Fig. 10.6 compare the measured computation times on a logarithmic scale. The CPU-time measured in seconds is shown on the vertical axes, the horizontal axes represent the solution methods. Computations prematurely terminated because of the time limit are marked by an asterisk (∗).

We can see that for all input programs, the approximation by isolated flows (a_1) and the stepwise approximation of isolated flows (a_2) produce an optimal result. The code produced by the stepwise approximation(a_3) exceeds the optimal number of instructions by one for three input programs and by two for one program; for 82.6% of the evaluated programs it produces an optimal result. The result of the list scheduling algorithm exceeds the optimal number of instructions in more than 43% of all cases. The maximum overhead has been measured for `whetp3` where it exceeds the optimal number of instructions by 21.05%. The programs where

Figure 10.3.: Compacted instructions produced by the SILP-based optimisations for the ADSP-2106x SHARC within loop boundaries (1).

Name	$t[sec]$	I	I_o	$\Delta_o[\%]$	I_{in}	Name	$t[sec]$	I	I_o	$\Delta_o[\%]$	I_{in}
dbiquadn	3.67	43	43	0	44	dncompup	2.29	37	37	0	38
dbiquado	0.9	25	24	4.16	30	dnrealup	0.61	23	23	0	24
dcompmul	0.24	18	17	5.88	20	drealup	0.10	13	12	8.3	13
dcompupd	2.02	35	34	2.94	36	mamu2	3.97	46	46	0	48
dconvolu	0.19	14	14	0	17	dft	0.46	14	14	0	26*
ddotprod	0.39	20	20	0	21	fir	0.17	9	8	12.5	18*
dfir2dim	5.35	51	51	0	52	histo	2.25	32	31	3.2	44*
dmatrix2	1.54	32	32	0	33	cascade	0.34	10	9	11.1	23*
dfir	0.38	17	17	0	18	waveleti	2.76	37	35	5.7	49*
dlms	1.43	32	32	0	35	waveletk	1.41	32	29	10.34	39*
dmat1x3	0.48	22	22	0	23	whetp3	0.86	23	19	21.05	22
dmatrix1	1.90	36	36	0	37						

Table 10.4.: Results of list scheduling.

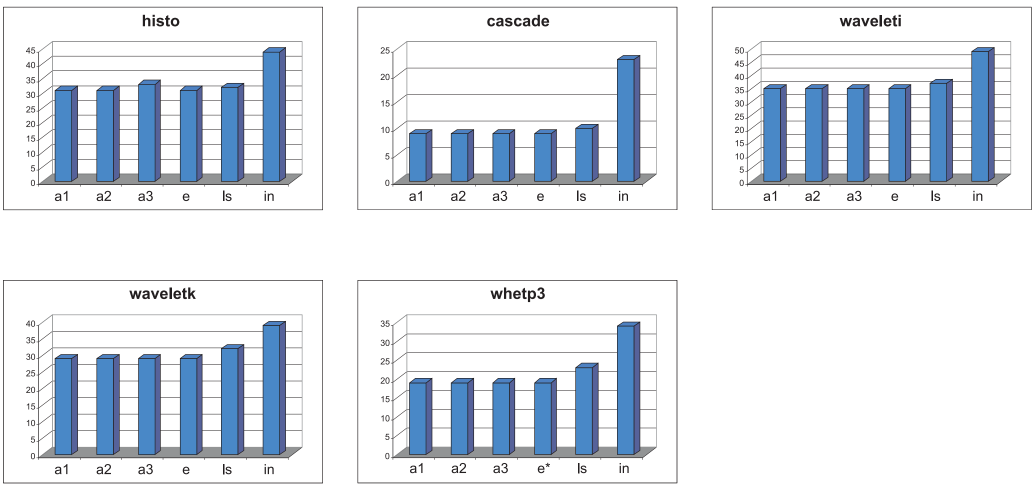


Figure 10.4.: Compacted instructions produced by the SILP-based optimisations for the ADSP-2106x SHARC within loop boundaries (2).

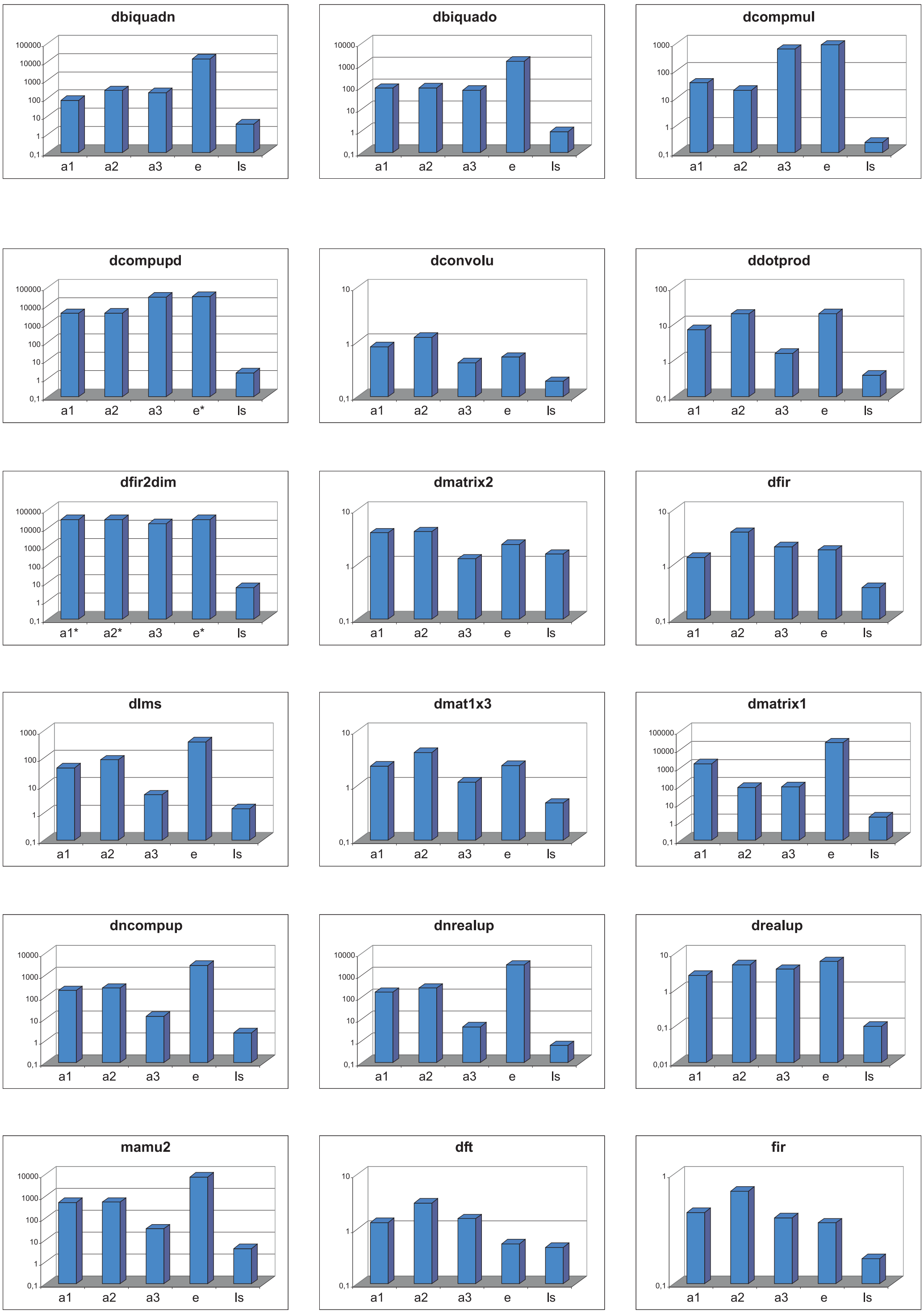


Figure 10.5.: CPU-time in seconds of the SILP-based optimisations for the ADSP-2106x SHARC within loop boundaries (1).

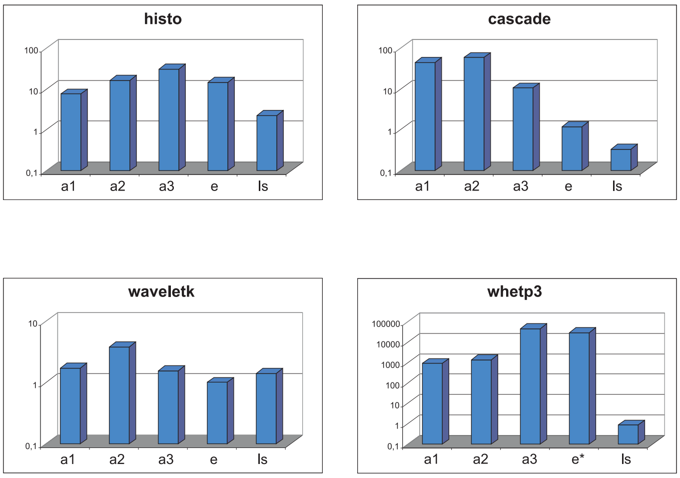


Figure 10.6.: CPU-time in seconds of the SILP-based optimisations for the Adsp-2106x Sharc within loop boundaries (2).

the list scheduling algorithm produces an optimal result mostly offer only little parallelism. The `g21k`-generated input code could be improved in all cases; on average, it exceeds the optimal number of instructions by 8.2%.

When analysing the measured computation times we can see that for the smaller input programs the exact ILP-based solution takes less time than the ILP-based approximations which is due to the setup overhead of the approximations. For the larger input programs however the computation time can be reduced significantly. The time required for the stepwise approximation strongly varies with the input programs; for some programs it is the fastest approximative method, for others it is the slowest. The performance of the approximation of isolated flows and of the stepwise approximation of isolated flows is comparable for the investigated input programs. With the exception of `dcompupd`, `dfir2dim` and `whetp3`, ILP-based approximations can be obtained for each input program within less than two minutes. The fastest ILP-based solution of `dcompupd` resp. `whetp3` is obtained by the approximation of isolated flows in 1 hour and 2 minutes resp. in 15 minutes. For `dfir2dim` the stepwise approximation is the fastest method; it produces its result in 4 hours and 48 minutes. Only for `dbiquado` and `dcompupd` this required splitting the largest basic block into two parts which are separatedly optimised. If the maximal computation time has to be decreased this can be achieved by specifying smaller upper bounds for the maximal length of the superblocks.

Optimisations across Loop Boundaries

In the following the second phase of the experimental evaluation is investigated. Here the superblocks have been maximally extended so that all input programs are modelled by a unique integer linear program. In Tab. 10.5 the most important characteristics of the generated integer linear programs are shown. Column x

Name	x	r	V	p	l	s	rs	C	size[KB]
`dbiquadn`	404	982	1697	99	22	181	262	2023	252.22
`dbiquado`	171	610	948	83	19	135	175	1220	156.92
`dcompmul`	69	226	376	52	15	55	64	596	6.78
`dcompupd`	253	676	1130	105	20	253	169	1672	191.27
`dconvolu`	37	40	134	25	4	31	10	344	20.47
`ddotprod`	80	128	303	36	7	56	32	558	45.0
`dfir`	58	124	284	38	6	52	31	550	45.01
`dmat1x3`	95	128	343	39	7	64	32	652	49.72
`dmatrix2`	209	600	1117	143	19	132	150	1408	167.99
`dncompup`	324	780	1298	85	20	130	195	1458	198.06
`dnrealup`	121	172	417	41	8	80	43	677	61.18
`drealup`	35	76	151	32	5	33	19	280	24.87
`dft`	106	408	647	53	22	93	102	883	103.68
`fir`	23	382	458	38	16	12	118	507	77.14
`histo`	358	1684	2224	161	33	200	421	1832	359.46
`cascade`	49	699	892	314	21	36	189	1111	153.73
`waveleti`	481	1820	2719	98	36	393	521	3114	435.19
`waveletk`	330	1020	1584	63	28	276	303	2043	256.49
`whetp3`	153	1376	1749	41	41	81	413	2213	438.92
`dfir2dim`	615	1075	2336	292	28	615	286	3436	362.01
`dlms`	198	426	813	63	18	198	138	1252	119.64
`mamu2`	750	684	1902	166	33	750	198	4101	331.82
`dmatrix1`	312	394	998	77	18	312	109	1791	148.5

Table 10.5.: Characteristics of the ILPs generated for the ADSP-2106X SHARC in the SILP formulation across loop boundaries.

respectively r shows the number of resource flow variables respectively register flow variables that are explicitly specified as binary. The number of all integer variables is presented in column V. In column p the numbers of the generated precedence constraints (equations 5.9 – 5.11) are listed. Column l shows the number of life range constraints (equation 5.16), column s the number of serial constraints (equation 5.12) and column rs the number of register serial constraints (equation 5.17). The total number of generated constraints is given in column C, and the last column shows the size of the generated integer linear programs in the uncompressed CPLEX LP-format [ILO99]. The total number of binary variables ranges from 134 to 2719, the number of constraints from 280 to 4101, and the size of the integer linear programs ranges from 6.78 KBytes to 438.92 KBytes.

For the largest input programs (`dncompup`, `histo`, `dfir2dim`, `mamu2` and for `dmatrix1`) the time limit of 8 hours leads to premature exits of the ILP-based methods when the complete input program is modelled by a single integer linear program. In order to achieve lower computation times for those programs the optimisation scope should remain restricted to loop boundaries. The information about the programs only consisting of one basic block (`dbiquado, dcompmul, dcompupd, drealup, whetp3`) is the same as in Tab. 10.2 and Tab. 10.3. In Tab. 10.6, the results of the ILP-based optimisations across loop boundaries are shown for the remaining 13 programs. Again column m represents the solution method where a_1 denotes the approximation of isolated flows, a_2 the stepwise approximation of isolated flows, a_3 the stepwise approximation and e the exact, i. e. provably optimal solution. In column t the measured computation times are listed and column I shows the number of compacted instructions in the result of each method. Again a time limit of 8 hours is specified for each input program. The computations where the time limit leads to a premature return of the ILP solver are marked by '(p)' in Tab. 10.6. The upper bound on the length of path constraints is set to three for all input programs except for `cascade` where the maximal path length is set to two.

The results are visualised in Fig. 10.7 and Fig. 10.8 where Fig. 10.7 compares the number of compacted instructions for each solution method and Fig. 10.8 shows the computation times on a logarithmic scale. For 67% of the programs where the exact ILP-based method is terminated after 8 hours, the best feasible solution found up to this point of time is already optimal. Due to the premature exit however the optimality of the result is not guaranteed. In nearly all cases, i. e. in 37 of the 39 approximative computations, the ILP-based approximations give an optimal result (the optimality was always verified by hand). It becomes apparent that for the large input programs the stepwise approximation of isolated flows is faster than the approximation of isolated flows while the computation time of the stepwise approximation again is subject to significant variations depending on the input program. For `dlms`, e. g., it is the fastest approximation while for `dbiquadn` no result is obtained within 12 hours. For most of the programs listed, a result can be produced by an ILP-based approximation in less than 10 minutes (except for `dbiquadn` where the fastest result is produced by the stepwise approximation of isolated flows in about 3.5 hours) while the computation time for an exact solution exceeds 8 hours for 46% of the input programs. The maximal speed-up is in the order of several magnitudes. Another observation is that no improvement could be achieved over the optimisation restricted to loop boundaries. This can be explained by several reasons: first the input programs offer few opportunities of moving operations between basic blocks. Moreover, since all operations can be executed in one clock cycle and the assignment of operations to resource types is uniquely defined, there are no interdependencies between the scheduling and resource allocation of different superblocks. Interdependencies are only caused by the register usage; in the optimisations restricted to loop boundaries however no disadvantageous decisions have been made due to the relatively low number of overlapping life ranges.

Name	m	S	t	I	Name	m	S	t	I
`dconvolu`	a_1	1	4.2”	14	`dnrealup`	a_1	1	1h 13’ 18.76”	23
	a_2		8.7”	14		a_2		48’ 40.19”	23
	a_3		3.29”	14		a_3		6’ 22.13”	23
	e		5.01”	14		e		8h 1’ 42.4” (p)	23
`ddotprod`	a_1	1	3’ 21.03”	20	`dft`	a_1	1	10.4”	14
	a_2		3’ 8.09”	20		a_2		30.63”	14
	a_3		2’ 10.05”	20		a_3		58.56”	14
	e		1h 24’ 18.21”	20		e		7.67”	14
`dfir`	a_1	1	84.62”	17	`fir`	a_1	1	2.65”	8
	a_2		2’ 8.59”	17		a_2		8.14”	8
	a_3		1’ 22.7”	17		a_3		4.43”	8
	e		31’ 25.88”	17		e		1.3”	8
`dmat1x3`	a_1	1	50.23”	22	`waveleti`	a_1	1	4h 16’ 44.84”	35
	a_2		49.78”	22		a_2		2’ 51.13”	36
	a_3		3’ 34.87”	22		a_3		1h 45’ 2.53”	35
	e		7’ 42.67”	22		e		8h 1’ 57.46” (p)	39
`dmatrix2`	a_1	1	44’ 57.99”	32	`cascade`	a_1	1	5’ 32.54”	9
	a_2		9’ 22.32”	32		a_2		2’ 45.43”	9
	a_3		5’ 10.64”	32		a_3		27.95”	9
	e		8h 1’ 8.65” (p)	32		e		3.66”	9
`waveletk`	a_1	1	9’ 25.9”	29	`dbiquadn`	a_1	1	9h 58’ 8.62” (p)	43
	a_2		40.22”	29		a_2		3h 35’ 59.23”	43
	a_3		2’ 58.6”	29		a_3		$>$ 12 h	–
	e		8h 1’ 8.33” (p)	30		e		8h 4’ 18.0” (p)	43
`dlms`	a_1	1	27’ 11.41”	32					
	a_2		15’ 12.94”	32					
	a_3		1’ 16.78”	32					
	e		8h 32’ 25.9” (p)	32					

Table 10.6.: Performance of the SILP-based optimisations for the ADSP-2106x SHARC across loop boundaries.

Figure 10.7.: Compacted instructions produced by the SILP-based optimisations for the ADSP-2106X SHARC across loop boundaries.

Figure 10.8.: CPU-time in seconds of the SILP-based optimisations for the ADSP-2106x SHARC across loop boundaries.

10.2. Philips TriMedia TM1000

10.2.1. Architecture

The TriMedia TM1000 is a media processor for high-performance multimedia applications such as real-time processing of audio, video, graphics and communications datastreams. It contains in a single chip a 100 MHz VLIW CPU (the DSPCPU), DMA-driven multimedia input/output units, DMA-driven multimedia coprocessors that operate independently and in parallel with the DSPCPU, a high-performance bus, and the memory system. The block diagram of the TM1000 is shown in Fig. 10.9.

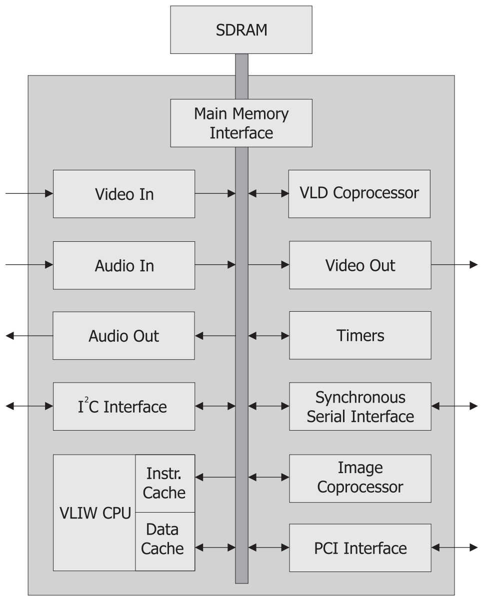


Figure 10.9.: Philips TriMedia TM1000 block diagram.

The DSPCPU is a 32-bit VLIW processor containing 128 general-purpose registers. The registers are not separated into banks; operations can use any register for any operand. The registers are named `r0` . . . `r127`; the register `r0` always contains the integer value 0, and the register `r1` the integer value 1. The DSPCPU contains separate 16-KB data and 32-KB instruction caches. The data cache is dual-ported to allow two simultaneous accesses, and both caches are eight-way set-associative with a 64-byte block size. The memory is byte-addressable.

The architecture supports unsigned integers, signed integers in two's complement representation and single-precision IEEE-compliant floating-point arithmetic. The TriMedia has a load/store architecture; memory accesses are restricted to dedicated load/store operations. The instruction set includes common RISC operations, multimedia operations accelerating standard video compression and decompression algorithms, special DSP operations that perform SIMD functions and IEEE-compliant floating-point operations. Multimedia operations are defined for

32-bit, 16-bit and 8-bit operands. As an example the `dspuquadaddui` operation implements four eight-bit additions; it treats the first operand of each addition as unsigned, the second as signed, and produces an unsigned result for each addition. The execution time of the operations ranges from one clock cycle for the most common operations up to 17 clock cycles.

The DSPCPU issues one long instruction every clock cycle. Each instruction word is composed of five microoperations that are issued simultaneously. Certain restrictions exist in the choice of what operations can be packed into one instruction. For example, the DSPCPU allows no more than two load/store operations to be packed into a single instruction; a detailed overview of the feasible operation groupings is given in Tab. 10.7. Also, no more than five results can be written during any one cycle. There are 27 functional units including integer and floating-point arithmetic units and data-parallel digital signal processing units. A schematic view of this architecture is given in Fig. 10.10.

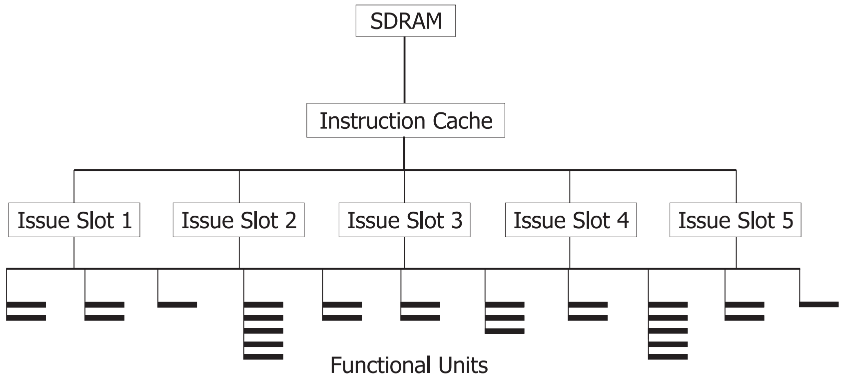


Figure 10.10.: Issue slots and functional units of the DSPCPU.

In the TM1000 architecture, most operations are optionally guarded. A guarded operation executes conditionally depending on the value of an explicitly specified general purpose register. Guarding controls the effect on the whole programmer visible state of the system, i. e. register values, memory content and device state.

10.2.2. Performance of the Optimisers

Since the DSPCPU disposes of a homogeneous general-purpose register file, the register assignment does not influence the available parallelism and thus can be neglected. However the instruction scheduling interacts with the resource allocation problem. Each operation has to be explicitly assigned to an issue slot of the instruction word. The TriMedia Tm1000 architecture imposes restrictions on the way operations can be placed within an instruction. The operations can be grouped into several categories that are implemented by a certain functional unit type. The feasible mappings of those categories to issue slots are summarised in Tab. 10.7; the individual mapping determines the functional unit binding. Thus in

Functional Unit Type	Slot1	Slot2	Slot3	Slot4	Slot5
`const`	x	x	x	x	x
`alu`	x	x	x	x	x
`dmem`				x	x
`dmemspec`					x
`shifter`	x	x			
`dspalu`	x		x		
`dspmul`		x	x		
`branch`		x	x	x	
`falu`	x			x	
`ifmul`		x	x		
`fcomp`			x		
`ftough`		x			

Table 10.7.: Mapping between operation categories and issue slots.

the TDL description the issue slots are defined as virtual resources and the resource allocation problem is concerned with mapping the microoperations to issue slots. Moreover, the operations have to be synchronised with respect to the result bus; no more than five operations may write their result simultaneously on the bus. Since not all microoperations have the same execution time, an explicit synchronisation is necessary. Thus for the TriMedia TM1000 the phase coupling problem between instruction scheduling and resource allocation has to be addressed. The corresponding optimisers are generated automatically from the TDL-description; the exact ILP-based methods allow an optimal solution of the problem of phase-coupled instruction scheduling and resource allocation with respect to the given code selection and the optimisation scope.

The computation routines of the `dspstone` benchmark are compiled to assembly code by the highly optimising Philips `tmcc` compiler. Additionally two hand-crafted assembly programs, `dctk` and `fidctk`, are evaluated which are excerpts of a discrete cosine transform and an inverse discrete cosine transform. In Tab. 10.8 an overview of the characteristics of the input programs is given. For each input program the number of machine operations and of compacted instructions are listed together with the number of basic blocks and the number of loops. The number of basic blocks and loops differs from the figures of Tab. 10.1 which is due to the loop unrolling and if-conversion algorithms employed by the `tmcc` compiler. The input programs contain between 9 and 95 microoperations; the hand-crafted assembly programs use special digital signal processing operations of the TriMedia TM1000 and offer a high degree of available parallelism.

The experimental results show that for the TriMedia TM1000 the time-indexed OASIC formulation is better suited than the order-indexed SILP formulation. In the following first the results of the optimiser using the OASIC modelling are

Name	Operations	Instructions	Blocks	Loops
dbiquadn	41	18	2	1
dbiquado	21	13	1	0
dcompmul	19	11	1	0
dcompupd	19	11	1	0
dconvolu	47	21	2	1
ddotprod	13	10	1	0
dfir2dim	76	76	2	1
dfir	50	17	2	1
dlms	73	25	3	2
dmat1x3	31	29	2	1
dmatrix1	73	45	2	1
dmatrix2	74	51	3	1
dncompup	33	14	2	1
dnrealup	41	15	2	1
drealup	9	9	1	0
mamu2	62	20	3	1
whetp3	11	34	1	0
dctk	95	21	1	0
fidctk	74	15	1	0

Table 10.8.: Statistics about the input programs for the TriMedia TM1000.

summarised; then the optimiser using the SILP formulation is presented.

OASIC-based Optimiser

Due to the significant instruction-level parallelism of the TriMedia TM1000 it is reasonable to exploit the schedule of the input program when computing the *alap* values of each operation. Since the code produced by the `tmcc` compiler often is already optimal, for the optimiser using the OASIC formulation the length of the input schedule is incremented by the longest execution time of any input operation and this value is used as a heuristic upper bound for the computation of the *alap* control steps. This way, the knowledge about the input schedule is exploited and there is enough scheduling freedom for efficiently computing the ILP-based approximation.

Optimisations restricted to Loop Boundaries. Again in a first phase the optimisations are evaluated where the superblock enlargement is stopped at loop boundaries. Tab. 10.9 summarises the result of the optimisations. The computation method is listed in column m, where a denotes the stepwise approximation, e the exact solution when the schedule of the input programs is used as a start solution and n the exact solution when no start solution is used. Column S shows the number of generated superblocks for each program, column t the total required CPU-time and the number of compacted instructions in the result of each method is shown in column I. The input programs `dfir2dim` and `matrix1` have basic blocks that are too large to allow the computation of an ILP-based solution within a time limit of 8 hours. Therefore for those programs a threshold of 50 operations for the maximal size of any superblock is specified (see Chap. 7).

Again the ILP-based solutions are compared to the result of the generic list scheduling algorithm presented in Sec. 9.3. The list scheduling is heuristically coupled with the resource allocation and selects an available functional unit on the fly when scheduling a machine operation. The results of the list scheduling algorithm are shown in Tab. 10.10. The optimal number of instructions (verified by hand) is listed in column I_o, the number of instructions produced by the list scheduling algorithm is shown in column I, the percentage deviation of it from the optimal solution in column Δ_o and the number of instruction of the input program is listed in column I_{in}.

The figures Fig. 10.11 – Fig. 10.14 give a visualisation of the experimental results. In Fig. 10.11 and Fig. 10.12 the number of compacted instructions in the result of the ILP-based methods is compared to the schedule of the input programs and the result of the list scheduling algorithm. Again the vertical axes show the number of compacted instructions, the horizontal axes the different computation methods. A visualisation of the computation time is given in Fig. 10.13 and Fig. 10.14; for each input program and each solution method the measured computation time in seconds is shown on a logarithmic scale.

Name	m	S	t	I	Name	m	B	t	I
`dbiquadn`	a	3	1.59”	18	`dmatrix1`	a, t50	4	27’ 4.01”	38
	e		2.93”	18		e, t50		2’ 28.91”	37
	n		20.91”	18		n, t50		4’ 52.44”	37
`dbiquado`	a	1	6.23”	13	`dmatrix2`	a	5	33’ 55.28”	45
	e		0.37”	13		e		5’ 9.33”	45
	n		0.73”	13		n		5’ 10.8”	46
`dcompmul`	a	1	4.34”	11	`dncompup`	a	3	21.7”	14
	e		0.48”	11		e		1.42”	14
	n		2.27”	11		n		28.57”	14
`dcompupd`	a	1	4.29”	11	`dnrealup`	a	3	34.46”	15
	e		0.47”	11		e		1.92”	15
	n		1.7”	11		n		1.91”	15
`dconvolu`	a	3	5’ 44.79”	21	`drealup`	a	1	1.09”	9
	e		5’ 34.95”	21		e		0.21”	9
	n		10’ 30.82”	21		n		0.24”	9
`ddotprod`	a	1	1.81”	10	`mamu2`	a	4	1’ 6.32”	19
	e		0.26”	10		e		10.25”	19
	n		0.36”	10		n		2’ 30.29”	19
`dfir2dim`	a, t50	4	24’ 19.14”	58	`whetp3`	a	1	1’ 2.98”	34
	e, t50		3’ 3.18”	58		e		2.4”	34
	n, t50		2’ 9.38”	58		n		3.64”	34
`dfir`	a	3	1’ 46.17”	17	`dctk`	a	1	9’ 30.25”	22
	e		1’ 4.9”	17		e		30.21”	21
	n		3’ 26.56	17		n		1h 20’ 7.75”	21
`dlms`	a	4	1’ 49.03”	26	`fidctk`	a	1	3’ 0.22”	16
	e		4.1”	25		e		6.93”	15
	n		4’ 22.0”	27		n		18’9.06”	15
`dmat1x3`	a	3	1’ 41.44”	25					
	e		16.16”	25					
	n		26.23”	25					

Table 10.9.: Performance of the OASIC-based optimisations for the TriMedia TM1000 not exceeding loop boundaries.

Figure 10.11.: Compacted instructions produced by the OASIC-based optimisations for the TriMedia Tm1000 within loop boundaries (1).

Name	$t[sec]$	I	I_o	$\Delta_o[\%]$	I_{in}	Name	$t[sec]$	I	I_o	$\Delta_o[\%]$	I_{in}
`dbiquadn`	0.33	20	18	11.1	18	`dmatrix1`	1.89	42	37	13.51	45
`dbiquado`	0.06	14	13	7.7	13	`dmatrix2`	1.83	46	45	2.2	51
`dcompmul`	0.05	11	11	0	11	`dncompup`	0.20	14	14	0	14
`dcompupd`	0.05	11	11	0	11	`dnrealup`	0.31	16	15	6.67	15
`dconvolu`	0.48	21	21	0	21	`drealup`	0.02	9	9	0	9
`ddotprod`	0.03	10	10	0	10	`mamu2`	1.14	19	19	0	20
`dfir2dim`	1.95	57	57	0	76	`whetp3`	0.04	34	34	0	34
`dfir`	0.56	19	17	11.76	17	`dctk`	3.89	30	21	42.85	21
`dlms`	1.67	29	25	16	25	`fidctk`	1.87	19	15	26.67	15
`dmat1x3`	0.17	25	25	0	29						

Table 10.10.: Results of list scheduling.

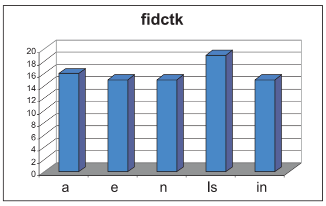


Figure 10.12.: Compacted instructions produced by the OASIC-based optimisations for the TriMedia TM1000 within loop boundaries (2).

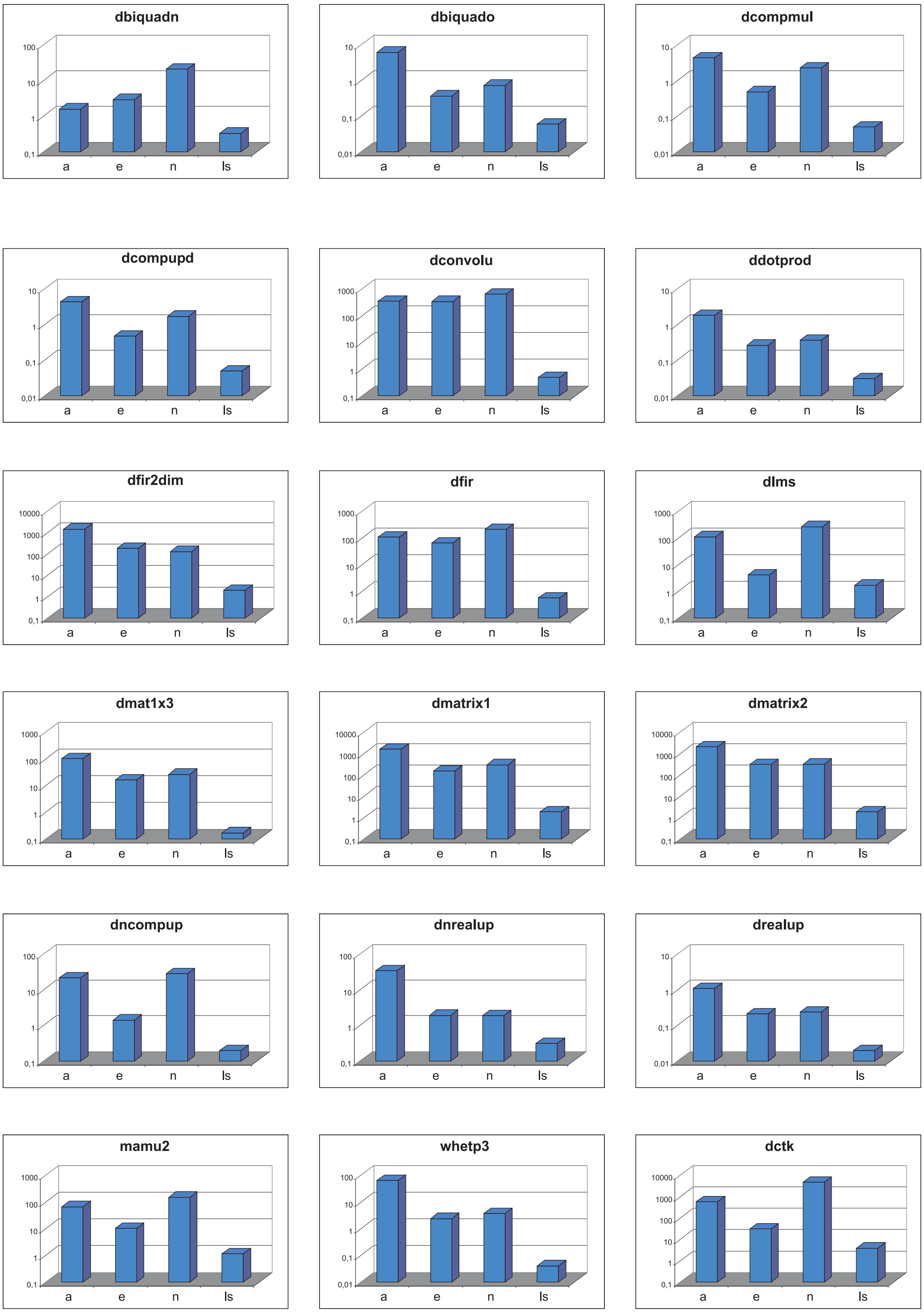


Figure 10.13.: CPU-time in seconds of the OASIC-based optimisations for the TriMedia TM1000 within loop boundaries (1).

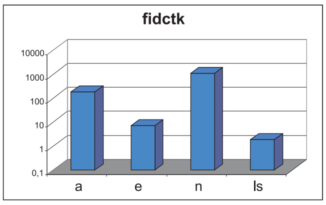


Figure 10.14.: CPU-time in seconds of the OASIC-based optimisations for the TriMedia TM1000 within loop boundaries (2).

In most cases the ILP-based approximation produces optimal results, only for 26% of the input programs (`dfir2dim`, `dlms`, `dmatrix1`, `dctk`, and `fidctk`) the number of compacted instructions exceeds the optimal number of instructions by one. Another observation is that for `dlms`, and `dmatrix2` the exact ILP-based solution without using a start solution does not produce a globally optimal result. This effect is due to the isolated optimisation of several superblocks. During the optimisation of a superblock s the impact of the scheduling and allocation decisions on subsequently addressed superblocks is not visible. Thus an optimal scheduling and allocation of one superblock may not be optimal if the whole program is considered. For `dlms` and `dmatrix2` the input schedule corresponds to the globally optimal solution. When using this schedule as a start solution the ILP optimisation (method e) states its optimality and reports it as an optimal solution. If the input schedule is not used as a start solution (method n), another solution is computed that is locally optimal for all the superblocks but does not represent a globally optimal solution. For both input programs, the scheduling and allocation decisions of the stepwise approximation lead to a better overall solution than method n. For `dfir2dim` all ILP-based optimisations exceed the globally optimal number of instructions by one while the list scheduling can produce an optimal result. In this case the suboptimality is due to the splitting of the largest basic block into two parts which is required to limit the computation time. In this case an improvement can be achieved by more elaborate partitioning algorithms for splitting basic blocks which is a topic of future research. So the notion of optimality has to be used carefully; the exact ILP-based methods produce optimal results with respect to the optimisation scope and the incorporated code generation problems.

The schedule produced by the generic list scheduling algorithm exceeds the optimal number of instructions on average by 7.29%. The programs where it gives an optimal result offer not enough instruction-level parallelism for the allocation decisions to negatively affect the code quality. The two hand-crafted assembly programs however offer a high level of parallelism and make use of special DSP operations that can only be assigned to few issue slots. For those programs, the interaction between instruction scheduling and resource allocation has a significant effect which becomes apparent in the result of the list scheduling algorithm. The optimal number of instruction is exceeded by more than 25% respectively more than 42%. The computation time of the ILP-based approximation has properties similar to the SILP-based approximations. Due to the approximation setup time, for the smaller input programs the exact ILP-based solution takes less time than the ILP-based approximation. With increasing size of the superblocks however the computation time is reduced compared to the exact solution (method n). For all investigated programs, an ILP-based solution could be obtained within less than 6 minutes. Another observation is that using the schedule of the input code as a start solution can speed up the calculation of the exact solution significantly (method e). Thus proving a given solution to be in fact optimal can be done comparatively fast. The list scheduling algorithm again produces its results within some seconds.

Optimisations across Loop Boundaries. In this paragraph the results of the optimisations when superblocks are maximally extended across loop boundaries and each input program is modelled by an individual integer linear program are shown. In Tab. 10.11 the most important characteristics of the generated integer linear programs are shown. Column x shows the number of binary variables associated with functional unit resources, and column w the number of binary variables associated with the result bus. Column V lists the total number of binary variables. The number of precedence constraints is given in column p, the total number of constraints in column C, and the last column lists the sizes of the generated integer linear programs in the CPLEX LP-format [ILO99]. Since the size of the integer linear program generated for `dfir2dim` exceeds 500 MB the sizes of the ILPs of each superblock are shown when no loop boundaries are crossed[1]. The total number of binary variables ranges from 198 to 11562, the number of constraints from 307 to 68432, and the size of the integer linear programs ranges from 32.52 KB to 166.2 MB.

The optimisation results of the input programs consisting of one basic block (`dbiquado`, `dcompmul`, `dcompupd`, `ddotprod`, `drealup`, `whetp3`, `dctk`, and `fidctk`) are the same as in Tab. 10.9. No ILP-based solution could be obtained within 8 hours for the ILPs modelling the complete input program in the case of `dmatrix2`, `dfir2dim`, `dlms`, and `dmatrix1`. In order to achieve lower computation times for those programs the optimisation scope should remain restricted to loop boundaries. For the remaining programs the results of the optimisations exceeding loop boundaries are shown in Tab. 10.12. Column m lists the computation method where a denotes the stepwise approximation, e the exact solution when the input schedule is used as start solution and n denotes the exact solution computed from scratch. Column t shows the measured CPU-time and column I the number of compacted instructions in the result of each method. The results are visualised in Fig. 10.15 and Fig. 10.16, where Fig. 10.15 displays the number of instructions in the result code and Fig. 10.16 compares the measured computation times in seconds on a logarithmic scale.

The results show again a high solution quality of the ILP-based approximation;

[1] One superblock corresponds to the exit block which for this program is empty; thus only three superblocks are listed (cf. Tab. 10.9).

Name	x	w	V	p	C	size[KB]
dbiquadn	1465	349	1815	480	4366	2361.78
dbiquado	283	94	378	86	660	81.59
dcompmul	363	95	459	98	782	102.70
dcompupd	318	95	414	114	777	105.09
dconvolu	2334	677	3012	1181	11098	7096.57
ddotprod	201	60	262	34	422	46.49
dfir2dim	228	48	276	0	323	40.47
	2970	671	3641	876	13125	19255.58
	1609	449	2058	595	5448	2121.04
dfir	1834	469	2304	980	7111	2617.14
dlms	3560	918	4478	2274	18693	11308.50
dmat1x3	2069	502	2572	7880	625	8404.96
dmatrix1	8312	1801	10113	3102	55754	124782.44
dmatrix2	9419	2142	11562	3595	68432	170159.07
dncompup	1264	311	1576	478	3808	1556.68
dnrealup	1447	372	1820	674	4918	2022.52
drealup	154	43	198	16	307	32.53
mamu2	3733	888	4622	1336	13785	9886.20
whetp3	695	245	941	222	2942	1640.54
dctk	1998	915	3312	891	6212	1404.74
fidctk	1557	388	1945	585	3996	1084.44

Table 10.11.: Characteristics of the ILPs for the TriMedia TM1000 in the OASIC formulation.

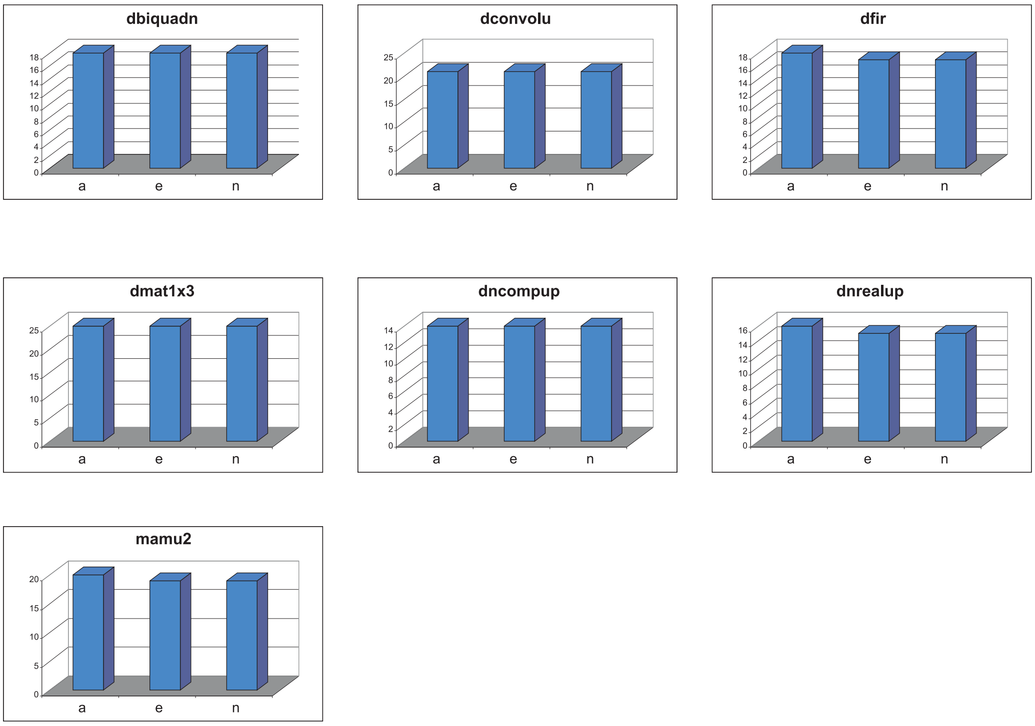


Figure 10.15.: Compacted instructions produced by optimisations across loop boundaries.

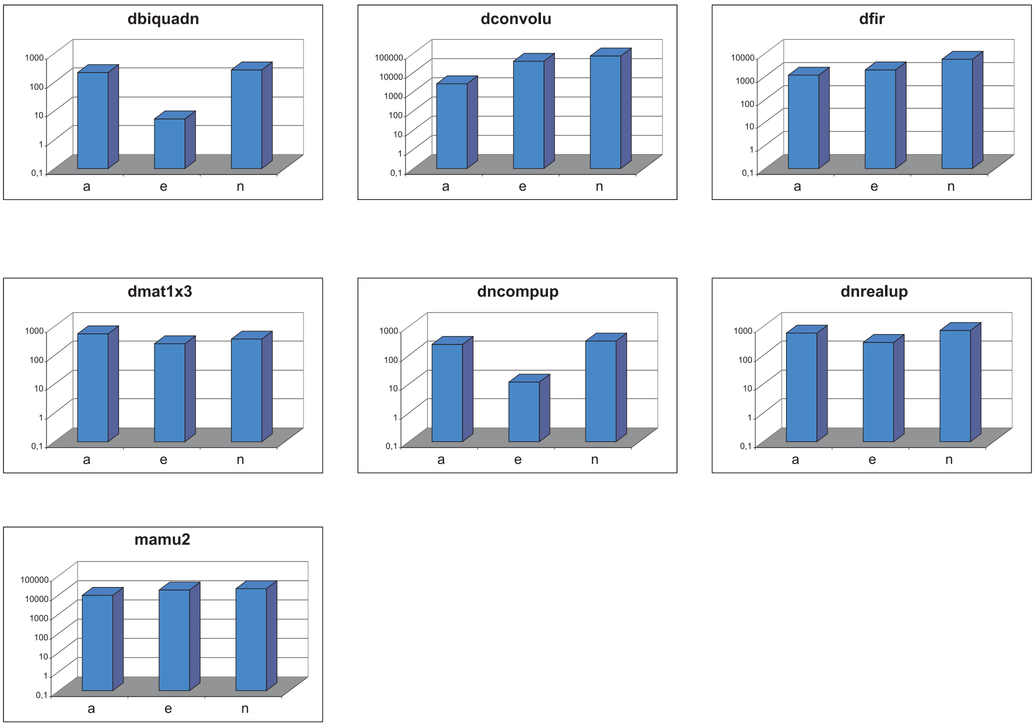


Figure 10.16.: CPU-time in seconds of optimisations across loop boundaries.

Name	m	S	t	I
`dbiquadn`	a	1	3' 31.73"	18
	e		5.23"	18
	n		4' 33.11"	18
`dconvolu`	a	1	43' 37.78"	21
	e		11h 2' 25.42"	21
	n		19h 47' 20.76"	21
`dfir`	a	1	18' 39.23"	18
	e		32' 0.28"	17
	n		1h 30' 15.15"	17
`dmat1x3`	a	1	9' 22.64"	25
	e		4' 12.53"	25
	n		6' 6.25"	25
`dncompup`	a	1	4' 0.16"	14
	e		11.92"	14
	n		5' 9.81"	14
`dnrealup`	a	1	9' 52.27"	16
	e		4' 47.33"	15
	n		12' 11.56"	15
`mamu2`	a	1	2h 32' 22.36"	20
	e		4h 50' 34.95"	19
	n		5h 33' 4.93"	19

Table 10.12.: Performance of the OASIC-based optimisations for the TriMedia TM1000 across loop boundaries.

for 4 input programs it produces an optimal result and for three input programs, the optimal number of instructions is exceeded by 1. It becomes apparent that for large input programs the computation time of the ILP-based approximation can be reduced significantly when compared to the exact ILP-based solutions. For 4 of the 7 input programs, an ILP-based solution could be obtained within less than 10 minutes; for the largest input program, `mamu2`, whose ILP contains 4622 binary variables and 13785 constraints, the approximative ILP-based solution could be obtained in 2.5 hours.

SILP-based Optimiser

In the following the evaluation of the optimiser using the SILP formulation is presented for the TriMedia TM1000. For most operations several issue slots are available that are modelled by different resource types in the generated integer linear programs. The consequence for the SILP formulation is that there is a large number of alternative resource flows for each operation. Thus a large number of binary variables is required which results in high computation times. In the experimental evaluation presented in the following the performance of the SILP-based optimiser is investigated when the superblocks are maximally extended across loop boundaries. Since for the larger input programs this results in computation times of several hours, only the results of the shorter programs are listed. By explicitly specifying an upper bound U for the execution time of the input program when calculating the *alap* control steps for each operation, U is imposed as an upper bound for the length of any feasible schedule produced by the ILP-based approximations. Starting from the length of the input schedule, this bound is incremented for each ILP-based approximation until it finds a feasible schedule not exceeding this bound.

The characteristics of the generated integer linear programs are shown in Tab. 10.13 where x denotes the number of flow variables for functional units, w the number of flow variables associated with the result bus, and V the total number of binary variables. The number of precedence constraints is listed in column p, column s shows the number of serial constraints for functional units, ws the number of serial constraints for the result bus and column C lists the size of the generated ILPs. It is obvious that the number of flow variables and of constraints is significantly larger than those generated by the ADSP-2106X SHARC-optimiser for comparable program sizes (cf. Tab. 10.5). Moreover we can see that the required number of binary variables also is considerably larger than in the integer linear programs of the OASIC formulation (cf. Tab. 10.11).

The results of the experimental evaluation are shown in Tab. 10.14. Again, column m shows the computation method where a_1 denotes the approximation of isolated flows, a_2 the stepwise approximation of isolated flows, a_3 the stepwise approximation and e the exact solution. Additionally the combinations of the approximations a_1, a_2 and a_3 with the approximation of isolated operations (a_4) are investigated. The required CPU-time is shown in column t, the number of

Name	x	w	V	p	s	ws	C	size[KB]
dbiquado	2120	206	2327	38	216	206	2508	277.89
dcompmul	2467	167	2635	34	201	167	3282	332.41
drealup	522	34	557	12	43	34	641	59.81
dmat1x3	7983	546	8529	60	668	546	24609	1644.99
ddotprod	1000	82	1083	21	91	82	1075	124.68
dcompupd	1969	160	2130	42	181	160	2822	274.90
whetp3	558	54	613	17	62	54	576	68.67

Table 10.13.: Characteristics of the ILPs generated for the TriMedia TM1000 in the SILP formulation across loop boundaries.

compacted instructions in the result of each method in column I.

When analysing the results of the ILP-based approximations we can see that the stepwise approximation produces the best results for the TriMedia TM1000; for all listed input programs an optimal result is computed. Due to the large number of alternative functional units between which a selection has to be made, the efficiency of the approximations based on the resource flows decreases. Since the approximation of isolated flows and the stepwise approximation of isolated flows address the resource types one after another, the operations are often fixed as late as possible to a resource type, i. e. when the iteration reaches the last resource type available for the operation. If this is the last available resource type for many different operations, they will all be assigned to that resource type, although a more efficient resource usage would have been possible. Since there is exactly one instance of each resource type, i. e. of each issue slot, this results in lower instruction-level parallelism. For `dbiquado`, both approximations exceed the optimal number of instructions by one, for `dcompmul` even by three. For the other input programs they produce an optimal result.

When combining the approximation of isolated flows (a_1) or the stepwise approximation of isolated flows (a_2) with the approximation of isolated operations (a_4) the code quality is improved for `dbiquado` and `dcompmul` but is decreased for `ddotprod`. The approximation of isolated operations iterates through all operations of the input program and decides which resource each operation should be mapped to. This results in a greedier resource assignment which, depending on the input program, can lead to a code improvement or deterioration. The results indicate that the combination of the stepwise approximation with the approximation of isolated operations is not profitable; for most input programs it leads to a decrease of code quality. Due to solving an individual mixed integer linear program for each operation prior to approximation a_1, a_2 or a_3, the computation time of the combined approximations is usually higher than without this combination.

For the example programs shown in Tab. 10.14 an ILP-based approximative solution could always be obtained less than one minute. For the smaller input programs the approximation of isolated flows and the stepwise approximation of

Name	m	t	I	Name	m	t	I
dbiquado	a_1	3.12”	14	ddotprod	a_1	1.07”	10
	$a_1 + a_4$	19.94”	13		$a_1 + a_4$	3.52”	11
	a_2	15.81”	14		a_2	3.42”	10
	$a_2 + a_4$	25.66”	13		$a_2 + a_4$	5.17”	11
	a_3	12.00”	13		a_3	4.88”	10
	$a_3 + a_4$	22.56”	13		$a_3 + a_4$	4.48”	11
	e	2.11”	13		e	2.17”	10
dcompmul	a_1	11.57”	14	dcompupd	a_1	6.07”	11
	$a_1 + a_4$	22.45”	12		$a_1 + a_4$	16.37”	11
	a_2	22.96”	14		a_2	27.41”	11
	$a_2 + a_4$	26.70”	12		$a_2 + a_4$	23.88”	11
	a_3	53.69”	11		a_3	17.87”	11
	$a_3 + a_4$	25.20”	13		$a_3 + a_4$	19.78”	11
	e	4h 21’ 38.08”	11		e	7h 14’ 9.87”	11
drealup	a_1	0.45”	9	whetp3	a_1	0.44”	34
	$a_1 + a_4$	0.97”	9		$a_1 + a_4$	1.30”	34
	a_2	1.07”	9		a_2	1.41”	34
	$a_2 + a_4$	1.43”	9		$a_2 + a_4$	2.17”	34
	a_3	0.74”	9		a_3	3.80”	34
	$a_3 + a_4$	1.17”	9		$a_3 + a_4$	2.40”	34
	e	0.30”	9		e	0.35”	34
dmat1x3	a_1	54.84”	25				
	$a_1 + a_4$	5’ 23.87”	25				
	a_2	2’ 1.37”	25				
	$a_2 + a_4$	4’ 21.71”	25				
	a_3	6’ 21.12”	25				
	$a_3 + a_4$	6’ 19.31”	26				
	e	5’ 8.63”	25				

Table 10.14.: Performance of the SILP-based optimisations for the TriMedia TM1000 across loop boundaries.

isolated flows mostly require more time than the stepwise approximation; for the largest input program, `dmat1x3`, however they are almost 7 times faster. `dcompmul` and `dcompupd` show that the ILP-based approximations can reduce the computation in the order of several magnitudes in comparison with the exact solution. With increasing program size however the computation time of the SILP-based optimisations increases fast so that also the approximative methods take several hours.

10.3. Summary

Because of the exponential space consumption of the OASIC model when integrating the register assignment, the optimiser for the ADSP-2106X SHARC is based on the SILP modelling. Our experimental results suggest that input programs up to a size of 30–50 machine operations can be modelled by a single integer linear program. For input programs exceeding this size it is advisable to stop the superblock enlargement at loop boundaries or to specify an explicit upper bound on the maximal superblock size. Then the program is covered by several superblocks each of which is modelled by an individual integer linear program. This can also imply splitting single basic blocks exceeding the code size threshold so that each partition is represented by a single superblock. In our experimental evaluation, solving the integer linear program that modelled the complete input program led to a computation time of more than 8 hours for 6 out of 23 investigated programs. For those programs the superblock construction was stopped at loop boundaries; then in most cases an ILP-based solution could be obtained within a few minutes. A splitting of basic blocks was only required for two input programs in order to restrict the optimisation time. In the experimental evaluation, bounds for the maximal superblock size have only been specified if the computation time would have exceeded several hours. The computation times can further be reduced by choosing smaller size limits. An additional speed-up can be achieved by specifying a time limit for the ILP-solver after which the best feasible solution found so far is returned.

The variation of the maximal superblock size for which a fast ILP-based solution can be obtained is caused by the individual properties of the input programs. Here the number of precedence constraints, overlapping life ranges and irregular hardware constraints like the explicit parallelisation prohibition of machine operations have to be considered as well as the number of generated path constraints. The maximal length of the paths modelled by path constraints can be bounded by an explictly specified parameter. Our results indicate that small bounds can be chosen without causing infeasibility. Thus the number of path constraints can be assumed to be very small in spite of their exponential worst case number.

The experimental evaluation shows that ILP-based approximations can reduce the computation time by orders of magnitude compared to the exact solution. In most cases the approximations produce an optimal result; the optimal number of instructions was only exceeded in 5 of 108 approximative computations and in one case the stepwise approximation could not produce its result within 12 hours. Four of the suboptimal results were produced by the stepwise approximation, one by the stepwise approximation of isolated flows. So for the ADSP-2106x SHARC the approximation of isolated flows which always produced an optimal result and the stepwise approximation of isolated flows which produced an optimal result in all but one case are superior to the stepwise approximation. For larger input programs, the computation time required by the stepwise approximation of isolated flows is considerably smaller than that of the approximation of isolated flows. The computation time of the stepwise approximation strongly depends on the input program and is subject to large variations. Another observation is that the limitation of the superblocks to loop boundaries did not negatively affect the code quality. This can be explained by the fact that all operations have an execution time of one clock cycle and the assignment of operations to functional units is uniquely determined. Thus interdependencies between the superblocks are only caused by the register assignment. For the investigated programs the number of overlapping life ranges was small enough so that an optimal register assignment could always be found, also when the input program was covered by several superblocks. The list scheduling algorithm is usually faster than the ILP-based methods; it produces its results in a few seconds. Although the register assignment in the input programs for the list scheduling algorithm is always optimal, its code quality is lower than that of the ILP-based methods. The largest deviation in the investigated programs is 21.05%; with a suboptimal register assignment of the input program this difference can grow significantly larger. The code produced by the `gcc`-based `g21k` compiler exceeds the optimal number of instructions on average by 8.2%.

The architecture of the TriMedia TM1000 is close to the worst-case architecture for the SILP formulation. There is a large number of alternative resource types (modelling the issue slots) between which a selection has to be made and of which exactly one instance exists. This leads to a large number of flow variables so that the computation time of the SILP-based optimiser for the TriMedia TM1000 is comparatively high. Although no register assignment is incorporated the code size limit in which fast ILP-based solutions can be obtained is 30–40 microoperations. Due to the overlapping resource usage (e. g. `shift` operations can be assigned to issue slots 1 and 2, `dspalu` operations to slot 2 and 3), the code quality of the approximation of isolated flows and the stepwise approximation of isolated flows decreases; they exceed the optimal number of instructions in two of seven input programs. The combination of these approximations with the approximation of isolated operations led to an improvement of code quality in two cases, in one case to a deterioration. The effect of the combined approximations is a greedier resource assignment whose efficiency depends on the input program. For the TriMedia TM1000, the stepwise approximation of the SILP formulation produced the best results; it gave optimal results for all investigated programs.

For the TriMedia TM1000, the OASIC modelling outperforms the SILP formulation. With tight upper bounds on the execution time of the input program the number of binary variables of the integer linear programs in the OASIC formulation is significantly lower than in the SILP formulation. The experimental evaluation indicates that input programs containing up to 70–100 microoperations can be modelled by a single integer linear program with acceptable computation times. For input programs exceeding this size it is again advisable to stop the superblock enlargement at loop boundaries or to specify an explicit upper bound on the maximal superblock size. For two of 19 investigated programs a splitting of basic blocks and for additional two programs a restriction of superblocks to loop boundaries were required to limit the computation time. Increasing the number of superblocks covering the input program can lead to a deterioration of the solution quality; this has been observed in three cases. The reasons are the restrictions of the resource assignment in combination with the fact that the execution of operations can take several machine cycles. Thus there is a trade off between the required computation time and the achieved code quality so that within the ILP-based methods the code quality can be scaled. The experimental evaluation shows that while the code quality of the `tmcc` compiler is already very high, in some cases it could be improved significantly. Another observation is that proving an optimal input solution to be optimal can be done comparatively fast by exactly solving the ILPs using the schedule of the input program as a start solution. The code quality of the ILP-based approximation is high. In 18 of 26 cases it produced an optimal result; for the remaining programs, the optimal number of instructions was exceeded only by one. For large input programs the approximative method allows to reduce the computation time significantly when compared to the exact solution. The list scheduling algorithm always produced its results in a few seconds; the generated code however exceeded the optimal number of instructions by up to 42%.

To summarise, the PROPAN system has been successfully retargeted to generate ILP-based postpass optimisers for two representative standard DSPs. Our experimental evaluation indicates that the ILP-based methods can be applied to realistic input programs representing digital signal processing kernels. The achieved code quality is very high. For large input programs exceeding the size of typical DSP kernels, the computation time can be kept low by restricting the application of the ILP-based methods to hot code sequences which are often executed and whose performance is critical, e. g. nested loops. Less important code sequences can be addressed by faster heuristic approaches.

11. Related Work

The PROPAN framework has been developed as a system for retargetable code optimisations and analyses. In this chapter an overview of related work in the fields of retargetable code generation and code generation for irregular architectures is given with special emphasis to phase-coupling issues. Additional surveys of related work are given in individual chapters where specific aspects of PROPAN are presented in detail. Classical methods for code generation, especially for code selection, register allocation and instruction scheduling are summarised in Chap. 2. An overview of publications about the structure of scheduling and code generation polytopes is given in Chap. 5. Chap. 6 includes a short summary of combinatorial approximation algorithms and general search-based heuristic algorithms. A survey of related work in the field of hardware description languages is given in Sec. 8.1.

The following overview is structured as follows: first a summary of historical and contemporary retargetable compilers and code generators is given. It is only during the last years that the issues of code generation for irregular architectures have gained increased attention. Thus the presentation first covers retargetable compiling for regular architectures and then turns to frameworks developed in the area of embedded systems. Because of the importance of phase-coupled code generation a dedicated survey is given that concentrates on phase coupling approaches based on graph-based heuristic methods. The last part of this overview focuses on exact, search-based strategies that have been developed for code generation issues, mostly for arriving at phase-coupled problem descriptions.

11.1. Retargetable Code Generation

One of the earliest retargetable systems in the area of code generation and optimisation is the PO system [DF80, DF84]. PO is a retargetable peephole optimiser that takes assembly files as input. The basic idea of PO is to perform all optimisations on a machine-independent representation of the machine operations. The input program is transformed into sequences of machine-independent register transfer lists (RTL code). PO considers the combined effect of lexically adjacent instructions and, where possible, it replaces such pairs with a single instruction having the same effect. In order to retarget PO a grammar has to be provided that describes a mapping of machine operations to RTL sequences. Only the operations that are relevant for the peephole optimisations have to be specified. Descendants of PO are the `vpo`-system and the `gcc` compiler.

In [Gie82] an approach for automatically generating machine specific code optimisers is presented. The optimiser generator takes a machine description in an ISP-like notation [Bar81], generates a machine specific program analyser based on abstract interpretation, and a code transformer. The generation of the code transformer is based on a set of standard rules that are part of the generator, additional user-defined rules and user-supplied implementation decisions. Supported transformations are, e. g., elimination of redundant instructions, changes of addressing modes, elementary replacements as in PO, or elimination of jump chains.

`vpo` [BD88, BD94] is a portable code improver for RISC and CISC processors that reads RTL code sequences as an input and performs a set of efficiency-increasing program transformations as, e. g., dead code elimination, common subexpression elimination, loop-invariant code motion, etc. There is no coupling of code generation phases. A frontend that transforms the source program into the RTL sequences has to be provided externally to `vpo`. `vpo` can be retargeted by specifying how to map the RTL code to machine operations and by implementing a set of target-specific C functions, e. g. in order to deal with calling conventions, etc. Additional target specific optimisations can be incorporated into the system.

The GNU `gcc` [Sta98, Nil98] compiler is a widely used portable compiler that has been retargeted to various processors. In contrast to `vpo`, `gcc`'s frontend and backend are tightly coupled. The source code is transformed into sequences of machine-independent register transfer operations (RTL code) that can be extended by target-specific information. All code optimisations work on the RTL code; there is no coupling of code generation phases. The code selection is based on pattern matching. In order to retarget the compiler the user has to specify pattern matching rules describing how to generate machine operations from the RTL code; for complex instructions it may be necessary to provide dedicated C functions. Additional information about the target architecture is provided in the form of C data structures and macros such as, e. g., endianess, the assembly representation of registers, delay slots, etc. Since the machine specification of `gcc` represents basically the implementation of the code selector, it tends to be very complex. `gcc` has been primarily designed for RISCs and CISCs; the code quality for irregular architectures mostly is not satisfactory [ZSWS95, BL99].

`lcc` [FH91, FHP92, FH95] is a portable compiler for ANSI-C that has originally been developed for use in teaching. For the sake of simplicity no global optimisations and only few local optimisations such as, e. g., the elimination of common subexpressions are performed. The frontend of the compiler and parts of the backend are architecture independent; the program parts required for code generation have to be implemented separately for each target processor. The code selector is based on tree pattern matching and is generated by the code selector generator `iburg`. The developer has to provide a tree grammar for the instruction set of the target processor. Further architecture specific functions are required for procedure calls, parameter passing, memory layout and for register allocation. The focus of `lcc` is on RISC and CISC architectures. Typical features of DSPs like heterogeneous register files are not supported. There is no phase coupling.

Marion [BHE91, Bra91] is a retargetable code generator system designed specifically for RISC architectures. Modelled target architectures are the MIPS R2000 [Kan87], the Motorola 88000 [Mot88], and the Intel i860 [Int89]. The code generators are built from specifications of the target architectures written in the machine description language Maril (see Sec. 8.1). Marion uses the `lcc` front end; the code selection is done by a recursive-descent tree pattern matcher. The emphasis of the Marion system is on instruction scheduling, register allocation and a heuristic phase coupling between both tasks. For instruction scheduling and register allocation heuristic graph based algorithms are used. Instruction scheduling is implemented by a list scheduling algorithm, and register allocation is based on graph colouring [Cha82, CH90]. The core of Marion is a code generation policy called RASE where instruction scheduling and register allocation communicate with each other. First, a pre-scheduler is invoked which computes schedule cost estimates that allow the subsequent register allocation phase to quantify the effect of its allocation choices on the scheduler. Then the final schedule is produced.

The Trimaran system [tri98] has especially been designed to provide a research platform for code generation issues for VLIW architectures. It is based on a parameterisable research architecture called HPL-PD supporting features like speculative and predicated execution, and compiler-visible cache hierarchy. The architecture is parameterisable with respect to the sizes of register files, the number of functional units and operation latency values; a dedicated machine description language Mdes [GHR96, RKA99] is available to specify the architecture parameters. The highly optimising frontend Impact [HHR97] transforms the input programs into an extensible intermediate representation. The backend Elcor is parameterised with respect to the machine specification and performs instruction scheduling, register allocation and machine dependent optimisations. Its focus is on transformations to increase the available parallelism like if-conversion [PS91, DT93] and software pipelining algorithms [AJLA95].

The Salto system [BCRS97] to our knowledge is the only existing retargetable framework explicitly supporting efficiency-increasing transformations on assembly programs apart from the retargetable peephole optimiser PO [DF80, DF84] and the Propan framework. The basic idea of Salto is to provide the user with an environment that allows to implement tools for analyses and transformations of assembly code. From the description of the target machine and the assembly language a parser is generated that transforms the input program into a generic control flow graph. The user is offered an object oriented interface to access and manipulate the data structures representing the CFG.

While the previously presented frameworks mostly have primarily been developed for RISC and CISC architectures, the following approaches have been developed in the area of digital signal processing. Mimola [Now87, MS93] is a system for hardware/software codesign of digital programmable processors. It comprises tools for hardware synthesis and simulation and a retargetable code generator MSSQ [Now87]. All components are based on the structural Mimola hardware description language. The source language of MSSQ is a hardware-oriented superset of the PASCAL language; it produces executable code. Code selection and register allocation are performed locally for each expression tree of the intermediate representation. In a subsequent compaction phase the final binary code is generated. Since there are no global optimisations, the quality of the generated code is not satisfactory [Leu97].

RECORD [Leu97, LM94, LM97] is a retargetable compiler for fixed point DSPs; however the only target architecture actually modelled is the TI TMS320C25. The input language of RECORD is DFL[1]; it produces executable binary code. The target architecture is specified by the MIMOLA description language. In a first stage, the RECORD system extracts the instruction set from the hardware-oriented processor specification. All transport paths between registers and memory in the data paths are enumerated yielding a superset of the feasible machine operations. Additionally the binary encodings of the extracted operations are determined in order to prevent violations of instruction word restrictions. The extracted machine operations are represented as tree patterns. An implicit assumption is that all machine operations can be executed in one clock cycle. From the tree patterns a tree grammar is synthesised that is used as input of the code selector generator `iburg` [FHP92]. The code selection is followed by a register allocation phase and a dedicated algorithm for calculating the layout of program variables in memory in order to efficiently use addressing hardware. The subsequent compaction phase is based on integer linear programming; the task of instruction scheduling is coupled with the selection among alternative binary encodings for the operations. Apart from that there is no phase coupling. Encoding restrictions are taken into account by reusing the information of the instruction set extraction phase.

The retargetable CBC compiler [Fau95, FK93a, FK93b] has mainly been designed for DSPs and ASIPs; the target architectures are specified by the machine description language nML [FVPF95]. The inputs of the compiler are given in the form of flowchart descriptions. In the code selection phase the nodes of the intermediate representation are mapped to complex machine operations [FHMK94]. Then a combined data routing and scheduling phase is executed. Data-routing performs register allocation and delivery path definition, i. e., for each intermediate value the storage location and the necessary transfer operations are determined. These decisions are incorporated into a list scheduling algorithm [Har92]. However none of the published articles gives any results about the quality of the produced code.

CHESS [LVPK$^+$95] is a commercial retargetable code generation framework for fixed point DSPs. The target processor is modelled by instruction set graphs that are automatically extracted from an nML-description of the processor [FVPF95]. Source languages of CHESS are C and DFL. A presupposition is that each instruction is executed in one machine cycle; there is no pipeline modelling. Code selection, register allocation, and instruction scheduling are implemented as separate modules. After code selection, operations are redistributed among basic blocks

[1] *Data Flow Language*

in order to balance the functional unit and register utilisation. The register allocator can take distributed register sets into account by using a data routing algorithm [LCGDM94]. The register allocation decisions are guided by estimates about the effects on instruction scheduling. This way, information is exchanged between the phases, but there is no true phase integration. Restrictions of instruction-level parallelism cannot be modelled. The scheduling is based on an extended list scheduling algorithm supporting software pipelining and different forms of code hoisting.

The retargetable Flexware system [PLMS95] has been especially developed for ASIPs and consists of the code generator CodeSyn and the instruction set simulator Insulin. Insulin is based on a VHDL description while CodeSyn uses a dedicated description language. The machine description used for CodeSyn consists of three parts. The first part specifies the mapping of nodes of a fixed intermediate representation to the machine operations. The second part contains a structural graph that models the data transfer paths between memory banks, register files and functional units of the processor. The third part provides a classification of the resources used in the structural graph that is used, e. g., for register assignment [LMP94]. Prior to the code generation proper the intermediate representation can be transformed by peephole optimisations based on a set of user-defined rules [LCS+97]. There is no phase coupling; code selection, instruction scheduling, and register allocation are performed in separate phases [LCS+97]. The generated code contains on average 20% more instructions than hand-crafted assembly code [PLMS95].

The SUIF library [Sta94] is basically an optimising C frontend. Its intermediate representation is an extension of the intermediate representation of `lcc` where high level information such as information about loop constructs and array accesses can be represented. A set of standard optimisations as, e. g., loop invariant code motion, induction variable detection and elimination or dead code elimination are available. The SPAM compiler [SPA97, Sud98] uses the SUIF library as an optimising C-Frontend. The backend, called TWIF, consists of a parameterisable optimisation library containing typical data structures and algorithms for code generation problems. The code selector is based on tree-pattern matching and is generated by the code selector generator OLIVE from a regular tree grammar. The backend has to be implemented separately for each target processor; in the implementation the parameterisable algorithms of the TWIF-library can be reused. These include algorithms for computing local data dependence graphs, for register allocation, memory bank allocation, and instruction scheduling. However the SPAM library is primarily designed for strongly encoded fixed-point processors with limited amount of instruction-level parallelism. There is no phase coupling; code selection, register allocation, and instruction scheduling are performed in separate phases. Instruction scheduling is performed by a local list scheduling algorithm; there are no global optimisations. Moreover due to the design of the system, developing a new compiler using SPAM requires detailed insight into the internal behaviour of all library routines. The algorithms of the TWIF library can only be reused if the target architecture only requires adaptations with respect to numerical values as, e. g., the number of registers in a register file.

The retargetable code generator AVIV [HD98] is based on the SPAM library. The target architecture is specified in the machine description language ISDL (see Chap. 8). AVIV focuses on DSP architectures exhibiting instruction-level parallelism including VLIW architectures. The intermediate representation of the input program consists of a set of expression DAGs connected by control flow edges. In a first step, the expression DAGs are extended by hardware-specific information; the resulting graphs are called split-node DAGs. In the split-node DAG all functional units available to execute an operation and all register sets or memory banks available to store the operands are explicitly represented. Data transfers between different register sets or memory banks are represented by dedicated nodes. For each split-node DAG a heuristically guided search algorithm performs simultaneous allocation of functional units, operation grouping, and register bank allocation. The search algorithm is similar to a branch-and-bound algorithm; however it does not yield a provably optimal solution since it uses ad-hoc heuristics to prune the search space. The detailed register allocation is calculated in a separate phase after the phase-coupled search algorithm; during the search estimations about the availability of registers are used. Therefore the quality of the generated code depends to a large degree on the quality of the chosen heuristics. There is no optimisation of spill code generated during register allocation which can lead to a significant deterioration of code quality. Each expression DAG is addressed separately; in consequence parallelising operations from different expression DAGs of the same basic block is not possible. Extending the compiler by additional optimisation and analysis algorithms is impeded by the monolithic structure of the code generator. No experimental results have been reported so far; thus the performance of the AVIV compiler cannot safely be judged yet.

In the retargetable compiler Express [HGG+99] the information about the hardware is extracted from the machine description language Expression that is also used to generate a cycle-accurate instruction set simulator. The code generation phase is initiated by a software pipelining algorithm and an algorithm for global code motion. Code selection, register allocation and instruction scheduling are heuristically coupled by mutation scheduling [NN94, NND95]. Each value in the program is associated with a set of mutations corresponding to different sets of machine operations computing that value. The decision whether to keep a value in a register or to store it in memory is considered as a mutation. The mutation sets can change dynamically during scheduling. All decisions of code selection, register allocation and instruction scheduling are guided by heuristics. In the machine description the pipeline paths and all valid data transfer paths are specified. From this specification reservation tables are automatically generated that can be used during mutation scheduling to model the resource usage of each machine operation. This way irregular resource restrictions can be modelled but it is not clear how encoding restrictions can be handled. While the Expression language has been used to model the Texas Instruments TI320C6x, in [NN94, NND95, HGG+99] experiments about the code quality of mutation scheduling have only been reported for artificial homogeneous VLIW architectures without significant irregularities. Details about the achieved code quality and the required compilation time have not been published.

11.2. Heuristic Phase Coupling

There are various approaches aiming at a coupling of code generation phases by heuristic methods. A comprehensive survey of those methods is given in [Bas95]; in the following only a short overview will be presented.

The phase coupling of instruction scheduling and register assignment is considered in [EM92, NPW91, NN93]. The register allocation proper is determined prior to instruction scheduling. During instruction scheduling, false dependences are eliminated by dynamic register renaming; thus the register assignment is incorporated into the scheduling process.

A large number of publications is concerned with the interaction of instruction scheduling and register allocation. In the integrated prepass scheduling algorithm of [GH88] the instruction scheduler precedes the register allocator but attempts to restrict the number of concurrently living local virtual registers by giving each basic block a register limit. If needed, the limit can be increased. After scheduling the register allocator assigns physical registers to the virtual registers and inserts spills to memory if the limit cannot be met. In the RASE algorithm of the Marion system [BHE91, Bra91] the decisions of the register allocator are guided by estimates of their effect on the scheduling (see Sec. 11.1). In [FR91] register allocation is integrated into trace scheduling by a greedy algorithm. In [NP93, Pin93] graph-colouring algorithms for register allocation are extended by considering aspects of parallelism. Further publications in this area are listed in [Bas95].

In the approaches listed above, distributed or irregular register sets as well as interdependencies between the usage of functional units and storage resources are not considered. The Bulldog compiler [Ell86] and the Cbc compiler [Har92] perform local instruction scheduling with greedy register allocation and data routing where data routes are selected on the fly. The algorithm of [LCGDM94] implemented in the Chess compiler selects between different data routes by estimating their effect on the schedule quality. In [RH88] and [Har92] local instruction scheduling is combined with greedy binding of functional units and register allocation on the fly. In [Hei93] a register allocation algorithm combining the selection of different data routes with delayed binding of functional units is invoked on the fly by a trace scheduling algorithm. A heuristically guided phase coupling of code selection, instruction scheduling, and register allocation is addressed in the mutation scheduling algorithm [NN94] (see Sec. 11.1).

11.3. Search-Based Methods in Code Generation

Apart from the heuristic approaches, there are also search-based code generation methods which allow to calculate exact, optimal solutions to the modelled problems — usually at the cost of higher calculation times.

Exact code generation algorithms have been developed for architectures that satisfy special conditions. For simple homogeneous architectures, optimal algorithms for simultaneous code selection and register allocation on expression trees have been presented in [SU70, AJ76]. Other algorithms [AM99, LDKT95, LDK+95] assume that all storage resources, i. e. register files or memory, can be considered to contain exactly one, or an infinite number of elements (accumulator-based architectures). In [AM99] a polynomial algorithm for instruction selection, register allocation and instruction scheduling on expression trees is presented. An additional criterion is given that the target architecture has to satisfy in order for the algorithm to generate optimal code. For the same class of architectures, [LDKT95] propose an algorithm for optimal code selection that works on expression DAGs and is based on binate covering. Since binate covering is NP-complete a heuristic procedure can be used to solve the covering. In [LDK+95] a branch-and-bound algorithm for instruction scheduling and register allocation is presented that minimises the number of accumulator spills.

However there are also approaches that are applicable to broader classes of architectures. In the area of architectural synthesis several ILP formulations have been developed that can be used for phase-coupled code generation. The goal of architectural synthesis is to design the fastest architecture for a given input algorithm that does not exceed a fixed cost maximum, or to design the cheapest architecture for the input algorithm that meets a fixed performance criterion. In order to evaluate the performance capacities of a hardware design it is important to determine the optimal code sequence for the given input algorithm. The ILP model used in the ALPS synthesiser [HLH91, BST93] is a time-indexed formulation for instruction scheduling and functional unit allocation. In [GE92, GE93] an improvement of the ALPS model has been developed by exploiting results of polyhedral theory in order to provide a tighter description of the feasible region of the problem and thus increase the solution efficiency. The resulting formulation has been implemented in the OASIC synthesiser and has been extended to incorporate the register assignment problem. The OASIC formulation represents one of the starting points of our work and is presented in Sec. 5.3 in more detail. While all previously mentioned ILP formulations are time-indexed formulations, in [Zha96] an order-indexed ILP formulation for instruction scheduling, functional unit allocation and register assignment has been developed. Since this formulation also plays an important role for this thesis it is presented in Sec. 5.2 in more detail. Modelling irregular restrictions of instruction-level parallelism and encoding restrictions of a given target architecture is not taken into account since the design of the architecture is part of the synthesis problem. Another part of the system synthesis problem where integer linear programming is used is the partitioning and mapping of algorithms onto processor arrays. Partitioning maps an arbitrary size algorithm onto a processor array with a restricted number of processing elements and searches a balance between local memory consumption, and I/O and communication rate. In [Thi93, Thi95] an ILP-based optimisation model for partitioning with a constrained number of functional units has been presented. ILP formulations for scheduling of partitioned regular algorithms on processor arrays with constrained resources have been presented in [Zha96, TTZ96].

An ILP formulation for simultaneous code selection and register allocation is presented in [Geb97]. The code generation problem is described by logical propositions in the form of Horn clauses [Hoo88] that are automatically translated to ILP constraints. Integer linear programs representing a set of horn clauses can efficiently be solved by a combination of linear programming algorithms with variable rounding. The basic blocks of the input program are represented by data flow graphs. In a first step a scheduling of data flow nodes, i. e. a total linear ordering of the nodes in the data flow graph is computed. Then the integer linear program for the code selection and register allocation is generated. For each set of operation instances a data flow node can be mapped to, a dedicated binary variable is introduced. The modelling of the register allocation requires operation instances that only differ in their register usage to be considered as different. Architectures with register files containing multiple registers — which is a common property for contemporary processor architectures — lead to an exponential problem size. For architectures exhibiting instruction-level parallelism the sequentialisation of the nodes in the data flow graph can lead to a significant decrease in code quality. Thus only a restricted class of architectures can be modelled efficiently. The modelling has been implemented for a subset of the instruction set of the TMS320C2x where, e. g., only direct memory addressing without using address registers is allowed.

There have been only few approaches to incorporate ILP-based methods into the code generation process of a compiler. An early approach for local ILP-based instruction scheduling for vector processors has been presented in [Ary85]. In the Record compiler [Leu97], integer linear programming is used to model local instruction scheduling and selection among alternative binary encodings of operations. Both ILP models are time-indexed formulations similar to the OASIC model [GE92, GE93]. Wilson et al. [MG95] use an ILP-formulation for simultaneously performing code selection, scheduling, register allocation and assignment; movements of operations across basic boundaries however are not addressed. The complexity of the resulting formulations leads to prohibitive computation times. In [GW96] an ILP formulation for the register allocation problem has been presented that in [KW98] is extended to irregular register architecture. The modelled architectural restrictions comprise combined source and destination operands, fixed combinations of register and memory operands, and overlapping registers. Most functions of the SPECint92 benchmark can be optimised within a time limit of 1024 seconds. An ILP model for local instruction scheduling and register allocation has been presented in [CCK97]; experimental results are only reported for some small example programs and show exhaustive computation times. In [HLW00] it is demonstrated that by using a well-structured ILP formulation, an optimal local instruction scheduling of large basic blocks for regular architectures can be computed in acceptable time. Other ILP-based approaches have been developed in the context of software pipelining; for more information see e. g. [RGSL96, GAG96].

The application of evolutionary algorithms (see Sec. 6.1) in the field of hardware and software synthesis is the subject of [TZB99, ZT99, ZTB00]. An overview of multiobjective evolutionary algorithms is given in [Bli96, ZT99] and the applicability to architectural synthesis is investigated. In the approach of [ZT99] the individuals of the evolutionary algorithm encode the set of allocated hardware resources and the binding of instructions to resources. The scheduling is performed by a list scheduling algorithm with loop pipelining. In [TZB99, ZTB00] evolutionary algorithms for software synthesis in rapid prototyping environments are investigated. A commonly used approach is to store optimised assembly code for so-called actors in a target-specific library. Then from a restricted data flow graph whose nodes represent the actors, code is generated by instantiating the actors' code in the final program. An evolutionary algorithm for the optimisation problem of minimising program memory usage, data memory usage, and execution time is presented. Considered are the memory usage, the effects of procedure inlining or subroutine calls as well as the effect of loop nesting and context switching. The outcome is a set of optimal trade-offs which consist of all solutions that cannot be improved in one criterion without degradation in another.

The ICG-compiler [BL99] models the code generation problem as a constraint satisfaction problem; it is implemented in the constraint logic programming language ECLiPSe. The basic blocks of the input program are represented by data flow graphs. In a first step a covering of each data flow graph by factorised register transfers is computed. Each factorised register transfer represents all machine operation instances that can be generated for a node in the data flow graph. Interdependencies between alternative machine operations and the usage of storage resources are modelled by dedicated constraints associated with the factorised register transfers. The resulting set of all possible coverings of the DFG by machine operations can be pruned by heuristic methods; when neglecting instruction-level parallelism, alternatively an optimal covering can be computed. Subsequently a list scheduling algorithm is executed that performs register allocation and data routing on the fly. The algorithms work on factorised register transfers such that decisions of the code selection can be delayed until the scheduling and allocation phase. The scheduling and allocation decisions are represented by additional constraints. Since each data flow graph is addressed separately, the parallelisation of operations from different data flow graphs of the same basic block is not supported. After all data flow graphs have been traversed a labelling has to be performed in order to check the global feasibility of the generated schedule and to select machine instructions from the factorised register transfers. The feasibility check is an NP-complete problem; however it is necessary since the generated constraints only guarantee local feasibility. If there is no feasible solution, correction code has to be inserted. The schedule produced so far does not contain instructions for memory addressing; the values are bound to memories but not yet to fixed addresses. Thus the address assignment and the insertion of addressing code is performed in a post-processing step. In a subsequent compaction phase the additional operations are integrated into the previously determined schedule. This may lead to a significant decrease in code quality especially for architectures that do not dispose of dedicated address register files. Experimental results are presented in [BL99] for the Analog Devices ADSP-210x[2]. It is shown that for some code excerpts of the DSPstone benchmark the code quality of ICG corresponds to the quality of hand-written assembly while the GNU `gcc` produces an overhead of 170% on average.

[2]The ADSP-210x is a fixed-point DSP representing a precursor of the ADSP-2106x family

12. Conclusion and Outlook

In this thesis the PROPAN system has been presented as a retargetable framework for search-based code optimisations and machine-dependent program analyses at assembly level. To the best of our knowledge, PROPAN is the first system where the issues of machine description driven retargetability and of high-quality postpass optimisations have been combined. The retargetability concept of PROPAN is based on the combination of generic and generative mechanisms. We have developed a novel machine description language called TDL for concisely specifying all relevant information about the target architecture. Apart from the assembly orientation the main innovation of TDL is the generic modelling of irregular hardware constraints that allows them to be exploited in generic search-based optimisation algorithms. The core system of PROPAN has been implemented in a generic way; if target-specific information is required, the architecture database generated from the TDL description is referenced. For each target architecture the architecture database is linked with the generic core system yielding a dedicated hardware-sensitive postpass optimiser.

The optimisations are based on integer linear programming and allow a phase-coupled modelling of instruction scheduling, register assignment and resource allocation taking precisely into account the hardware characteristics of the target architecture. Two well-structured ILP-formulations have been implemented: an extension of the order-indexed SILP model and an extension of the time-indexed OASIC model. The modelling extensions are concerned with adaptations to the code generation problem, the representation of the control flow structure and the incorporation of irregular hardware characteristics. The polytope structure of both models has been investigated and the relationship between the hardware design and the appropriate ILP modelling style has been worked out. Order-indexed modelling allows an efficient integration of the register assignment problem and is well suited for irregular architecture where the resource competition is high. Time-indexed modelling presents itself for architectures with a high degree of instruction-level parallelism where a large number of functional unit types is available for the execution of each microoperation. In contrast to most previous search-based code generation methods, the optimisation scope is not restricted to a single basic block. Instead a novel superblock mechanism allows to exceed basic block and loop boundaries.

The integer linear programs can be solved exactly providing provably optimal solutions with respect to the problem dimension, i. e. the modelled code generation tasks, and to the optimisation scope. As an alternative to the exact solution, we have developed novel ILP-based approximations that are based on the iterative solution of partial relaxations of the original problem. Experimental results show that the approximative methods allow to reduce the computation time significantly compared to the exact solution while producing a very high solution quality. By specifying upper bounds for the maximal size of code sequences modelled by a single integer linear program the required computation times can be reduced. This way the user can choose a suitable trade-off between computation time and code quality.

The PROPAN framework has been retargeted to several representative standard processors. It has been used to generate ILP-based optimisers for the Analog Devices ADSP-2106x SHARC and the Philips TriMedia TM1000. Practical experiments demonstrate the applicability of the optimisers for typical applications of digital signal processing. PROPAN has also been used as a platform for generic program analyses and user-supplied hardware-dependent program optimisations. It is integrated in a framework for calculating worst-case execution time guarantees for real-time systems; in this context a TDL specification of the Infineon TriCore μC/DSP has been developed. For the Infineon C16x microprocessor family, PROPAN has been used as a platform for implementing hardware-sensitive postpass optimisations that are part of a commercial postpass optimiser. Practical experiments and industrial experience have shown that due to the postpass orientation, PROPAN can be integrated in existing tool chains with moderate effort.

Future research aims at different directions. There is ongoing work to extend the TDL language by a modelling of superscalar architectures with complex pipelines. The control flow reconstruction algorithm is being generalised based on analysing the specification of the operation semantics in the TDL description. The generation of cycle-accurate instruction set simulators from the TDL specification is another goal. There is ongoing work to develop value analyses that allow to disambiguate memory accesses and remove spurious data dependences in guarded code. Furthermore, the modelling of additional processors is planned. By investigating the polytope structure of both ILP formulations, the constraints could be identified that contribute most to the required computation times. Future research will aim at tightening the formulations in order to further improve the solution efficiency.

13. List of Symbols

A. Appendix

A.1. Instruction Set of the SHARC

In the following, an overview of the instruction set of the ADSP-2106X SHARC is given [Ana95]. In Sec. A.1.1 an overview of the terminology is given; Sec. A.1.2 lists the instruction formats of the ADSP-2106X SHARC. An overview of the compute operations is given in Sec. A.1.3.

A.1.1. Notation

Condition Codes. Most operations of the ADSP-2106X SHARC can optionally be guarded; then their execution depends on the value of certain bits in control and status registers that have been set by preceding operations. In the following the assembly representation of the available condition codes is listed.

Condition	*Description*	*Condition*	*Description*
EQ	ALU equal zero	SZ	Shifter zero
NE	ALU not equal zero	NOT SZ	Not shifter zero
LT	ALU less than zero	FLAG0_IN	Flag 0 input
GT	ALU greater than zero	NOT FLAG0_IN	Not flag 0 input
AC	ALU carry	FLAG1_IN	Flag 1 input
NOT AC	Not ALU carry	NOT FLAG1_IN	Not flag 1 input
AV	ALU overflow	FLAG0_IN	Flag 2 input
NOT AV	Not ALU overflow	NOT FLAG2_IN	Not flag 2 input
MV	Multiplier overflow	FLAG0_IN	Flag 3 input
NOT MV	Not multiplier overflow	NOT FLAG3_IN	Not flag 3 input
MS	Multiplier sign	TF	Bit test flag
NOT MS	Not multiplier sign	NOT TF	Not Bit test flag
SV	Shifter overflow	BM	Bus master
NOT SV	Not shifter overflow	NBM	Not bus master
TRUE	Always true (IF)		
FOREVER	Always false (DO UNTIL)		
LCE	Loop counter expired (DO UNTIL)		
NOT LCE	Loop counter not expired (IF)		
GE	ALU greater than or equal zero		
LE	ALU less than or equal zero		

Universal Registers.

Register	*Function*
Data Register File	
R15-R0	Fixed-point registers
F15-F0	Floating-point registers
Program Sequencer	
PC	Program counter (read-only)
PCSTK	Top of PC stack
PCSTKP	PC stack pointer
FADDR	Fetch address (read-only)
DADDR	Decode address (read-only)
LADDR	Loop termination address
CURLCNTR	Current loop counter
LCNTR	Loop count for next nested counter-controlled loop
Data Address Generators	
I7-I0	DAG1 index registers
M7-M0	DAG1 modify registers
L7-L0	DAG1 length registers
B7-B0	DAG1 base registers
I15-I8	DAG2 index registers
M15-M8	DAG2 modify registers
L15-L8	DAG2 length registers
B15-B8	DAG2 base registers
Bus Exchange	
PX1	PMD-DMD bus exchange 1 (16 bits)
PX2	PMD-DMD bus exchange 2 (32 bits)
PX	48-bit combination of PX1 and PX2
Timer	
TPERIOD	Timer period
TCOUNT	Timer counter
System Registers	
MODE1	Mode control and status
MODE2	Mode control and status
IRPTL	Interrupt latch
IMASK	Interrupt mask
IMASKP	Interrupt mask pointer (for nesting)
ASTAT	Arithmetic status flags, bit test flag, etc.
STKY	Sticky arithmetic status flags, stack status flags, etc.
USTAT1	User status register 1
USTAT2	User status register 2

A.1.2. Instruction Formats

In the following the instruction formats of the ADSP-2106X SHARC are listed. Instructions parts printed in italics are optional, capitals represent explicit assembly syntax. The assembly code itself is case-insensitive. The following abbreviations are used:

Notation	*Meaning*
;	Instruction delimiter
,	Operation delimiter
$\begin{vmatrix}\text{Option1}\\ \text{Option2}\end{vmatrix}$	List of options among which one has to be selected
compute	Compute operations (cf. Sec. A.1.3)
cond	Condition code (cf. Sec.A.1.1)
term	Loop termination conditions (cf. Sec.A.1.1)
ureg	Universal register
sreg	System register
dreg	Data register (register file): R15-R0 or F15-F0
Ia	I7-I0 (DAG1 index register)
Mb	M7-M0 (DAG1 modify register)
Ic	I15-I8 (DAG2 index register)
Md	M15-M8 (DAG2 modify register)
<data*n*>	*n*-bit constant
<addr*n*>	*n*-bit address
<reladdr*n*>	*n*-bit relative address
(DB)	Delayed branch
(LA)	Loop abort
(CI)	Clear interrupt

Instruction Formats

$$\textit{compute}, \begin{vmatrix}\text{DM(Ia, Mb)=dreg1}\\ \text{dreg1=DM(Ia, Mb)}\end{vmatrix}, \begin{vmatrix}\text{PM(Ic, Md)=dreg2}\\ \text{dreg2=PM(Ic, Md)}\end{vmatrix};$$

IF cond compute;

$$\textit{IF cond compute}, \begin{vmatrix}\text{DM(Ia, Mb)}\\ \text{PM(Ic, Md)}\end{vmatrix}\text{=ureg;}$$

$$\textit{IF cond compute}, \begin{vmatrix}\text{DM(Mb, Ia)}\\ \text{PM(Md, Ic)}\end{vmatrix}\text{=ureg;}$$

$$\mathit{IF\ cond\ compute},\ \ \text{ureg=}\left|\begin{array}{l}\text{DM(Ia, Mb)}\\ \text{PM(Ic, Md)}\end{array}\right|;$$

$$\mathit{IF\ cond\ compute},\ \ \text{ureg=}\left|\begin{array}{l}\text{DM(Mb, Ia)}\\ \text{PM(Md, Ic)}\end{array}\right|;$$

$$\mathit{IF\ cond\ compute},\ \ \left|\begin{array}{l}\text{DM(Ia, <data6>)}\\ \text{PM(Ic, <data6>)}\end{array}\right|\text{=dreg;}$$

$$\mathit{IF\ cond\ compute},\ \ \left|\begin{array}{l}\text{DM(<data6>, Ia)}\\ \text{PM(<data6>), Ic}\end{array}\right|\text{=dreg;}$$

$$\mathit{IF\ cond\ compute},\ \ \text{dreg=}\left|\begin{array}{l}\text{DM(Ia, <data6>)}\\ \text{PM(Ic, <data6>)}\end{array}\right|;$$

$$\mathit{IF\ cond\ compute},\ \ \text{dreg=}\left|\begin{array}{l}\text{DM(<data6>, Ia)}\\ \text{PM(<data6>), Ic}\end{array}\right|;$$

$$\mathit{IF\ cond\ compute},\ \ \text{ureg1=ureg2;}$$

$$\mathit{IF\ cond}\ \text{shiftimm,}\ \left|\begin{array}{l}\text{DM(Ia, Mb)}\\ \text{PM(Ic, Md)}\end{array}\right|\text{=dreg;}$$

$$\mathit{IF\ cond}\ \text{shiftimm, dreg=}\left|\begin{array}{l}\text{DM(Ia, Mb)}\\ \text{PM(Ic, Md)}\end{array}\right|;$$

$$\mathit{IF\ cond\ compute},\ \ \text{MODIFY}\left|\begin{array}{l}\text{(Ia, Mb)}\\ \text{(Ic, Md)}\end{array}\right|;$$

$$\mathit{IF\ cond}\ \text{JUMP}\ \left|\begin{array}{l}\text{<addr24>}\\ \text{(PC, <reladdr24>)}\end{array}\right|\ \left|\begin{array}{l}\mathit{(DB)}\\ \mathit{(LA)}\\ \mathit{(CI)}\\ \mathit{(DB,\ LA)}\\ \mathit{(DB,\ CI)}\end{array}\right|;$$

$$\mathit{IF\ cond}\ \text{CALL}\ \left|\begin{array}{l}\text{<addr24>}\\ \text{(PC, <reladdr24>)}\end{array}\right|\ \mathit{(DB)};$$

$$\textit{IF cond}\ \text{JUMP}\ \left|\begin{array}{l}\text{(Md, Ic)}\\ \text{(PC, <reladdr6>)}\end{array}\right|\ \left|\begin{array}{l}\textit{(DB)}\\ \textit{(LA)}\\ \textit{(CI)}\\ \textit{(DB, LA)}\\ \textit{(DB, CI)}\end{array}\right|,\ \left|\begin{array}{l}\textit{compute}\\ \textit{ELSE compute}\end{array}\right|;$$

$$\textit{IF cond}\ \text{CALL}\ \left|\begin{array}{l}\text{(Md, Ic)}\\ \text{(PC, <reladdr6>)}\end{array}\right|\ \textit{(DB)},\ \left|\begin{array}{l}\textit{compute}\\ \textit{ELSE compute}\end{array}\right|;$$

$$\textit{IF cond}\ \text{JUMP}\ \left|\begin{array}{l}\text{(Md, Ic)}\\ \text{(PC, <reladdr6>)}\end{array}\right|,\ \text{ELSE}\ \left|\begin{array}{l}\textit{compute}\text{, DM(Ia, Mb)=dreg)}\\ \\ \textit{compute}\text{, dreg=DM(Ia, Mb))}\end{array}\right|;$$

$$\textit{IF cond}\ \text{RTS}\ \left|\begin{array}{l}\textit{(DB)}\\ \textit{(LR)}\\ \textit{(DB, LR)}\end{array}\right|,\ \left|\begin{array}{l}\textit{compute}\\ \textit{ELSE compute}\end{array}\right|;$$

$$\textit{IF cond}\ \text{RTI}\ \textit{(DB)},\ \left|\begin{array}{l}\textit{compute}\\ \textit{ELSE compute}\end{array}\right|;$$

$$\text{LCNTR=}\left|\begin{array}{l}\text{<data16>}\\ \text{ureg}\end{array}\right|,\ \text{DO}\ \left|\begin{array}{l}\text{<addr24>}\\ \text{(PC, <reladdr24>)}\end{array}\right|\ \text{UNTIL LCE;}$$

$$\text{DO}\ \left|\begin{array}{l}\text{<addr24>}\\ \text{(PC, <reladdr24>)}\end{array}\right|\ \text{UNTIL term;}$$

$$\left|\begin{array}{l}\text{DM(<addr32>)}\\ \text{PM(<addr24>)}\end{array}\right|\text{=ureg;}$$

$$\text{ureg=}\left|\begin{array}{l}\text{DM(<addr32>)}\\ \text{PM(<addr24>)}\end{array}\right|;$$

$$\left|\begin{array}{l}\text{DM(<addr32>, Ia)}\\ \text{PM(<addr24>, Ic)}\end{array}\right|\text{=ureg;}$$

$$\text{ureg=}\left|\begin{array}{l}\text{DM(<addr32>, Ia)}\\ \text{PM(<addr24>, Ic)}\end{array}\right|;$$

$$\left|\begin{array}{l}\text{DM(Ia, Mb)}\\ \text{PM(Ic, Md)}\end{array}\right|\text{=<data32>;}$$

ureg=<data32>;

$$\text{BIT} \left|\begin{array}{l}\text{SET}\\ \text{CLR}\\ \text{TGL}\\ \text{TST}\\ \text{XOR}\end{array}\right| \text{sreg} <\text{data32}>;$$

$$\text{MODIFY} \left|\begin{array}{l}(\text{Ia}, <\text{data32}>)\\ (\text{Ic}, <\text{data24}>)\end{array}\right|;$$

$$\text{BITREV} \left|\begin{array}{l}(\text{Ia}, <\text{data32}>)\\ (\text{Ic}, <\text{data24}>)\end{array}\right|;$$

$$\left|\begin{array}{l}PUSH\\ POP\end{array}\right| LOOP, \left|\begin{array}{l}PUSH\\ POP\end{array}\right| STS, \left|\begin{array}{l}PUSH\\ POP\end{array}\right| PCSTK, FLUSH\ CACHE;$$

NOP;

IDLE;

$$\text{CJUMP} \left|\begin{array}{l}\text{function}\\ (\text{PC}, <\text{reladdr24}>)\end{array}\right| (\text{DB});$$

RFRAME;

A.1.3. Compute Operations

ALU Operations

Rn=Rx+Ry	Fn=ABS(Fx+Fy)
Rn=Rx–Ry	Fn=ABS(Fx–Fy)
Rn=Rx+Ry+CI	Fn=(Fx+Fy)/2
Rn=Rx+Ry+CI–1	COMP(Fx, Fy)
Rn=(Rx+Ry)/2	Fn=–Fx
COMP(Rx, Ry)	Fn=ABS Fx
Rn=Rx+CI	Fn=PASS Fx
Rn=Rx+CI–1	Fn=RND Fx
Rn=Rx+1	Fn=SCALB Fx BY Fy
Rn=Rx–1	Rn=MANT Fx
Rn=–Rx	Rn=LOBG Fx

Rn=ABS Rx	Rn=FIX Fx BY Ry
Rn=PASS Rx	Rn=FIX Fx
Rn=Rx OR Ry	Fn=FLOAT Rx BY Ry
Rn=Rx XOR Ry	Fn=FLOAT Rx
Rn=NOT Rx	Fn=RECIPS Fx
Rn=MIN(Rx, Ry)	Fn=RSQRTS Fx
Rn=MAX(Rx, Ry)	Fn=Fx COPYSIGN Fy
Rn=CLIP Rx BY Ry	Fn=MIN(Fx, Fy)
Fn=Fx+Fy	Fn=MAX(Fx, Fy)
Fn=Fx–Fy	FN=CLIP Fx BY FY

Multiplier Operations

$$\left|\begin{array}{l}\mathrm{Rn}\\\mathrm{MRF}\\\mathrm{MRB}\end{array}\right| = \mathrm{Rx*Ry} \qquad \left(\left|\begin{array}{l}\mathrm{S}\\\mathrm{U}\end{array}\right| \left|\begin{array}{l}\mathrm{S}\\\mathrm{U}\end{array}\right| \left|\begin{array}{l}\mathrm{F}\\\mathrm{I}\\\mathrm{FR}\end{array}\right|\right)$$

$$\left|\begin{array}{lcl}\mathrm{Rn} & = & \mathrm{MRF}\\\mathrm{Rn} & = & \mathrm{MRB}\\\mathrm{MRF} & = & \mathrm{MRF}\\\mathrm{MRB} & = & \mathrm{MRB}\end{array}\right| + \mathrm{Rx*Ry} \qquad \left(\left|\begin{array}{l}\mathrm{S}\\\mathrm{U}\end{array}\right| \left|\begin{array}{l}\mathrm{S}\\\mathrm{U}\end{array}\right| \left|\begin{array}{l}\mathrm{F}\\\mathrm{I}\\\mathrm{FR}\end{array}\right|\right)$$

$$\left|\begin{array}{lcl}\mathrm{Rn} & = & \mathrm{MRF}\\\mathrm{Rn} & = & \mathrm{MRB}\\\mathrm{MRF} & = & \mathrm{MRF}\\\mathrm{MRB} & = & \mathrm{MRB}\end{array}\right| - \mathrm{Rx*Ry} \qquad \left(\left|\begin{array}{l}\mathrm{S}\\\mathrm{U}\end{array}\right| \left|\begin{array}{l}\mathrm{S}\\\mathrm{U}\end{array}\right| \left|\begin{array}{l}\mathrm{F}\\\mathrm{I}\\\mathrm{FR}\end{array}\right|\right)$$

$$\left|\begin{array}{lcl}\mathrm{Rn} & = & \mathrm{SAT\ MRF}\\\mathrm{Rn} & = & \mathrm{SAT\ MRB}\\\mathrm{MRF} & = & \mathrm{SAT\ MRF}\\\mathrm{MRB} & = & \mathrm{SAT\ MRB}\end{array}\right| \qquad \left|\begin{array}{l}\mathrm{(SI)}\\\mathrm{(UI)}\\\mathrm{(SF)}\\\mathrm{(UF)}\end{array}\right|$$

$$\left|\begin{array}{lcl}\mathrm{Rn} & = & \mathrm{RND\ MRF}\\\mathrm{Rn} & = & \mathrm{RND\ MRB}\\\mathrm{MRF} & = & \mathrm{RND\ MRF}\\\mathrm{MRB} & = & \mathrm{RND\ MRB}\end{array}\right| \qquad \left|\begin{array}{l}\mathrm{(SF)}\\\mathrm{(UF)}\end{array}\right|$$

$$\left|\begin{array}{l}\mathrm{MRF}\\\mathrm{MRB}\end{array}\right| = 0$$

$$\left|\begin{array}{l}\mathrm{MRxF}\\\mathrm{MRxB}\end{array}\right| = \mathrm{Rn}$$

$$\mathrm{Rn} = \left|\begin{array}{l}\mathrm{MRxF}\\\mathrm{MRxB}\end{array}\right|$$

Shifter Operations

Rn=LSHIFT Rx BY Ry
Rn=LSHIFT Rx BY <data8>
Rn=Rn OR LSHIFT Rx BY Ry
Rn=Rn OR LSHIFT Rx BY <data8>
Rn=ASHIFT Rx BY Ry
Rn=ASHIFT Rx BY <data8>
Rn=Rn OR ASHIFT Rx BY Ry
Rn=Rn OR ASHIFT Rx BY <data8>
Rn=ROT Rx BY Ry
Rn=ROT Rx BY <data8>
Rn=BCLR Rx BY Ry
Rn=BCLR Rx BY <data8>
Rn=BSET Rx BY Ry
Rn=BSET Rx BY <data8>
Rn=BTGL Rx BY Ry
Rn=BTGL Rx BY <data8>
BTST Rx BY Ry
BTST Rx BY <data8>
Rn=FDEP Rx BY Ry
Rn=Rn OR FDEP Rx BY Ry
Rn=FDEP Rx BY Ry (SE)
Rn=Rn OR FDEP Rx BY Ry (SE)
Rn=FEXT Rx BY Ry
Rn=FEXT Rx BY Ry (SE)
Rn=EXP Rx (EX)
Rn=EXP Rx
Rn=LEFTZ Rx
Rn=LEFTO Rx
Rn=FPACK Fx
Fn=FUNPACK Rx
Rn=FDEP Rx BY <bit6>:<len6>
Rn=Rn OR FDEP Rx BY <bit6>:<len6>
Rn=FDEP Rx B> <bit6>:<len6> (SE)
Rn=Rn OR FDEP Rx B> <bit6>:<len6> (SE)
Rn=FEXT Rx BY <bit6>:<len6>
Rn=FEXT Rx BY <bit6>:<len6> (SE)

Multifunctional Operations

The multifunctional operations trigger parallel execution of the ALU and the multiplier. Each of the four input operands for computations that use both the ALU and multiplier are constrained to a different set of four register file locations as explained in Sec. 10.1.1.

Rm=R3-0*R7-4 (SSFR), Ra=R11-8+R15-12
Rm=R3-0*R7-4 (SSFR), Ra=R11-8–R15-12
Rm=R3-0*R7-4 (SSFR), Ra=(R11-8+R15-12)/2

MRF=MRF+R3-0*R7-4 (SSF), Ra=R11-8+R15-12
MRF=MRF+R3-0*R7-4 (SSF), Ra=R11-8–R15-12
MRF=MRF+R3-0*R7-4 (SSF), Ra=(R11-8+R15-12)/2

Rm=MRF+R3-0*R7-4 (SSFR), Ra=R11-8+R15-12
Rm=MRF+R3-0*R7-4 (SSFR), Ra=R11-8–R15-12
Rm=MRF+R3-0*R7-4 (SSFR), Ra=(R11-8+R15-12)/2

MRB=MRB+R3-0*R7-4 (SSF), Ra=R11-8+R15-12
MRB=MRB+R3-0*R7-4 (SSF), Ra=R11-8–R15-12

MRB=MRB+R3-0*R7-4 (SSF), Ra=(R11-8+R15-12)/2

Rm=MRB+R3-0*R7-4 (SSFR), Ra=R11-8+R15-12
Rm=MRB+R3-0*R7-4 (SSFR), Ra=R11-8–R15-12
Rm=MRB+R3-0*R7-4 (SSFR), Ra=(R11-8+R15-12)/2

Fm=F3-0*F7-4, Fa=F11-8+F15-12
Fm=F3-0*F7-4, Fa=F11-8–F15-12
Fm=F3-0*F7-4, Fa=FLOAT F11-8 BY R15-12
Fm=F3-0*F7-4, Fa=FIX F11-8 BY R15-12
Fm=F3-0*F7-4, Fa=(F11-8+F15-12)/2
Fm=F3-0*F7-4, Fa=ABS F11-8
Fm=F3-0*F7-4, Fa=MAX(F11-8, F15-12)
Fm=F3-0*F7-4, Fa=MIN(F11-8, F15-12)

A.2. Excerpts from the TDL Specification of the SHARC

```
/*-----------------------------------------------------------------*/
/*                Resources                                        */
/*-----------------------------------------------------------------*/

Resources-Section

// Functional Units
FuncUnit ALU 1;
FuncUnit MUL 1;
FuncUnit SHI 1;

// Integer Registers
Register ireg   "r%d" [0:15] size=40, type=signed<32>, renaming=true;
ResourceClass iregA   {ireg[0:3]};
ResourceClass iregB   {ireg[4:7]};
ResourceClass iregC   {ireg[8:11]};
ResourceClass iregD   {ireg[12:15]};
SetProperties ireg ilpres={iregA, iregB, iregC, iregD};

// Index Registers
Register indreg "i%d" [0:15] size=40, type=signed<32>, usage=Index;
SetProperties indreg[6] usage=FP;
SetProperties indreg[7] usage=SP;

// Modify Registers
```

```
Register mreg   "m%d" [0:15] size=40, type=signed<32>, usage=Offset;

// Length Registers
Register lreg   "l%d" [0:15] size=40, type=signed<32>;

// Base Registers
Register breg   "b%d" [0:15] size=40, type=signed<32>;

// Floating point registers are aliases of the integer registers
RegisterAlias freg "f%d" ireg mapping=[1:1], type=float<24,8>;

// MR registers, used as accumulators
Register MRFreg "MR%dF" [0:2] size=32, type = signed, usage=accu;
SetProperties MRFreg[2] size=16;

Register MRBreg "MR%dB" [0:2] size=32, type=signed, usage=accu;
SetProperties MRBreg[2] size=16;

RegisterAlias MRF "MRF" MRFreg mapping=[3:1], size=80;
RegisterAlias MRB "MRB" MRBreg mapping=[3:1], size=80;

// Now the control flow registers
Register sPC            "pc"            size=24, type=signed, usage=PC;
Register FADDR          "faddr"         size=24, type=signed;
Register DADDR          "daddr"         size=24, type=signed;
Register PCSTK          "pcstk"         size=24, type=signed;
Register PCSTKP         "pcstkp"        size=5,  type=signed;
Register LSTKP          "lstkp"         size=5,  type=signed;
Register LADDR          "laddr"         size=32, type=signed;
Register CURLCNTR       "curlcntr"      size=32, type=signed;
Register LCNTR          "lcntr"         size=32, type=signed;
Register MODE1          "mode1"         size=32, type=signed;
Register MODE2          "mode2"         size=32, type=signed;
Register IRPTL          "irptl"         size=32, type=signed;
Register IMASK          "imask"         size=32, type=signed;
Register IMASKP         "imaskp"        size=32, type=signed;
Register ASTAT          "astat"         size=32, type=signed;
Register STKY           "stky"          size=32, type=signed;
Register USTAT1         "ustat1"        size=32, type=signed;
Register USTAT2         "ustat2"        size=32, type=signed;

// Data Memory
Memory DM "dm" type=data,  align=16, access=32;
Memory PM "pm" type=mixed, align=16, access=32;

// Cache
```

```
Cache InstrCache assoc=2, size=32, linesize=6, type=instr;

// Standard-Resource, used for ALU and Shifter in ILP-Modelling
DefineResource Standard "std" 1;

// Resource Classes
ResourceClass nonifreg {indreg,mreg,breg,lreg};
ResourceClass ifreg    {ireg, freg};
ResourceClass indregA {indreg[0:7]};
ResourceClass indregC {indreg[8:15]};
ResourceClass mregB   {mreg[0:7]};
ResourceClass mregD   {mreg[8:15]};
ResourceClass sreg    {MODE1, MODE2, IRPTL, IMASK, sPC,
                       IMASKP, ASTAT, STKY, USTAT1, USTAT2};
ResourceClass ureg    {ifreg,nonifreg,sreg};

ResourceClass CallerSaved {ireg[0], ireg[1], ireg[2], ireg[4],
        ireg[8], ireg[12], indreg[4], indreg[12], mreg[4],
        mreg[12]};

ResourceClass UsedByCall {ireg[4], ireg[8], ireg[12]};

ResourceClass UsedByRet {ireg[0], ireg[1]};


/*---------------------------------------------------------------*/
/*               Instruction Set                                 */
/*---------------------------------------------------------------*/
InstructionSet-Section

// User-defined attributes
DefineAttribute condname {"c_eq", "c_lt", "c_gt", "c_le", "c_ge",
 "c_ac", "c_av", "c_ms", "c_mv", "c_sv", "c_sz", "c_flag0_in",
 "c_flag1_in", "c_flag2_in", "c_flag3_in", "c_tf", "c_bm",
 "c_lce", "c_not_lce", "c_not_ac", "c_not_av", "c_not_mv",
 "c_not_ms", "c_not_sv", "c_ne", "c_not_sz", "c_not_flag0_in",
 "c_not_flag1_in", "c_not_flag2_in", "c_not_flag3_in",
 "c_not_tf", "c_nbm", "c_forever", "c_true"} associated to
 Operation;

DefineAttribute smode {"ssi", "sui", "usi", "uui", "ssf", "suf",
 "usf", "uuf", "ssfr", "sufr", "usfr", "uufr",  "si", "ui",
        "sf", "uf", "signext", "extexp", "db", "lr", "dblr", "la",
 "ci", "dbla", "dbci", "ndb"} associated to Operation;

DefineAttribute modtype {"s_simple"} associated to Operation;
```

```
// Non-terminal definitions
OpNT flag "eq" {guard=true, condname=c_eq},{;},
         {unsigned<1> cond; cond := ASTAT<0>;}
        | "ne" {guard=true, condname=c_ne},{;},
         {unsigned<1> cond; cond := ASTAT<0>;}
        | "lt" {guard=true, condname=c_lt},{;},
          { unsigned<1> cond;
          if ((ASTAT<0>==0b0)&&((ASTAT<2>==0b1)^((ASTAT<1>==0b1)
              &&(MODE1<13>==0b0)))||((ASTAT<10>==0b1)&&
              (ASTAT<2>==0b1)&&(ASTAT<0>==0b0))) {
            cond := 0b1; } else {cond:= 0b0;}}
        | "gt" {guard = true, condname=c_gt},{;},
          { unsigned<1> cond;
           if ((((ASTAT<0>==0b0)&&((ASTAT<2>==0b1)^((ASTAT<1>==0b1)&&
               (MODE1<13>==0b0)))||((ASTAT<10>==0b1)&&
               (ASTAT<2>==0b1)))||(ASTAT<0>==0b1))) {
            cond:=0b1;} else {cond:=0b0;}}
        | "le" {guard = true, condname=c_le},{;},
          { unsigned<1> cond;
            if (((ASTAT<0>==0b0)&&((ASTAT<2>==0b1)^
               ((ASTAT<1>==0b1)&&(MODE1<13>==0b0)))
                ||((ASTAT<10>==0b1)&&(ASTAT<2>==0b1)))||
               (ASTAT<0>==0b1)) {cond:=0b1;}
           else {cond:=0b0;}};

OpNT optguard "if %!(flag) " {guard = true}, {;}, {}
        | "" {guard = false}, {;}, {unsigned<1> cond; cond:=0b0;};

OpNT dbmod "( DB )" {smode=db},{PM(slots=2);},{}
        | "" {smode=ndb},{;},{};

OpNT jumpmod "%!(dbmod)" {modtype=s_simple},{;},{}
        | "( LA )" {smode=la},{;},{}
        | "( CI )" {smode=ci},{;},{}
        | "( DB , LA )" {smode=dbla},{PM(slots=2);},{}
        | "( DB , CI )" {smode=dbci},{PM(slots=2);},{};

OpNT retmod "%!(dbmod)" {modtype=s_simple},{;},{}
        | "( LR )" {smode=lr},{;},{}
        | "( DB , LR )" {smode=dblr},{PM(slots=2);},{};

OpNT imm24_t "%24D" {target="$1" as signed<24>},{;},{}
        | "%24X" {target="$1" as signed<24>},{;},{}
        | "%s" {target="$1" as signed<24>},{;},{};
```

```
OpNT mod1 "" | "( SI )" {smode=si},{;},{} |"(UI)" {smode=ui},{;},{}
        |"( SF )" {smode=sf},{;}, {} |"(UF)" {smode=uf},{;},{};

OpNT mod2 "" | "( SSI )" {smode=ssi},{;},{}
        |"( SUI )" {smode=sui},{;},{}
        |"( USI )" {smode=usi},{;},{}
        |"( UUI )" {smode=uui},{;},{}
        |"( SSF )" {smode=ssf},{;},{}
        |"( SUF )" {smode=suf},{;},{}
        |"( USF )" {smode=usf},{;},{}
        |"( UUF )" {smode=uuf},{;},{}
        |"( SSFR )" {smode=ssfr},{;},{}
        |"( SUFR )" {smode=sufr},{;},{}
        |"( USFR )" {smode=usfr},{;},{}
        |"( UUFR )" {smode=uufr},{;},{};

// Definitions of machine operations

DefineOp DMwriteIMif "%!(optguard)dm ( %s , %s ) = %s" {
        dst1=DM, base="$2" in {indregA} as signed<32>,
        offset = "$3" in {mregB} as signed<32>,
        src1 = "$4" in {ifreg}, mode =post},
        {DM(latency=1, slots=0, exectime=1);},
        {external unsigned<1> cond;
         if (cond==1) {base:=base+offset; DM[base]:=src1;}};

DefineOp PMreadIMf "%!(optguard)%s = pm ( %s , %s )" {
        dst1="$2" in {freg}, base="$3" in {indregC} as signed<32>,
        offset = "$4" in {mregD} as signed<32>, src1 = PM,
        mode = post}, {PM(latency=1, slots=0, exectime=1);},
        {external unsigned<1> cond;
         if (cond==1) {base:=base+offset; dst1:=PM[base];}};

DefineOp RegAsgni "%!(optguard)%s = %s" {
        dst1="$2" in {ireg}, src1="$3" in {ureg}},
        {DM(latency=1, slots=0, exectime=1);},
        {external unsigned<1> cond;
         if (cond==1) {dst1:=src1;}};

DefineOp LJumpAbs "%!(optguard)jump %!(imm24_t) %!(jumpmod)"
        {type=Jmp, noreorder}, {PM(latency=1, exectime=1);},
        {external unsigned<1> cond;
         if (cond == 1) {sPC:=target;}};

DefineOp LCallAbs "%!(optguard)call %!(imm24_t) %!(dbmod)"
        {dst1=CallerSaved, src1=UsedByCall, noreorder,
```

```
         type=Call}, {PM(latency=1, exectime=1);},
        {external unsigned<1> cond;
         if (cond == 1) {PCSTKP:=PCSTKP+1; PCSTK:=sPC;
                  sPC:=target;}};

DefineOp ImmDMwrite "%!(optguard) dm ( %!(anyimm_b) ) = %s"
        {dst1=DM, src1 = "$3" in {ureg}},
        {Standard(latency=1, exectime=1, slots=0);},
        {DM[base]:=src1;};

DefineOp AluFixed1 "%!(optguard)%s = %s + %s"
        {dst1="$2" in {ireg}, src1="$3" in {ireg}
        , src2="$4" in {ireg}},
        {Standard(latency=1, exectime=1, slots=0);},
        {dst1:=_iadd(src1, src2, 0b1, MODE1<13>, ASTAT<1>);
         if(dst1==0) {ASTAT<0>:=0b1;}
         if(dst1<0) {ASTAT<2>:=0b1;}};

DefineOp MulFixed1 "%!(optguard)%s = %s * %s %!(mod2)"
        {dst1="$2" in {ireg}, src1="$3" in {ireg},
         src2="$4" in {ireg}},
        {MUL(latency=1, exectime=1, slots=0);},
        {unsigned<1> s1; unsigned<1> s2;
         if (smode==ssi) {s1:=0b1;s2:=0b1;}
         if (smode==sui) {s1:=0b1;s2:=0b0;}
         if (smode==usi) {s1:=0b0;s2:=0b1;}
         if (smode==uui) {s1:=0b0;s2:=0b0;}
         dst1:=_imul(src1, src2, s1, s2, 32, ASTAT<1>);};

DefineOp Shift1 "%!(optguard)%s = lshift %s by %s"
        {dst1="$2" in {ireg}, src1="$3" in {ireg},
         src2="$4" in {ireg}},
        {Standard(latency=1, exectime=1, slots=0);},
        {dst1:=_lsh(src1, src2); if (dst1==0) {ASTAT<12>:=0b1;}
         if (src2>0) {ASTAT<11>:=0b1;}};

// Definition of operation classes
OperationClass MoveOrModifyClass {DMwriteIMif, DMreadIMi,
         DMreadIMf, PMwriteIMif, PMreadIMi, PMreadIMf,
         DMwriteIMnonif, DMreadIMnonif, PMwriteIMnonif,
         PMreadIMnonif, DMwriteMI, DMreadMIi, DMreadMIf,
         DMreadMIo, PMwriteMI, PMreadMIi, PMreadMIf, PMreadMIo,
         DMwriteIcon, PMreadIconi, PMreadIconf, PMreadIcono,
         PMwriteIcon, DMwriteconI, DMreadconIi, DMreadconIf,
         DMreadconIo, PMwriteconI, PMreadconIi, PMreadconIf,
         PMreadconIo, RegAsgni, RegAsgnf, RegAsgno, Modify};
```

```
/* The following operation classes are defined additionally:
       NonParaDM, NonParaPM, PMClass, DMClass, RAandMoDM,
       PrgFlowClass, ParaPrgFlow, NonParaPrgFlow, ShiftClass,
       AluClass, MulClass, ComputeClass, MultiAluFixed,
       MultiAluFloat, MultiMulFixed, MultiMulFloat */

/*--------------------------------------------------------------------*/
/*               Constraints                                           */
/*--------------------------------------------------------------------*/
Constraint-Section

// Irregular restrictions of instruction-level parallelism
op1 in {ShiftClass} & op2 in {MulClass}: !(op1 && op2);
op1 in {MiscClass} &
        op2 in {PrgFlowClass, MulClass, MoveOrModifyClass}:
        !(op1 && op2);
op1 in {ImmediateClass} & op2 in {PrgFlowClass, MulClass,
        MoveOrModifyClass}: !(op1 && op2);
op1 in {NonParaDM} & op2 in {PMClass}: !(op1 && op2);
op1 in {NonParaPM} & op2 in {DMClass}: !(op1 && op2);
op1 in {RAandMoDM} & op2 in {PMClass}: !(op1 && op2);
op1 in {PrgFlowClass} & op2 in {DMClass}: !(op1 && op2);
op1 in {NonParaPrgFlow} & op2 in {AluFixedClass, AluFloatClass,
        MulClass}: !(op1 && op2);

// Constraints for multifunctional operations
op1 in {MultiAluFixed} & op2 in {MultiMulFixed}:
        !(op1 && op2) | op1.src1 in {iregC} & op1.src2 in {iregD}
        & op2.src1 in {iregA} & op2.src2 in {iregB};
op1 in {MultiAluFloat} & op2 in {MultiMulFloat}:
        !(op1 && op2) | op1.src1 in {iregC} & op1.src2 in {iregD}
        & op2.src1 in {iregA} & op2.src2 in {iregB};

/*--------------------------------------------------------------------*/
/*              Assembly Section                                       */
/*--------------------------------------------------------------------*/
Assembly-Section

OperationDelimiter ",";
InstructionDelimiter ";";
AnnotationDelimiter "@";
Comment ("/*", "*/"), ("{", "}");
LineComment "!";

DefineDirective DirSegStartDM ".segment /pm %s;" {type=SegStart,
```

```
        name="$2"};
DefineDirective DirSegStartPM ".segment /dm %s;" {type=SegStart,
        name="$2"};

DefineDirective DirSegEnd ".endseg;" {type=SegEnd};

DefineDirective DirGcc ".gcc_compiled;" {};

DefineDirective DirGlobVar ".global %s;" {type=GlobalDecl,
                                          opnd1="$1"};

DefineDirective DirExtVar ".extern %s;" {type=ExternDecl,
                                         opnd1="$1"};
```

A.3. Instruction Set of the TM1000

Each instruction word of the TriMedia TM1000 is composed of five microoperations that are issued simultaneously (cf. Sec. 10.2.1) [Phi97]. Each operation has the following format (optional fields are enclosed in brackets (`[`,`]`)):

```
[if <guard>] <opcode> [(<modifier>)][<operand_1>][<operand_2>]
[-> destination]
```

- `guard` denotes an optional guard. A guarded operation is executed if and only if the least-significant bit of the general purpose register specified as guard is 1. Immediate operations cannot be guarded.
- The optional `modifier` field is used to indicate shift distances, displacements for memory accesses, etc.
- Since the TriMedia TM1000 is a load/store architecture, memory accesses are restricted to explicit load and store operations. The fields `guard`, `operand_1`, `operand_2` and `destination` must be general purpose registers. The registers are represented as rn where n is the number of the register.

In the following a list of all microoperations of the TriMedia TM1000 is given. For each operation, the assembly syntax is listed, and the functional unit type used for executing the operation (cf. Tab. 10.7) is given with the corresponding latency. The feasible issue slots are enumerated in the last column.

Syntax	*Execution Time, FU Type*	*Issue Slots*
asl $src1$ $src2 \to dst$	1, `shifter`	1,2
asli(n) $src1 \to dst$	1, `shifter`	1,2

asr *src1 src2* → *dst*	1, `shifter`	1,2
asri(n) *src1* → *dst*	1, `shifter`	1,2
bitand *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
bitandinv *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
bitinv *src1* → *dst*	1, `alu`	1,2,3,4,5
bitor *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
bitxor *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
carry *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
cycles → *dst*	1, `fcomp`	3
dcb(d) *src1*	3, `dmemspec`	5
dinvalid(d) *src1*	3, `dmemspec`	5
dspiadd *src1 src2* → *dst*	2, `dspalu`	1,3
dspidualadd *src1 src2* → *dst*	2, `dspalu`	1,3
dspidualmul *src1 src2* → *dst*	3, `dspmul`	2,3
dspidualsub *src1 src2* → *dst*	2, `dspalu`	1,3
dspimul *src1 src2* → *dst*	3, `ifmul`	2,3
dspisub *src1 src2* → *dst*	2, `dspalu`	1,3
dspuadd *src1 src2* → *dst*	2, `dspalu`	1,3
dspumul *src1 src2* → *dst*	3, `ifmul`	2,3
dspuquadaddui *src1 src2* → *dst*	2, `dspalu`	1,3
dspusub *src1 src2* → *dst*	2, `dspalu`	1,3
fabsval *src1* → *dst*	3, `falu`	1,4
fabsvalflags *src1* → *dst*	3, `falu`	1,4
fadd *src1 src2* → *dst*	3, `falu`	1,4
faddflags *src1 src2* → *dst*	3, `falu`	1,4
fdiv *src1 src2* → *dst*	17, `ftough`	2
fdivflags *src1 src2* → *dst*	17, `ftough`	2
feql *src1 src2* → *dst*	1, `fcomp`	3
feqlflags *src1 src2* → *dst*	1, `fcomp`	3
fgeq *src1 src2* → *dst*	1, `fcomp`	3
fgeqflags *src1 src2* → *dst*	1, `fcomp`	3
fgtr *src1 src2* → *dst*	1, `fcomp`	3
fgtrflags *src1 src2* → *dst*	1, `fcomp`	3
fmul *src1 src2* → *dst*	3, `ifmul`	2,3
fmulflags *src1 src2* → *dst*	3, `ifmul`	2,3
fneq *src1 src2* → *dst*	1, `fcomp`	3
fneqflags *src1 src2* → *dst*	1, `fcomp`	3
fsign *src1* → *dst*	1, `fcomp`	3
fsignflags *src1* → *dst*	1, `fcomp`	3
fsqrt *src1* → *dst*	17, `ftough`	2
fsqrtflags *src1* → *dst*	17, `ftough`	2
fsub *src1 src2* → *dst*	3, `falu`	1,4
fsubflags *src1 src2* → *dst*	3, `falu`	1,4
funshift1 *src1 src2* → *dst*	1, `shifter`	1,2

funshift2 *src1 src2* → *dst*	1, `shifter`	1,2
funshift3 *src1 src2* → *dst*	1, `shifter`	1,2
h_dspiabs *r*0 *src2* → *dst*	2, `dspalu`	1,3
h_dspidualabs *r*0 *src2* → *dst*	2, `dspalu`	1,3
h_iabs *r*0 *src2* → *dst*		
h_st16d(*d*) *src1 src2*	3, `dmem`	4,5
h_st32d(*d*) *src1 src2*	3, `dmem`	4,5
h_st8d(*d*) *src1 src2*	3, `dmem`	4,5
hicycles → *dst*	1, `fcomp`	3
iadd *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
iaddi(*n*) *src1* → *dst*	1, `alu`	1,2,3,4,5
iavgonep *src1 src2* → *dst*	2, `dspalu`	1,3
ibytesel *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
iclipi *src1 src2* → *dst*	2, `dspalu`	1,3
iclr		
ieql *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
ieqli(*n*) *src1* → *dst*	1, `alu`	1,2,3,4,5
ifir16 *src1 src2* → *dst*	3, `dspmul`	2,3
ifir8ii *src1 src2* → *dst*	3, `dspmul`	2,3
ifir8ui *src1 src2* → *dst*	3, `dspmul`	2,3
ifixieee *src1* → *dst*	3, `falu`	1,4
ifixieeeflags *src1* → *dst*	3, `falu`	1,4
ifixrz *src1* → *dst*	3, `falu`	1,4
ifixrzflags *src1* → *dst*	3, `falu`	1,4
iflip *src1 src2* → *dst*	2, `dspalu`	1,3
ifloat *src1* → *dst*	3, `falu`	1,4
ifloatflags *src1* → *dst*	3, `falu`	1,4
ifloatrz *src1* → *dst*	3, `falu`	1,4
ifloatrzflags *src1* → *dst*	3, `falu`	1,4
igeq *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
igeqi(*n*) *src1* → *dst*	1, `alu`	1,2,3,4,5
igtr *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
igtri(*n*) *src1* → *dst*	1, `alu`	1,2,3,4,5
iimm(*n*) → *dst*	1, `const`	1,2,3,4,5
ijmpf *src1 src2*	delay = 3, `branch`	2,3,4
ijmpi(*address*)	delay = 3, `branch`	2,3,4
ijmpt *src1 src2*	delay = 3, `branch`	2,3,4
ild16d(*d*) *src1* → *dst*	3, `dmem`	4,5
ild16r *src1 src2* → *dst*	3, `dmem`	4,5
ild16x *src1 src2* → *dst*	3, `dmem`	4,5
ild8d(*d*) *src1* → *dst*	3, `dmem`	4,5
ild8r *src1 src2* → *dst*	3, `dmem`	4,5
ileqi(*n*) *src1* → *dst*	1, `alu`	1,2,3,4,5
ilesi(*n*) *src1* → *dst*	1, `alu`	1,2,3,4,5

imax *src1 src2* → *dst*	2, `dspalu`	1,3
imin *src1 src2* → *dst*	2, `dspalu`	1,3
imul *src1 src2* → *dst*	3, `ifmul`	2,3
imulm *src1 src2* → *dst*	3, `ifmul`	2,3
ineq *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
ineqi(*n*) *src1* → *dst*	1, `alu`	1,2,3,4,5
inonzero *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
isub *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
isubi(*n*) *src1* → *dst*	1, `alu`	1,2,3,4,5
izero *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
jmpf *src1 src2*	delay = 3, `branch`	2,3,4
jmpi(*address*)	delay = 3, `branch`	2,3,4
jmpt *src1 src2*	delay = 3, `branch`	2,3,4
ld32d(*d*) *src1* → *dst*	3, `dmem`	4,5
ld32r *src1 src2* → *dst*	3, `dmem`	4,5
ld32x *src1 src2* → *dst*	3, `dmem`	4,5
lsr *src1 src2* → *dst*	1, `shifter`	1,2
lsri(*n*) *src1* → *dst*	1, `shifter`	1,2
mergelsb *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
mergemsb *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
pack16lsb *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
pack16msb *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
packbytes *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
quadavg *src1 src2* → *dst*	2, `dspalu`	1,3
quadumulmsb *src1 src2* → *dst*	3, `dspmul`	2,3
rdstatus(*d*) *src1* → *dst*	3, `dmemspec`	5
rdtag(*d*) *src1* → *dst*	3, `dmemspec`	5
readdpc → *dst*	1, `fcomp`	3
readpcsw → *dst*	1, `fcomp`	3
readspc → *dst*	1, `fcomp`	3
rol *src1 src2* → *dst*	1, `shifter`	1,2
roli(*n*) *src1* → *dst*	1, `shifter`	1,2
sex16 *src1* → *dst*	1, `alu`	1,2,3,4,5
ubytesel *src1 src2* → *dst*	1, `alu`	1,2,3,4,5
uclipi *src1 src2* → *dst*	2, `dspalu`	1,3
uclipu *src1 src2* → *dst*	2, `dspalu`	1,3
ueqli(*n*) *src1* → *dst*	1, `alu`	1,2,3,4,5
ufir16 *src1 src2* → *dst*	3, `dspmul`	2,3
ufir8uu *src1 src2* → *dst*	3, `dspmul`	2,3
ufixieee *src1* → *dst*	3, `falu`	1,4
ufixieeeflags *src1* → *dst*	3, `falu`	1,4
ufixrz *src1* → *dst*	3, `falu`	1,4
ufixrzflags *src1* → *dst*	3, `falu`	1,4
ufloat *src1* → *dst*	3, `falu`	1,4

ufloatflags *src1* → *dst*	3, `falu`	1,4
ufloatrz *src1* → *dst*	3, `falu`	1,4
ufloatrzflags *src1* → *dst*	3, `falu`	1,4
ugeq *src1* *src2* → *dst*	1, `alu`	1,2,3,4,5
ugeqi(n) *src1* → *dst*	1, `alu`	1,2,3,4,5
ugtr *src1* *src2* → *dst*	1, `alu`	1,2,3,4,5
ugtri(n) *src1* → *dst*	1, `alu`	1,2,3,4,5
uimm(n) → *dst*	1, `const`	1,2,3,4,5
uld16d(d) *src1* → *dst*	3, `dmem`	4,5
uld16r *src1* *src2* → *dst*	3, `dmem`	4,5
uld16x *src1* *src2* → *dst*	3, `dmem`	4,5
uld8d(d) *src1* → *dst*	3, `dmem`	4,5
uld8r *src1* *src2* → *dst*	3, `dmem`	4,5
uleqi(n) *src1* → *dst*	1, `alu`	1,2,3,4,5
ulesi(n) *src1* → *dst*	1, `alu`	1,2,3,4,5
ume8ii *src1* *src2* → *dst*	2, `dspalu`	1,3
ume8uu *src1* *src2* → *dst*	2, `dspalu`	1,3
umul *src1* *src2* → *dst*	3, `ifmul`	2,3
umulm *src1* *src2* → *dst*	3, `ifmul`	2,3
uneqi(n) *src1* → *dst*	1, `alu`	1,2,3,4,5
writedpc *src1*	1, `fcomp`	3
writepcsw *src1* *src2*	1, `fcomp`	3
writespc *src1*	1, `fcomp`	3

A.4. Excerpts from the TDL Specification of the TM1000

```
/*------------------------------------------------------------------*/
/*                Resources                                         */
/*------------------------------------------------------------------*/

Resources-Section

// Functional Units
FuncUnit ALU 5;
FuncUnit BRNCH 3;
FuncUnit CONST 5;
FuncUnit DMEM 2;
FuncUnit DMEMSPEC 1;
FuncUnit DSPALU 2;
FuncUnit DSPMUL 2;
FuncUnit FALU 2;
```

```
FuncUnit FCOMP 1;
FuncUnit FTOUGH 1;
FuncUnit IFMUL 2;
FuncUnit SHIFTER 2;

// General Purpose Registers
Register gpr "r%d" [0:127] size=32, type=signed<32>;

// The general purpose registers r0 and r1 have fixed values.
SetProperties gpr[0] value=0x00000000;
SetProperties gpr[1] value=0x00000001;

RegisterAlias rrp "rp" gpr[2] mapping=[1:1];
RegisterAlias rfp "fp" gpr[3] mapping=[1:1], usage=FP;
RegisterAlias rsp "sp" gpr[4] mapping=[1:1], usage=SP;
RegisterAlias rrv "rv" gpr[5] mapping=[1:1];

// ... and only the remaining registers are true GPRs.
ResourceClass tgpr {gpr[2:127]};

// Program Counter
Register tPC "PC" size=32, usage=PC;

// Program Control and Status Word
Register PCSW "pcsw" size=32;

// Destination Program Counter
Register DPC "dpc" size=32;

// Source Program Counter
Register SPC "spc" size=32;

// Clock Cycle Counter
Register CCCOUNT "ccount" size=64;

// Memory
Memory MEM "Mem" access=8, align=8, type=mixed;

// Cache
Cache DataCache assoc=8, size=256, linesize=64, type=data;
Cache InstrCache assoc=8, size=512, linesize=64, type=instr;

// Resources representing the issue slots, used for ILP-Modelling
DefineResource ISlot1 "islot1" 1;
DefineResource ISlot2 "islot2" 1;
DefineResource ISlot3 "islot3" 1;
```

```
DefineResource ISlot4 "islot4" 1;
DefineResource ISlot5 "islot5" 1;

// Resource for the write-back bus, used for ILP modelling
DefineResource WbBus "Write-back Bus" 5;


/*-------------------------------------------------------------------*/
/*               Instruction Set                                     */
/*-------------------------------------------------------------------*/
InstructionSet-Section

// Examples for macros allowing to shorten the description by
// using the C preprocessor

#define DUALCOMP32 if (temp1<0xffff8000) {temp1:=0x00008000;} \
         if (temp2<0xffff8000) {temp2:=0x00008000;} \
         if (temp1>0x00007fff) {temp1:=0x00007fff;} \
         if (temp2>0x00007fff) {temp2:=0x00007fff;} \
         _bext(dst1,31,16):=_bext(temp2,15,0); \
         _bext(dst1,15,0) :=_bext(temp1,15,0);

#define LD32MACRO  \
  _bext(dst1,7,0):= \
       _signed(MEM[tempaddr+_signed(_zext(tempbsxor,32),32)],8); \
  tempbsxor:=_conc(0b0,tempbs^0b10); \
  _bext(dst1,15,8):= \
        _signed(MEM[tempaddr+_signed(_zext(tempbsxor,32),32)],8); \
  tempbsxor:=_conc(0b0,tempbs^0b01); \
  _bext(dst1,23,16):= \
        _signed(MEM[tempaddr+_signed(_zext(tempbsxor,32),32)],8);\
  tempbsxor:=_conc(0b0,tempbs^0b00); \
  _bext(dst1,31,24):= \
        _signed(MEM[tempaddr+_signed(_zext(tempbsxor,32),32)],8);


OpNT asmexp "%!(asmexp) %!(binary_asm_operator) %!(asm_opnd)"
         {},{;},{}
        | "%!(asm_opnd)" {},{;},{};

OpNT binary_asm_operator "+" {},{;},{} | "-" {},{;},{}
        | "." {},{;},{};

OpNT asm_opnd "%d" {},{;},{} | "%x" {},{;},{}
        | "%s" {},{;},{} | "(%!(asmexp))" {},{;},{};
```

```
OpNT optguard "if %s" {guard=true, src3="$1" in {gpr}},{;},{}
        | ""{guard=false},{;},{};

OpNT jtarget "%!(asmexp)" {target="$1"},{;},{};

OpNT imms7_s1 "%!(asmexp)" {src1="$1" as signed<7>},{;},{};

DefineOp opNop "%!(optguard) nop" {type=nop},
        {ISlot1(exectime=1, latency=1, slots=0)
         |ISlot2(exectime=1, latency=1, slots=0)
         |ISlot3(exectime=1, latency=1, slots=0)
         |ISlot4(exectime=1, latency=1, slots=0)
         |ISlot5(exectime=1, latency=1, slots=0);},{};

DefineOp opASL "%!(optguard) asl %s %s \[->\] %s"
        {src1="$2" in {gpr}, src2="$3" in {gpr},
         dst1="$4" in {tgpr}},
        {ISlot1(exectime=1, latency=1, slots=0)
         |ISlot2(exectime=1, latency=1, slots=0); WbBus},
        {if ((guard==true)&&(src3<0>==0b1)) {
           signed<6> n; n:=_conc(0b1,_bext(src2, 4, 0));
           dst1:=_ash(src1,n);} };

DefineOp opDSPIDUALADD "%!(optguard) dspidualadd %s %s \[->\] %s"
        {src1="$2" in {gpr}, src2="$3" in {gpr},
         dst1="$4" in {tgpr}},
        {ISlot1(exectime=2, latency=1, slots=0)
         |ISlot3(exectime=2, latency=1, slots=0); WbBus},
        {if (guard==true) {if (src3<0>==0b1) {
           signed<32> temp1; signed<32> temp2;
           temp1:=_sext(_signed(_bext(src1,15,0),16),32)+
                  _sext(_signed(_bext(src2,15,0),16),32);
           temp2:=_sext(_signed(_bext(src1,31,16),16),32)+
                  _sext(_signed(_bext(src2,31,16),16),32);
           DUALCOMP32 } } };

DefineOp opDSPUQUADADDUI
        "%!(optguard) dspuquadaddui %s %s \[->\] %s"
        {src1="$2" in {gpr}, src2="$3" in {gpr},
         dst1="$4" in {tgpr}},
        {ISlot1(exectime=2, latency=1, slots=0)
         |ISlot3(exectime=2, latency=1, slots=0); WbBus},
        {if (guard==true) {if (src3<0>==0b1) {
           signed<32> i; signed<32> m; signed<32> n;
           signed<32> temp; i:=0; m:=31; n:=24;
           while(i<4) {
```

```
            temp:=_zext(_signed(_bext(src1,m,n),8),32)+
                  _sext(_signed(_bext(src2,m,n),8),32);
            if (temp<0) {_unsigned(_bext(dst1,m,n),8):=0x00;}
            else { if (temp>0x000000ff) {
                       _unsigned(_bext(dst1,m,n),8):=0xff;}
                    else { _signed(_bext(dst1,m,n),8):=
                                 _bext(temp,7,0);} }
            i:=i+1; m:=m-8; n:=n-8; } } } };

DefineOp opFGTRFLAGS "%!(optguard) fgtrflags %s %s \[->\] %s"
        {src1="$2" in {gpr} as float<24,8>,
         src2="$3" in {gpr} as float<24,8>,
         dst1="$4" in {tgpr} as float<24,8>},
        {ISlot3(exectime=1, latency=1, slots=0);WbBus},
        {if ((guard==true)&&(src3<0>==0b1)) {
           float<24,8> ftemp; dst1:=0;
           ftemp:=_fgt(src1,src2,24,8,
                       round_nearest,_bext(dst1,6,0));} } };


DefineOp opIFIR16 "%!(optguard) ifir16 %s %s \[->\] %s"
        {src1="$2" in {gpr}, src2="$3" in {gpr},
         dst1="$4" in {tgpr}},
        {ISlot2(exectime=3, latency=1, slots=0)
         |ISlot3(exectime=3, latency=1, slots=0); WbBus},
        {if (guard==true) {if (src3<0>==0b1) {
         dst1:=_sext(_signed(_bext(src1,31,16),16),32) *
                _sext(_signed(_bext(src2,31,16),16),32)
         + _sext(_signed(_bext(src1,15,0),16),32) *
                _sext(_signed(_bext(src2,15,0),16),32); } } };

DefineOp opIMUL "%!(optguard) imul %s %s \[->\] %s"
        {src1="$2" in {gpr}, src2="$3" in {gpr},
         dst1="$4" in {gpr}},
        {ISlot2(exectime=3, latency=1, slots=0)
         |ISlot3(exectime=3, latency=1, slots=0); WbBus},
        {if ((guard==true)&&(src3<0>==0b1)) { signed<64> temp;
           temp:=_sext(src1,64) * _sext(src2,64);
           dst1:=_bext(temp,31,0);} } };

DefineOp opJMPI "%!(optguard) jmpi ( %!(jtarget) )"
        {type=jmp},
        {ISlot2(exectime=3, latency=1, slots=3)
         |ISlot3(exectime=3, latency=1, slots=3)
         |ISlot4(exectime=3, latency=1, slots=3); WbBus},
        {if (guard==true) {if (src3<0>==0b1) {
```

```
         if (src1<0>==0b0) {tPC:=src2; } } } };

DefineOp opLD32D "%!(optguard) ld32d ( %!(imms7_s1) ) %s \[->\] %s"
        {src2="$3" in {gpr}, dst1="$4" in {gpr}, src4=MEM},
        {ISlot4(exectime=3, latency=1, slots=0)
         |ISlot5(exectime=3, latency=1, slots=0); WbBus},
        {if (guard==true) {if (src3<0>==0b1) {unsigned<2> tempbs;
          signed<32> tempaddr; signed<3> tempbsxor;
          if (PCSW<9>==0b1) {tempbs:=0b11;} else {tempbs:=0b00;}
          tempbsxor:=_conc(0b0,tempbs^0b11);
          tempaddr:=src2+_sext(src1,32); LD32MACRO } } };


OperationClass DMemspecClass {opDCB, opDINVALID, opRDSTATUS, opRDTAG};
OperationClass DMemClass {opH_ST16D, opH_ST32D, opH_ST8D, opILD16D,
        opILD16R, opILD16X, opILD8D, opILD8R, opLD32D, opLD32R,
        opLD32X, opULD16D, opULD16R, opULD16X, opULD8D, opULD8R};
OperationClass BugClass {opDINVALID, opDCB};

/*-------------------------------------------------------------------*/
/*               Constraints                                         */
/*-------------------------------------------------------------------*/
Constraints-Section

op1 in {DMemspecClass} & op2 in {DMemClass}: !(op1 && op2);
op1 in {DMemspecClass, DMemClass} & op2 in {BugClass}: !(op1->(1)op2);


/*-------------------------------------------------------------------*/
/*               Assembly Section                                    */
/*-------------------------------------------------------------------*/
Assembly-Section

OperationDelimiter ",";
InstructionDelimiter ";";
Comment ("(*", "*)");

DefineDirective DirData  "\[.data\]"  {type=SegStart, name="data"};
DefineDirective DirData1 "\[.data1\]" {type=SegStart, name="data1"};
DefineDirective DirText  "\[.text\]"  {type=SegStart, name="text"};

DefineDirective DirGlobal "\[.global\] %!(dirExpList)"
        {type=GlobalDecl, opnd1="$1"};

DefineDirective DirAscii "\[.ascii\] %s" {opnd1="$1"};
DefineDirective DirAlign "\[.align\] %d" {opnd1="$1"};
```

```
DirNT dirExp "%s" | "%d";
DirNT dirExpList "%!(dirExp) %!(dirExpList)" | "%!(dirExp)";
DefineDirective DirByte "\[.byte\] %!(dirExpList)" {opnd1="$1"};
DefineDirective DirHalf "\[.half\] %!(dirExpList)" {opnd2="$1"};
DefineDirective DirWord "\[.word\] %!(dirExpList)" {opnd3="$1"};
```

Index

Bibliography

[Abs00a] AbsInt Angewandte Informatik GmbH. *aiPop166. Code Compaction for the C166/ST10. User Documentation – Version 1.0*, 2000. http://www.absint.com.

[Abs00b] AbsInt Angewandte Informatik GmbH. *aiSee. Graph Visualization. User Documentation*, 2000. http://www.absint.com.

[AG85] A.V. Aho and M. Ganapathi. Efficient Tree Pattern Matching: An Aid to Code Generation. *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 334–340, 1985.

[AGT89] A.V. Aho, M. Ganapathi, and S.W.K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[AJ76] A.V. Aho and S.C. Johnson. Optimal Code Generation for Expression Trees. *Journal of the ACM*, 23(3):488–501, 1976.

[AJLA95] V.H. Allan, R.B. Jones, R.M. Lee, and S.J. Allan. Software Pipelining. *Computing Surveys*, 27(3):367–432, September 1995.

[AJU77] A.V. Aho, S.C. Johnson, and J.D. Ullman. Code Generation for Expressions with Common Subexpressions. *Journal of the ACM*, 24(1):146–160, 1977.

[AM99] G. Araujo and S. Malik. Optimal Code Generation for Embedded Memory non-homogeneous Register Architectures. *Proceedings of the ISSS*, pages 36–41, 1999.

[Ana95] Analog Devices. *ADSP-2106x SHARC User's Manual*, 1995.

[Ary85] S. Arya. An Optimal Instruction Scheduling Model for a Class of Vector Processors. *IEEE Transactions on Computers*, C-34, November 1985.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[Bal74a] E. Balas. Cutting Planes from Logical Conditions. *University of Wisconsin Nonlinear Programming Symposium*, April 1974.

[Bal74b] E. Balas. Intersection Cuts from Disjunctive Constraints. Technical Report MSSR No.330, Carnegie-Mellon University, 1974.

[Bal98] E. Balas. Disjunctive Programming: Properties of the Convex Hull of Feasible Points. *Discrete Applied Mathematics*, 89:3–44, 1998. Elsevier Science.

[Bar81] M.R. Barbacci. Instruction Set Processor Specifications (ISPS): The Notation and Its Applications. *IEEE Transactions on Computers*, C-30(1):24–40, January 1981.

[Bas95] S. Bashford. Code Generation Techniques for Irregular Architectures. Technical Report 596, University of Dortmund, 1995.

[BCE+98] F. Bodin, Z. Chamski, C. Eisenbeis, E. Rohou, and A. Seznec. GCDS: A Compiler Strategy for Trading Code Size Against Performance in Embedded Applications. Technical Report 3346, INRIA, January 1998.

[BCRS97] F. Bodin, Z. Chamski, E. Rohou, and A. Seznec. *Functional Specification of SALTO: A Retargetable System for Assembly Language Transformation and Optimization, rev. 1.00 beta.* INRIA, 1997.

[BCT94] P. Briggs, K. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, 1994.

[BD88] M.E. Benitez and J.W. Davidson. A Portable Global Optimizer and Linker. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, in SIGPLAN Notices*, 23(7):329–338, July 1988.

[BD94] M.E. Benitez and J.W. Davidson. Target-Specific Global Code Improvement: Principles and Applications. Technical report, Department of Computer Science, University of Virginia, Charlottesville, 1994.

[Bea91] S.J. Beaty. *Instruction Scheduling Using Genetic Algorithms.* PhD thesis, Department of Mechanical Engineering, Colorado State University, Fort Collins, Colorado, 1991.

[Ben94] M.E. Benitez. *Register Allocation and Phase Interactions in Retargetable Optimizing Compilers.* PhD thesis, University of Virginia, May 1994.

[BEP+96] J. Blazewicz, K.H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. *Scheduling Computer and Manufacturing Processes*. Springer-Verlag, 1996.

[BGS94] D. Bacon, S. Graham, and O. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 4, 1994.

[BHE91] D.G. Bradlee, R.R. Henry, and S.J. Eggers. The Marion System for Retargetable Instruction Scheduling. *Proceedings of the PLDI*, pages 229–240, 1991.

[BL99] S. Bashford and R. Leupers. Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths. *Design Automation for Embedded Systems*, pages 1–50, 1999.

[Bli96] T. Blickle. *Theory of Evolutionary Algorithms and Applications to System Design*. PhD thesis, ETH Zürich, 1996.

[Bra91] D.G. Bradlee. Retargetable Instruction Scheduling for Pipelined Processors. Phd thesis, Technical Report 91-08-07, University of Washington, 1991.

[Bri92] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston, Texas, April 1992.

[BST93] A. Bachmann, M. Schöbinger, and L. Thiele. Synthesis of Domain Specific Multiprocessor Systems including Memory Design. In *VLSI Signal Processing VI*, pages 417–425, New York, 1993. IEEE Press.

[Cam98] M.E. Campbell. *Evaluating ASIC, DSP, and RISC Architectures for Embedded Applications*. Northrop Grumman Corporation, 1998.

[CCK97] C-M. Chang, C-M. Chen, and C-T. King. Using Integer Linear Programming for Instruction Scheduling and Register Allocation in Multi-Issue Processors. *Computers and Mathematics with Applications*, 34(9):1–14, November 1997.

[CFRW91] R. Cytron, J. Ferrante, B.K. Rosen, and M.N. Wegman. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CH90] F.C. Chow and J.L. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, 1990.

[Cha82] G.J. Chaitin. Register Allocation and Spilling via Graph Coloring. In *Proc. SIGPLAN'82 Symp. on Compiler Construction, SIGPLAN Notices*, volume 17(6), pages 201–207, 1982.

[Chv83] V. Chvatal. *Linear Programming*. Freeman and Company, New York, 1983.

[CJP83] H. Crowder, E. Johnson, and M. Padberg. Solving Large Scale Zero-One Linear Programming Problems. *Operations Research*, 31(5):803–834, 1983.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1990.

[CWM93] S. Chaudhuri, R.A. Walker, and J.E. Mitchell. The Structure of Assignment, Precedence, and Resource Constraints in the ILP Approach to the Scheduling Problem. *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 25–31, 1993.

[CWM94] S. Chaudhuri, R.A. Walker, and J.E. Mitchell. Analyzing and Exploiting the Structure of the Constraints in the ILP-Approach to the Scheduling Problem. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):456–471, December 1994.

[Dak65] R.J. Dakin. A Tree-Search Algorithm for Mixed Integer Programming Problems. *The Computer Journal*, 8:250–255, 1965.

[Dan51] G.B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In Tj. C. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347. Wiley, New York, 1951.

[DF80] J.W. Davidson and C.W. Fraser. The Design and Application of a Retargetable Peephole Optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, April 1980.

[DF84] J.W. Davidson and C.W. Fraser. Code Selection through Object Code Optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, October 1984.

[Die95] T.A. Diep. *A Visualization-based Microarchitecture Workbench*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 1995.

[DK96] W. Dinkelbach and A. Kleine. *Elemente einer betriebswirtschaftlichen Entscheidungslehre*. Springer, 1996. In German.

[DR98] J.W. Davidson and N. Ramsey. Machine Descriptions to Build Tools for Embedded Systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 172–188. Springer LNCS, Volume 1474, June 1998.

[DT93] J.C. Dehnert and R.A. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, 1/2:181–228, May 1993.

[Eis00] F. Eisenbrand. *Gomory-Chvatal Cutting Planes and the Elementary Closure of Polyhedra.* PhD thesis, Saarland University, 2000.

[Ell86] J.R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1986.

[EM92] K. Ebcioglu and S. Moon. An Efficient Resource Constrained Global Scheduling Technique for Superscalar and VLIW Processors. *MICRO*, December 1992.

[Emm89] H. Emmelmann. *BEG – a Back End Generator.* GMD Forschungsstelle an der Universität Karlsruhe, November 1989.

[Emm92] H. Emmelmann. Code Selection by Regularly Controlled Term Rewriting. *Proceedings of the CODE91 Workshop*, 1992.

[Fau95] A. Fauth. Beyond Tool-Specific Machine Descriptions. In *[MG95]*, chapter 8, pages 138–152. Kluwer, 1995.

[Fer97] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems.* PhD thesis, Saarland University, 1997.

[FH91] C.W. Fraser and D. Hanson. A Retargetable Compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[FH95] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design And Implementation.* Benjamin/Cummings Publishing Company, Inc., 1995.

[FHMK94] A. Fauth, G. Hommel, C. Müller, and A. Knoll. Global Code Selection for Directed Acyclic Graphs. *Proceedings of the ACM International Conference on Compiler Construction*, pages 128–142, April 1994.

[FHP92] C.W. Fraser, D.R. Hanson, and T.A. Proebsting. Engineering a Simple, Efficient Code-Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, 1992.

[Fis81] J.A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[Fis92] P.C. Fishburn. Induced Binary Probabilities and the Linear Ordering Polytope: A Status Report. *Mathematical Social Sciences*, 23:67–80, 1992.

[FK93a] A. Fauth and A. Knoll. Automated Generation of DSP Program Development Tools Using a Machine Description Formalism. *Proceedings of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing ICASSP '93*, pages 457–460, apr 1993. http://www.techfak.uni-bielefeld.de/techfak/ags/ti/forschung/publikationen/icassp93.ps.gz.

[FK93b] A. Fauth and A. Knoll. Translating Signal Flowcharts into Microcode for Custom Digital Signal Processors. *Proceedings of the IEEE International Conference on Signal Processing*, pages 65–68, October 1993.

[FKL+99] C. Ferdinand, D. Kästner, M. Langenbach, F. Martin, M. Schmidt, J. Schneider, J. Theiling, S. Thesing, and R. Wilhelm. Run-Time Guarantees for Real-Time Systems - The USES Approach. In *Informatik '99. Informatik überwindet Grenzen. Jahrestagung der GI*, Informatik Aktuell. Springer, 1999.

[Fly95] M.J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett, 1995.

[Fou81] L.R. Foulds. *Optimization Techniques: An Introduction*. Springer, 1981.

[FOW87] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[FR91] S.M. Freudenberger and J.C. Ruttenberg. Phase Ordering of Register Allocation and Instruction Scheduling. In R. Giegerich and S.L. Graham, editors, *"Code Generation - Concepts, Tool, Techniques", Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, 1991*, pages 146–172. Workshops in Computing, Springer, 1991.

[FSW94] C. Ferdinand, H. Seidl, and R. Wilhelm. Tree Automata for Code Selection. *Acta Informatica*, 31:741–760, 1994.

[FVPF95] A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction Set Processors Using nML. In *Proceedings of the European Design and Test Conference*, pages 503–507. IEEE, 1995.

[GAG96] R. Govindarajan, E.R. Altman, and G.R. Gao. A Framework for Resource Constrained Rate Optimal Software Pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 7(11), November 1996.

[Gas89] F. Gasperoni. Compilation Techniques for VLIW-Architectures. Technical report, Courant Institute of Mathematical Science, New York University, March 1989.

[GE92] C.H. Gebotys and M.I. Elmasry. *Optimal VLSI Architectural Synthesis.* Kluwer Academic, 1992.

[GE93] C.H. Gebotys and M.I. Elmasry. Global Optimization Approach for Architectural Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1266–1278, 1993.

[Geb97] C.H. Gebotys. An Efficient Model for DSP Code Generation: Performance, Code Size, Estimated Energy. In *Proceedings of the 10th International Symposium on System Synthesis*, pages 41–47. IEEE Computer Society Press, 1997.

[GG78] R.S. Glanville and S.L. Graham. A new Method for Compiler Code Generation. *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, pages 231–240, January 1978.

[GH88] J. Goodman and W. Hsu. Code Scheduling and Register Allocation. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[GHK+98] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, and A. Nicolau. EXPRESSION: An ADL for System Level Design Exploration. Technical Report 1998-29, University of California, Irvine, 1998.

[GHR96] J.C. Gyllenhaal, M.M.W. Hwu, and B.R. Rau. HMDES Version 2.0. Specification. Technical Report IMPACT-96-3, University of Illinois at Urbana-Champaign, 1996.

[Gie82] R. Giegerich. Automatic Generation of Machine Specific Code Optimizers. In Richard DeMillo, editor, *Conference Record on the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 75–81, Albuquerque, NM, 1982. ACM Press.

[GJ79] M. Garey and D.S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* Freemann and Company, 1979.

[Goe97] M.X. Goemans. Improved Approximation Algorithms for Scheduling with Release Dates. *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 591–598, 1997.

[Gom63] R.E. Gomory. An Algorithm for Integer Solutions to Linear Programs. In R. Graves and P. Wolfe, editors, *Recent Advances in Mathematical Programming*, pages 269–302. McGraw-Hill, 1963.

[GS90] R. Gupta and M.L. Soffa. Region Scheduling: An Approach for Detecting and Redistributing Parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, 1990.

[GW96] D. Goodwin and K. Wilken. Optimal and Near-Optimal Global Register Allocation Using 0-1 Integer Programming. *Software—Practice and Experience*, 26(8):929–965, August 1996.

[Had98] G. Hadjiyiannis. ISDL: Instruction Set Description Language Version 1.0. Technical report, MIT RLE, April 1998.

[Hal97] L.A. Hall. Approximation Algorithms for Scheduling. In D.S. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*, pages 1–43. PWS Publishing Company, 1997.

[Har92] R. Hartmann. Combined Scheduling and Data Routing for Programmable ASIC Systems. *Proceedings of the European Conference on Design Automation*, pages 486–490, March 1992.

[HD89a] R.R. Henry and P.C. Damron. Algorithms for Table-Driven Code Generators Using Tree Pattern Matching. Technical Report 89-02-03, University of Washington, Seattle, 1989.

[HD89b] R.R. Henry and P.C. Damron. Encoding Optimal Pattern Selection in a Table-Driven Bottom-Up Tree-Pattern Matcher. Technical Report 89-02-04, University of Washington, Seattle, 1989.

[HD98] S. Hanono and S. Devadas. Instruction Scheduling, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *Proceedings of the Design Automation Conference 1998*, San Francisco, California, 1998. ACM.

[Hei93] W. Heinrich. *Formal Description of Parallel Computer Architectures as a Basis of Optimizing Code Generation.* PhD thesis, TU Munich, 1993.

[Hen84] R.R. Henry. *Graham-Glanville Code Generators.* PhD thesis, University of California, Berkeley, 1984.

[HGG$^+$99] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. *Proceedings of the DATE99*, 1999.

[HHR97] R.E. Hank, W.W. Hwu, and B.R. Rau. Region-Based Compilation: An Introduction and Motivation. *International Journal of Parallel Programming*, 25(2):113–146, 1997.

[HLH91] C. Hwang, J. Lee, and Y. Hsu. A Formal Approach to the Scheduling Problem in High Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(4):464–475, 1991.

[HLW00] M. Heffernan, J. Liu, and K. Wilken. Optimal Instruction Scheduling Using Integer Programming. *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 121–133, June 2000.

[HMC+93] W.-m. W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, and D.M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing, Kluwer Academic Publishers*, pages 229–248, 1993.

[Hoo88] J.N. Hooker. Resolution vs. Cutting Plane Solution of Interference Problems: Some Computational Experience. *Operations Research Letters*, 7(1), 1988.

[HP96] J.L. Hennessy and D.A. Patterson. *Computer Architecture a Quantitive Approach.* Morgan Kaufmann, San Francisco, 1996.

[ILO99] ILOG S.A. *ILOG CPLEX 6.5. User's Manual*, 1999.

[Inf00] Infineon, http://www.infineon.com. *TriCore v1.3. Architecture Manual*, 2000.

[Int89] Intel Corp., Santa Clara. *i860 64-bit Microprocessor Programmer's Reference*, 1989.

[JNS97] E.L. Johnson, G.L. Nemhauser, and M.W.P. Savelsbergh. Progress in Integer Programming: An Exposition. Technical Report LEC-97-02, Georgia Institute of Technology, School of Industrial and Systems Engineering, Atlanta, CA 30332-0205, January 1997. http://tli.isye.gatech.edu/reports.html.

[Kan87] G. Kane. *MIPS R 2000 RISC Architecture.* Prentice Hall, Englewood Cliffs, 1987.

[Kar84] N. Karmarkar. A new Polynomial-Time Algorithm for Linear Programming. *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 302–311, 1984.

[Käs97] D. Kästner. Instruktionsanordnung und Registerallokation auf der Basis ganzzahliger linearer Programmierung für den digitalen Signalprozessor ADSP-2106x. Master's thesis, Saarland University, 1997. In German.

[Käs99a] D. Kästner. TDL - Eine Architekturbeschreibungssprache für Postpassoptimierungen und -analysen. *Proceedings of the DSP Deutschland*, September 1999. In German.

[Käs99b] D. Kästner. TDL: A Hardware and Assembly Description Language. Technical Report TDL1.3, Transferbereich 14, Saarland University, 1999.

[Käs00a] D. Kästner. PROPAN: A Retargetable System for Postpass Optimisations and Analyses. *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, June 2000.

[Käs00b] D. Kästner. PROPAN: Ein Retargierbares System für Postpassoptimierungen und -analysen. In K. Mehlhorn and G. Snelting, editors, *Informatik 2000 – Neue Horizonte im neuen Jahrhundert: 30. Jahrestagung der Gesellschaft für Informatik*. Springer, September 2000. In German.

[Kha80] L.G. Khachiyan. Polynomial Algorithms in Linear Programming (in Russian). *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 20:51–68, 1980.

[KL98] D. Kästner and M. Langenbach. Integer Linear Programming vs. Graph Based Methods in Code Generation. Technical Report A/01/98, Saarland University, Saarbrücken, Germany, January 1998.

[KL99] D. Kästner and M. Langenbach. Code Optimization by Integer Linear Programming. In Stefan Jähnichen, editor, *Proceedings of the 8th International Conference on Compiler Construction CC99*, pages 122–136. Springer LNCS 1575, March 1999.

[KT98] D. Kästner and S. Thesing. Cache Sensitive Pre-Runtime Scheduling. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Montreal, CA, June 1998.

[KT99] D. Kästner and S. Thesing. Cache-Aware Pre-Runtime Scheduling. *Journal of Real-Time Systems*, 17:235–256, 1999.

[KW98] T. Kong and K.D. Wilken. Precise Register Allocation for Irregular Architectures. *Proceedings of the 31st International Microarchitecture Conference*, December 1998.

[KW99] D. Kästner and R. Wilhelm. Operations Research Methods in Compiler Backends. *Journal of Mathematical Communications*, 1999.

[Lam88] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *Proceedings of the SIGPLAN88 Conference on*

Programming Language Design and Implementation, pages 318–328, June 1988.

[Lan99] M. Langenbach. CRL – A Uniform Representation for Control Flow. Technical report, TFB 14, Saarland University, November 1999.

[LBSL97] P. Lapsley, J. Bier, A. Shoham, and E.A. Lee. *DSP Processor Fundamentals. Architectures and Features.* IEEE Press Series on Signal Processing. IEEE Press, New York, 1997.

[LCGDM94] D. Lanneer, M. Cornero, G. Goossens, and H. De Man. Data Routing: a Paradigm for Efficient Data-Path Synthesis and Code Generation. *Proceedings of the 7th ACM/IEEE International Symposium on High-Level Synthesis*, pages 17–22, May 1994.

[LCS+97] C. Liem, M. Cornero, M. Santana, P. Paulin, A. Jerraya, J.-M. Gentit, J. Lopez, X. Figari, and L. Bergher. An Embedded System Case Study: the Firm Ware Development Environment for a Multimedia Audio Processor. *34th Design Automation Conference*, 1997.

[LD60] A.H. Land and A.G. Doig. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(497–520), 1960.

[LDK+95] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Code Optimisation Techniques for Embedded DSP Microprocessors. *Design Automation Conference*, pages 599–604, 1995.

[LDKT95] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction Selection Using Binate Covering for Code Size Optimisation. *International Conference on Computer-Aided Design*, pages 393–399, 1995.

[LDS80] D. Landskov, S. Davidson, and B. Shriver. Local Microcode Compaction Techniques. *ACM Computing Surveys*, 12(3):261–294, September 1980.

[Leu97] R. Leupers. *Retargetable Code Generation for Digital Signal Processors.* Kluwer Academic Publishers, 1997.

[LLKS85] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *The Traveling Salesman Problem.* John Wiley and Sons, New York, 1985.

[LM94] R. Leupers and P. Marwedel. Instruction Set Extraction from Programmable Structures. *European Design Automation Conference*, pages 156–161, 1994.

[LM97] Rainer Leupers and Peter Marwedel. Time-constrained Code Compaction for DSPs. *IEEE Transactions on VLSI Systems*, 5(1), September 1997.

[LMP94] C. Liem, T. May, and P. Paulin. Register Assignment through Resource Classification for ASIP Microcode Generation. *Proceedings of the International Conference on Computer-Aided Design*, pages 397–403, 1994.

[LSU93] R. Lipsett, C. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, 12. edition, 1993.

[LT79] T. Lengauer and R.E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[LVPK+95] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: Retargetable Code Generation For Embedded DSP Processors. In *[MG95]*, pages 85–102. Kluwer, 1995.

[Mar99] F. Martin. *Generation of Program Analyzers*. PhD thesis, Saarland University, 1999.

[Meh88] K. Mehlhorn. *Datenstrukturen und effiziente Algorithmen 1*. Teubner, Stuttgart, 1988. In German.

[MG95] P. Marwedel and G. Goossens. *Code Generation for Embedded Processors*. Kluwer, Boston; London; Dortrecht, 1995.

[Mic94] G.D. Micheli. *Synthesis and Optimisation of Digital Circuits*. McGraw-Hill, Inc., New York, 1994.

[Mot88] Motorola, Inc. *MC88100 RISC Microprocessor User's Manual*, 1988.

[MP97] A. Mignotte and O. Peyran. Reducing the Complexity of ILP Formulations for Synthesis. In *Proceedings of the 10th International Symposium on System Synthesis*, pages 58–64. IEEE Computer Society Press, 1997.

[MS93] P. Marwedel and W. Schenk. Cooperation of Synthesis, Retargetable Code Generation and Test Generation in the MIMOLA Software System. *European Conference on Design Automation*, pages 63–69, 1993.

[Nic85] A. Nicolau. Uniform Parallelism Exploitation in Ordinary Programs. In *International Conference on Parallel Processing*, pages 614–618. IEEE Computer Society Press, August 1985.

[Nil98] H.-P. Nilsson. Porting the GNU C Compiler to the CRIS architecture. Master's thesis, Axis Communications AB, August 1998.

[NN92] A. Nicolau and S. Novack. An Efficient Global Resource Constrained Technique for Exploiting Instruction Level Parallelism. *Proceedings of the International Conference on Parallel Processing*, pages 297–301, August 1992.

[NN93] S. Novack and A. Nicolau. Trailblazing: A Hierarchical Approach to Percolation Scheduling. Technical Report TR-92-56, Irvine University, August 1993.

[NN94] S. Novack and A. Nicolau. Mutation scheduling: A Unified Approach to Compiling for fine-grain Parallelism. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 16–30. Springer LNCS, 1994.

[NND95] S. Novack, A. Nicolau, and N. Dutt. A Unified Code Generation Approach Using Mutation Scheduling. In *[MG95]*, pages 203–218. Kluwer, 1995.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, Berlin; Heidelberg; New York, 1999.

[Now87] L. Nowak. Graph Based Retargetable Microcode Compilation in the MIMOLA Design System. *20th Annual Workshop on Microprogramming*, pages 126–132, 1987.

[NP93] C. Norris and L. Pollok. A Scheduler-Sensitive Global Register Allocator. *Proceedings of Supercomputing*, 1993.

[NPW91] A. Nicolau, R. Potasman, and H. Wang. Register Allocation, Renaming and their Impact on Parallelism. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing*. Springer, 1991.

[NS92] G.L. Nemhauser and M.W.P. Savelsbergh. A Cutting Plane Algorithm for the Single Machine Scheduling Problem with Release Times. *Combinatorial Optimization: New Frontiers in the Theory and Practice, NATO ASI Series F: Computer and System Sciences*, 82:63–84, 1992. Springer, Berlin.

[NW88] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

[NW89] G.L. Nemhauser and L.A. Wolsey. Integer Programming. In G.L. Nemhauser, A.H.G. R. Kan, and M.J. Todd, editors, *Handbooks in*

Operations Research and Management Science, chapter VI, pages 447–527. North-Holland, Amsterdam; New York; Oxford, 1989.

[PCL96] P. Paulin, M. Cornero, and C. Liem. Trends in Embedded Systems Technology. In M.G. Sami and G. De Micheli, editors, *Hardware/Software Codesign, An Industrial Perspective*. Kluwer Academic Publishers, 1996.

[Pet88] R. Peters. L'ordonnancement sur une machine avec des constraintes de délai. *Journal of Operations Research, Statistics and Computer Science*, 28:22–76, 1988. In French.

[PF92] T.A. Proebsting and C.N. Fisher. Probabilistic Register Allocation. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 300–310, January 1992.

[Phi97] Philips Electronics North America Corporation. *TriMedia TM1000 Preliminary Data Book*, 1997.

[PHZM99] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA: Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. *Proceedings of the Design Automation Conference*, 1999.

[Pin93] S.S. Pinter. Register Allocation with Instruction Scheduling. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 248–257, 1993.

[PL88] E. Pelegri-Llopart. *Rewrite Systems, Pattern Matching, and Code Selection*. PhD thesis, University of California, Berkeley, 1988.

[PLG88] E. Pelegri-Llopart and S.L. Graham. Optimal Code Generation for Expression Trees: An Application of BURS Theory. *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 294–308, 1988.

[PLMS95] P.G. Paulin, C. Liem, T.C. May, and S. Sutarwala. FLEXWARE: A Flexible Firmware Development Environment for Embedded Systems. In *[MG95]*, pages 67–84. Kluwer, 1995.

[Pot80] C.N. Potts. An Algorithm for the Single Machine Sequencing Problem with Precedence Constraints. *Mathematical Programming Study*, 13:78–87, 1980.

[PR87] M.W. Padberg and G. Rinaldi. Optimisation of a 532 City Symmetric Traveling Salesman Problem by Branch and Cut. *Operations Research Letters*, 6:1–7, 1987.

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization, Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, 1982.

[PS91] J.C.H. Park and M.S. Schlansker. On Predicated Execution. Technical Report HPL-91-58, Hewlett-Packard Laboratories, Palo Alto CA, May 1991.

[QS94] M. Queyranne and A.S. Schulz. Polyhedral Approaches To Machine Scheduling. Technical Report 408/1994, Technische Universität Berlin, Fachbereich 3 Mathematik, 1994.

[RF93] B.R. Rau and J.A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[RF97] N. Ramsey and M.F. Fernandez. Specifying Representations of Machine Instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.

[RGSL96] J. Ruttenberg, G.R. Gao, A. Stoutchinin, and W. Lichtenstein. Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler. *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI 96)*, 31(5):1–11, May 1996.

[RH88] K. Rimey and P.N. Hilfinger. Lazy Data Routing and Greedy Scheduling. *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture*, 21:111–115, 1988.

[RKA99] B.R. Rau, V. Kathail, and S. Aditya. Machine-Description driven Compilers for EPIC and VLIW processors. *Design Automation for Embedded Systems*, 4(2/3):71–118, 1999.

[Rüt98] O. Rüthing. *Interacting Code Motion Transformations: Their Impact and Their Complexity.* Springer LNCS 1539, 1998.

[San94] G. Sander. Graph Layout through the VCG Tool. In R. Tamassia and I.G. Tollis, editors, *Proceedings of the DIMACS International Workshop on Graph Drawing*, pages 194–205. Springer LNCS 894, 1994.

[San96] G. Sander. *Visualisierungstechniken für den Compilerbau.* PhD thesis, Universität des Saarlandes, Fachbereich 14, 1996. In German.

[San99] G. Sander. Graph Layout for Applications in Compiler Construction. *Theoretical Computer Science*, 217(2):175–214, 1999.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, 1986.

[Sch93] H.R. Schwarz. *Numerische Mathematik.* Teubner, Stuttgart, 1993. In German.

[Sch96a] A.S. Schulz. *Polytopes and Scheduling.* PhD thesis, Technische Universität Berlin, February 1996.

[Sch96b] A.S. Schulz. Scheduling to Minimize Total Weighted Completion Time: Performance Guarantees of LP-Based Heuristics and Lower Bounds. *Proceedings of the 5th International Conference on Integer Programming and Combinatorial Optimization*, pages 301–315, June 1996.

[SCL96] M.A.R. Saghir, P. Chow, and C.G. Lee. Exploiting Dual Data-Memory Banks in Digital Signal Processors. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[Set75] R. Sethi. Complete Register Allocation Problems. *SIAM Journal of Computing*, 4(3):226–248, 1975.

[Sie96] Siemens. *C165/C163 User's Manual 10.96 Version 2.0.* Siemens AG, 1996. http://www.infineon.com.

[SN98] M.W.P. Savelsbergh and G.L. Nemhauser. *Functional Description of MINTO, a Mixed INTeger Optimizer. Version 3.0.* Georgia Institute of Technology. School of Industrial and Systems Engineering., Atlanta, USA, March 1998.

[SPA97] SPAM Research Group, http://www.ee.princeton.edu/spam. *SPAM Compiler User's Manual*, September 1997.

[ST94] D.L. Springer and D.E. Thomas. Exploiting the Special Structure of Conflict and Compatibility Graphs in High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7):843–856, July 1994.

[Sta94] Stanford Compiler Group. *SUIF Compiler System: The SUIF Library*, 1994.

[Sta98] R. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Cambridge/Massachusetts, 1998.

[SU70] R. Sethi and J.D. Ullman. The Generation of Optimal Code for Arithmetic Expressions. *Journal of the ACM*, 17(4):715–728, 1970.

[Sud98] A. Sudarsanam. *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors.* PhD thesis, University of Princeton, November 1998.

[SUW97] Martin W.P. Savelsbergh, R.N. Uma, and Joel Wein. An Experimental Study of LP-Based Approximation Algorithms for Scheduling Problems. Technical Report LEC-97-11, Georgia Institute of Technology, Atlanta, GA 30332-0205, 1997. http://tli.isye.gatech.edu/reports.html.

[Tex97] Texas Instruments. *TMS320C62xx Programmer's Guide*, 1997.

[Tex98a] Texas Instruments. *TMS320C5x User's Guide*, 1998.

[Tex98b] Texas Instruments. *TMS320C62x/C67x CPU and Instruction Set. Reference Guide.*, March 1998.

[The00] H. Theiling. Extracting Safe and Precise Control Flow from Binaries. *7th International Conference on Real-Time Computing Systems and Applications*, July 2000. To appear.

[Thi93] L. Thiele. Resource Constraint Scheduling of Uniform Algorithms. *Proceedings of the Conference on Application Specific Processor Arrays ASAP93*, pages 29–40, October 1993.

[Thi95] L. Thiele. Resource Constraint Scheduling of Uniform Algorithms. *International Journal of VLSI Signal Processing*, 10:295–310, 1995.

[TM95] D. Thomas and P. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 2. edition, 1995.

[tri98] TRIMARAN: An Infrastructure for Research in Instruction-Level Parallelism. http://www.trimaran.org, 1998.

[TTZ96] J. Teich, L. Thiele, and Li Zhang. Scheduling of Partitioned Regular Algorithms on Processor Arrays with Constrained Resources. *Proceedings of the Conference on Application Specific Processor Arrays ASAP96*, pages 131–144, 1996.

[TZB99] J. Teich, E. Zitzler, and S.S. Bhattacharyya. 3D Exploration of Software Schedules for DSP Algorithms. *7th International Workshop on Hardware/Software Codesign*, pages 168–172, May 1999.

[WGB94] T.C. Wilson, G.W. Grewal, and D.K. Banerji. An ILP Solution for Simultaneous Scheduling, Allocation, and Binding in Multiple Block Synthesis. In *Proceedings of the International Conference on Computer Design : VLSI in Computers and Processors*, pages 581–586. IEEE Computer Society Press, 1994.

[WGHB95] T. Wilson, G. Grewal, S. Henshall, and D. Banerji. An ILP-Based Approach to Code Generation. In *[MG95]*, chapter 6, pages 103–118. 1995.

[Wil93a] H.P. Williams. *Model Building in Mathematical Programming.* John Wiley and Sons, New York, 3. edition, 1993.

[Wil93b] H.P. Williams. *Model Solving in Mathematical Programming.* John Wiley and Sons, 1993.

[WM95] R. Wilhelm and D. Maurer. *Compiler Design.* Addison-Wesley, 1995.

[WM97] R. Wilhelm and D. Maurer. *Übersetzerbau. Theorie, Konstruktion, Generierung; zweite, überarbeitete und erweiterte Auflage.* Springer, Berlin; Heidelberg; New York, 1997. In German.

[WW89] B. Weisgerber and R. Wilhelm. Two Tree Pattern Matchers for Code Selection (Including Targeting). In D. Hammer, editor, *Compiler Compilers and High Speed Compilation*, pages 215–229. Springer LNCS 371, 1989.

[ZABT00] E. Zehendner, W. Amme, P. Braun, and F. Thomasset. Data Dependence Analysis of Assembly Code. *International Journal of Parallel Programming*, 28(5):431–467, 2000.

[Zha96] L. Zhang. *SILP. Scheduling and Allocating with Integer Linear Programming.* PhD thesis, Saarland University, 1996.

[ZSWS95] V. Zivojnovic, H. Schraut, M. Willems, and R. Schoenen. DSPs, GPPs, and Multimedia Applications – An Evaluation Using DSPstone. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, pages 1779–1783. DSP Associates, October 1995.

[ZT99] E. Zitzler and L. Thiele. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, November 1999.

[ZTB00] E. Zitzler, J. Teich, and S.S. Bhattacharyya. Evolutionary Algorithms for the Synthesis of Embedded Software. *IEEE Transactions on VLSI Systems*, 2000. To appear.

[ZVSM94] V. Zivojnovic, J.M. Velarde, C. Schläger, and H. Meyr. DSPSTONE: A DSP-Oriented Benchmarking Methodology. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, 1994.