TDL: A Hardware Description Language for Retargetable Postpass Optimizations and Analyzes

Daniel Kästner

Universität des Saarlandes &
AbsInt Angewandte Informatik GmbH
Saarbrücken, Germany
kaestner@absint.com

Abstract. The hardware description language TDL has been designed with the goal to generate machine-dependent postpass optimizers and analyzers from a concise specification of the target processor. TDL is assembly-oriented and provides a generic modeling of irregular hardware constraints that are typical for many embedded processors. The generic modeling supports graph-based and search-based optimization algorithms. An important design goal of TDL was to achieve extendibility, so that TDL can be easily integrated in different target applications. TDL is at the base of the PROPAN system that has been developed as a retargetable framework for high-quality code optimizations at assembly level. For two contemporary microprocessors, the Analog Devices SHARC 2106x, and the Philips TriMedia TM1000, significant improvements of the code produced by production-quality compilers could be achieved with short retargeting time. TDL has also been used for implementing postpass optimizations for the Infineon C16x/ST10 processor that are part of a commercial postpass optimizer. TDL specifications are concise and can be produced in short time.

1 Introduction

During the past years, the markets for telecommunication, embedded systems, and multimedia applications have been growing rapidly. Real-time applications, which are part of many embedded systems, impose stringent performance requirements. Cost constraints and performance requirements have led to the development of irregular hardware architectures specially designed for real-time applications and digital signal processing. For this type of architecture the quality of the code generated by traditional high-level language compilers is often not satisfactory [36, 22]. Generating efficient code for irregular architectures requires sophisticated optimization techniques exploiting specific hardware features of the target processor. The shortcomings of available compilers have resulted in many applications being developed in assembly language, or at least in part. Due to the growing complexity of embedded applications and the shrinking design cycles of embedded products the usage of high-level programming languages is becoming increasingly imperative. There is an urgent need for *retargetable* code generation and optimization techniques that can be quickly adapted to different target architectures, e. g. by deriving machine-specific properties from a dedicated hardware description. At the same time these techniques must be able to produce high-quality code for the modeled architectures.

The Propan system [19, 16, 17] has been developed as a retargetable framework for high-quality code optimizations and machine-dependent program analyzes at assembly level. In the past, research on retargetability has mainly focused on closed compilation systems. Examples for retargetable research compilers are, e. g. CHESS[21], RECORD[22], AVIV[12], SPAM [32], or EXPRESS[11]. 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. 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. The machine description language Tdl (*Target Description Language*) allows to specify the hardware resources of the target processor, its instruction set, irregular hardware constraints and the assembly language in a concise way [15]. Irregular hardware constraints are modeled in a generic way enabling them to be exploited in generic graph-based and search-based optimization algorithms. Tdl offers a hierarchical structuring of operation semantics specification, and semantical consistency checks.

The article is structured as follows: after a brief overview of related work in Sec. 2, the basic design principles and the most important language elements of Tdl are described in Sec. 3. To illustrate the usage of TDL, the Propan-framework is sketched in Sec. 4. Sec. 5 summarizes experimental results gained with applications of Tdl. Sec. 6 contains a summary and gives an outlook to future work.

2 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 modeling and simulating, widely used languages are VHDL [23] and Verilog [34]; well-known approaches used in code generation are Isps [3], Maril [6, 5], the Mimola language [27], the Salto language [4], Sled [30], and nML [8]. Languages currently under development are Lisa [28] aiming at generating cycle-accurate simulators for architectures with complex pipelines, Isdl [10], Expression [11], and λ−RTL [7].

Hardware description languages can be categorized as behavioral, structural or mixed behavioral/structural languages [25]. A behavioral description specifies the instruction set of the target processor and focuses on the semantics, i. e. the behavior 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.

An assembly-oriented architecture description language is used in the SALTO system [4]. SALTO has been designed to support analyzes and transformations of low-level code. The architecture specification covers the hardware resources and the instruction set, but without means for specifying detailed operation semantics. Complex interactions among operations or interdependencies between scheduling and allocation decisions cannot be modeled.

ISDL [10] 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. Structural information is restricted to the declaration of storage resources; behavioral components are the specification of the instruction set and a mechanism for specifying constraints on operation combinations. Since only storage resources can be declared the problem of functional unit binding cannot be addressed with an ISDL description. The constraint mechanism is purely syntactical and does not exploit structural and semantical information from other specification parts. There is no semantical analysis of the machine description itself.

nML [8] is used in the retargetable code generator CBC, the retargetable compiler CHESS and the instruction set simulator CHECKERS [21]. The instruction set is modeled as an attribute grammar; each terminal of the grammar corresponds to a valid instruction. Structural information is provided by declaring the available storage resources. Specifying the timing of machine operations, irregular hardware constraints, and building long instruction words from independent operations, i. e. the VLIW execution model, is not explicitly supported. Since nML has not been designed for processing assembly files, specifying the syntax of assembly expressions, comments, or directives is not supported.

SLED [30] is part of the CSDL (*Computer Systems Description Languages*) language family [7]. SLED focuses on instruction decoding and encoding; for specifying the semantics of machine operations a dedicated language called λ-RTL has been developed [7]. 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. A detailed specification of the timing behavior of operations is not directly supported.

EXPRESSION [11] is a mixed-level language for supporting architectural design space exploration and automatic generation of compilers and simulators. Structural information is given by specifying the hardware resources of the architecture, including its memory hierarchy, the pipeline mechanism, and the data transfer paths. However, the assembly syntax cannot be specified and the modeling of irregular hardware characteristics seems to be restricted. As an example it is not clear whether encoding restrictions limiting instruction word packing can be specified. In [11, 9] no information about the specification of the detailed semantics of machine operations is given.

3 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 modeled. A TDL specification provides all information about the target architecture that can influence program analyzes and optimizations. 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 offers a generic modeling of irregular hardware constraints and a hierarchical structuring of the instruction set specification. TDL is easily extendible and flexible in use which is a presupposition for modeling a wide range of target architectures and for supporting different kinds of optimizations and analyzes.

TDL can be classified as a mixed structural/behavioral description formalism [25]. The resources are declared in a structural style while the machine operations are mainly described from the view of the operation behavior, 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. Each TDL description is checked for semantical consistency so that input errors and inconsistencies in the machine description are detected early.

In the remainder of this chapter, the different sections of a TDL description are presented.

3.1 Resource Specification

Hardware-sensitive program analyzes and optimizations 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. 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. The designer can extend the domain of the predefined attributes and declare user-defined attributes if additional properties have to be taken into account. It is also possible to declare user-defined resource types which allows for maximum flexibility with respect to the range of supported hardware architectures and target applications. Those declarations are used during the semantical analysis of the machine description. Different views of hardware resources are supported in TDL by a dedicated alias mechanism. As an example, in the Analog Devices

ADSP-2106x [2] the same register file is used as integer and as floating-point register file. One view is associated with fixed-point operations, the other with floating-point operations; the assembly representation of both views differs.

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 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 modeled in the constraint section. Currently TDL cannot be used to completely specify architectures with complex superscalar pipelines. However there is ongoing work to develop a specification mechanism for complex superscalar pipelines and integrate it into TDL.

```
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;
```

Fig. 1. Example of a Resource Section.

An example of a resource section is shown in Fig. 1. First a functional unit type *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 analyzes and optimizations. 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`.

3.2 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 behavior 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 [35]. 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.

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

Fig. 2. Example Operation Declaration

An example for an operation declaration is shown in Fig. 2 where a machine operation with unambiguous name `IAdd` is declared. This name is followed by the definition of the assembly representation of the operation. The assembly representation is specified in a syntax similar to C format strings. The expression `%!(optguard)` represents an occurrence of the 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}` specifies that the first occurrence of `%s` in the format string (the second placeholder) denotes the destination register of the operation which has to be a register from 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 for the instruction set that contains 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` 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`. If a condition register is specified (with a leading exclamation mark) the operation is executed only if the least significant condition bit has the value 1 (0).

The subsequent two blocks represent the timing and the semantics of the operation. The reservation table specification of Fig. 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.

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.

Specifying 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, etc. In TDL the semantics is specified by a dedicated register transfer language (RTL) that is similar to the mechanism of register transfer lists (cf. [8, 10, 7]). A register transfer represents the transfer of 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.

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. Thus 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. 3 in BNF form; the complete definition is given in [15].

```
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 )
```

Fig. 3. Skeleton of the RTL grammar in BNF form.

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 comprise 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. 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 also has the possibility of declaring user-defined "canonical" functions. This way the specification can be modularized and shortened. 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 analyzing 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. 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.

3.3 Constraint and Assembly Specification

In the constraint section of TDL, restrictions of instruction-level parallelism and resource usage as well as interdependencies between scheduling and allocation are modeled 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. 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 previous approaches, e. g. [10], and it allows modeling 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 automatically transformed into C functions for generic graph-based optimizers (e. g. list scheduling) and in integer linear constraints for generic search-based optimizers. This way, irregular hardware properties can be modeled in a completely generic way.

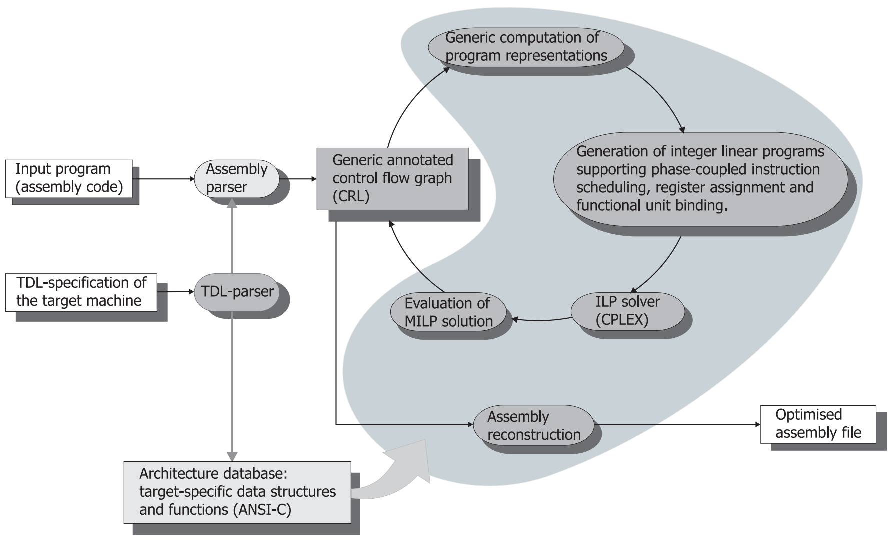

Fig. 4. The PROPAN system.

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 optimization 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 optimizations 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 specified architectural restrictions. For each rule, two functions are generated, one for the use in ILP-based optimization, 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. 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 optimization phases have to individually analyses the representation of the rules and extract the information relevant for them.

The syntax of the constraint language is shown in Fig. 5 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 optimizers. 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

```
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
```

Fig. 5. Excerpt of the grammar of the constraint language in BNF form.

operation attributes can be compared; the feasible comparisons depend on the type of the attribute values (character strings and resource references can only compared for (in)equality, numerical values also for the relations $<, >, \geq, \leq$).

```
Constraints-Section

op in {C0}: op.dst1 = op.src1;
op1 in {C1} & op2 in {C2}: !(op1 && op2);
(op1 in {AluOps} & op2 in {MulOps}):
 (op1 && op2) -> ((op1.src1 in {GroupA})
  & (op1.src2 in {GroupB})
  & (op2.src1 in {GroupC})
  & (op2.src2 in {GroupD}));
```

Fig. 6. Exemplary constraint section.

Some exemplary rules are shown in Fig. 6. 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 digital signal processor Analog Devices ADSP-2106x SHARC[2]. Some operations can only be executed in parallel if all operands reside in uniquely defined register groups within the heterogeneous register file. If other registers are used, the operations still are feasible but cannot be executed in parallel [19, 17].

The assembly section deals with syntactic details of the assembly language such as instruction or operation delimiters, assembly directives, assembly expressions, 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 [4]. While the specification of the syntax of assembly directives would also be possible in other languages as, e. g. nML [8], those could not be distinguished from regular machine operations. An example of a simple directive declaration in TDL is shown in the following; more details can be found in [15, 17].

```
DefineDirective DirGlobVar ".global %s;"
  {type = GlobalDecl, name="$1"};
```

4 The PROPAN Framework

The PROPAN system (*Postpass-oriented Retargetable Optimizer and Analyzer*) [19, 16, 17] has been developed as a retargetable framework for high-quality code optimizations and machine-dependent program analyzes at assembly level. An overview of PROPAN is shown in Fig. 4.

PROPAN combines the issues of machine description driven retargetability and of search-based postpass optimisations to improve the quality of machine code especially for irregular hardware architectures.

The retargetability concept of PROPAN is based on the combination of generic and generative techniques. *Generic* program parts are independent from the target architecture and can be used for different processors without any modification. Hardware-specific information is retrieved in a standardized way from an architecture 'database' which is *generated* from the TDL-description.

The input of PROPAN consists of a TDL-description of the target machine and of the assembly programs that are to be analysed or optimized. 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 optimizer.

The generated assembly parser reads the input programs and reconstructs their control flow graphs [20]. From the control flow graph, other required program representations as, e. g., the data dependence and the control dependence graphs are calculated. The optimizations are based on integer linear programming (ILP) and allow a phase-coupled modeling of instruction scheduling, register assignment and resource allocation taking precisely into account the hardware characteristics of the target architecture [16, 17]. The generated integer linear programs can be solved exactly, or by using novel ILP-based approximations. The approximations allow the calculation time for both models to be drastically reduced while obtaining a solution quality that is superior to conventional graph-based approaches [18]. The optimization scope is not restricted to basic blocks: a dedicated superblock concept allows the optimization scope to be extended across basic block and loop boundaries. The superblock mechanism also allows the ILP-based high-quality optimizations to be combined with fast graph-based heuristics. This way, ILP optimizations can be restricted to frequently used code sequences like inner loops, providing for computation times that are acceptable for practical use [17].

5 Experimental Results

This section focuses on the experience gained with TDL and PROPAN to illustrate the design goals and the application field of the TDL description language.

The PROPAN framework has been retargeted to several different target architectures. It has been used to generate ILP-based postpass optimizers for two widely used contemporary digital signal processors with considerably different hardware characteristics, the Analog Devices ADSP-2106X SHARC [2] and the Philips TriMedia TM1000 [29].

All hardware characteristics of the ADSP-2106X SHARC and the TriMedia TM1000 that have to be known for phase-coupled code optimization are provided by the TDL-descriptions. This especially includes constraints for modelling irregular restrictions of instruction-level parallelism. The SHARC-optimizer performs integrated instruction scheduling and register assignment; the TM1000 optimizer couples instruction scheduling and functional unit binding. In Sec. 5.1 and Sec. 5.2 the experimental evaluation of both optimizers is shortly summarized. The evaluation shows that using the hardware description language TDL enables significant improvements of the code produced by production-quality compilers to be achieved with short retargeting time.

PROPAN has also been used as a platform to implement hardware-specific postpass optimizations for the Infineon C16x [31] microprocessor family that are part of a commercial postpass optimizer [1]. TDL descriptions also exist for the Infineon TriCore μC/DSP [14], the Motorola Coldfire 5307 [26] and the TI320C6x [33].

Including comments, the TDL files contain between 1000 and 3000 lines; by using the `cpp` preprocessor the size can be further reduced. The time to develop and test a TDL description for a new architecture ranges from two weeks for an experienced user to two months for unexperienced users.

5.1 The ADSP-2106x SHARC

The ADSP-2106X SHARC is a digital signal processor with irregular architecture and restricted instruction-level parallelism. It has a VLIW execution model: several machine operations can be grouped to long instructions and be executed in parallel. Two memory banks are available that can be configured as program and data memory where program memory can be used to store instructions and data.

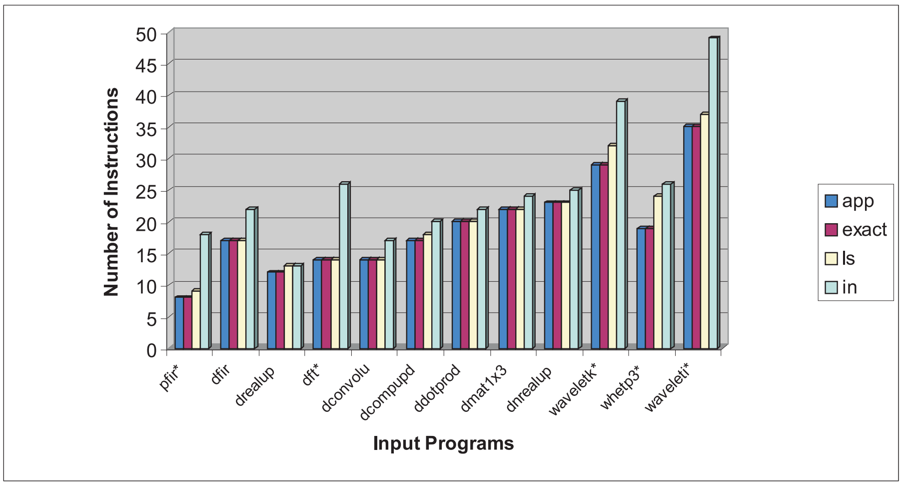

Fig. 7. Optimization Results (ADSP-2106X SHARC).

Data accesses to program memory and data memory can be executed in parallel for a restricted set of addressing modes, if specific address registers are used. 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. A presupposition is that each of the four input operands is located in a register from a specific register group within the heterogeneous general purpose register file. If any operand is located in a register from another group, the operation is feasible but cannot be part of a multifunctional instruction (cf. Sec. 3.3). Thus, generating high-quality code requires a phase-coupled modeling of instruction scheduling and register assignment.

The experiments have been performed under SunOS 5.7 on a SPARC Ultra-Enterprise 10000; the generated integer linear programs are solved by the CPLEX library [13]. The investigated input programs are typical applications of digital signal processing. They comprise the computation routines of the DspStone benchmark [36] and some hand-written assembly programs. The input programs from the DspStone benchmark [36] have been compiled to assembly code by the `gcc`-based `g21k` compiler. The hand-crafted assembly programs have been explicitly sequentialized so that in the input programs the available instruction-level parallelism is not exploited. The input programs contain between 14 and 55 microoperations.

An overview of the results for some representative input programs is given in Fig. 7. In Fig. 7 a comparison of code quality is shown; the number of compacted instructions in the exact, i. e., provably optimal ILP-based solution (`exact`) is compared to the result of the fastest ILP-based approximation (`app`) [17], a generic list scheduling algorithm with highest-level-first heuristic (`ls`) and to the schedule of the input programs (`in`). In order to ensure a conservative comparison, an optimal register assignment is given as an input to the list scheduling algorithm while it is explicitly computed by the ILP-based methods. The programs marked with an asterisk are the sequentialized hand-crafted assembly programs; for those programs the number of instructions in the input program corresponds to the number of microoperations.

In most cases the ILP-based approximations [18] produce optimal results; the optimal number of instructions was only exceeded in 5 of 108 approximative computations. The computation time of the ILP-based approximations ranges from a few seconds up to 15 minutes for input programs containing up to 30 – 50 instructions. The superblock mechanism allows to combine them with fast heuristic methods to generate highly optimized code for hotspots, i. e. important and often executed code sequences like inner loops within acceptable computation time.

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%.

5.2 The Philips TriMedia TM1000

The TriMedia TM1000 is a multimedia processor with a VLIW architecture where the instruction scheduling interacts with the *slot allocation problem.* Each long instruction word is composed of five microoperations that are issued simultaneously. All operations have to be assigned to issue slots in the instruction word and the assignment of operations to issue slots is restricted. The assignment to issue slots also determines which operations can be executed in parallel. Moreover, the operations have to be synchronized with respect to the write-back bus; no more than five operations may write their result simultaneously on the bus. Since different operations have different execution times, this is a non-trivial problem.

An overview of the results for some representative input programs is presented in Fig. 8. The input programs have been generated with the highly optimizing Philips `tmcc` compiler. In Fig. 8 the number of compacted instructions in the exact, i. e., provably optimal ILP-based solution (`exact`) is compared to the result of the fastest ILP-based approximation (`app`), of a generic list scheduling algorithm with highest-level-first heuristic that performs resource allocation on the fly (`ls`) [17] and to the schedule of the input programs (`in`). In most cases the `tmcc` compiler produces optimal results, but for some input programs the code can be improved significantly. The code quality of the ILP-based approximation is high. For 18 of 26 investigated programs it produces an optimal result; for the remaining programs, the optimal number of instructions is exceeded only by one. The computation time ranges from a few seconds up to 9 minutes for input

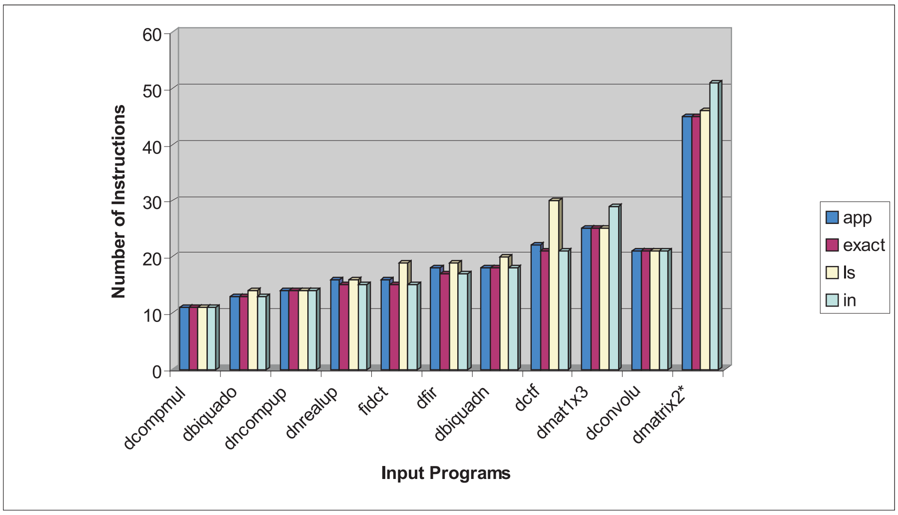

Fig. 8. Optimization Results (TM1000).

programs containing up to 70 – 100 instructions. Again the superblock mechanism allows to combine them with fast heuristic methods to generate highly optimized code for hotspots within acceptable computation time.

The schedule produced by the generic list scheduling algorithm exceeds the optimal number of instructions on average by 7.29%. The two hand-crafted assembly programs (`fidct,dctf`) 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. Here the optimal number of instruction is exceeded by more than 25% respectively more than 42%.

6 Conclusion and Outlook

In this article the machine description language TDL has been presented. TDL has been designed to support generating machine-dependent postpass optimizers and program analyzers on assembly level. The focus is on embedded processors and irregular hardware architectures for which traditional code generation and optimization techniques fail to achieve a satisfactory code quality. TDL is assembly-oriented and provides a generic modeling of irregular hardware constraints that allows them to be exploited in generic graph-based and search-based optimization algorithms. The extendibility of TDL and dedicated abstraction mechanisms provide for an easy integration of TDL in different target applications. It is used within the PROPAN system that has been developed as a retargetable framework for high-quality code optimizations and machine-dependent program analyzes at assembly level. PROPAN allows machine-specific postpass optimizers to be automatically generated from a TDL specification. The optimizers generated can be based on heuristic algorithms as well as exact or approximative ILP-based methods. With short retargeting time significant improvements of the code produced by production-quality compilers can be achieved.

TDL has also been used for implementing hardware-sensitive postpass optimizations based on the PROPAN framework that are part of a commercial postpass optimizer. TDL descriptions have been written for various contemporary embedded processors; the specifications are concise and can be produced in short time.

Future research aims at different directions. There is ongoing work to extend the TDL language by a modeling of superscalar architectures with complex pipelines. The generation of cycle-accurate instruction set simulators from the TDL specification is another goal. Moreover there is ongoing work to develop an interface between TDL and the program analyzer generator PAG [24]. This allows to use the same program analyzer specification to generate hardware-sensitive program analyzers for different target architectures. Examples are value analyzes that allow to disambiguate memory accesses and remove spurious data dependences in guarded code. Furthermore, the modeling of additional processors is planned.

References

1. AbsInt Angewandte Informatik GmbH. *aiPop166. Code Compaction for the C166/ST10. User Documentation – Version 1.0*, 2000. http://www.absint.com.
2. Analog Devices. *ADSP-2106x SHARC User's Manual*, 1995.
3. M. Barbacci. Instruction Set Processor Specifications (ISPS): The Notation and Its Applications. *IEEE Transactions on Computers*, C-30(1):24–40, Jan. 1981.
4. 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.
5. D. Bradlee. Retargetable Instruction Scheduling for Pipelined Processors. Phd thesis, Technical Report 91-08-07, University of Washington, 1991.
6. D. Bradlee, R. Henry, and S. Eggers. The Marion System for Retargetable Instruction Scheduling. *Proceedings of the PLDI*, pages 229–240, 1991.
7. J. 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.
8. 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.
9. 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.
10. G. Hadjiyiannis. ISDL: Instruction Set Description Language Version 1.0. Technical report, MIT RLE, Apr. 1998.

11. 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.
12. 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.
13. ILOG S.A. *ILOG CPLEX 6.5. User's Manual*, 1999.
14. Infineon, http://www.infineon.com. *TriCore v1.3. Architecture Manual*, 2000.
15. D. Kästner. TDL: A Hardware and Assembly Description Language. Technical Report TDL1.3, Transferbereich 14, Saarland University, 1999.
16. 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.
17. D. Kästner. *Retargetable Code Optimisation by Integer Linear Programming*. PhD thesis, Saarland University, 2000.
18. D. Kästner. ILP-based Approximations for Retargetable Code Optimization. *International Conference on Optimization: Techniques and Applications (ICOTA01)*, 2001.
19. D. Kästner and M. Langenbach. Code Optimization by Integer Linear Programming. In S. Jähnichen, editor, *Proceedings of the 8th International Conference on Compiler Construction CC99*, pages 122–136. Springer LNCS 1575, Mar. 1999.
20. D. Kästner and S. Wilhelm. Generic Control Flow Reconstruction from Assembly Code. *Proceedings of the ACM SIGPLAN Joined Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02)*, June 2002.
21. 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 *[25]*, pages 85–102. Kluwer, 1995.
22. R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, 1997.
23. R. Lipsett, C. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, 12. edition, 1993.
24. F. Martin. *Generation of Program Analyzers*. PhD thesis, Saarland University, 1999.
25. P. Marwedel and G. Goossens. *Code Generation for Embedded Processors*. Kluwer, Boston; London; Dortrecht, 1995.
26. Motorola Inc. *MCF5307 ColdFire Integrated Microprocessor User's Manual*, Aug. 2000. MCF5307UM/D, Rev. 2.0.
27. L. Nowak. Graph Based Retargetable Microcode Compilation in the MIMOLA Design System. *20th Annual Workshop on Microprogramming*, pages 126–132, 1987.
28. 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.
29. Philips Electronics North America Corporation. *TriMedia TM1000 Preliminary Data Book*, 1997.
30. N. Ramsey and M. Fernandez. Specifying Representations of Machine Instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.
31. Siemens. *C165/C163 User's Manual 10.96 Version 2.0.* Siemens AG, 1996. http://www.infineon.com.

32. SPAM Research Group, http://www.ee.princeton.edu/spam. *SPAM Compiler User's Manual*, Sept. 1997.
33. Texas Instruments. *TMS320C62xx Programmer's Guide*, 1997.
34. D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 2. edition, 1995.
35. R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995.
36. V. Zivojnovic, J. 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.