Reconstructing Control Flow from Predicated Assembly Code

Björn Decker[1] and Daniel Kästner[2]

[1] Saarland University,
bdecker@cs.uni-sb.de
[2] AbsInt GmbH,
kaestner@absint.com

Abstract. Predicated instructions are a feature more and more common in contemporary instruction set architectures. Machine instructions are only executed if an individual guard register associated with the instruction evaluates to true. This enhances execution efficiency, but comes at a price: the control flow of a program is not explicit any more. Instead instructions from the same basic block may belong to different execution paths if they are subject to disjoint guard predicates. Postpass tools processing machine code with the purpose of program analyses or optimizations require the control flow graph of the input program to be known. The effectiveness of postpass analyses and optimizations strongly depends on the precision of the control flow reconstruction. If traditional reconstruction techniques are applied for processors with predicated instructions, their precision is seriously deteriorated. In this paper a generic algorithm is presented that can precisely reconstruct control flow from predicated assembly code. The algorithm is incorporated in the PROPAN system that enables high-quality machine-dependent postpass optimizers to be generated from a concise hardware specification. The control flow reconstruction algorithm is machine-independent, and automatically derives the required hardware-specific knowledge from the machine specification. Experimental results obtained for the Philips TriMedia TM1000 processor show that the precision of the reconstructed control flow is significantly higher than with reconstruction algorithms that do not specifically take predicated instructions into account.

1 Introduction

Many of today's microprocessors use instruction-level parallelism to achieve high performance. They typically have multiple execution units and provide multiple issue slots (EPIC, VLIW) or deep pipelining (superscalar architectures). However, since the amount of parallelism inherent in programs tends to be small [1], it is a problem to keep the available execution units busy. For architectures with static instruction-level parallelism this problem is especially virulent, since if not enough parallelism is available the issue slots of the long instruction words are filled with nops. For embedded processors this means a waste of program memory and energy.

Guarded (predicated) execution [2–4] has been implemented in many different microprocessors such as the TriMedia TM1000, the ADSP-2106X SHARC processor, and the Intel IA-64 architecture [5–7]. It provides an additional boolean register to indicate whether the instruction is executed or not. This register is called the *guard* or the *guard register* of the instruction. A guard register having the value `true` forces the processor to execute the corresponding instruction. If the value of the guard is `false` the operation typically is dismissed without having any effect. An example is shown in Fig. 1. The original program consists of three basic blocks; if predicated execution is exploited only one basic block remains. If supported by the target architecture, i_2 and i_4 resp. i_3 and i_5 can be allocated in the same VLIW instruction.

Predicated execution can significantly improve code density since it allows to fill issue slots of the same instruction with micro-operations from different control paths. Moreover it enhances performance since it allows conditional branches to be removed from the program. Conditional branches can degrade the performance since they interrupt the sequential instruction stream. Mispredicted branches can introduce bubbles in the pipeline and may degrade cache performance

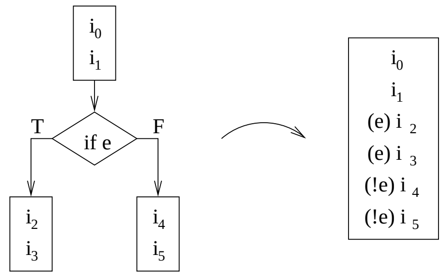

Fig. 1. Guarded Code.

if code sequences are prefetched but have to be discarded again. Thus, predicated execution can enhance performance for architectures with static parallelism (EPIC, VLIW), and for superscalar pipelined architectures.

Embedded processors are used in a variety of application fields: healthcare technology, telecommunication, automotive and avionics, multimedia applications, etc. Common characteristics of many applications is that high computation performance has to be obtained at low cost and low power consumption. The incorporation of application-specific functionality has the additional consequence that the architectural design of these microprocessors often is highly irregular. In the area of the classical general-purpose processors, compiler technology has reached a high level of maturity. However, for irregular architectures, the code quality achieved by traditional high-level language compilers is often far from satisfactory [8, 9]. Generating efficient code for irregular architectures requires highly optimizing techniques that have to be aware of specific hardware features of the target processor.

The PROPAN system [10–13] has been developed as a retargetable framework for high-quality code optimizations and machine-dependent program analyses at assembly level. From a concise hardware specification a machine-sensitive postpass optimizer is generated that especially addresses irregular hardware architectures. The generated optimizer reads assembly programs and performs efficiency-increasing program transformations.

A precondition for the code transformations performed by PROPAN-generated optimizers is that the control flow graph of the input program is known. In the presence of guarded code, whether an instruction is executed or not depends on the contents of the guard register. Code sequences that compute guard values look just like 'normal' computations — with the exception that the end result is stored in a guard register and this influences the control flow of the program. Thus, an important part of control flow reconstruction from guarded code is detecting the operations that determine the control flow. Moreover, in order to recognize that some operations are executed under mutually exclusive conditions, relations between the contents of guard registers have to be computed. Determining relations between register contents requires simulating the effect of operations on the machine state, i. e. evaluating the instruction semantics. In cases where an exact evaluation is not statically possible conservative approximations have to be available. Thus, a symbolic evaluation is required that is generic to ensure retargetability and that is very precise to enable accurate control flow reconstruction. How this can be achieved is described in this paper.

The article is structured as follows: Sec. 2 gives an overview of related work in the area of control flow reconstruction with the focus on predicated code. Sec. 3 addresses the PROPAN framework. The guarded code semantics which is at the base of our work is presented in Sec. 4; Sec. 5 gives an overview of the control flow reconstruction problem and the approach chosen in PROPAN. Our algorithm to compute the control flow graph is detailed in Sec. 6. The experimental results are presented in Sec. 7, and Sec. 8 concludes.

2 Related Work

Reconstructing control flow for predicated code has not been an issue in most previous approaches. The Executable Editing Library EEL reconstructs control flow graphs from binary code to support editing programs without knowledge of the original source code [14]. Based on a simple high-level machine description EEL can be retargeted to new architectures. The reconstructed control flow graphs are reported to be very precise for some machines, e. g. the SPARC architecture. However, [15] reports that the system is not sufficiently generic to deal with complex architectures and compiler techniques. Reconstructing control flow from predicated instructions is not supported.

`exec2crl` [15] uses a bottom-up approach for reconstructing the basic control flow graph which solves some problems specific to control flow reconstruction from executables. The targets of control flow operations are computed precisely for most indirections occurring in typical DSP programs. The reconstructed control flows graphs are used for static analyses of worst case execution times of binary programs. There is no support for reconstructing control flow from predicated instructions.

`asm2c` is an assembly to C translator for the SPARC architecture [16]. The translation requires a CFG which is computed using extended register copy propagation and program slicing techniques. Extended register copy propagation was first used in the `dcc` decompiler [17] which was developed to recover C code from executable files for the Intel 80286. In contrast to `EEL` and `exec2crl`, `asm2c` and `dcc` are not retargetable by specification of a high-level machine-description; the problem of reconstructing control flow from predicated code is not considered. [16] and [17] do not contain any information about the precision of the reconstructed control flow graphs.

An algorithm for reconstructing control flow from guarded (predicated) code, called *reverse if-conversion*, is presented in [18] as a part of a code generation framework. In this framework first a local part of the control flow is if-converted (see Sec. 4) in order to enlarge the scope of the scheduling process. Then the resulting guarded code is scheduled. Subsequently, the reverse if-conversion retranslates the obtained guarded code segment back into a control flow graph which offers precise control flow information to the final analysis and optimization steps. During the if-conversion performed in the early stages of the code generation process all operations which are responsible for control flow joins and forks are marked. The reverse if-condition is depending on those markings to detect operations which directly alter the 'control flow' of the program. Relying on the presence of such markings is contradictory to the retargetability principle of Propan, since this would severely restrict the set of supported assembly languages. Thus, we have to explicitly compute all reconstruction information from the assembly source.

3 The PROPAN Framework

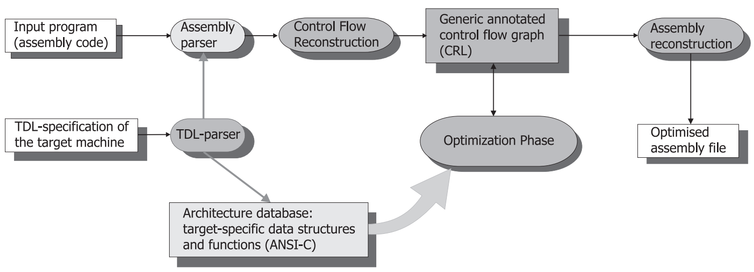

Fig. 2. The Propan System.

The Propan system [10–12] 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. 2. The input of Propan consists of a Tdl-description of the target machine and of the assembly programs that are to be analyzed or optimized. The Tdl specification is processed once for each target architecture; from the Tdl description a parser for the specified assembly language and the 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 initialize, access and manipulate them are defined. The core system is composed of generic and generated program parts. 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 the architecture 'database'. For each target architecture, the generic core system is linked with the generated files yielding a dedicated hardware-sensitive postpass optimizer.

The GECORE-module (GEneric COntrol flow REconstruction) of PROPAN performing the reconstruction of control flow graphs from assembly programs is liable to the same requirements as the PROPAN core system itself. Its core has to be generic while the required target-specific information is retrieved from the architecture database. The first part of the GECORE-module is a generic control flow reconstruction algorithm that reconstructs control flow from assembly programs [13]. Input is a sequence of assembly instructions. Using the architecture description, branch operations are detected and a control flow graph of the input program is determined. In that part guarded execution is not taken into account. The second part is subject of this paper: here an explicit representation of control flow information coded in guard registers is computed.

The optimizations modules of PROPAN are based on integer linear programming 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. By using ILP-based approximations, the calculation time can be drastically reduced while obtaining a solution quality that is superior to conventional graph-based approaches [11, 19]. The optimizations are not restricted to basic block level; instead a novel superblock concept allows to extend the optimization scope across basic block and loop boundaries. The superblock mechanism also allows to combine the ILP-based high-quality optimizations 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 [12].

4 Guarded Code Semantics

If-conversion [20, 2, 3, 21, 4] is a compiler algorithm that removes conditional branches from programs by converting programs with conditional branches into *guarded code*. Guarded code contains less branches since the conditions under which an instruction is executed are represented by its guard register. Thus, if-conversion transforms *explicit control flow* via branch and jump operations into *implicit control flow* based on the information of the guard registers.

Given a previously if-converted piece of code, the implicit control flow has to be reconstructed from the guarded code before other analyses or optimizations are performed. Otherwise the implicit control flow information would be lost and the precision of the control flow graph would be degraded which could severely reduce the effectiveness of postpass analyses and optimization techniques. As an example consider the following two predicated instructions:

```
(r3) r5 = load (r9)
(!r3) r7 = r8 + r5
```

If the information that the instructions are guarded by disjoint control flow predicates (`r3` and `!r3`) was not available to a data dependency analysis, a data dependency between both instructions would be reported. This would prevent any reordering or parallelization of both instructions although this would be perfectly feasible.

Our approach for control flow reconstruction is based on the static semantics inference mechanism of [4], which will be summarized in the remainder of this section. The semantics of a guard is a logical formula consisting of branch conditions represented by predicate variables. In these formulas the operators $\wedge$, $\vee$ and $\neg$ are allowed. There also exist constants, such as `true` and `false`. An operation is executed if and only if its guard's semantics is `true`.

A piece of guarded code is a sequence of guarded operations `<guard>? <operation>`. A statement $C \vdash S$ denotes that, from a piece of guarded code C, a set S of semantics of all occurring guards is deducible. For the reduction of the guard semantics three inference rules exist: *taut*, *fork*, *join*. This analysis requires all guard registers to be initialized to `false` and to be assigned at most once on each control flow path.

$$[\text{taut}] \ \vdash \{g_0 = \texttt{true}\}$$

$$[\text{fork}] \ \frac{C \vdash S \cup \{g_1 = l_1\}}{C;\ g_1 \,?\, g_2 := l_2 \vdash S \cup \{g_1 = l_1\} \cup \{g_2 = (l_1 \wedge l_2)\}}$$

$$[\text{join}] \ \frac{C \vdash S \cup \{g_1 = l_1\} \cup \{g_2 = l_2\}}{C;\ g_1 \,?\, g_2 := l_3 \vdash S \cup \{g_1 = l_1\} \cup \{g_2 = ((l_1 \wedge l_3) \vee l_2)\}}$$

The first rule (*taut*) specifies that g_0 always evaluates to `true`; it is used, e. g., as guard of the entry block.

For forks of the control flow a second inference rule (called *fork*) is introduced. From a given code segment $C;\ g_1 \,?\, g_2 := l_2$ it can be deduced that $S \cup \{g_1 = l_1\} \cup \{g_2 = (l_1 \wedge l_2)\}$ holds if the statement $C \vdash S \cup \{g_1 = l_1\}$ is deducible. Let $S \cup \{g_1 = l_1\}$ be the semantical information of the guard registers obtained by analyzing the operation sequence C. Then, for the sequence $C;\ g_1 \,?\, g_2 := l_2$ the set of guard semantics $S \cup \{g_1 = l_1\} \cup \{g_2 = (l_1 \wedge l_2)\}$ can be derived. Intuitively formulated, if the semantical information derived for C contains a binding of g_1 to l_1, then from the guarded statement $g_1 ? g_2 := l_2$ the additional information that g_2 is bound to $l_1 \wedge l_2$ can be deduced. Since the assignment of l_2 to g_2 is only executed if l_1 is `true`, the effective condition associated with g_2 is $l_1 \wedge l_2$.

The third rule (called *join*) is applied at joins of control flow. In contrast to the *fork* rule the semantical value of g_2, l_2, is already known. l_2 represents all values of g_2 reaching the current instruction on control flow paths $\pi_0, \ldots, \pi_x$. The semantical value of g_2 on path π_{x+1} (containing the operation $g_1 \,?\, g_2 := l_3$) is $l_1 \wedge l_3$. The semantical value of g_2 after the current instruction is the disjunction of its semantical values reaching the instruction $g_1 \,?\, g_2 := l_3$ on paths $\pi_0, \ldots, \pi_x$ or π_{x+1}.

5 Control Flow Reconstruction

The control flow reconstruction module GECORE of the PROPAN system works in two phases. In the first phase control flow reconstruction is done without taking predicated instructions into account. The input of this phase is a generic representation of the assembly instructions of the input program which is provided by the assembly parser generated from the TDL description. An extended program slicing algorithm is used that can deal with unstructured control flow instructions typical for assembly programs. The data structure used for representing control flow is the interprocedural control flow graph (ICFG) [22] which completely represents the control flow of programs. It consists of two components:

1. The *call graph* (CG) describes relationships between procedures of the program. Its nodes represent procedures, its edges represent procedure calls.
2. The *basic block graph* (BBG) describes the intraprocedural control flow of each procedure. Its nodes are the *basic blocks* of the program. A *basic block* is a sequence of instructions that are executed under the same control conditions, i. e., if the first instruction of the block is executed, the others are executed as well. The edges of the BBG represent jumps and fall-through edges[3].

Details about this phase can be found in [13].

After the explicit control flow has been reconstructed in the first phase, the second phase deals with the implicit control flow represented by instruction predicates. In the ideal case the reconstructed ICFG represents the control flow precisely. Whenever this is not possible, a safe

[3] Fall-through edges point to successors that are reached by sequential execution of the instructions instead of following a branch.

approximation has to be computed. Another important requirement is that the reconstruction algorithms are generic, i. e. that they can be used for any target architecture without modification. All information about the architecture should be retrieved from the TDL description. From these requirements, several problems arise that have to be addressed when recovering implicit control flow information from guarded code:

1. Each operation possibly affects control flow.
2. The contents of registers cannot always be statically determined at every instruction. Thus, a symbolic representation of register contents is necessary. In this representation, also the semantical relations to other registers have to be established.
3. In the presence of frequent memory accesses statically determining register contents becomes even more difficult. Enabling the reconstruction algorithm to identify the contents of memory cells requires a precise memory analysis to be incorporated. Since a precise control flow graph is not yet available during the reconstruction process, dedicated analysis approaches are required.

6 The Reconstruction Algorithm

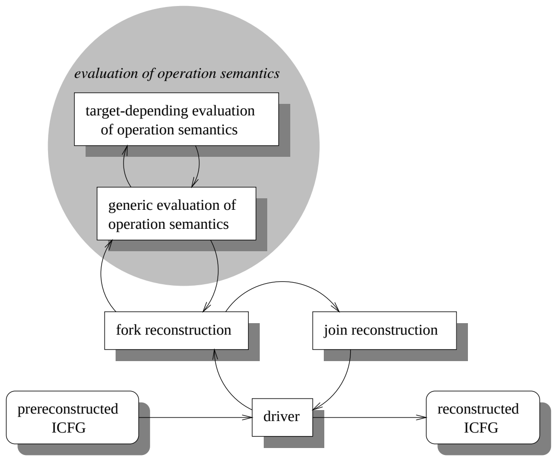

Fig. 3. Recovering Implicit Control Flow.

Recovering control flow from guarded code is performed by refining the pre-reconstructed CFG (see Fig. 3). The reconstruction algorithm is applied to each basic block of the pre-reconstructed ICFG. It incorporates two subtasks:

1. For each basic block an equivalent micro-block structure is built which represents implicit forks in the control flow (see Sec. 6.3). During the reconstruction of forks the semantics of assembly operations has to be evaluated (see Sec. 6.2) as part of the value analysis performed.
2. In the second subtask, the micro-block structure is refined to represent control flow joins (see Sec. 6.4); the result is the refined basic block graph where implicit control flow has been made explicit.

Finally, the input basic block is replaced by the computed basic block graph.

6.1 Definitions

An *instruction* is defined as a set of microoperations whose execution is started simultaneously. This definition is mainly used in the context of VLIW architectures. However, a processor not

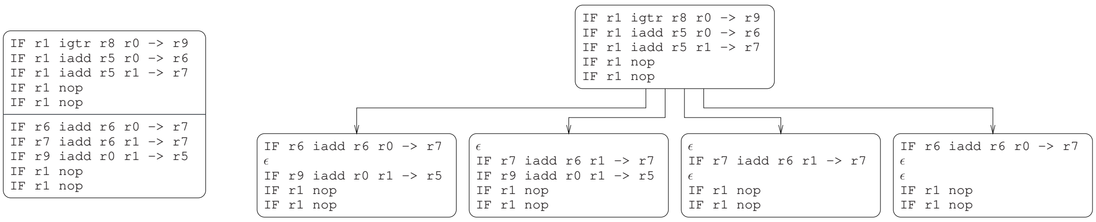

Fig. 4. A procedure and its instruction occurrence graph

exhibiting instruction-level parallelism can be seen as a special case of a VLIW architecture with each instruction containing only one microoperation.

We will conceptually distinguish between a microoperation (in the following called operation) and the instantiation of a microoperation in the input program. We will use *operation* to denote the operation type provided by the processor, and use the term *operation instance* to refer to an occurrence of a microoperation with concrete arguments in the input program. To give an example, an operation instance of operation `add` could be `add r1,r2,r3`. The same terminology is canonically applied to instructions.

While reconstructing control flow from guarded code, it can become necessary to duplicate operations or replace them by `nop` if a basic block is decomposed into different control flow paths. For this purpose we use the notion of *operation variation* resp. *instruction variation*. Let o be an operation and $\tilde{o}$ an instance of o. Then a *variation* $\hat{o}$ of $\tilde{o}$ is an instance of o that has exactly the same operands as $\tilde{o}$ or is the empty operation ϵ. The empty operation ϵ is equivalent to an unconditionally executed `nop`. For a processor with k instruction slots, a variation $\hat{i}$ of an instruction i is represented by a $k+1$-tuple $(a, \hat{o}_1, \ldots, \hat{o}_k)$ where $\hat{o}_i$ are variations of operations contained in the instruction instance with address a.

An *execution sequence* π of a procedure is a possible sequence of instruction variations containing only the operations that are executed at run-time, i. e. for which the guard register evaluates to `true`. The occurrence of the variation $\hat{o}$ of an operation o in the execution sequence π is called an *operation occurrence* of the operation o.

The example in Fig.4 shows a block consisting of two TriMedia TM1000 instructions on the left. Paths through the graph on the right are exactly the feasible execution paths through the block on the left. One instruction, shown as a box, consists of five microoperations that are executed simultaneously. The nodes of the graph on the right hand side are feasible instruction variations of the two instructions on the left. Edges represent their ordering. A guard is interpreted as `true` if the least significant bit is set. Each execution path can contain instructions guarded either by $r6$ or by $r7$, but not both. In the second and third operation of the first instruction they are set to values that cannot be `true` at the same time. The contents of $r5$ is unknown, but adding it to $r0$ (hardwired to `0x0`) always results in a different truth-value (least significant bit) than adding it to $r1$ (hardwired to `0x1`). Thus, in the second instruction operations guarded by $r6$ and those guarded by $r7$ are never executed at the same time. Therefore, feasible instruction variations of the second instruction contain operations that are guarded by either $r6$ (first and fourth instruction variation) or $r7$ (second and third instruction variation). Without information about the contents of $r8$ we are not able to exactly evaluate the greater-than-comparison in the first operation of the first instruction. Therefore, we assume $r9$ to evaluate to either `true` (first and second instruction variation of the second instruction) or `false` (third and fourth instruction variation).

Since during static analyses register contents are not necessarily known at every point of execution, symbolic values have to be introduced. The set of *concrete values* V contains natural numbers, strings and floating point values; *symbolic values* are contained in V' (see Eq. 6.1). Additionally, we have to keep track of the development of register contents over time. Therefore, we introduce the term *register instance* to denote the value of some register at a given point in time. A register instance is a register tagged with a timestamp of a point in time when a value is assigned to the register. We allow register instances to be written only once. Let $\mathcal{R}_I$ be the set of register instances defined in the TDL specification. Then, the set of symbolic values V' is defined as follows:

$$V' = \left\{ \begin{array}{c} \top, \mathtt{true}, \mathtt{false}, \mathit{ref}(r), \mathit{not}(v_x), \\ \mathit{and}(v_x, v_y), \mathit{or}(v_x, v_y) \end{array} \middle| \begin{array}{c} r \in \mathcal{R}_I, \\ v_x, v_y \in V' \cup V \end{array} \right\}$$

While evaluating operation semantics it is not guaranteed that each condition of an if-statement can be properly evaluated. These if-conditions can consist of comparisons or logical computations. However, we require all if-conditions ($\mathcal{C}_I$) to be interpreted either as `true` or `false`. Therefore, whenever an if-condition is reached that cannot be evaluated it is necessary to make assumptions on the truth-value of the condition. In order to face this problem the concept of *meta-environments* is introduced.

Definition 1 (Environment) Let $\mathcal{R}_I$ denote the set of instances of all registers specified in the TDL-description of the target processor and let $\mathcal{C}_I$ be the set of if-condition instances. Furthermore let V be the set of concrete values and V' the set of symbolic values. A *symbolic environment* $\sigma_{V \cup V'}$ is a triple $(\mathit{map}, \mathit{act}, \mathit{force})$. The function $\mathit{map} : \mathcal{R}_I \cup \mathcal{C}_I \to V \cup V'$ maps register instances and if-condition instances to (concrete or symbolic) values. The function $\mathit{act} : \mathcal{R} \to \mathcal{R}_I$ maps a generic register to its active instance. The function *force* is used to force a register to evaluate to a certain truth-value.

A *meta-environment* is a set of environments; in each environment every occurring condition can be evaluated to `true` resp. `false` during semantics evaluation. For each combination of occurring conditions, a dedicated environment has to be contained in a meta-environment.

During the reconstruction of control flow from guarded code, for each basic block in the input ICFG increasingly refined versions of the *micro-block graph* are computed which explicitly represents the implicit control flow of the basic block. Before defining the micro-block graph some additional definitions have to be given.

Definition 2 (Instruction Occurrence Graph) Let a basic block B of the control flow graph of a procedure p be given. The *instruction occurrence graph* of B is a minimal directed graph $G_I = (N_I, E_I, N_A, N_\Omega)$ with node labels. For each instruction occurrence i' of each instruction i in B which belongs to an execution sequence of p, there is a node $n_{i'} \in N_I$ that is marked by i'. Edges (n', m') exist in E_I if and only if n' and m' are subsequent instruction occurrences of the same execution sequence. N_A is the set of occurrences of the entry instruction of B and N_Ω is the set of occurrences of the exit instruction of B.

Definition 3 (Micro-Block) A *micro-block* of an instruction occurrence graph is a path of maximal length which has no joins except possibly at the beginning and no forks except possibly at the end.

Definition 4 (Micro-Block Graph) The micro-block graph $G_M = (N_M, E_M, m_A, m_\Omega)$ of an instruction occurrence graph $G_I = (N_I, E_I, N_A, N_\Omega)$ is formed from G_I by combining each micro-block into a node. Edges of G_I leading to the first node of a micro-block lead to the node of that micro-block in G_M. Edges of G_I leaving the last node of a micro-block, lead out of the node of that micro-block in G_M. m_A denotes the (possibly empty) entry micro-block that has an edge to each micro-block containing an entry node. b_Ω denotes the set of micro-blocks containing the exit nodes.

During the process of building the micro-block graph, all executions of the basic block are simulated such that all feasible execution paths are covered. Let π be the path in the partially reconstructed micro-block graph from the entry node to the leaf micro-block b. The meta-environment of b in the partially reconstructed micro-block graph represents the contents of registers after the execution of all instruction variations on the path in the micro-block graph from the entry node to the leaf node b.

Within the scope of the reconstruction of guarded code, a *safe approximation* of the micro-block graph is the micro-block graph of a safe approximation of the instruction occurrence graph.

An approximation of the instruction occurrence graph IOG^0 is *safe* if it contains at least all paths of the IOG^0.

Definition 5 (Fitting Instruction) Let i be an instruction and $\Sigma_{V \cup V'}$ be a meta-environment. i_F is the *fitting instruction* of i and $\Sigma_{V \cup V'}$ if for all operations o_F contained in i_F and the corresponding operations o of i holds that

- $o_F = \epsilon \iff$ the guard register of o evaluates to `false` within all environments of $\Sigma_{V \cup V'}$ or
- $o_F = o \iff$ the guard register of o evaluates to `true` within all environments of $\Sigma_{V \cup V'}$.

A *fitting operation* is a single operation for which one of the conditions above holds.

The existence of a fitting instruction is not guaranteed. The guard register of an operation could evaluate to `true` as well as `false` within the meta-environment of a micro-block. Assume the meta-environment $\{\{r3 \mapsto \top\}\}$. A fitting operation does not exist for the following operation because the guard register, $r3$, cannot be uniformly evaluated to `true` or `false`: `IF r3  add r0 r1 -> r4`. For `IF r1  add r0 r1 -> r4`, the fitting operation is the operation itself since $r1$ evaluates to `true`.

6.2 Instruction Semantics Evaluation

The operation semantics is defined in the instruction set section of the TDL specification. TDL provides its own *register transfer language RTL*, which is statement-oriented in order to generate cycle-accurate instruction-set simulators requiring a precise specification of what happens in which cycle. It is described in detail in [23, 12]; a formal approach defining the operation semantics using derivation rules is presented in [24].

The symbolic evaluation of instruction semantics must be aware of the definition of truth values by the processor modeled. For instance, the TriMedia TM1000 interpretes register contents as `true` or `false` depending on the least significant bit. In the ADSP-2106x SHARC on the other hand, a register evaluating to `false` must have all bits set to 0. For different interpretations slightly different derivation rules have to be defined. Within the scope of this paper we model memory locations as unknown values since no value analysis for memory cells and no alias analysis is performed. Incorporating these analysis in the control flow reconstruction process is a goal of future work.

Our approach is based on an extended constant propagation analysis supporting symbolic values. The relevant program state comprises the contents of all registers and is represented by meta-environments. During the reconstruction of implicit control flow for a basic block increasingly refined versions of the micro-block graph are computed. The micro-block graph is built bottom-up. Whenever the analysis determines that an instruction occurrence has to be arranged within a specific micro-block the meta-environment of that block is updated by evaluating the instruction semantics. This simulates multiple executions: the instruction occurrence is "executed" within each binding of registers represented by the meta-environment. In order to properly evaluate if- and while-statements the corresponding condition is always required to evaluate to `true` or `false`. To ensure this, appropriate environments are added to the current meta-environment. In order reduce the number of environments in a meta-environment, environments which are indistinguishable with respect to the truth value of all registers are replaced by a single representative. Detailed information about the symbolic evaluation can be found in [24].

6.3 Fork Reconstruction

While building up the micro-block graph, its leaf blocks are called *visible*. The *env* function is used to retrieve the meta-environment from a micro-block; the *instr* function is used to access the set of instructions of a micro-block.

Starting point of the reconstruction is a basic block of the precomputed CFG and a micro-block graph containing only one empty micro-block. The empty micro-block contains no instructions, has no successors and is associated with an environment that maps all registers to $\top$, i. e. that does not force any register to evaluate to a special value. First, we successively arrange the instructions of the input block into the visible blocks of the micro-block graph. For this purpose we compute the fitting instruction for each instruction in every visible block.

The fitting operation of an operation and a meta-environment is computed as follows: In the case the operation is not guarded or the guard register evaluates to `true`, the fitting operation is o. If the operation is guarded but the guard register evaluates to `false`, the operation cannot change the environment; the fitting operation is ϵ. The result is undefined if it cannot be uniformly determined whether the guard register evaluates to `true` or `false` within the meta-environment Σ'.

If a fitting instruction exists we add it to this block and update the meta-environments using semantics evaluation. In case the fitting instruction does not exist for a certain block we introduce two empty successor blocks with the same meta-environment as the block. In one block the guard register preventing the existence of the fitting instruction is forced to evaluate to `true`, in the other to `false`. Then, these blocks are considered for arranging the instruction instead of their parent block. Once all visible blocks are processed for an instruction, the subsequent instruction is arranged. Using this technique we separate different control flow paths from each other.

An example of an input block containing TriMedia TM1000 assembly instructions is given in Fig.5. Instructions are referred to as i_0, i_1 and i_2. The micro-block graph obtained from recon-

```
i0  IF r1 igtr r8 r0 -> r9
    IF r1 ileq r8 r0 -> r6
    IF r1 iadd r0 r1 -> r7
    IF r1 nop
    IF r1 nop
i1  IF r6 iadd r7 r0 -> r8
    IF r9 iadd r7 r0 -> r8
    IF r1 nop
    IF r1 nop
    IF r1 nop
i2  IF r8 iadd r0 r1 -> r5
    IF r1 nop
    IF r1 nop
    IF r1 nop
    IF r1 nop
```

Fig. 5. An input block containing TriMedia TM1000 assembly instructions

structing forks of the input block given in Fig.5 is illustrated in Fig.6.

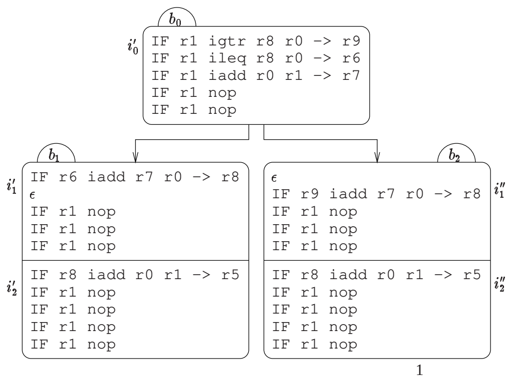

Fig. 6. Micro-block graph after reconstructing forks of the block in Fig.5

We refer to the instruction variations in Fig. 5 by i'_0 for the instruction variation of i_0 and i'_1, i'_2 for the instruction variations of i_1 and i_2 in block b_1. Block b_2 contains instruction variations i''_1 and i''_2. Instruction i_0 can be arranged without problems in block b_0 resulting in i'_0 because all operations are guarded by $r1$ which is hardwired to `0x1`. Since the contents of $r8$ is unknown ($\top$) at the beginning of the analysis, we cannot compute the exact value of $r9$. We split the environment into a meta-environment containing two environments: one where $r9$ is `true` and one where it is `false`. Within the environments of that meta-environment we are able to evaluate the less-or-equal-comparison of the second operation: $r6$ is `true` in the environment where $r9$ evaluates to `false` and it evaluates to `false` for the environment containing $r9$ to be `true`.

In both environments of the meta-environment $r7$ is set to `0x1`. Next we try to arrange i_1 into b_0, but within b_0 no fitting instruction for i_1 exists. In the meta-environment associated with b_0, $r9$ can possibly evaluate to `true` as well as to `false`. Thus, successor blocks (b_1 and b_2) are introduced. b_1 is associated with the meta-environment containing only those environments with $r9$ evaluating to `true`. The meta-environment of b_2 contains only environments where $r9$ is `false`. This implies that $r6$ is `false` in b_1 resp. `true` in b_2. In both blocks (b_1, b_2) a fitting instruction for i_1 exists: i_1' in b_1, i_1'' in b_2. Since within b_1 $r9$ evaluates to `true` and $r6$ to `false` i_1' contains the operation guarded by $r9$ but the operation guarded by $r6$ is replaced by ϵ; i_1'' is handled likewise. These instruction variations set $r8$ to $r7$ which is `0x1` in the meta-environments of both blocks. Therefore, instruction variations i_2' and i_2'' both contain the operation guarded by $r8$.

6.4 Join Reconstruction

From the first phase of the reconstruction, an approximated micro-block graph in form of a tree is obtained. This graph explicitly represents all forks in the control flow of an input block. It is a safe approximation of the micro-block graph (a proof is given in [24]). Different control flow paths are separated from each other but joins of control flow are not represented yet. Presuming no additional paths are introduced, reconstructing joins is equivalent to computing a smaller solution of the approximated micro-block graph, i. e. the resulting micro-block graph is more precise.

We recognize joins of control flow by identifying equal instruction occurrence sequences at the end of paths through the micro-block graph. Assume two equal subpaths starting with instruction i. Then, these paths can be combined into one single subpath starting with i and representing a join of control flow.

Join detection is initiated at the lowest address of instructions in the input basic block. We look for pairs of instruction occurrences at address a that are roots of equivalent subgraphs in the micro-block graph. In the case such a pair is found at address a the join is reconstructed by modifying the micro-block graph in such a way that the common subgraphs are shared. If no pair of equivalent instruction occurrences can be found at address a the subsequent address $a + 1$ is inspected.

Considering two instruction occurrences as equivalent requires introducing the notion of *similar* operations. Operations are similar either if they are equal or if one of them is ϵ and the other is `nop`. Similar operations have the same effect on environments. Then, two single instructions can be denoted equivalent if they are similar themselves, i. e. all operations contained are similar, and for each immediate successor instruction of the first instruction an equivalent successor instruction of the second can be found and vice versa.

The micro-block graph for the input block in Fig.5 after reconstructing also joins of the control flow is shown in Fig.6. For reconstructing joins we successively inspect instruction occurrences at the same address of the micro-block graph obtained from fork reconstruction. For i_0' there is no other instruction occurrence to compare with. Instructions i_1' and i_1'' are not equivalent since they themselves are not similar. Instruction occurrences i_2' and i_2'' (in Fig.6) can be combined into the single instruction occurrence at the bottom of Fig.7. They are similar and since they do not have successors they are considered as equivalent.

7 Experimental Results

The algorithm for recovering control flow from guarded code has been evaluated using the TriMedia TM1000 [5]. The TriMedia TM1000 is a multimedia VLIW processor providing several hardware

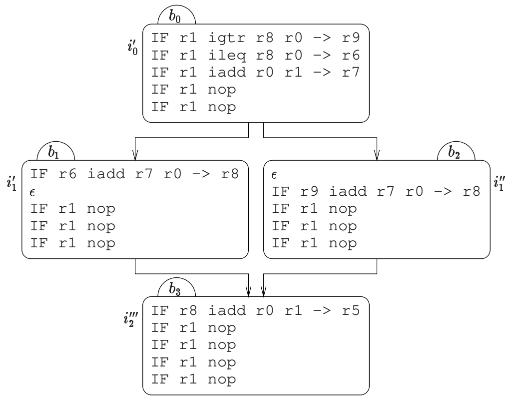

Fig. 7. Micro-block graph after reconstructing joins of the block in Fig.5

characteristics that make control flow reconstruction difficult: it exhibits significant instruction level parallelism, implements procedure calls and returns by jump instructions and uses predicated execution for all machine operations.

Our input programs comprise the DSPSTONE benchmark [8] and some additional typical digital signal programming applications. The experiments have been executed on an AMD Athlon 1400 processor with 512 MByte RAM running Linux; the assembly files have been generated by the Philips `tmcc` compiler [5] at highest optimization level.

Fig. 8 shows the statistics of the control flow reconstruction. Column $\#I$ gives the number of assembly instructions for each input program. Columns $\#B_{ex}$ shows the number of blocks after the reconstruction of explicit control flow [13]; column $\#B_{em}$ shows the number of blocks after reconstructing the implicit control flow. Before implicit control flow reconstruction there is only one path through every block. Hence, the number of paths through the blocks is equal to the number of blocks. During the guard-sensitive reconstruction these blocks are split and additional paths are introduced (see Fig.4); the number of these additional paths through the reconstructed blocks are shown in column $\#P$. Edges representing explicit control flow are not taken into account in the figures of column $\#P$. Column $\#P/\#B_{ex}$ shows the number of intra-block paths after reconstructing implicit control flow divided by the number of intra-block paths before implicit reconstruction. The execution time of the reconstruction in milliseconds is presented in column t. It shows the time for reconstructing the implicit control flow only and does not include the time needed for building the initial CFG.

The numbers of paths shown in the fifth column give a hint on how much precision is gained by reconstructing implicit control flow from guarded code. To give an example, for `whet` the control flow graph obtained by reconstructing explicit control flow contains 46 blocks. After reconstructing implicit control flow from predicated instructions the control flow graph contains 62 additional blocks. While after reconstructing explicit control flow exactly one path is visible through each basic block, the reconstruction of implicit control flow makes 22 additional "intra-block" paths visible. The number of "intra-block" paths is lower than the number of blocks after recovering implicit control flow because in situations where joins are reconstructed (see Fig.7) more additional blocks are introduced than additional intra-block paths become visible. For example the micro-block graph in Fig.7 contains 3 additional blocks but only 1 additional intra-block path compared to its input block. An illustration of the number of intra-block paths before and after the reconstruction of implicit control flow for each input program is given in Fig. 9.

Since our approach works at basic block level, we have to assume that at the entry of each basic block register contents are unknown. Values read from memory also have to be considered as unknown. Thus, we may overestimate the number of possible control flow paths. However, this overestimation does not reduce the enhanced precision of analyses and optimizations gained by making implicit control flow paths explicitly visible. Nevertheless, if the control flow graph contains infeasible control flow paths the computation time of algorithms working with the control flow

file name	$\#I$	$\#B_{ex}$	$\#B_{em}$	$\#P$	$\#P/\#B_{ex}$	t [msec]
`biquad_N_sections`	56	6	12	8	1.33	27
`biquad_one_section`	28	7	7	7	1	12
`c_fir`	250	54	144	87	1.61	599
`c_firfxd`	168	23	50	32	1.39	249
`c_vecsum`	681	120	309	184	1.53	2,069
`complex_multiply`	10	4	4	4	1	2
`complex_update`	8	4	4	4	1	2
`convolution`	49	7	16	10	1.43	34
`dot_product`	25	7	7	7	1	11
`fft`	436	52	108	71	1.37	507
`fir`	56	11	23	15	1.36	41
`fir2dim`	193	15	50	27	1.8	830
`iir1`	27	3	6	4	1.33	26
`iir2`	27	9	15	11	1.22	26
`lms`	65	10	28	16	1.6	51
`mat1x3`	40	6	9	7	1.17	25
`matrix1`	202	42	226	134	3.19	781
`matrix2`	238	45	196	116	2.58	452
`n_complex_updates`	57	14	29	19	1.36	37
`n_real_updates`	70	12	27	17	1.42	44
`puzzle`	392	82	396	226	2.76	1,035
`real_update`	24	7	7	7	1	9
`vec_mpy1`	58	3	6	4	1.33	86
`vec_mpy2`	26	8	11	9	1.13	19
`whet`	648	46	108	68	1.48	1,618

Fig. 8. Statistics of Control flow Reconstruction.

graph may increase, and their scope may be reduced. The control flow graph resulting after the reconstruction of implicit control flow provides a safe basis for global value analyses and memory analysis. Such analyses can be used to remove infeasible control flow paths from the reconstructed control flow graph; incorporating them into our framework is subject of future work.

8 Conclusion

We have presented a generic algorithm that can precisely reconstruct control flow from predicated assembly code. The algorithm has been implemented as a part of the Gecore module of the Propan framework. The control flow reconstruction algorithm is machine-independent, and automatically derives the required hardware-specific knowledge, e. g., the semantics of machine instructions, from the machine specification. Thus, in order to retarget the analysis to another processor, only developing a Tdl description is necessary.

The reconstruction algorithm consists of two phases. In the first stage a micro-block graph is built for each basis block which explicitly represents implicit forks in the control flow. Instructions from the same micro-block are always executed unconditionally since the guard register of each contained instruction definitely evaluates to `true` when the control flow reaches it. In the second stage the micro-block graph is refined by detecting control flow flow joins. In the end a refined control block graph is obtained where implicit control flow has been made explicit. The algorithm is based on a symbolic evaluation of instruction semantics which is aware of the definition of truth values by the processor modeled.

Practical experiments demonstrate the applicability of the reconstruction algorithm for typical applications of digital signal processing. For all input programs investigated reconstructing the implicit control flow is completed within a few seconds. The implicit control flow is completely transformed into explicit control flow. The experimental analysis shows that the precision of the

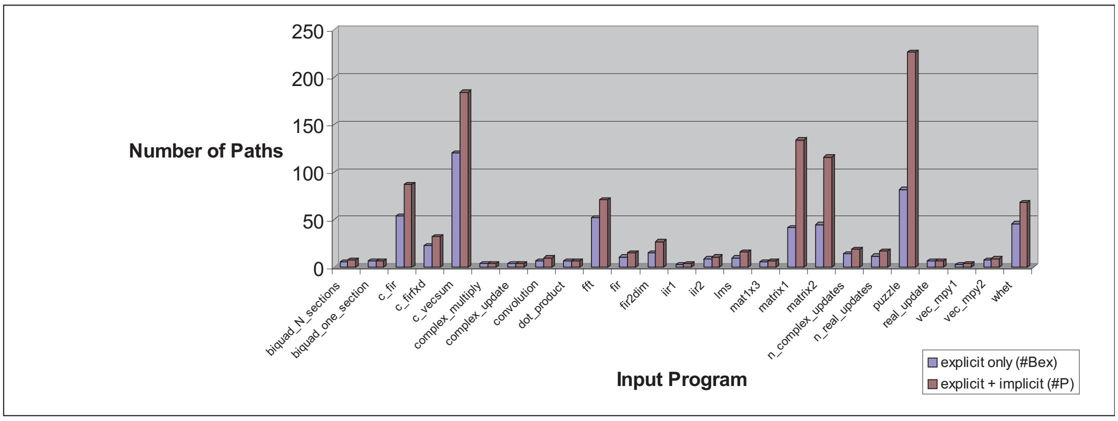

Fig. 9. Paths within each input program's basic block before and after implicit reconstruction.

reconstructed control flow is significantly higher than with reconstruction algorithms that do not specifically take predicated instructions into account.

Due to conservative assumptions concerning register contents at basic block entries and values read from memory the algorithm may overestimate the number of possible control flow paths. This overestimation does not reduce the enhanced precision of analyses and optimizations working with the reconstructed control flow graph that has been gained by making implicit control flow paths explicitly visible. Nevertheless, if the control flow graph contains spurious control flow paths the computation time of algorithms working with the control flow graph may increase, and their scope may be reduced. The control flow graph resulting after the reconstruction of implicit control flow provides a safe basis for global value analyses and memory alias analyses. Incorporating suitable analyses to remove spurious control flow paths into the GECORE module is subject of our future work. Another goal is to apply the reconstruction to other processors featuring predicated execution like the Intel IA64-architecture [7].

References

1. B. Rau and J. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective," *The Journal of Supercomputing*, vol. 7, pp. 9–50, 1993.
2. J. Park and M. Schlansker, "On Predicated Execution," Tech. Rep. HPL-91-58, Hewlett-Packard Laboratories, Palo Alto CA, May 1991.
3. J. Dehnert and R. Towle, "Compiling for the Cydra 5," *The Journal of Supercomputing*, vol. 1/2, pp. 181–228, May 1993.
4. P. Hu, "Static Analysis for Guarded Code," in *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pp. 44–56, 2000.
5. Philips Electronics North America Corporation, *TriMedia TM1000 Preliminary Data Book*, 1997.
6. Analog Devices, *ADSP-2106x SHARC User's Manual*, 1995.
7. Intel, *IA-64 Architecture Software Developer's Manual, Volume 1: IA-64 Application Architecture, Revision 1.1*, July 2000.
8. 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.
9. R. Leupers, *Retargetable Code Generation for Digital Signal Processors.* Kluwer Academic Publishers, 1997.
10. D. Kästner and M. Langenbach, "Code Optimization by Integer Linear Programming," in *Proceedings of the 8th International Conference on Compiler Construction CC99* (S. Jähnichen, ed.), pp. 122–136, Springer LNCS 1575, Mar. 1999.
11. 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.

12. D. Kästner, *Retargetable Code Optimisation by Integer Linear Programming.* PhD thesis, Saarland University, 2000.
13. 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.
14. J. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing," in *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 291–300, 1995.
15. H. Theiling, "Extracting Safe and Precise Control Flow from Binaries," in *7h International Conference on Real-Time Computing Systems and Applications*, July 2000.
16. C. Cifuentes, D. Simon, and A. Fraboulet, "Assembly to High-Level Language Translation," pp. 228–237, Aug. 1998.
17. C. Cifuentes, "Interprocedural Data Flow Decompilation," Tech. Rep. 4(2), June 1996.
18. N. J. Warter, S. A. Mahlke, W.-M. W. Hwu, and B. R. Rau, "Reverse If-Conversion," *ACM SIGPLAN Notices*, vol. 28, no. 6, pp. 290–299, 1993.
19. D. Kästner, "ILP-based Approximations for Retargetable Code Optimization," *Proceedings of the 5th International Conference on Optimization: Techniques and Applications (ICOTA 2001)*, 2001.
20. J. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Conference record of the 10th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 177–189, 1983.
21. J. Hoogerbrugge and L. Augusteijn, "Instruction Scheduling for TriMedia," 1999.
22. F. Martin, *Generation of Program Analyzers.* PhD thesis, Saarland University, 1999.
23. D. Kästner, "TDL: A Hardware and Assembly Description Language," Tech. Rep. TDL1.4, Transferbereich 14, Saarland University, 2000.
24. B. Decker, "Generic Reconstruction of Control Flow for Guarded Code from Assembly," Master's thesis, Saarland University, 2002.