Validation of Translation to Optimized Machine Code

Thesis for the degree of Ph.D.

by

Raya Leviathan

Under the Supervision of
Prof. Amir Pnueli

Presented to the Scientific Council of
the Weizmann Institute of Science

August 25, 2004

Acknowledgment

I would like to express my deep gratitude to my advisor, Prof. Amir Pnueli, for sharing with me his wisdom, knowledge, experience and confidence. His commendable insights and invaluable guidance inspired me all along my work.

Many thanks to Henny Sipma for her assistance in using the STeP tool, for providing me with relevant material and, for willingly adding features to suit my needs.

Special thanks to Orna Lichtenstein, for taking the time and effort to carefully read this thesis and related work. Her advised comments assisted me to improve my work.

Also, I would like to thank all the members of our formal verification group at the Weizmann Institute, for both the scientific atmosphere and the pleasant environment they have created.

Special thanks are relayed to my parents, Ester and Pinchas, who showed me the beauty of learning. My thanks are also conveyed to my children Yaniv, Thalma, Yarden and Roni for walking silently around the house, so as not to disturb me working. Their understanding and approval of the scarce quality time I could spend with them during my work, considerably contributed to the peace of mind I needed. In particular, I would like to thank Yaniv for his knowledgeable advice and help with regard to software issues, as well as for his enlightening comments to my work.

Finally, I would like to thank my husband, Nissan, for helping me at home with the kids and the dishes, for his encouragement during difficult times, as well as for his innumerable comments on my thesis.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

There is a growing awareness of the importance of formally proving the correctness of safety critical portions of software systems. However, proving the correctness of the implementation, written in a high level language, is not sufficient. It should also be verified that whatever correctness has been achieved on the higher level, is not impaired in the translation done by the compiler. This is the reason that standards and regulations require to qualify a compiler, to be used in a safety critical system.

To ensure mathematical correctness of a compiled code, several methods are possible:

- Verifying the compiler program, using the formal semantics of the source program and the target processor machine code, as well as the knowledge about the compiler internal structure and algorithms.
- Enhancing the compiler to produce assertions, whose validity is an evidence to the correctness of the compiler, or of a specific translation.
- Validating each translation as it is generated, by comparing the source program to its translated version and, formally verifying their equivalence.

Formally verifying a full-fledged compiler is a huge engineering effort due to its size, evolution over time and possibly, proprietary considerations. It

also requires a freeze of the compiler development process. In many cases, the qualified compiler is inferior to its unqualified updated version, with regard to the performance of the translation, as well as to the number of errors, since qualification does not guarantee the mathematical correctness of the tool. As far as modifications to the compiler are concerned, they may be implemented on an open source compiler. However, should the compiler be a commercial one, such modifications are rendered impossible.

In view of the above considerations we choose the **translation validation** [PSS98c] approach, which allows an alternative to the verification of translators in general and, of compilers in particular. It offers the validation of each specific translation, whenever the compiler is invoked. Another advantage of this approach is derived from the fact that most industrial compilers have known bugs, yet it is rather difficult to identify the code in the application source program, that exposes such a bug. The translation validation approach allows the use of such a compiler in the development of a safety critical system, by ensuring that any translation error is discovered.

Translation validation should handle four kinds of differences between the source program and its translation:

- **Syntactic differences.**
- **Semantic differences** (e.g. synchronous languages versus sequential languages).
- **Program transformations** (code optimizations).
- **Variables names and representation differences**.

When a straightforward translation is involved, without optimizations, the validation of the translation is basically a syntactic issue. This task becomes much more difficult when transformations are applied and, change the source program flow of control, as well as the source language data-structures and their semantics. This is the case of **optimizing compilers**.

The importance of compiler optimizations is increasing, together with the development of modern CPU architectures, such as RISC (Reduced Instruction Set Computer), ILP (Instruction Level Parallelism) or EPIC (Explicitly Parallel Instruction Computing). It is the task of optimizing compilers to ensure correct and efficient use of the CPU. Without optimizations, only a small fraction of the computing power of these CPUs can be exploited. For

these architectures, as opposed to previous generations of architectures, a significant part of the optimizations is done at the code generation stage of the compiler. It is the aim of this research, among other things, to handle the validation of this kind of program transformations.

Another challenge of this research is to handle an **industrial environment**, where the source language, the processor, the compiler as well as the applications, are all industrial software. However, in order to be useful to the industrial sector, **a fully automatic process** is obligatory.

Our main goal is to develop a method, which can validate the machine dependent stage of an optimizing compiler, with a focus on advanced CPU architectures. Since we believe that automation is a key feature of such a method, we also want to check and demonstrate the feasibility of a fully automatic process. Another issue we want to investigate is the kind of assistance needed from the compiler in order to achieve this goal. To complete the validation process, we should provide the needed decision procedures, by developing our own algorithms and using existing tools.

We chose two extreme cases of the problem:

- **Case 1**: Verifying the optimized translation of programs, written in a general imperative language (C subset) for an EPIC type CPU. This problem is characterized by aggressive loop optimizations in the code generation stage of the compiler, in particular, loop unrolling and software pipelining. The compiler under research is an open source compiler and provides internal information upon request.

- **Case 2**: Verifying a translation from *synchronous* C programs, compiled with a commercial compiler, into an optimized code for the PowerPC family of architectures. Here, the only available information is the binary output of the compiler. The lack of assistance from the compiler, imposes limitations on the language of the source programs we handle. Synchronous programs are used very often to design safety-critical systems. These programs, by definition, have no loops, except for one implicit external loop in their main function. This fact helps us to overcome the difficulties raised by the industrial compiler. Thus another characterizing description for synchronous C programs is "loop-free programs".

Along with developing a theoretical basis and methods, which can handle these two problems, it was required to develop tools, capable of producing a

common semantics for different languages (C language as well as IA64 and, PowerPC assembly languages). These tools should automatically produce proof obligations, in order to establish the correctness of the translation. Algorithmic simplification is needed as well, to make the proof obligations provable by external validity checkers, to which our tools should interface.

1.2 Related Work

Several methods are used in order to mechanically solve the translation validation problem. Most methods start by constructing a data abstraction mapping. Then, either proof obligations or checkers, are produced. The proof obligations can be validated by an "off the shelf" validity checker, or by a theorem prover [ZG97, GZ99, GZG00, GZ00]. In other cases, the validation process is based on heuristics and, interaction between the heuristic part and the decision procedures, or the formulas' simplifier [Nec00, RM00]. Yet another approach, lets the compiler produce the proofs that are needed to establish its correctness [RM00, GZ00].

The CVT tool [PSS98a, PSS99] is an example of an industrial quality translation validation tool. It demonstrates an automatic procedure for validating the translation from the synchronous language SCADE to a sequential C program. Its main problems are the semantic difference between a synchronous language and a sequential C code, as well as the data abstraction mapping. The proof obligations that are produced, include equalities and uninterpreted functions, relying on the fact that the compiler does not change the way arithmetic functions are applied. The final validation is done by general decision procedures. Part of our thesis is to further validate the translation of the C programs, that are produced from SCADE source, to optimized machine code.

The VOC project [ZPFG02] concentrates on the translation of C programs by an optimizing compiler, where the investigated optimizations are machine independent. The language of both source and target systems is the intermediate representation of the compiler. Two different proof rules are presented: one which handles loop transformations and, another for "structure preserving" transformations. The first proof rule ensures the correctness of non "structure preserving" transformations, which are permutations on the loop iterations' order. The other proof rule is based on Floyd's inductive-assertion method [Flo67] and, proves the correctness of other machine independent

transformations. The methodology we are using in part II of our work is the same as the one used in VOC. However, applying it to EPIC code generator introduces new problems. For example, the EPIC-specific transformations of software pipelining and loop unrolling which occupy a major part of the thesis are not covered by the VOC project.

Other works are done as part of the Verifix project [GZ99, GZG00]. These rely on the PVS theorem prover [COR+95], in order to mechanize the verification, but full automation is not supported. One part of the Verifix project concentrates on compiler back-ends. It formalizes the semantics of the assembly code, as well as of an intermediate representation, using higher-order logic which are supported by PVS, whereas our goal is to use formulas whose validity is decidable. Another aspect of the Verifix project, which relates to our work, is the *program checker* approach. This approach, when applied to compiler verification, is the same as translation validation, since a validation checker is produced for each translation. However, contrary to our work, the checker is produced by the compiler. In the work described in [RM00], it is the compiler that produces the proofs, but the validity of the proof is established by the tool itself. The correctness proof consists of two sub-proofs: a sub-proof which shows that the analysis of the input program produces a correct result and, a sub-proof that establishes an equivalence between the original and the transformed programs.

Another method is described in [Nec00]. It is based on heuristics and a special purpose simplifier. The latter is composed of inference rules and is built in a tool. The tool is able to validate translations automatically. It can handle a wide range of optimizations, but as reported, may fail for loop unrolling and does not handle software pipelining. This work also uses the intermediate representation and, validates each optimization pass separately.

All the above mentioned works use compiler assistance, to some degree. It is a major challenge in translation validation research to minimize this dependence.

1.3 Contribution

Theoretical Framework

The theoretical framework, is based on [MP95], [ZPG00], [ZPL01] and [ZPFG02]. In this thesis, we define block programs syntax and semantics, which we use to formulate the problem. Based on this formulation we describe a criterion for the validation of a translation. This criterion also holds for infinite computation (as usually the case for reactive systems). We prefer to use *basic blocks* rather than *basic paths,* because we handle assembly language programs. We restrict the validation procedure in order to ensure that the validity of the verification conditions can be decided algorithmically. The framework includes the following items:

- A general syntax and formal semantics of *block programs*, for machine code programs. As semantics we use transition systems [MP95].
- A definition of correct translation for terminating and non-terminating programs.
- The AVALIDATE (Assembly VALIDATE) procedure, which is an adaptation and extension of the VALIDATE procedure of [ZPG00], [ZPL01] and [ZPFG02], was developed to support *block programs* and advanced machine specific loop optimizations. We prove the soundness of the procedure, for both terminating and non-terminating (though "correct") programs. We formulate the procedure to produce only formulas of decidable logics.
- In order to validate the translation of synchronous programs we define compressed transition system and a simplified procedure that ensure the correctness of the translation.

In this work we choose to handle two cases of the problem, as described hereby.

Case 1: Validating the Code Generator Optimizations of an EPIC type CPU

We developed algorithmic methods that cooperate with the AVALIDATE procedure, based on internal information from the compiler, to prove the validity of machine specific loop optimizations - loop unrolling and software pipelining. It should be emphasized that although information from the compiler is used, the soundness of the method does not depend on the correctness of this information. An algorithm to produce program invariants, which is based on symbolic evaluation was developed and enables the validation of software pipelining optimizations. The *SPV* (Software Pipelining Validator) tool that we developed, demonstrates the automatic validation of software pipelining optimizations, which are performed by the SGI-PRO64 [GADT00] compiler for Intel's IA64 processors family [Int]. SPV uses the intermediate representation of the code generator, that is obtained from the compiler. It produces verification conditions, that are validated by the CVC [SBD02] validity checker. At the early stages of the project, we used STEP[BBC+95], to check the basic concepts, and verify manually produced verification conditions. Our method is based on dividing the program into blocks, which are usually small. Therefore, the verification conditions are small as well. The implementation supports a subset of the IA64 architecture and, handles small programs. No performance problems were identified in the application of the tool.

Case 2: Validating the Translation from Synchronous C Programs to PowerPC Binary Code, by a Commercial Compiler

As opposed to case 1 above, here we could not apply the AVALIDATE procedure. The reason is that the basic building blocks of the validation methodology that underlies AVALIDATE, are the control and data abstraction mapping, between the source and the target (translated) systems. In case 1 above, we use the compiler internal information that can be obtained upon request, which assists us to construct these abstractions, whereas in the present case, there is no access to internal information of the commercial compiler. Therefore, we had to develop a different approach.

Hence, we developed the ANNOTATION algorithm. This algorithm produces a compressed form of a block program without loops and thus, the need to map the control and local variables is eliminated. The algorithm performs forward analysis of the assembly code, while annotating the machine instructions with the conditions on the input variables values, under which they are executed in a specific computation. The soundness of the algorithm is proved.

However, constructing the abstraction mapping between the observable variables of the source and the target systems, is still required. This is achieved by using standard debug symbolic information. Due to the presentation of both source and target systems in a compressed form, it is easy to break up the verification conditions, in order to handle each observable variable by a different verification condition.

Another problem we had to overcome, is the fact that the compiler uses special machine code sequences, which contains bit manipulation machine-instructions, to optimize some C language integer operations in the source program. In order to solve this problem, we developed a mechanism that interprets these sequences correctly, yet in a simplified way, so as to shorten the run time of the validity checker.

The tool we developed - *MCVT* (*Machine Code Validation Tool*) - validates the translation done by the DiabData C compiler of WindRiver [Dia], for the PowerPC CPU. It produces verification conditions and then invokes CVC to prove their validity. The MCVT tool is capable of proving the validity of the translation, of non-trivial programs, as is shown in the SafeAir case study, which is a simple communication controller program. In some cases, we expect the MCVT tool to fail in validating a correct translation within reasonable time limits. We believe that these cases can be solved, by further applying simplification methods to the verification conditions, produced by MCVT.

1.4 Overview

The methods we developed handle a variety of processor architectures. The tools we developed are retargetable to different processors and support already a couple of machine codes (IA64 and PowerPC). The proof obligations that are produced are formulas, whose validity is decidable. The tools are re-configurable to use different validators.

This thesis is organized in three parts, where the common theoretical framework is described in Part I, validating the code generator optimizations of an EPIC type CPU is detailed in Part II and, validating the translation from synchronous C programs by a commercial compiler is presented in Part III.

Part I

Theoretical Framework

Chapter 2

Transition Systems

This chapter describes the common framework of the two cases we handle in this thesis. In the following chapters we extend the relevant theory, separately for each case.

In order to be able to check that the "meanings" of two programs are equivalent, the semantics of both programs should be formalized by common means. Transition Systems, as introduced in [MP95] turned out to be a convenient common semantics. In this chapter we describe transition systems. In later chapters of this work we describe the syntax of different programming languages and their corresponding transition systems.

We assume that all variables that describe states of the program, or that appear in formulas specifying properties of programs, are taken from a universal set of variables $\mathcal{V}$, called the *vocabulary*. Variables in $\mathcal{V}$ are typed, where the type of a variable such as *boolean* or *integer*, indicates the domain over which the variables range. In particular, we assume that, for each $x \in \mathcal{V}$, its *primed version* x' is also in $\mathcal{V}$. To define the syntax of transition systems, we use the language $\mathcal{L}$, which is the language of first-order logic without quantifiers over $\mathcal{V}$, with arithmetic functions, the *ite (if-then-else)* and *case* operators, the natural order of integers and booleans, uninterpreted functions, arrays and constants. For arrays, the operations *with* and [ ] are part of the language $\mathcal{L}$. *Expressions* in this language are constructed out of variables of $\mathcal{V}$ and constants, to which functions and operations are applied. Expressions whose value is of integer type are *integer expressions* and, those whose value is boolean are *conditional expressions* (also called *assertions*). The array operation *with* is used to update an element of an array and [ ]

is used to get a value of an array element. For example, for an array a of integers, the expression $a\ with([2]:5)[2]$ has the integer value 5.

Let $V \subseteq \mathcal{V}$ be a finite set of typed variables. We define a *V-state* s to be a type-consistent assignment of $\mathcal{V}$, assigning to each variable $x \in V$ a value $s[x]$ over its domain. We denote by Σ_V the set of all states of V. The value of an expression $Exp(x_1, \ldots, x_n)$ in state s, where $x_1, \ldots, x_n \in V$, is denoted by $s[Exp]$ and, is defined as $Exp(s[x_1], \ldots, s[x_n])$. For an assertion p over V we use the notation $s \vDash p$ to denote that p holds in state s. We use the notion $\langle s^1, s^2 \rangle \vDash R$ to denote that the pair of states $\langle s^1, s^2 \rangle$, where $s^1 \in \Sigma^1, s^2 \in \Sigma^2$, belongs to the binary relation $R \subseteq \Sigma^1 \times \Sigma^2$.

The system - A *transition system (TS)* $\mathcal{S} = \langle V, O, \beta, \Theta, \rho \rangle$ consists of the following components:

- $V \subseteq \mathcal{V}$ is a finite set of typed variables. One of the variables, $\pi \in V$, is the *control variable* which represents the location of the control of the program. 0 is the *entry location* value. *exit* is the *exit location* value.

- $O \subseteq V$ is the set of *observable variables*. The observable variables are the variables which are visible to the external world. An *observation* is a projection of a state $s \in \Sigma_V$ on the observable variables. We denote by Obs the set of observations.

- β an *observability condition*. An assertion characterizing the states which should be observed. It is required that Θ implies β, i.e. all initial states are observable. A state satisfying β is called an *observable state*.

- Θ is an assertion characterizing the initial states of the system.

- ρ - *transition relation*. This is an assertion $\rho(V, V')$, relating a state s to its *successor* s', by referring to both unprimed and primed versions of the state variables. The transition relation $\rho(V, V')$ identifies state s' as a successor of state s if $\langle s, s' \rangle \vDash \rho(V, V')$, where $\langle s, s' \rangle$ is the joint interpretation which interprets $x \in V$ as $s[x]$ and, x' as $s'[x]$. Thus, for example, the transition relation may include "$y' = y + 1$" to denote that the value of the variable y in the successor state, is greater by 1 than its value in the previous state.

In writing a transition relation, we often use the notation $Pres(U)$ for $U \subseteq V$ which is an abbreviation for $\bigwedge_{v \in U} (v' = v)$, stating that all U variables are preserved.

Computations - A *computation* of a transition system $\mathcal{S} = \langle V, O, \beta, \Theta, \rho \rangle$ is a maximal-length finite or infinite sequence of states $\sigma : s_0, s_1, \ldots$, starting with an initial state $s_0 \models \Theta$ and such that every two consecutive states are related by the transition relation, i.e. $\langle s_i, s_{i+1} \rangle \models \rho$ for every i $0 \leq i, i+1 < |\sigma|$ [1]$(|\sigma| \leq \infty)$. A finite computation is called *terminating* if its last state is observable.

Observable behavior - In order to formally define the external behavior of a transition system, we use *observation trajectory*, which defines **when** and **what** is externally observable during a computation. Given a computation $\sigma : s_0, s_1, \ldots$, the observation trajectory corresponding to σ is obtained by taking the subsequence of observable states and projecting each of them on the observable variables O.

2.1 Programs and Translations

In this work we handle a few types of programming languages, the translation of one into the other and, transformations of programs in the same programming language. Each program P has a **syntactic description** and, **formal semantics** which is expressed by a **transition system**. Let P be a program written in a programming language PL and let the transition system S^P express the semantics of program P. We say that S^P is the transition system of program P. A translation that is done by an optimizing compiler, includes transformations of a *source* program P^S into a *target* program P^T, in the same programming language, usually in order to decrease the run-time of the program (*optimization* transformation). It also includes transformations of a source program into a target program in a different, lower level programming language.

[1] Let $\sigma : s_0, \ldots, s_{k-1}$ be a finite sequence, then $k = |\sigma|$ is the *length* of the sequence of states σ. When σ is infinite $k = |\sigma| = \infty$. In both cases an element of the sequence is s_i; where $0 \leq i < |\sigma|$.

To examine the correctness of a translation, we have to check the semantics of the source and target systems, and ensure the source semantics is preserved by the target. Thus, we consider the transition systems of the relevant source and target programs and require that certain relations hold between these systems. A similar approach can be found in other works on compiler verification, e.g. [PSS98c], [GZ00]. In the following subsection we define the correctness of a translation, based on the transition system semantics of the source and target programs, regardless of their programming languages. Later in this thesis, we describe syntax and transition system semantics of programs in specific programming languages and apply the notion of a correct translation.

2.2 Correctness of Translations

Informally, a correct translation needs only to ensure the preservation of the observable behavior [ZG97]. A target system is a correct translation of a source system, if every external behavior that exists in the target system, also exists in the source system. This characteristic is formalized by the notion of *refinement* [EdR+99].

Let $S^S = \langle V^S, O^S, \beta^S, \Theta^S, \rho^S \rangle$ and $S^T = \langle V^T, O^T, \beta^T, \Theta^T, \rho^T \rangle$ be two systems to which we refer as the *source* and *target* systems, respectively. Let $\tau = \tau\left\langle O^S, O^T \right\rangle$ be an assertion to which we refer as the *translation mapping*. Usually, the translation mapping expresses values of source observables in terms of target observables.

A source observation $o^S \in Obs^S$ and a target observation $o^S \in Obs^S$ are said to be τ-related if $\left\langle o^S, o^T \right\rangle \vDash \tau$.

We define system S^T to be a $\tau-$*refinement* of system S^S if, for every target observation trajectory $\omega^T : o_0^T, o_1^T, \ldots$, there exists a source observation trajectory $\omega^S : o_0^S, o_1^S, \ldots$, such that

1. $\left|\omega^T\right| = \left|\omega^S\right|$, i.e., they are both infinite or are both finite and of equal lengths,

2. For every $i < \left|\omega^T\right|$, o_i^S and o_i^T are τ-related.

Refinement means that for every computation of the target system, there exists a corresponding computation of the source system, in the sense that

both computations assume related values of the corresponding observable variables, at their observable states.

The definition of refinement, allows for finite and infinite computations in both the source and target systems. We include infinite computations, in order to support translations of reactive programs which run forever. For the same reason, we do not enforce determinism, i.e. for one initial state, we allow more than one computation to be possible. We say that a target program P^T is a *correct translation* of a source program P^S, if the transition system S^T of P^T refines the transition system S^S of P^S.

It should be emphasized, that since we choose the **translation** validation approach [PSS98c], rather than **translator** validation, we are not concerned with the compiler (translator) program validity. The **translation validation problem** is the problem of proving that a target transition system is a refinement of a source transition system. In this work we developed methods to establish refinement between source and target systems.

Part II

Validating an EPIC Code Generator

Chapter 3

A Compiler Structure

The translation validation method we describe in this part, relies significantly on our knowledge of compiler structure and on information that the compiler provides upon request. For this reason, we include a brief description of a compiler structure. This description is based on [ASU88] and [GADT00]. A typical structure of an EPIC compiler is shown in Fig. 3.1. Each block in the diagram represents a compiler phase. After syntax and semantics analysis, most compilers generate an *intermediate representation (IR)* as a program for an abstract machine.

A *basic block* is a sequence of consecutive statements, in which flow of control enters at the beginning and leaves at the end, without halt or a possibility of branching, except at the end. Basic blocks are the building blocks of the IR. Commonly used IR is a graph representation called a *flow graph*. Nodes of the flow graph are basic blocks, hence represent computations, and the edges represent the flow of control.

The optimizer phase attempts to improve the intermediate representation, resulting in a faster running machine code. A common practice in compilers' construction, is to perform each optimization in a different *pass*. Both the input and the output of an optimization pass are IRs. Some of the passes are executed more than once. The optimized IR is moved on to the code generator phase, which usually produces the final code. The machine code optimizer characterizes EPIC (Explicitly Parallel Instruction Computing) compilers. It applies optimizations that highly depend on the machine architecture and the exact code issued. These optimizations include register allocation, instruction scheduling, loop unrolling, predicating instructions,

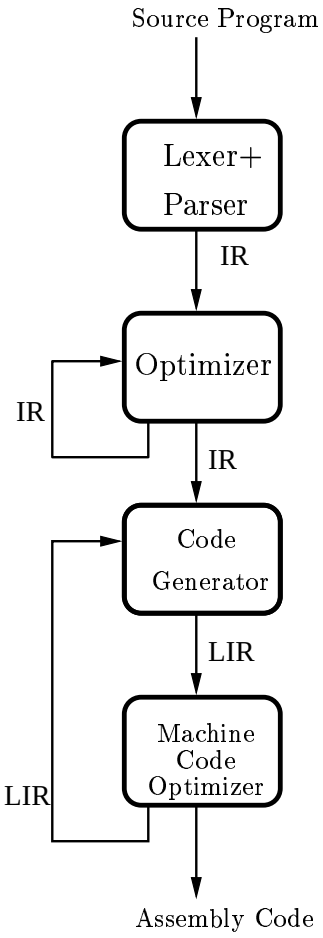

Figure 3.1: Compiler structure

use of speculative load instructions and software pipelining. The machine code optimizer uses a *low level intermediate representation (LIR)*, which is closer to the concrete architecture, and is used as the interface between its passes. The output of this compiler phase is the final assembly code.

3.1 Validating Each Pass Separately

As mentioned above, optimizers usually have many passes. The code generator of SGI-PRO64 compiler makes no exception. Based on this, we decide to validate each pass of the code generator separately, knowing that refinement is a transitive relation. I.e., let S^1, S^2, S^3 be transition systems, then if S^2 refines S^1 and S^3 refines S^2, then S^3 also refines S^1 [EdR+99]. The translation validation tool knows which optimization is applied on the target code in each pass and benefits from this knowledge. The LIR that the code generator produces in each pass is annotated with data. The validator tool uses this data to conclude which optimizations are applied, as well as their exact application details.

Chapter 4

EPIC Architectures

EPIC architecture is a way of achieving high instruction level parallelism, with reduced hardware complexity [KSR00]. The principal way of reducing the hardware complexity is by exposing the parallel micro-architecture to the compiler, using a processor model which provides an abstract description of the processor. Parallel execution, as well as latencies, are exposed to the compiler. The responsibility of achieving parallel execution is shifted from the hardware to the compiler. The cost of simplifying the hardware is complicating the compiler in general and the code generation stage, which is responsible for adjusting the code to the specific target, in particular.

We describe here some of the special features of EPIC architecture, in order to show the kind of problems that it raises for to the translation validator.

- Parallel execution - A typical EPIC processor allows the parallel execution of some instructions. It is the compiler's responsibility to bundle instructions together, in order to be executed in parallel and, to ensure that access to registers is consistent and legal.
- Speculative execution - In some cases the execution of an instruction may be discovered as erroneous and therefore, the effect of this execution should be canceled. This feature is called *speculative execution*. It is the compiler's responsibility to use this execution mode correctly.
- Register renaming - Registers' names are changed (logically - the register file is rotated), so that the very same instruction actually affects

different registers in two different executions.

- Software pipelining support (see section 7.1).
- Predicated instructions - The execution of an instruction depends on a bit predicate. The compiler generates code that assigns values to this predicate.

We choose to concentrate on two main loop optimizations: loop unrolling and software pipelining, which use most of the advanced features in the list above. For loop unrolling we use the TRIMARAN compiler [TR0]. This compiler produces code for the research architecture PlayDoh [SRM+94]. To demonstrate validation of software pipelining we use the open source SGI-PRO64 [GADT00] compiler, which produces code for the industrial EPIC processor IA64 of Intel [Int].

Chapter 5

Validating the Translation

We first define a general syntax for machine code programs, as well as their semantics, using transition systems. These definitions do not include the details of a specific machine code, but rather serve as an abstract model which fits most of the processors we know. We describe a general procedure to validate optimizations of machine code programs, and prove the soundness of this procedure. The following chapters describe how loop unrolling and software pipelining optimizations are validated, using this procedure. Finally, we describe the tool *SPV (Software Pipelining Validator),* that automatically verifies loop pipelining optimization for the SGI-PRO64 compiler and the IA64 architecture.

5.1 Block Programs

The machine code programs that we define here, are similar to flowchart programs [Flo67, Man72, LS84]. However, we prefer to use a syntax which is closer to the machine code.

An instruction $Inst$, is a legal machine instruction of the underlying physical processor. An instruction changes the state of the processor (the *machine state*), by assigning new values to the processor memory and registers. We divide the group of possible instructions to three subgroups, as follows:

1. *Procedure instructions* which are procedure calls. They have the following form:

- `call` <procedure-name>

An example of a procedure instruction is a call to `print` procedure which prints memory variables values.

2. *branch instructions* which have one of the two forms:

   - `branch` <condition> <label> - a conditional branch, where <label> is of type integer and, <condition> is a boolean expression.
   - `branch` <label> - an unconditional branch where <label> is of type integer.

   The <label> is the branch instruction *destination.*

3. *regular instructions* of the form `x:=` <expression> which are assignment instructions. (e.g. `x:= a+1`).

A *block* is a finite sequence of instructions $B = Inst_1, \ldots, Inst_k$. A block that contains a procedure instruction, may not contain any other instruction. Only the last instruction of a block may be a branch or a function instruction. The rest of the block's instructions are regular instructions. Also, only the first instruction of a block may be the destination of a branch instruction. A *block program* is a finite sequence of blocks $P = B_0, \ldots, B_n$, where the blocks are sequentially indexed. The index of a block is its *label.* The definition of a block conforms with the one of a basic block that is found in chapter 3.

In one block, at most one assignment is allowed for each variable. The meaning of a sequence of regular instructions in one block, is the simultaneous assignment of the expressions' values to the variables on the left side of the assignment. The framework built in Chapter 11 is the reasoning behind these restrictions.

Let $P = B_0, \ldots, B_n$ be a block program, then B_0, the block whose index is 0, is the *entry block.* Sometimes we call this block B_{entry}. Each block $B_i, i = 0, \ldots, n$ of a program may have one or two *successors,* denoted by $succ(B)$. If the last instruction of B_i is regular, then $succ(B_i) = \{i + 1\}$. If the last instruction is an unconditional branch with destination j, then $succ(B_i) = \{j\}$. If the last instruction is a conditional branch with destination j, then $succ(B_i) = \{j, i + 1\}$. The entry block is not a successor of any block of the program. We assume an additional implicit empty block B_{exit}, whose index is the *exit location* of the program. It is the *exit block.* A `branch`

instruction with destination $i > n$ is equivalent to a `branch` instruction with destination *exit*. Let *Proc(P)* denote the indices of blocks which consist of procedure instructions.

Each block program P, corresponds to a *transition system* $S^P = \langle V, O, \beta, \Theta, \rho \rangle$ where:

- $V : \{v_1, ..., v_m\} \cup \{pc\}$ is a finite set of system *variables*. V includes all the program variables, where $pc \in \{0, ..., n, exit\}$ is the control variable. Its value denotes the index of the block in which control currently resides. *pc* stands for *program counter*. When the location of control equals a block's label, this block is about to be executed. Block label i and block name B_i, are used interchangeably.

- O : The observable variables are memory variables.

- $\beta : pc \in \{0, exit\} \cup Proc(P)$. This identifies the entry, exit and call-locations as locations at which we wish to observe the values of the observable variables. Since β depends only on *pc*, we will write $\beta(i)$ or $i \in \beta$ to denote that $pc = i$ implies β.

- Θ : The *initial condition* assertion. $\Theta \rightarrow pc = 0$.

- ρ : The *transition relation*.

Each block $B_i = Inst_1, \ldots, Inst_k$ in the program is associated with a *block transition relation*:

$$\rho_i(V, V') : pc = i \wedge \bigwedge_{v \in V - \{pc\}} v' = Exp_v(V) \wedge pc' = comput\_pc(V),$$

where:

1. If there is a regular instruction in B_i such that `v := Exp(V)`, then $Exp_v(V)$ is `Exp(V)`, otherwise $Exp_v(V)$ is `v`. In case of data dependencies within a block, updated values replace the corresponding variables in $Exp_v(V)$. Recalling that there is at most one assignment for each variable, in a block, $Exp_v(V)$ is well defined.

2.

$$compute\_pc(V) = \begin{cases} \textbf{if} & \text{the block's last instruction is} \\ & \text{regular or a function} \\ \textbf{then} & pc+1 \\ \textbf{else if} & \text{the block's last instruction is} \\ & \text{'}\texttt{branch}\ \text{<label>'} \\ \textbf{then} & label \\ \textbf{else if} & \text{the block's last instruction is} \\ & \text{'}\texttt{branch}\ \text{<condition> <label>'} \\ \textbf{then} & ite(condition, label, pc+1) \end{cases}$$

The program transition relation is $\rho = \bigvee_{i=0}^{n} \rho_i(V, V')$.

Two consecutive states in a computation of a block program, correspond to the machine states before and after the execution of one block. No transition relation is associated with the exit block.

Claim 1 *Every computation $\sigma : s_0, \ldots, s_i, \ldots$ of a block program, corresponds to a finite or infinite sequence of blocks' labels: $B_0, \ldots, B_{j_i}, \ldots$, where $j_i = s_i[pc]$, $B_{j_{i+1}} \in succ(B_{j_i})$ and $\langle s_i, s_{i+1} \rangle \vDash \rho_{j_i}$. If the sequence is finite, then its last block is B_{exit} (i.e. the exit block).*

Proof. A direct consequence of the soundness proof of the inductive assertion method by Floyd [Man74] ■

Figure 5.1 shows a block program and its corresponding transition system.

5.2 The AVALIDATE Procedure

Let $P^S = B_0^S, \ldots, B_m^S$ and $P^T = B_0^T, \ldots, B_n^T$, be a source block program and a target block program respectively, with the corresponding transition systems $\mathcal{S}^S = \langle V^S, O^S, \beta^S, \Theta^S, \rho^S \rangle$ and $\mathcal{S}^T = \langle V^T, O^T, \beta^T, \Theta^T, \rho^T \rangle$. As a convention, we use capital letters to indicate a source program variable (e.g. PC) and, small letters for a target variable (e.g. pc). We assume that there is a $1-1$ correspondence between the source and the target observable variables and that the translation relation τ is given by $\tau : O = o$. Based

Block Program	**Transition System**
`L0: s := 0; i := 0;`	$V = \{s, i, a, pc\}$
`L1: s := a[i]+s;`	$O = \{a, b\}, \beta : pc \in \{0, 2, exit\}$
`i := i+1;`	$\Theta : pc = 0$
`branch (i<100) L1`	$\rho : (pc = 0) \wedge (i' = 0) \wedge (s' = 0) \wedge (pc' = 1) \quad \vee$
`L2: b := print(s)`	$(pc = 1) \wedge (s' = a[i] + s) \wedge (i' = i + 1)$
	$\wedge (pc' = ite(i + 1 < 100), 1, 2)) \quad \vee$
	$(pc = 2) \wedge (b' = call(print, s)) \wedge pc' = exit$

Figure 5.1: A block program and its transition system

on the translation validation procedure VALIDATE which is introduced in [PZP00] and [ZPL01], we define the AVALIDATE procedure, a variation of VALIDATE, which better fits block programs. The procedure describes how to produce the verification conditions, sufficient to prove that the target system $\mathcal{S}^T$ refines the source system $\mathcal{S}^S$. AVALIDATE uses some constructions without describing how to implement them. Based on these constructions, a set of verification conditions is produced. The soundness of AVALIDATE implies that every concrete implementation of these constructions, is a sound proof of the translation validation.

The AVALIDATE procedure is defined as follows:

1. Construct a control abstraction mapping

$$\kappa : \{B_0^T, \ldots, B_{exit}^T\} \longmapsto \{B_0^S, \ldots, B_{exit}^S\},$$

   such that $\kappa(0) = 0$ and $\kappa(exit) = exit$. Also, the mapping κ should preserve location observability, i.e. $\beta^T(i)$ iff $\beta^S(\kappa(i))$, where i is a block index.

2. For each block B_i of the target system, form an invariant φ_i, that states a target property which should hold true whenever $pc = i$. Also, $\varphi_0 = true$. Only target variables may appear in φ_i.

3. Construct a data abstraction mapping

$$\alpha : \bigwedge_{V \in V^S \wedge V \neq pc} (V = E_v(V^T)) \wedge PC = \kappa(pc),$$

where E_v is a (possibly conditional) expression over target variables. For example:

$$x = case\ p_1 : e_1;\ p_2 : e_2;\ otherwise\ e_3;.$$

It is required that for every $i \in \beta^T : \varphi_i \wedge \alpha \rightarrow O = o$[1]. The data abstraction α is a mapping, which maps every target state s^t to a source state s^s, such that $\langle s^s, s^t \rangle \vDash \alpha$

4. For each $i \in \{0, \ldots, n\}$ a block index in P^T, construct a verification condition VC_i :

   $\varphi_i \wedge \alpha \wedge \alpha' \wedge \rho_i^T \rightarrow$

   $$(\bigvee_{j \in succ(B_i)} (pc' = j) \wedge \varphi'_j) \wedge \tag{5.1a}$$

   $$(\rho^S_{\kappa(i)} \vee \tag{5.1b}$$

   $$(Pres(V^S) \wedge\ e < c \wedge e' > e \wedge pc' \notin \beta^T)) \tag{5.1c}$$

   where e is an integer valued expression which depends on the target variables. The disjunct $Pres(V^S)$ allows a non-singleton sequence of target states to be mapped to a single source state (stuttering). The constraint $e < c \wedge e' > e$ is needed, in order to bound the length of such stuttering sequence. In this constraint, e is an integer valued expression and c is an integer constant. α' is α, where each source or target variable is replaced by its primed version.

5. Establish the validity of the verification conditions.

It is also allowed to add auxiliary variables to the target system, as needed. Auxiliary variables are program variables, to which assignments are added inside a program, not for influencing the flow of control or the computation, but in order to assist in the verification of the translation (see [AL91], [MP95]-page 144).

Let $\sigma^S : s_0, s_1, \ldots$ and $\sigma^T : t_0, t_1, \ldots$ be source and target computations respectively. A *contracting mapping* from $|\sigma^T|$ to $|\sigma^S|$ is a surjective function $f : |\sigma^T| \longrightarrow |\sigma^S|$, such that $f(0) = 0$ and $f(i{+}1) \in \{f(i), f(i){+}1\}$ for every i,

[1] We recall that capital letters are used for source system variables, and small letters are used for target system variables.

$0 < i+1 < \left|\sigma^T\right|$ .For an abstraction mapping α, we say that the computation σ^S is α-*related* to σ^T if there exists a contracting mapping $f : \left|\sigma^T\right| \longrightarrow \left|\sigma^S\right|$ such that $\left\langle s_{f(i)}, t_i \right\rangle \vDash \alpha$, for every $i < \left|\sigma^T\right|$.

5.3 Soundness of the AVALIDATE Procedure

Let $\mathcal{S}^S = \langle V^S, O^S, O^S_L, \Theta^S, \rho^S \rangle$ and $\mathcal{S}^T = \langle V^T, O^T, \beta^T, \Theta^T, \rho^T \rangle$, be a source transition system and a target transition system respectively. We want to prove that if the verification conditions are valid, then S^T refines S^S. The proof includes:

1. Proving that if $\sigma = s^T_0, \ldots, s^T_i, \ldots$ is a computation of the target system $\mathcal{S}^T$, then φ_j holds at s^T_i where $s^T_i[pc] = j$, $j = 0, \ldots, n$ and $i < \left|\sigma^T\right|$.

2. Showing that for each target computation σ^T there exists a corresponding computation σ^S of the source system.

3. Showing that the observation trajectories induced by σ^T and σ^S are α-related.

Proving the invariants

Let $\sigma = s^T_0, \ldots, s^T_i, \ldots$ be a computation of S^T. The proof is done by induction on i. The base case is trivially true ($\varphi_0 = true$).

Let $\varphi_{s^T_i[pc]}$ be valid at state s^T_i of the computation.

Induction step: $\varphi_{s^T_{i+1}[pc]}$ holds at s^T_{i+1} because s^T_{i+1} follows s^T_i in the computation, hence $s^T_{i+1}[pc] \in succ(s^T_i)$ and $\left\langle s^T_i, s^T_{i+1} \right\rangle \vDash \rho^T$, in particular $\left\langle s^T_i, s^T_{i+1} \right\rangle \vDash \rho^T_{s_i[pc]}$ (see claim 1). The validity of the verification condition $VC_{s^T_i[pc]}$ proves that $\varphi_{s^T_{i+1}[pc]}$ holds at state s^T_{i+1} (see 5.1a).

We showed that if for each $i \in \{0, \ldots, n\}$,

$$\varphi_i \wedge \rho^T_i \rightarrow \bigvee_{j \in succ(B_i)} (pc' = j) \wedge \varphi'_j \tag{5.2}$$

then φ_j holds at s^T_i where $s^T_i[pc] = j$, $j = 0, \ldots, n$ and $i < \left|\sigma^T\right|$. Since the invariants are target system invariants in which source system variables do not appear, and the mapping α **defines** the source variables in terms of target variables, we may ignore the conjuncts α and α' which appear in the verification conditions.

Constructing a corresponding source computation

We first define a *computation prefix.*

Definition 2 *Computation prefix - Let* $\mathcal{S} = \langle V, O, \beta, \Theta, \rho \rangle$ *be a transition system, and* $\sigma = s_0, \ldots, s_N$ *be a sequence of states of* $\mathcal{S}$ *such that* $s_0 \models \Theta$ *and, for all* $i, 0 \leq i \leq N-1$, *we have* $\langle s_i, s_{i+1} \rangle \models \rho$ *then* σ *is a computation prefix of* $\mathcal{S}$.

Let $\sigma^T = t_0, \ldots, t_i, \ldots, 0 \leq i < |\sigma|$, be a computation of the target system S^T. We inductively construct a source computation which is α-related to σ^T.

The construction considers increasing prefixes $\sigma_j^T : t_0, \ldots t_j$ of the computation σ^T and shows how to construct a corresponding α-related prefix σ_j^S. Together with the construction of σ_j^S, we also inductively define the construction mapping f.

As the base case, we take $\sigma_j^T : t_0$ and $\sigma_j^S : s_0$, where s_0 is obtained by mapping t_0 to values of the variables V^S according to α. We also define $f(0) = 0$. Obviously σ_0^S is α-related to σ_0^T.

Next assume that we have already constructed the prefix $\sigma_j^S : s_0, \ldots s_k$ corresponding to the prefix $\sigma_j^T : t_0, \ldots t_j$, and defined the value of $f(i)$, for all $i = 0, \ldots, j$, We will show how to construct σ_{j+1}^S and define $f(j+1)$. Let $i = t_j[pc]$. By induction we know that $f(j) = k$ and that s_k and t_j are α-related. In case $i =$*exit,* σ_j^T is the complete target computation, and we are done. Otherwise, $j < \left|\sigma^T\right|$ and there exists a successor state t_{j+1}. Let V_j^T and V_{j+1}^T denote the values of the variables V^T at states t_j and t_{j+1} respectively. Let V_j^S and V_{j+1}^S denote their respective α images. Consider the expression

$$\varphi_i(V_j^T) \wedge \alpha(V_j^S, V_j^T) \wedge \alpha(V_{j+1}^S, V_{j+1}^T) \wedge \rho_i^T(V_j^T, V_{j+1}^T)$$

which is obtained from the sub-formula $\varphi_i \wedge \alpha \wedge \alpha' \wedge \rho_i^T$ by the substitution

$$\gamma : \left[V^T \longleftarrow V_j^T, (V^T)' \longleftarrow V_{j+1}^T, V^S \longleftarrow V_j^S, (V^S)' \longleftarrow V_{j+1}^S\right]$$

We claim that this expression evaluates to 1 (*true*). The invariant $\varphi_i(V_j^T)$ holds due to part 1 of the proof and the fact that $pc_j = i$. The mapping $\alpha(V_j^S, V_j^T) = \alpha(V_{j+1}^S, V_{j+1}^T) = 1$ because we computed V_j^S and V_{j+1}^S using α. The transition relation $\rho_i^T(V_j^T, V_{j+1}^T)$ holds because t_{j+1} is a ρ^T-successor of t_j.

Since VC_i is assumed to be valid, it follows that the r.h.s of this implication holds under the substitution γ. We consider two cases:

Case I: The r.h.s of $VC_i[\gamma]$ holds due to the disjunct 5.1b ($\rho^S_{\kappa(i)}$). Then, we take $\sigma^S_{j+1} = \sigma^S_j; s_{k+1}$, where $s_{k+1} = \alpha(t_{j+1})$ which implies $s_{k+1}[V^S] = V^S_{j+1}$. Obviously s_{k+1} is a successor of s_k since $\langle s_k, s_{k+1}\rangle \models \rho^S_{\kappa(i)}$ and, by the induction hypothesis α-relating t_j to s_k, $s_k[pc] = \kappa(i)$. In this case, we define $f(j+1) = k+1 = f(j)+1$. It is obvious that σ^S_{j+1} and $\sigma^T_j; s_{j+1}$ are α-related.

Case II: The r.h.s of $VC_i[\gamma]$ holds due to the disjunct (5.1c). Note that this is possible only if $i \notin \beta$, that is, i is not an observation-requiring location. In this case, we take $\sigma^S_{j+1} = \sigma^S_j$ and define $f(j+1) = f(j) = k$. Due to the conjunct $Pres(V^S)$, we know that $V^S_{j+1} = V^S_j$, which shows that t_{j+1} and s_k are α-related (due to $\alpha(V^S_{j+1}, V^T_{j+1})$). It follows again that σ^S_{j+1} and $\sigma^T_{j+1} = \sigma^T_i; s_{j+1}$ are α-related. Case II can be consecutively repeated only a bounded number of times due to the bounded progress measure e.

The Observations Induced by σ^S and σ^T are τ-related

In the previous part we have shown how to construct a source computation σ^S which is α-related to a given target computation σ^T. We now consider the induced observation trajectories ω^S and ω^T and show that they are of equal length and their respective observations are τ-related. According to the construction, whenever a state t_j was associated with an observation-requiring location i, we could not use case II, which implies that $f(j+1) = f(j)+1$. As a result, while a source state s_k which is not observation-requiring can be mapped to an arbitrary long (but finite) sequence of target states, each observation-requiring state corresponds to a unique target state. It follows that between σ^S and σ^T there is a $1-1$ correspondence between the observation-requiring states. Since these are the only states retained while forming an observation sequence, we conclude that $\left|\omega^S\right| = \left|\omega^T\right|$. Furthermore, let t_j and s_j be two states corresponding to the same observation, and let $t_j[pc] = i \in \beta$. Since φ_i holds at t_j, and s_j, t_j are α-related, the implication $\varphi_i \wedge \alpha \rightarrow O = o$ leads to $s_j[O] = t_j[o]$. ■

Chapter 6

Loop Unrolling

6.1 Description of the Optimization

Loop unrolling generates more opportunities for instruction-level parallelism by duplicating the code of the loop's body a number of times equal to the unrolling factor. Instead of moving the control to the beginning of a loop body, at the end of an iteration (in the case that the loop termination condition is not met), the body of the loop is copied by the compiler, one or more times, and the control is moved to the next instruction. Loop unrolling is a preparatory optimizer stage, which enables further optimizations. When the number of loop iterations is known to the compiler, this stage is followed by elimination of branch instructions at the end of most iterations, thus increasing the basic block size. In other cases, the unrolled loop instructions are rescheduled, so as to avoid stalling of the processor pipeline and to obtain a better parallelization [LE95].

Figure 6.1 gives a schematic description of loop unrolling optimizer stage. The left side of the figure describes the source loop, where the number of iterations is N. The right side describes the optimized loop, with unrolled loop factor of C. The target loop iterates $((N-1)\ div\ C)+1$. The last target iteration may not be completed. Another difference between the source and the target loops is the value of the loop counter. In the source program it assumes all values between 0 to N, while in the optimized program its values are $0, C, 2C, \ldots, N$.

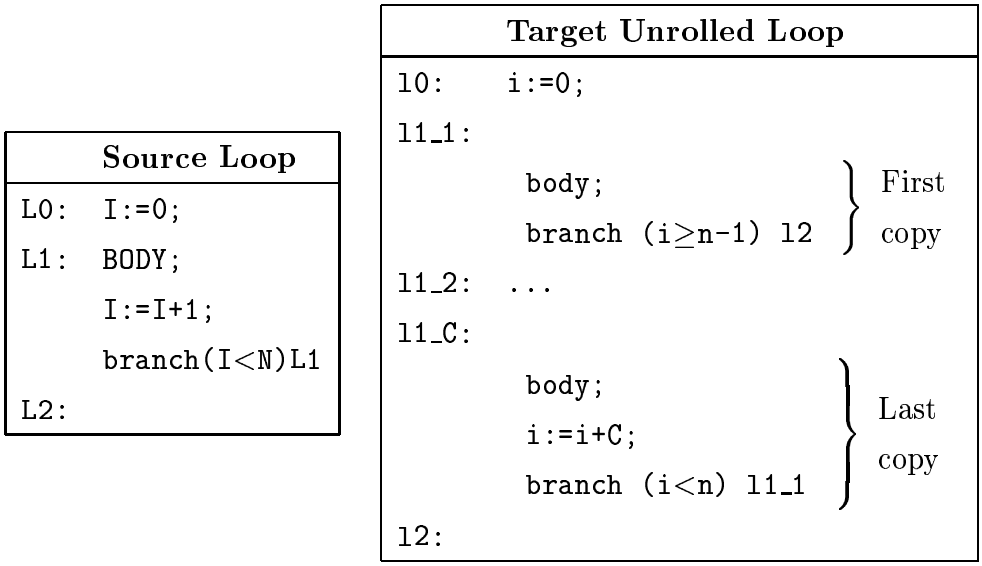

Figure 6.1: General form of loop unrolling

6.2 Validating Loop Unrolling

In this section we describe the data and control abstraction mapping that can be used in the framework of the AVALIDATE procedure in order to produce the verification conditions required to prove the validity of loop unrolling optimization. To validate this particular optimization technique, there is no need to compute invariants as described in the second step of the procedure AVALIDATE , and we can use the invariant $\varphi = true$.

We consider the general loop unrolling form which is presented in figure 6.1. We assume that BODY is only one block, knowing that this optimization is usually applied to the most-inner loop. For simplicity, we also assume that BODY code does not change I or N. Usually, loop unrolling is not applied in cases which violate this assumption. Consequently, a similar assumption is made for the target's body. According to the AVALIDATE procedure, we first have to construct the control abstraction mapping. The target loop is divided into C blocks, all of them are mapped into the loop body block of the source. Thus, the control mapping κ is: $(\kappa(l0) = L0) \wedge \ldots \wedge (\kappa(l1\_1) =$

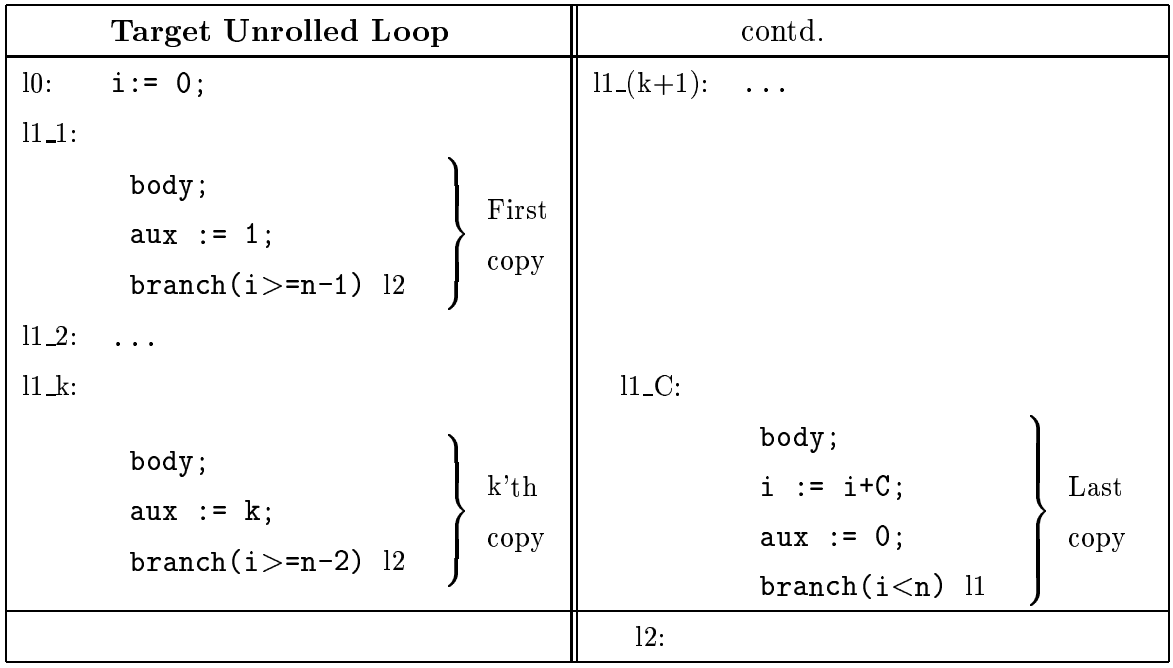

Figure 6.2: Adding an auxiliary variable

$\ldots = \kappa(l1\_c) = L1) \wedge (\kappa(l2) = L2)$.

An auxiliary variable aux, is added at the end of each copy of the body, as shown in figure 6.2, in order to enable a valid data abstraction outside the loop. The auxiliary variable is needed, because exiting from the target loop can occur at any location inside the target loop. We could equally use $N \bmod C$, but in order to simplify automatic validation of the verification condition, we choose to add an auxiliary variable. The data abstraction α is:

$$(pc \leq l1\_1 \rightarrow I = i) \wedge pc = l1\_2 \rightarrow I = i + 1 \ldots \wedge$$
$$(pc = l1\_C \rightarrow I = i + C - 1) \wedge (pc = l2 \rightarrow I = i + aux) \wedge (N = n) \wedge \ldots .$$

Verification conditions are needed for each $l1\_k$ $(k = 1, \ldots, C)$. For example, VC_{l1_1}, in Fig. 6.3 is produced for the first copy of `body` $(k = 1)$, whereas the verification condition for the last copy of `body` $(k = C)$ is presented in Fig. 6.4.

Appendix A shows a complete example of the validation process of loop unrolling optimization. The compiler we use is the TRIMARAN compiler

$$
\begin{array}{l}
VC_{l1\_1}: \\
\quad \alpha: \left\{ \begin{array}{l} (pc \leq l1\_1 \rightarrow I = i) \wedge (pc = l1\_2 \rightarrow I = i+1) \wedge \ldots \wedge \\ (pc = l1\_C \rightarrow I = i + C - 1) \wedge \\ (pc = l2 \rightarrow I = i + aux) \wedge (N = n) \wedge (PC = \kappa(pc)) \wedge \ldots \end{array} \right. \\
\quad \alpha': \left\{ \begin{array}{l} (pc' \leq l1\_1 \rightarrow I' = i') \wedge (pc' = l1\_2 \rightarrow I' = i'+1) \wedge \ldots \wedge \\ (pc' = l1\_C \rightarrow I' = i' + C - 1) \wedge \\ (pc' = l2 \rightarrow I' = i' + aux') \wedge (N' = n') \wedge (PC' = \kappa(pc')) \\ \wedge \ldots \end{array} \right. \\
\quad \rho^T_{l1\_1}: \left\{ \begin{array}{c} (pc = l1\_1) \wedge \rho^T_{body} \wedge (pc' = ite(i \geq n-1, l2, l1\_2)) \wedge \\ (aux' = 1) \wedge (i' = i) \wedge (n' = n) \end{array} \right. \\
\quad \rightarrow \\
\quad \rho^S_{L1}: \left\{ \begin{array}{c} (PC = L1) \wedge \rho^S_{BODY} \wedge (I' = I + 1) \wedge \\ (PC' = ite(I < N, L1, L2)) \wedge (N' = N) \end{array} \right.
\end{array}
$$

Figure 6.3: Verification condition for the first copy

$$
\begin{array}{ll}
VC_{l1\_k}: & \alpha \wedge \alpha' \wedge \\
& \rho^T_{l1\_C}: \left\{ \begin{array}{c} (pc = l1\_C) \wedge \rho^T_{body} \wedge (pc' = ite(i + C < n, l1\_1, l2)) \wedge \\ (aux' = 0) \wedge (i' = i + C) \end{array} \right. \\
& \rightarrow \\
& \rho^S_{L1}: \left\{ \begin{array}{c} (PC = L1) \wedge \rho^S_{BODY}(V^S, V^{S\prime}) \wedge (I' = I + 1) \wedge \\ (PC' = ite(I + 1 < N, L1, L2)) \wedge (N' = N) \end{array} \right.
\end{array}
$$

Figure 6.4: Verification condition for copy Number C

[TR0]. A special proof rule was implemented in STeP [BBC+95] by Henny Sipma to support code validation. The user provides source and target transition systems, data abstraction and invariants. Upon invoking the proof rule, STeP automatically produces the verification conditions. Then, the verification conditions are mechanically, though interactively, proved by STeP[BBC+95].

Chapter 7

Software Pipelining

7.1 Architecture Definition

In this section we define the syntax of a pseudo machine code, VIR - *virtual intermediate representation*, for a virtual architecture, based on the IA64 family of architectures. This syntax conforms with that of block programs, as defined in 5.1. To simplify the presentation, we prefer not to use the original machine-language syntax, but to present machine programs in a higher-level, C-like, programming style. In particular, we use "`do-while`" instead of conditional branch, which is used to implement loops. In other cases we use "`if-else`" instead of conditional branch. However, the actual implementation works directly on the genuine IA64 machine language syntax.

7.1.1 Machine Registers

The processor has the following registers:

- $\texttt{r}_\texttt{i}$ $(1 \leq i \leq m)$- A general integer register.
- `lc`- Loop counter register. A special register which counts the iterations within a loop and controls the loop completion.
- $\texttt{t}_\texttt{i}$ $(1 \leq i \leq n)$- An integer register in the rotating register file.

- $\mathtt{p_i}$ $(1 \leq i \leq l)$- A one bit register, called *predicate register,* in the rotating predicate register file. $\vec{p}$ is the vector of all the predicate registers.

7.1.2 Machine Instructions

Assigning a value to a memory variable is a *store-to-memory* instruction. The occurrence of a memory variable on the right side of an assignment, stands for a *memory-load* instruction. Thus, `a := t1` stands for storing the value of register $\mathtt{t_1}$ into memory variable `a`. Direct assignment of a memory variable to another memory variable is not allowed, since this instruction is not supported by the underlying machine. Arithmetic instructions are used with their usual meaning. We use *wait* to indicate no operation. Usually, this is used when the CPU waits for the completion of an instruction, such as a *load-from-memory.*

A special syntax represents the rotation of the register file:

$$\langle \mathtt{t_n} := \mathtt{t_{n-1}} := \cdots := \mathtt{t_1} \rangle$$

with the following semantics: $t'_n = t_{n-1} \wedge \cdots \wedge t'_2 = t_1$. An equivalent notation is used for the rotation of the predicate register file. The predicated execution feature of the architecture, is expressed by using an "$\mathtt{if}\ (\mathtt{p_{c_k}})$" prefix. Before starting a loop with N iterations and with Sc pipeline stages, the predicate array $\vec{p}$ is initialized to 10^{Sc-1} (equally written as $\vec{\mathtt{p}} := \mathtt{10^{Sc-1}}$). This means that the first predicate register is initiated to 1 and, the remaining $Sc-1$ predicate registers are initiated to 0. All instructions, appearing on a single command line, are executed in parallel within a single step. In other cases, the parallel execution is deduced from the context[1].

7.1.3 Architecture Support for Software Pipelining

Software pipelining is a technique that takes advantage of advanced architecture features, such as parallelism (multiple memory and arithmetic

[1]Note that although instructions may be executed in parallel, the architecture defines the semantics of parallel execution, to be the same as the sequential execution of the instructions' sequence.

units), rotating register file, predicate register and special branch instructions [AJLA95, Huf93]. Software pipelining increases the loop throughput by overlapping the loop's iterations, that is, by initiating successive iterations before the completion of previous ones, and achieving saturation of functional units. To pipeline a loop, the compiler should find an instruction schedule that best utilizes the functional units, to achieve minimal execution time, yet without causing a register jam.

One technique for loop scheduling is *Modulo Scheduling* [RST92]. To find an overlapped schedule, the compiler must take into account the constraints imposed by the availability of functional units and registers. Suppose that execution of one iteration takes c cycles. Considerations of the data dependencies within the loop body and instructions latency lead to a calculation of an *initiation interval* - Int. Int is the number of instruction cycles issued for iteration i, before iteration $i+1$ can be initiated. Let O_1, O_2 be two instructions using the same machine resource, which are scheduled to cycles c_1 and c_2 of a loop body, respectively. Then, it is required that ($c_1 \mod Int$) be different from ($c_2 \mod Int$). When all such constraints are satisfied, we have a sound *modulo schedule*. It looks as though the loop body is divided into *stages* whose execution time is Int cycles. The number of stages is $Sc = c/Int$

To demonstrate the modulo scheduling technique consider the C program in Fig. 7.1. In this loop, there is no data dependency between the iterations.

```
int a[100], b[100], N;
main {
   int I:=0;
   do {
      a[I]:=b[I]+5;
      I++;
   } while(I < N);
}
```

Figure 7.1: C source program

The target program that appears in Fig. 7.2, is expressed in VIR, without any optimization. Since the *load* delay is 2 cycles, one iteration time is 4

```
i1 := 0; i2 := 0; lc := 0;
do
{
   t1 := b[i1]; i1 := i1 + 1;                    –Load
   wait one cycle for the load delay to expire;  –Wait
   t2 := t1 + 5;                                 –Add
   a[i2] := t2; i2 := i2 + 1;                    –Store
   lc + +;
}while (lc < N);
```

Figure 7.2: Unoptimized target code

cycles. For this loop, the compiler calculates an initiation interval of 1 cycle. The number of stages Sc, is thus 4. Pipelining can be achieved by initiating source iteration *i+1,* one cycle after iteration *i*. A pipelined execution of the resulting instruction scheduling for this loop is demonstrated in Fig. 7.3 with N, number of iterations, equal to 6. *ld* stands for load, *w* for wait, *st* for store and *add* for add. $op(i) : op = ld, w, st, add$ stands for execution instruction *op* of source iteration *i*. For example, $ld(2)$ stands for the load instruction of iteration number 2.

The iterations of the target program are composed of steady state (*kernel*) iterations, which execute all the loop instructions (cycles 4,5,6) of the loop, *prologue* iterations (cycles 1,2,3), and *epilog* iterations (cycles 7,8,9). The target code contains predicated kernel instructions. When executed, different values of the predicate registers cause different executions that correspond to prologue, kernel and epilog iterations. One iteration is completed at each cycle. The number of target iterations is *N+Sc-1* $= 9$. The execution time of the source loop is $4 \cdot 6 = 24$ cycles, while the optimized loop time is only 9 cycles.

After allocating registers, using the rotating register file, the target code is the one presented in Fig. 7.4, or equivalently, by the IA64 assembly code in Fig. B.1 of appendix B.

	Cycle						
Prologue	1	*ld(1)*					
	2	*w(1)*	*ld(2)*				
	3	*add(1)*	*w(2)*	*ld(3)*			
Kernel	4	*st(1)*	*add(2)*	*w(3)*	*ld(4)*		
	5		*st(2)*	*add(3)*	*w(4)*	*ld(5)*	
	6			*st(3)*	*add(4)*	*w(5)*	*ld(6)*
Epilog	7				*st(4)*	*add(5)*	*w(6)*
	8					*st(5)*	*add(6)*
	9						*st(6)*

Figure 7.3: Loop pipeline schedule

7.2 Validation of the Optimization

In this section we describe a method to validate software pipelining loop optimization. It includes the production of assertions which hold at the beginning of each target iteration, the production and the decomposition of the verification conditions.

7.2.1 A General Software Pipelining Representation

Consider a general C loop as presented on the left side of Fig. 7.5, and its target pipelined code, presented on the right side of the same figure. B is the source loop body and B_T is the target loop body, $\mathtt{t}_1, \ldots, \mathtt{t}_l$ are the rotating registers used for this loop, and $\mathtt{p}_1, \ldots \mathtt{p}_{\mathtt{Sc}}$ are the predicate registers. Sc is the number of stages the optimizing compiler chooses for this loop. $\mathtt{s}_j$ is a list of instructions whose execution is predicated by the value of $\mathtt{p}_{\mathtt{c}_j}$ ($\mathtt{p}_{\mathtt{c}_j}$ is a predicate register, and thus $1 \leq c_i \leq Sc$, for $1 \leq i \leq k$).

Each of the instructions, $\mathtt{s}_j$,$1 \leq j \leq k$, which depends on the same predicate register ($\mathtt{p}_{\mathtt{c}_j}$), belongs to the same pipeline stage. Also, $\forall j \in [1..k]$:

```
p1 := 1; p2 := 0; p3 := 0; p4 := 0;
lc := 0; i1 := 0; i2 := 0;
do{
l0 :
   if (p4) a[i1] := t2; i1 := i1 + 1;                    Stage 4
   if (p3) t1 :=  t4 + 5;                                Stage 3
   if (p1) t2 := b[i2]; i2 := i2 + 1;                    Stage 1
   ⟨t4 := t3 := t2 := t1⟩;                               Register rotation
   ⟨p4 := p3 := p2 := p1⟩;                               Predicate rotation
   lc + +;
   if (lc < N) p1 := 1;  else  p1 := 0;
while (lc < N + 3);
l1 :
```

Figure 7.4: Target pipelined loop

$1 \leq c_j \leq Sc$. The number of target iterations is $N + Sc - 1$. Note that the first and last $Sc - 1$ iterations of the loop, execute only some of the stages contained in the loop's body, since some of the values of the predicate registers equal 0.

7.2.2 The General Idea

In the following sections of this chapter, we describe the algorithm we developed to compute the required invariants using symbolic evaluation. We show how to produce the required verification conditions. Then the algorithm is demonstrated on the running example.

Let ρ^S and ρ^T stand for the transition relations, representing the loop body of the source and target systems, respectively, and α be the data abstraction mapping. We want to compute an invariant φ, which can be used in the AVALIDATE procedure. We first note that while the source loop iterates N times, the target loop iterates $N+Sc-1$ times. We handle this by choosing

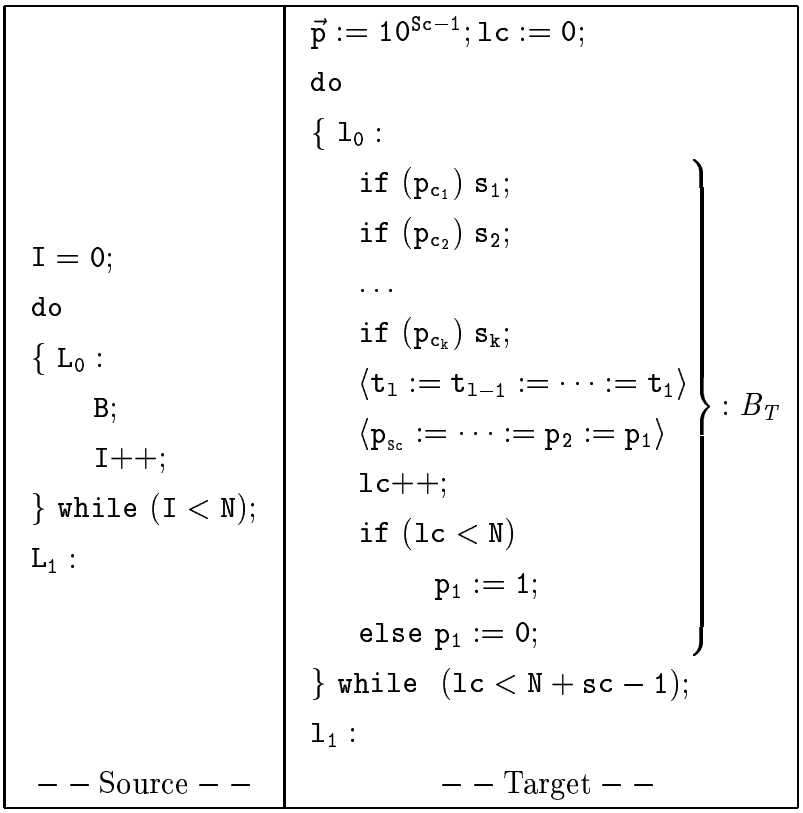

Figure 7.5: General form of a pipelined loop

the idle source transition to emulate the first $Sc - 1$ target iterations.

Obviously, the transition relation of a target iteration is different from the one of the source iteration. However, the use of the invariant makes the verification condition valid. To demonstrate this we use the source code and its corresponding target code, shown in Fig. 7.6. Let the invariant $\varphi_1 : pc = 1 \rightarrow t_1 = 5$ hold for the target code. From the figure we see that $\rho_1^T : (a' = t_1) \wedge (pc = 1) \wedge (pc' = 2)$, while $\rho_1^S : (A' = 5) \wedge (PC = 1) \wedge (PC' = 2)$. Let α be defined as: $(A = a) \wedge (PC = pc)$. Then, the verification condition $\varphi_1 \wedge \alpha \wedge \alpha' \wedge \rho_1^T \rightarrow \rho_1^S$ is valid. In this case, the value 5 was previously assigned to the temporary variable t_1 of the target system. This value is assigned to A in block one of the source program, while in block one of the target program, the value that is assigned to a is t_1. The invariant φ states that the value of

t_1 is the right value (5).

...	$\mathtt{t}_1 := 5$
$\mathtt{L}_1 : \mathtt{A} := 5$	...
$\mathtt{L}_2 :$	$\mathtt{l}_1 : \mathtt{a} := \mathtt{t}_1$
...	$\mathtt{l}_2 :$
	...
$--$ Source $--$	$--$ Target $--$

Figure 7.6: Using invariants to validate the translation - an example

We want to compute the invariant as allowed in step 2 of the AVALIDATE procedure (see page 31). Although we choose iteration number $i + Sc - 1$ of the target program to emulate source iteration i, the computations of source iteration i are actually spread over iterations $i, \ldots, i + Sc - 1$ of the target program. In other words, each target iteration emulates parts of different source iterations. The role of the invariant φ, is to characterize the state resulting from partial execution of an iteration (see Fig. 7.7).

Cycle	Operation	Invariant
1	*ld(1)*	$\varphi_1 : i = 1 \rightarrow t_3 = b[1]$
2	*w(1)* *ld(2)*	$\varphi_2 : i = 2 \rightarrow t_4 = b[1] \wedge t_3 = b[2]$
3	*add(1)* *w(2)* *ld(3)*	$\varphi_3 : i = 3 \rightarrow t_4 = b[2] \wedge t_3 = b[3] \wedge$
		$t_2 = b[1] + 5$

Figure 7.7: The States at the end of target prologue iterations

In the following sections, we describe a method for the automatic computation of the invariant φ.

7.2.3 Computing φ

Rather than using a single invariant, we distinguish between the versions of the invariant which apply to the various prologue, epilog, and kernel iterations. We call these invariants *software pipeline invariants*. We generate different versions of the invariant φ_i for $i = 0, \ldots, Sc-2, N, N+1, \ldots, N+Sc-2$, and a common version φ_{st}, corresponding to all steady-state iterations, which span the range $Sc-1 \leq i < N$. In this way, independently of the value of N, we always deal with $2 \cdot Sc - 1$ different versions of the invariant.

In order to compute the different versions of the invariant, we use a restriction of the notion of *symbolic evaluation* and *symbolic state,* as defined in [FS97]. A *symbolic state* is an assertion of the form

$$\varphi : \bigwedge_{v_i \in V} v_i = e_i,$$

where $v_i \in V$ are target system variables, and e_i are expressions.

Definition 3 *Symbolic Evaluation- Let V be a set of variables, φ be an assertion describing a symbolic state and, let ρ be a transition relation. The symbolic state, resulting from the application of the transition ρ to the symbolic state described by φ (also known as the* post-condition *of φ relative to ρ), is given by:*

$$\varphi \circ \rho \quad \triangleq \quad \exists V^- : (\varphi(V^-) \ \wedge \ \rho(V^-, V),$$

where V^- is another copy of the variables V, intended to capture their values before the transition is taken.

The algorithm in Fig. 7.8 successively computes the $2 \cdot Sc - 1$ different cases of the invariant φ_i, by symbolically applying the transition ρ_B, corresponding to a single execution of the loop's body B_T, to the previous symbolic state. The assertion φ_0 is computed, based on the initiation phase before the loop starts (see Section 7.4). We then proceed to compute $\varphi_1, \ldots, \varphi_{Sc-2}$, the invariants which hold at the beginning of each prologue iteration. The assertion φ_{st} holds at the loop steady state, while the assertions $\varphi_N, \ldots, \varphi_{N+Sc-2}$ hold at the beginning of the corresponding epilog iterations.

$$
\begin{array}{l}
\varphi_0 \ := \ \varphi := \ \mathit{Init} \ \wedge \ \vec{p} = 10^S c - 1 \\
\mathit{for}(i := 1;\ i \leq Sc - 2;\ i := i + 1) \\
\quad \{\varphi_i := \varphi := (\varphi \ \wedge \ lc + 1 < N) \circ \rho_B\} \\
\varphi_{st} := \varphi := (\varphi \ \wedge \ lc + 1 < N) \circ \rho_B \\
\mathit{for}(i := 0;\ i \leq Sc - 2;\ i := i + 1) \\
\quad \{\varphi_{N+i} := \varphi := (\varphi \ \wedge \ lc + 1 \geq N) \circ \rho_B\}
\end{array}
$$

Figure 7.8: Algorithm for computing φ

The condition $lc{+}1 < N$ added to φ in the computation of $\{\varphi_1, \ldots, \varphi_{Sc-2}, \varphi_{st}\}$, guarantees that the new value of $\mathtt{p}_1$ is 1. Similarly, the condition $lc + 1 \geqq N$ appearing in the computation of the invariants $\{\varphi_N, \ldots, \varphi_{N+Sc-2}\}$, guarantees that the new value of $\mathtt{p}_1$ is 0.

Following the AVALIDATE procedure, the verification condition to be validated is:

$$
VC_0 : \left(\begin{array}{l} \varphi \wedge \alpha \wedge \alpha' \wedge \rho_B \rightarrow \\ (pc' = 0 \wedge \varphi' \vee pc' = 1) \wedge (\rho^S_{L0} \vee Pres(V^S)) \end{array} \right)
$$

However, since we already split the invariant, when computing it, into $2{\cdot}Sc{-}1$ different cases, it is necessary to generate a similar number of verification conditions. These conditions, after appropriate simplification, are listed in Fig. 7.9.

using: $\mu \triangleq \alpha \wedge \alpha'$

Prologue verification conditions

$$
\begin{array}{lll}
VC_0: & \mu \wedge lc = 0 \wedge \varphi_0 \wedge \rho_B & \rightarrow \quad lc' = 1 \wedge \varphi'_1 \wedge Pres(V^S) \\
VC_1: & \mu \wedge lc = 1 \wedge \varphi_1 \wedge \rho_B & \rightarrow \quad lc' = 2 \wedge \varphi'_2 \wedge Pres(V^S) \\
\vdots & & \\
VC_{Sc-2}: & \mu \wedge lc = Sc-2 \wedge \varphi_{Sc-2} \wedge \rho_B & \rightarrow \quad lc' = Sc-1 \wedge \varphi'_{st} \wedge Pres(V^S)
\end{array}
$$

Steady state verification conditions

$$
\begin{array}{ll}
VC^a_{st}: & \mu \wedge Sc-1 \leq lc < N-1 \wedge \varphi_{st} \wedge \rho_B \quad \rightarrow \quad Sc-1 \leq lc' < N \wedge \varphi'_{st} \wedge \rho^S_{L_0} \\
VC^b_{st}: & \mu \wedge lc = N-1 \wedge \varphi_{st} \wedge \rho_B \quad \rightarrow \quad lc' = N \wedge \varphi'_N \wedge \rho^S_{L_0}
\end{array}
$$

Epilog verification conditions

$$
\begin{array}{lll}
VC_N: & \mu \wedge lc = N \wedge \varphi_N \wedge \rho_B & \rightarrow \quad lc' = N+1 \wedge \varphi'_{N+1} \wedge \rho^S_{L_0} \\
VC_{N+1}: & \mu \wedge lc = N+1 \wedge \varphi_{N+1} \wedge \rho_B & \rightarrow \quad lc' = N+2 \wedge \varphi'_{N+2} \wedge \rho^S_{L_0} \\
\vdots & & \\
VC_{N+Sc-2}: & \mu \wedge lc = N+Sc-2 \wedge \varphi_{N+Sc-2} \wedge \rho_B & \rightarrow \\
 & lc' = N+Sc-1 \wedge \ \wedge pc' = 1 \wedge \rho^S_{L_0} &
\end{array}
$$

Figure 7.9: Verification conditions for a pipelined loop

φ_0 :

$i_2 = lc \ \wedge \ i_1 = lc \ \wedge \ \vec{p} = (0,0,0,1)$

φ_1 :

$(\varphi_0 \wedge lc+1 < N) \circ \rho_B \quad \sim$

$t_3 = b[lc-1] \ \wedge \ i_2 = lc \ \wedge \ i_1 = lc-1 \ \wedge \ \vec{p} = (0,0,1,1)$

φ_2 :

$(\varphi_1 \wedge lc+1 < N) \circ \rho_B \quad \sim$

$t_4 = b[lc-2] \ \wedge \ i_2 = lc \ \wedge \ i_1 = lc-2 \ \wedge \ t_3 = b[lc-1] \ \wedge \ \vec{p} = (0,1,1,1)$

φ_{st} :

$(\varphi_2 \wedge lc+1 < N) \circ \rho_B \quad \sim$

$t_3 = b[lc-1] \ \wedge \ t_2 = b[lc-3]+5 \ \wedge \ t_4 = b[lc-2] \ \wedge \ i_1 = lc-3 \ \wedge \ i_2 = lc \ \wedge \ \vec{p} = (1,1,1,1)$

φ_N :

$(\varphi_{st} \wedge lc+1 \geq N) \circ \rho_B \quad \sim$

$t_3 = b[lc-1] \ \wedge \ t_2 = b[lc-3]+5 \ \wedge \ t_4 = b[lc-2] \ \wedge \ i_1 = lc-3 \ \wedge \ \vec{p} = (1,1,1,0)$

φ_{N+1} :

$(\varphi_N \wedge lc+1 \geq N) \circ \rho_B \quad \sim$

$t_2 = b[lc-3]+5 \ \wedge \ t_4 = b[lc-2] \ \wedge \ i_1 = lc-3 \ \wedge \ i_2 = lc-1 \ \wedge \ t_3 = b[lc-2] \ \wedge \ \vec{p} = (1,1,0,0)$

φ_{N+2} :

$(\varphi_{N+1} \wedge lc+1 \geq N) \circ \rho_B \quad \sim$

$t_2 = b[lc-3]+5 \ \wedge \ i_1 = lc-3 \ \wedge \ i_2 = lc-2 \ \wedge \ t3 = b[lc-3] \ \wedge \ t_4 = b[lc-3] \ \wedge \ \vec{p} = (1,0,0,0)$

Figure 7.10: Symbolic evaluation of φ - an example

7.3 An Example

In this section we demonstrate the algorithm to produce the software pipeline invariants of the example shown in Fig. 7.1 (the source) and in Fig. 7.4 (the target). The symbolic evaluation process of φ_i is listed in Fig. 7.10. The verification condition VC^a_{st}, for the steady state, as produced by SPV, (thus some simplifications are already applied) is shown below:

$$\alpha : I = lc - 3 \quad \wedge \quad A = a \quad \wedge \quad B = b \quad \wedge \quad PC = pc \quad \wedge$$
$$\alpha' : I' = lc' - 3 \quad \wedge \quad A' = a' \quad \wedge \quad B' = b' \quad \wedge \quad PC' = pc' \quad \wedge$$
$$\varphi_{st} \quad \wedge \quad 3 \leq lc < N - 1 \quad \wedge$$
$$\rho_B : \begin{cases} a' = a\ with([i1] : t_2) \quad \wedge \\ i'_1 = i_1 + 1 \quad \wedge \quad i'_2 = i_2 + 1 \quad \wedge \quad t'_3 = b[i_2] \quad \wedge \\ t'_2 = t_4 + 5 \quad \wedge \quad t'_1 = t_4 + 5 \quad \wedge \quad t'_4 = t_3 \quad \wedge \\ p'_4 = p_3 \quad \wedge \quad p'_3 = p_2 \quad \wedge \quad p'_2 = p_1 \quad \wedge \\ \quad p'_1 = ite(lc + 1 < N, 1, 0) \quad \wedge \\ lc' = lc + 1 \quad \wedge \quad pc' = ite(lc + 1 < N + 3, 0, 1) \end{cases}$$

$$\rightarrow$$

$$\rho_{L_0} : \begin{cases} A' = A\ with([I] : B[I] + 5) \quad \wedge \\ I' = I + 1 \quad \wedge \quad (I + 1 < N) \quad \wedge \quad \bigwedge \varphi'_{st} \quad \wedge \quad 3 \leq lc' < N \\ \quad \wedge \quad PC' = ite\ (I + 1 < N, 0, 1) \end{cases}$$

The SPV tool which we developed, computes the invariants and produces a set of verification conditions. The verification condition are automatically verified by CVC (CVC is a validity checker that was developed at Stanford University, [SBD02]). Since the basic blocks are usually small, no performance problems were encountered. The following section describes the SPV tool in detail.

7.4 SPV: Software Pipelining Validator

In this section we give an overview of the SPV tool. We intend to run the tool separately, for each code generator pass. Note that the code generator

of SGI-PRO64 has about 15 passes, some of which are performing the same type of optimization. Currently, we validate the software pipelining stage and produce the verification conditions for pipelined loops.

From LIR to a transition system - The tool input, source and target programs, uses syntax and semantics of the code generator internal language LIR (Low level Intermediate Representation) of the SGI-PRO64 compiler. The source program is the LIR before running software pipelining optimization and the target is the pipelined code. LIR represents the program as a list of pseudo machine instructions, but also includes annotations that describe the program as a flow graph. The nodes of the graph are basic blocks. The edges connect each block to its predecessors and successors. For each block, SPV produces the transition relation of the block, which has the form $\bigwedge_{v \in V} v' = Expression(V)$, i.e. each primed version of the system variables is equal to an expression over the unprimed version of all system variables. The LIR annotations are used by SPV as hints. The *expression* class (files: *expression.cc, expression.h)* enables the efficient production of the transitions, as well as substitution and some simplification. The substitution is used for the symbolic state evaluation. This class supports basic arithmetic and logic operations, *ite* operation as well as array lookup ([ ]) and update (*with).*

Producing the control abstraction - We use the annotations that describe the control flow, to produce the control abstraction. In most cases, block B_i in the target program, is mapped to a block with the same index in the source program.

Identifying loop linear inductive variables - The current preliminary version of the tool, produces the initial invariant of the software pipeline loop φ_0, by using the compiler annotations. In particular, registers and temporary variables which point to arrays, are annotated by the code generator. It is also possible to apply data-flow analysis methods, that are used by compilers (see for example [Muc97]), or to construct these invariants algorithmically (see [MP95] page 207).

Producing the data abstraction mapping - Mainly based on the code generator annotations.

Constructing φ for software pipelining loops - (*swp.cc, swp.h)*. SPV computes the assertions for the prologue, epilog and steady state, following the algorithm described in subsection 6.2.1. It uses the expression class to substitute and simplify expressions. A special simplification is performed by substituting constant p_i values, for the different pipeline stages. SPV outputs

$2 \cdot Sc - 1$ assertions to be proved. The tool is able to communicate with more than one verification engine (currently, ICS [FORS01] and CVC [SBD02]).
Validation using CVC - The produced verification conditions are all in the decidable logics of Presburger Arithmetic, arrays and uninterpreted functions. The output of SPV is a set of verification conditions which are fed into CVC. Then, CVC checks them and outputs either **Valid** or **Invalid**. It is a matter of less than a second, for CVC, to validate all verification conditions of the example in Fig. 7.1.

7.5 Conclusions

The SPV tool was used to validate a few optimized small programs. Each verification condition is verified separately and, its size is linear in the size of the basic block. However, basic blocks are relatively small (a few machine instructions), hence the validation process is very fast. The verification conditions produced by SPV belong to decidable logics, thus there validity can be algorithmically checked. Most of the other code generator optimizations are local to the block and, do not pose special problems. Software pipeline optimization is preceded by a loop unrolling pass, whose validation method is described in chapter 6, above. The main problems of validating the code generator optimizations, are solved by applying the two methods to the appropriate passes of the code generator.

Part III

Validating an Industrial Compiler

Chapter 8

The Programs and the Compiler

This research was done as part of the SafeAirII project of IST (*Information Society Technologies program of the European Union*), which aims towards the development of Advanced Design Tools for Safety Critical Software. The project is developed in an industrial context, intending to develop industrial tools and methods. Thus, we have chosen the WindRiver DiabData [Dia] compiler for the PowerPC [PPC] family of processors, a compiler which is widely used in the development of embedded systems. Typically, industrial compilers are proprietary software. This is a major obstacle for the code validation effort. Intermediate representation is not available to the user and a major part of the knowledge about optimizations is confidential. The compiler output is a standard object file format or assembly code, and standard debug information. When optimizations are activated, the debug information is only partially reliable. Information about the calling conventions, used in the code, is provided as part of the standard compiler documentation.

Since the PowerPC is a RISC (Reduced Instruction Set Computer) processor, the role of optimizations is very important, and the run time of the produced code is highly affected by the compiler. The compiler claims to perform a wide range of optimizations. These include target independent optimizations: tail recursion, inlining, argument address optimization, structure members to registers, range optimization, constant propagation, branch optimization, global common sub-expression elimination, loop strength re-

duction, loop count-down optimization, loop unrolling, unused assignment deletion, register coloring, use of scratch registers, loop invariant code motion and live-range analysis, as well as target dependent optimizations, peephole optimizations and basic reordering.

To overcome the obstacles caused by the compiler, we had to take advantage of the special type of program we handle. Though the programs are written in the HLL (High Level Language) C, they are automatically produced from programs written in a synchronous language. So we limit ourselves, in this part of the thesis, to handle C programs of a special type, called synchronous programs. To describe the formal semantics of these programs, a limited notion of transition systems- a singleton transition system, is offered.

We show that, in order to prove the validity of the translation of synchronous programs, a method which is simpler then the one described in the previous part of this work, can be used. The proof procedure, that is proposed in this part, has the advantage that it is capable of proving the correctness of a translation without previous knowledge about the optimization done by the compiler, and the compiler internal intermediate representation. Neither a construction of a control abstraction mapping nor of a local variable mapping is needed. Also, the proposed proof procedure does not require invariants construction. In the following chapters, we rebuild a theoretical framework that supports compressed transition systems, we define a reduced validation method, prove its soundness, and describe its implementation.

Chapter 9

Singleton Transition Systems

In part I we defined transition systems. Here, we specialize this general notion to deal with synchronous programs.

A transition system $\mathcal{S} = \langle V, O, \beta, \Theta, \rho \rangle$ is called a *singleton transition system* if it satisfies the following restrictions:

- Its transition relation has the form

$$\rho(V, V') : \pi = 0 \bigwedge_{v \in V - \{\pi\}} v' = Exp_v(V) \wedge \pi' = exit,$$

  where for every $v \in V$, Exp_v is a legal expression in the language $\mathcal{L}$.

- $\Theta \rightarrow \pi = 0$.

- $\beta : \pi \in \{0, exit\}$.Thus, such a system has only two observable locations: the entry and the exit locations, whose values are 0 and $exit$, respectively. Since these are the only values π can assume, β is equivalent to 1 ($true$).

Notation 4 *We use the notation $\rho(v)$, to stand for the term which is extracted from ρ for the variable v.*

The notation is naturally extended to expressions. For example: let $\rho : a' = b + 2 \wedge b' = 5$ then $\rho(a)$ is the expression $b + 2$, $\rho(b)$ is the expression 5 and $\rho(a + b)$ is the expression $(b + 2) + 5$.

Due to the limitation on the use of π, it may as well be omitted from the transition relation and, a short form of $\rho(V, V')$, $\bigwedge_{v \in V - \{\pi\}} v' = Exp_v(V)$, can be used. Also, since the component β is equivalent to 1 for all singleton transition systems, we omit β from the transition system, and get a shorter form: $\mathcal{S} = \langle V, O, \Theta, \rho \rangle$.

A *computation* σ,of a singleton transition system is a pair $\sigma = s_0, s_{exit}$, where $\langle s_0, s_{exit} \rangle \models \rho$. Obviously, a singleton transition system has only finite computations.

9.1 Correctness of Translations

As in part I, we use the notion of refinement in order to define a correct translation. We recall that having P^S be a source program with the transition system $\mathcal{S}^S$ and, P^T a target program with the transition system $\mathcal{S}^T$, then $\mathcal{S}^T$ refines $\mathcal{S}^S$, if for every computation of $\mathcal{S}^T$ there exists a computation of $\mathcal{S}^S$, with matching values of the observable variables at the observable states. If this refinement relation holds, we say that P^T is a correct translation of P^S.

In the case of source and target singleton transition systems, $\mathcal{S}^S = \langle V^S, O^S, \Theta^S, \rho^S \rangle$ and $\mathcal{S}^T = \langle V^T, O^T, \Theta^T, \rho^T \rangle$, to prove that $\mathcal{S}^T$ is a refinement of $\mathcal{S}^S$ is easier than the case of general block programs. The reason is that, by definition, a computation has only two states: the entry and exit states. Let $\sigma^T = s_0^T, s_{exit}^T$ be a target computation. We need to construct a data mapping α, which maps every target state to a source state. This data mapping, α, induces a pair of source states $\sigma^S = \alpha(s_0^T), \alpha(s_{exit}^T)$. If we prove that σ^S is indeed a source computation, it means that $\mathcal{S}^T$ refines $\mathcal{S}^S$. In order to show that σ^S is a computation of $\mathcal{S}^S$, we have to prove that the following holds:

$$\left\langle \alpha(s_0^T), \alpha(s_{exit}^T) \right\rangle \models \rho^S. \tag{9.1}$$

Hence, when both the source and the target systems are singleton transition systems, a simpler proof rule is sufficient, in order to prove that the target system is a refinement of the source system, and thus a correct translation.

Let $\mathcal{S}^S = (V^S, O^S, \Theta^S, \rho^S)$ and $\mathcal{S}^T = (V^T, O^T, \Theta^T, \rho^T)$ be the singleton transition systems that represent the source and the target programs, respectively. The following REDUCE procedure proves that $\mathcal{S}^T$ is a refinement of $\mathcal{S}^S$:

1. Construct a data mapping $\alpha : V_1 = v_1 \wedge \ldots V_i = v_i, \ldots, V_k = v_k$, where $O^S = \{V_1, \ldots .V_k\}$ are the source observable variables and, $O^T = \{v_1, \ldots .v_k\}$ are the target observable variables.

2. For each observable variable of the source system :$V_i \in O^S, 1 \leq i \leq k$, construct a verification condition VC_i :

$$\alpha \wedge v_i' = \rho^T(v_i) \wedge \alpha' \rightarrow V_i' = \rho^S(V_i)$$

3. Establish the validity of all the verification conditions.

In the following chapters we show that singleton transition systems provide an adequate semantics for high level (C), as well as low level (assembly) synchronous programs.

Chapter 10

Synchronous Programs

The source programs we handle are produced from a high-level synchronous language SCADE [SC9]. In this language, a *node* is a network of operators. A node has a formal interface (input and output variables) and local variables. The formal interface defines the observable part of the node. The node represents a computation of the values of the output variables as functions of the input variables. A SCADE program is a collection of nodes, which are activated by an external loop. Each node is translated into one compilation unit of the C language. The C code of the different nodes is compiled into a machine code. The code of each node usually contains no loops. In rare cases, it may contain loops with constant number of iterations which can easily be unrolled.

The papers [PSS98a, PSS99] describe a tool, CVT, which verifies the translation from SCADE to C. In our work, we concentrate on the compilation of the code of one node from C to machine code, in order to prove its correctness. For this reason, our notion of *synchronous program* relates to a program that represents one node, thus containing no loops. In this chapter we define two types of synchronous programs: synchronous machine code programs, and synchronous C programs. In the following chapter we show that synchronous programs can be represented by a singleton transition system, therefore the proof rule REDUCE can be used to validate the translations.

10.1 Synchronous Machine Code Programs

In the previous part of this work, we define block programs. there the smallest piece of code that interests us is a block. Here, we are concerned with a single instruction level. An *instruction* is a legal machine instruction of the underlying physical processor. Each instruction has a *label,* whose value is its index. There are a few types of instructions:

- For a variable `v` and an expression `Exp(V)` of the same type:

  `v := Exp(V)`

  is a *regular* instruction, also called an assignment instruction. `Exp(V)` can include calls to uninterpreted functions. $regular(Inst)$ holds iff $Inst$ is a regular instruction. In section 10.1.1 we describe regular instructions which have special importance.

- We also define two types of *branch* instructions:

  - For a non-negative integer `<label>`:

    `branch <label>`

    is an *unconditional branch* instruction. $branch(Inst, l)$ holds iff $Inst$ is a branch instruction with destination l.

  - For a non-negative integer `<label>` and a boolean expression `<condition>`:

    `branch <condition>,<label>`

    is a *conditional branch* instruction. $cond\_branch(Inst, Cond, l)$ holds iff $Inst$ is a conditional branch instruction, with condition $Cond$ and destination l.

The label in a branch instruction is the branch *destination* and its value is the index of an instruction.

A *machine code program* P is a sequence of *instructions* $P = Inst_{0,...}, Inst_n$. A branch to a label whose value is $l > n$, is interpreted as program termination. All destinations of this type assume a unique value $exit$, where $exit > n$.

Let $P = Inst_{0,...}, Inst_n$ be a machine-code program, then $Inst_0$ is the *entry* instruction. Each instruction $Inst_i$, $i = 0, \ldots, n$, of a program, may have one or two *successors,* denoted by $succ(i)$. If $Inst_i$ is an assignment

instruction, then $succ(i) = \{i+1\}$. If $Inst_i$ is an unconditional branch with destination j, then $succ(i) = \{j\}$. If an instruction $Inst_i$ is a conditional branch with destination j, then $succ(i) = \{j, i+1\}$. The successor of the last instruction, $Inst_n$, is $exit$. It is also a successor of any branch instruction with destination $l > n$.

The operands of an instruction of program P are either CPU registers or memory addresses. These comprise V, the set of *program variables*. We use the term *memory variable* to denote an operand which resides in the memory. These comprise the set of observable variables $O \subseteq V$.

Definition 5 *Synchronous machine code program - A machine code program $P = Inst_{0,}..., Inst_n$ is synchronous, if for every branch instruction of P with label i and destination j, we have $i < j$. This means that there are no backward branch instructions in the program and thus, no loops.*

Each machine code program P corresponds to a transition system $\mathcal{S}^P = < V, O, \beta, \Theta, \rho >$ such that:

- $V = \{v_1, ..., v_k\} \cup \{pc\}$ is a finite set of system variables. $\{v_1, ..., v_k\}$ are the program variables. $pc \in \{0, ..., n, exit\}$ is the control variable, whose value denotes the index of the instruction in which the control currently resides. When $pc = exit$, the program is terminated.
- O : The observable variables are all the memory variables.
- β : $pc \in \{0, exit\}$. Only the entry and exit locations are observable locations.
- $\Theta : pc = 0$
- ρ : The transition relation.

  Each instruction $Inst_i$ in the program is represented by an *instruction transition relation,* as follows:

  Let $inst_i$ be a regular instruction `v := Exp(V)`, then its instruction transition is

$$\rho_i(V, V') : v' = Exp(V) \wedge pc' = pc + 1 \wedge Pres(V - \{pc, v\})$$

Let $inst_i$ be a branch instruction `branch` <label>, then

$$\rho_i(V, V') : pc' = l \wedge Pres(V - \{pc\}).$$

Let $inst_i$ be a conditional branch instruction
`branch` <`condition`>,<`label`>, then

$$\rho_i(V, V') : pc' = ite(Cond, l, pc + 1) \wedge Pres(V - \{pc\}$$

The program transition relation[1] is $\rho : \bigvee_{i=0}^{n} (pc = i) \wedge \rho_i(V, V')$.

Claim 6 *Let $P = Inst_0, \ldots, Inst_n$ be a synchronous machine-code program, then all computations of P terminate.*

Proof. The pc value resides in a finite set $\{0, .., n, exit\}$ and is increasing along a computation. ■

We will show later, that every synchronous machine-code program corresponds to a "compressed" singleton transition system.

10.1.1 Special Machine Instructions

Most compilers use only a subset of the processor instruction set. Our implemented tool, as well as the formal semantics definition, consider only those instructions that are used by the compiler. We devote special attention to a few instructions which are used in a special way by the compiler, and which require a special treatment.

The Compare Instruction - The PowerPC processor has eight condition registers (see [PPC]), each consisting of a triple of bits. The instruction `compare i,`<`val1`> <`val2`> compares the values of <`val1`> <`val2`> and assigns values to condition register number i. The first bit of the register is set if <`val1`> is greater than <`val2`>, the second bit is set if <`val1`> is smaller than <`val2`>, and the third bit is set if both values are equal. A bit in the condition register that was not set, is reset. In the transition system of a machine program, every bit of a condition register is represented by a boolean variable. The set of condition registers is represented by a two dimensional array of boolean type elements $cr[i, j]$ (also denoted as $cr_{i,,j}$),

[1] When it is obvious from the context, the conjuct $\Pr es(U)$ is omitted form the text.

where $i = 0 \ldots 7$ and $j = 0, 1, 2$ which stand for eight triples. The value of each array element is a boolean expression. The `compare` instruction assigns value to one condition register, which consists of three elements of the array. For example, the transition semantics for the instruction: `compare 1,0,r0` is:

$$cr'_{1,0} = (r_0 < 0) \wedge (cr'_{1,1} = (r_0 > 0) \wedge (cr'_{1,2} = (r_0 = 0)))$$

A use of the `compare` instruction is demonstrated in Fig. 10.1. The C source code that corresponds to this example is : `v1 := (v2+3=1)`. The corresponding transition relation of the program in this figure is:

$$(pc = l_1) \wedge (v'_1 = ite((v_2 + 3 = 1), 1, 0)) \wedge (r'_0 = v_2 + 3) \wedge (pc' = l_3) \wedge$$
$$(cr'_{0,0} = (1 < v_2 + 3)) \wedge (cr'_{0,1} = (1 > v_2 + 3)) \wedge (cr_{0,2} = (0 = v_2 + 3))$$

Bit Instructions - Other interesting instructions are those that manipulate bits. For example, the instruction `rlwinm` (Rotate Left Word I bits, then And with Mask) is used by the compiler to extract some bits from an integer, or a condition register. Its semantics is defined, using the functions:

$$bit\_array : array\ of\ booleans \longmapsto integer$$
$$extract\_bits : integer \times integer \longmapsto integer$$

However, when these functions are called in a special sequence with predefined values, their common semantics becomes a specific known arithmetic function. For example, the code in Fig. 10.2 computes the same value for v'_1 as the code in Fig. 10.1. The transition semantics of this machine code program is:

$$v'_1 = extract(bit\_array(cr\ with(\ [2] : (v_3 + 3 = 1))), 2) \wedge \ldots,$$

which is equivalent to:

$$v'_1 = ite((v_3 + 3 = 1), 1, 0) \wedge \ldots$$

```
l1:   add       r0,v2,3     add 3 to v2 and put the result in integer register r0
      compare   cr0,r0,1    compare r0 to 1 and, set the three bits of condition
                            register cr0
      branch    cr0.2 l2    branch to l2 if bit no. 2 of cr0 is set
      move      v1,0        move 0 to v1
      branch    l3
l2:   move      v1,1        move 1 to v1
l3:
```

Figure 10.1: Machine code example - 1

```
l1:   add       r0,v2,3     add 3 to v2 and put the result in integer register r0
      compare   cr0,r0,1    compare r0 to 1 and, set the three bits of condition
                            register cr0
      move      r1,cr       move the content of the condition registers to
                            integer register r1
      rlwinm    v1,r1,2,1   rotate the content of r1 into v1, so that bit no. 2
                            moves to the least significant bit and extracts it.
```

Figure 10.2: Machine code example - 2

10.2 Synchronous C Programs

Synchronous C programs are C programs without loops. The following statements are not allowed: *while, for, do-while* and *goto*. The syntax of synchronous C programs is defined as follows:

Let V be a set of integer variables and, $O \subseteq V$ is a set of memory variables, d_O is a sequence of declarations of the memory variables and, d_L is a sequence of declarations of all the other variables. $v \in V$ is a variable, P_1 and P_2 are programs and $Exp(V)$ is any expression over V. The syntax[2] of a program P is defined as follows:

$P ::= \{d_O \mathbf{\{} d_L, S_1, \ldots, S_n \mathbf{\}}\}$
$S_i ::= Assignment \mid if \mid if-else \mid else \mid end$
$Assignment ::= v := Exp(V);$
$if ::=$ `if` $(Exp(V));$
$if-else ::=$ `if` $(Exp(V))$ `then`
$else ::=$ `else`
$end ::= \}$

It is also required that every if, and $if-else$ has its corresponding end, and that there are no redundant end. However, we assume that all programs we handle are syntactically correct. The semantic definition of synchronous C programs is the usual semantics for the C language.

Definition 7 *Synchronous program - A synchronous program is either a synchronous machine-code program or a synchronous C program.*

Theorem 8 *Let P be a synchronous program (either machine-code or C), then the semantics of P can be expressed by a compressed transition system.*

The ANNOTATION algorithm described in section 11.2, constructs a compressed transition system for a synchronous machine-code program. An algorithm to produce a compressed transition system for a synchronous C program, is described as well (section 11.4). The correctness of these algorithms proves theorem 8.

[2]Unlike C, we use := to stand for an assignment, in order to distiguish between an assignment and an equality.

In the rest of this thesis we always handle synchronous programs. For brevity, we omit the word "synchronous".

Chapter 11

Compressing a TS into a Singleton TS

11.1 Root Substitution

We apply forward analysis to a synchronous program $P = Inst_0, \ldots, Inst_n$, with a set of variables, $V = \{v_0, \ldots v_k\}$, in order to produce its corresponding singleton transition relation. The transition relation is produced incrementally. At each step, one instruction is considered, and a compressed transition relation is produced, based on the previously produced transition relation. When the instruction is an assignment instruction the step performs *root substitution*.

Definition 9 *Root substitution - Let* $\rho^A : \bigwedge_{v \in V} v' = E_v$ *be a singleton transition relation and,* $Inst_i : x := Exp^i(V)$, *an assignment instruction of a program* P, *where* $0 \leq i \leq n$. *Let* $x \in V$, *be a variable of* P. *Then the singleton transition relation* ρ^B, *resulting from applying root substitution on* ρ^A *and* $Inst_i$ *is:*

$$\rho^B : \rho^A \circ [\mathtt{x} := \mathtt{E}] \triangleq \bigwedge_{v \neq x} v' = E_v \wedge x' = \rho_A(E)$$

P : 0 : a := b + 1; 1 : b := a * 3; 2 : a := 2;		
$\rho^0 =$		$a' = a \wedge b' = b$
$\rho^1 =$	$\rho^0 \circ [a := b + 1]$:	$a' = b + 1 \wedge b' = b$
$\rho^2 =$	$\rho^1 \circ [b := a \cdot 3]$:	$a' = b + 1 \wedge b' = (b + 1) \cdot 3$
$\rho^3 =$	$\rho^2 \circ [a := 2]$:	$a' = 2 \wedge b' = (b + 1) \cdot 3$

Figure 11.1: Root substitution applied to program P

For brevity we use $\rho \setminus x = \bigwedge_{v \neq x} v' = E_v$. Fig. 11.1 demonstrates a root substitution applied iteratively on the transition relation and the instructions of a synchronous machine-code program, starting from $\rho^0 : V' = V$. ρ^{i+1} is the result of root substitution of ρ^i with $Inst_i$, for $i = 0, \ldots, 2$. Note that ρ^3 is the transition relation of the program P.

In the following sections we describe the ANNOTATION algorithm, which extends root substitution on assignment instructions, to handle branch instructions as well. In the description of the algorithm, as well as in the definition of root substitution, we assume that only one assignment occurs in each machine-instruction. Extending the algorithm to handle multiple assignment is straightforward, and is not described here, for simplicity.

11.2 The ANNOTATION Algorithm

The ANNOTATION algorithm is used to produce a compressed transition system for a machine-code program. Let $P = Inst_0, Inst_1, \ldots, Inst_n$ be a program, where each $Inst_i, i = 0, \ldots, n$ is a machine instruction. Assume that $V = v_1, ..., v_k$ are the program variables. The ANNOTATION algorithm computes the annotations $Ann(0), ..., Ann(n)$ and the compressed transition relations $\rho^0, \rho^1, ..., \rho^n, \rho^{n+1}$, inductively. For each instruction $i = 0, ..., n$, the annotation $Ann(i)$ expresses the condition under which instruction $Inst_i$ is executed, in terms of the input variables (the values of the variables before the program is started). The transition relation ρ^{i+1} is the one which corresponds to the program $P^{i+1} = Inst_0, \ldots, Inst_i$, where ρ^{n+1} is the transition relation of the whole program. The initial relation ρ^0 has the trivial form $v'_1 = v_1 \wedge \cdots \wedge v'_k = v_k$. For variable $x \in V$ and $i \geq 0$, we denote by $\rho^i(x)$ the value of x in terms of V^0, as determined by ρ^i. For example, using the tran-

sition relation of Fig. 11.1, it follows that $\rho^3(a) = 2$ and $\rho^3(b) = (b+1) \cdot 3$. This notation can be extended to expressions, including boolean expression. Thus $\rho^3(a+b) = (b+1) \cdot 3 + 2$.

The ANNOTATION Algorithm - This algorithm scans the program from $Inst_0$ to $Inst_n$, in order to compute ρ, the transition relation of program P.

1. Initially:
   $Ann(0) = true,\ Ann(1) = \ldots = Ann(n) = false,\ \rho^0 : V' = V, i = 0.$
2. Examine $inst_i$,
   (a) If $inst_i$ is an assignment instruction: `i:x := Exp;`
   do:
   $Ann(i+1) := Ann(i+1) \vee Ann(i)$.
   $\rho^{i+1} : \rho^i \setminus x \wedge x' = ite(Ann(i), \rho^i(Exp), \rho^i(x))$
   $i := i + 1$.
   (b) If $inst_i$ is a conditional branch instruction :
   `i: branch <Cond> j;` (where $j > i$).
   do:
   $Ann(i+1) := Ann(i+1) \vee (Ann(i) \wedge \neg\rho^i(Cond))$.
   $Ann(j) := Ann(j) \vee (Ann(i) \wedge \rho^i(Cond))$.
   $\rho^{i+1} : \rho^i$.
   $i := i + 1$.
   (c) If $inst_i$ is an unconditional branch instruction :
   `i: branch j;`
   do:
   $Ann(j) := Ann(j) \vee Ann(i)$.
   $\rho^{i+1} : \rho^i$.
   $i := i + 1$.
   (d) If $i > n$ (the *exit* location is encountered) , let $\rho : \rho^i \wedge pc' = exit$, and exit from ANNOTATION.

11.3 Soundness of ANNOTATION

The annotation algorithm computes a compressed transition relation ρ. We claim that this transition relation is "equivalent" to the transition relation of program P. This is what theorem 10 states.

Theorem 10 *Let $P = inst_0, \ldots, inst_n$ be a machine-code program and let $S = \langle V, O, \beta, \Theta, \rho \rangle$ be its transition system. Let ρ^C be the compressed transition relation constructed by the* ANNOTATION *algorithm. Let s_0 and s be two V-states, $s_0[pc] = 0$, then $\langle s_0, s \rangle \models \rho^C$ iff there exists a computation of P, $\sigma : S_0, \ldots, S_m$ such that the states S_0 and S_m are equal to s_0 and s, respectively.*

Lemma 11 *Let $\sigma : s_0, \ldots, s_m$ be a computation of a synchronous machine code program P. Let inst be a machine instruction then σ is a computation prefix of the program $P' = P, inst$ (i.e., there is a computation of the program $P, inst$ whose prefix is σ).*

Lemma 12 *Let $\sigma : s_0, \ldots, s_m$ be a computation prefix of a synchronous machine code program $P = inst_0, \ldots, inst_n$, then σ is a computation of P iff $s_m[pc] > n$*

Lemma 13 *Let t_0, t_1 and t_2 be states of a set V. Let ρ^1 be transition relation in a compressed form, $x \in V$ and E an expression of the variables of V. Let $\langle t_0, t_1 \rangle \models \rho^1$ and $\langle t_0, t_2 \rangle \models \rho^1 \setminus x \wedge x' = E$, then $\langle t_1, t_2 \rangle \models x' = E \wedge Pres(V - \{x\})$.*

Proof. Direction one: We start by assuming that $\langle s_0, s \rangle \models \rho^C$. We will show how to construct a P-computation $s_0, \ldots, s_m$, such that $s_m = s$.

Let $P = \{inst_0, \ldots, inst_n\}$ be a machine-code program with a corresponding transition system $S^P = \{V, \Theta, O, \rho\}; \rho : \bigvee_{i=0}^{n} (pc = i) \wedge \rho_i$. Let $\rho^0, \ldots, \rho^{n+1} = \rho^C$ be the sequence of relations obtained from the algorithm. We define $\sigma^A : t_0, \ldots, t_n, t_{n+1}$; a sequence of states on the set of variables $V^t = V \cup \{Ann, L\}$ for which $t_0 \Downarrow_V = s_0$, $t_0[Ann] = 1, 0, \ldots, 0$, $t_0[L] = 0$, $\langle t_0 \Downarrow_{V-\{pc\}}, t_i \Downarrow_{V-\{pc\}} \rangle \models \rho^i$, and $\langle t_i \Downarrow_{\{Ann,L\}}, t_{i+1} \Downarrow_{\{Ann,L\}} \rangle \models \rho^A$, where ρ^A is a transition relation on $\{Ann, L\} \cup V$, induced by the ANNOTATION algorithm and defined as follows:

$$
\begin{array}{llll}
\rho^A\colon & L' & = L+1 & \wedge \\
 & Ann' & = case & \\
 & & & regular(inst_L) : Ann\ with([L+1] : Ann[L] \vee Ann[L+1]) \\
 & & & branch(Inst_L, j) : Ann\ with([j] : Ann[L] \vee Ann[j]) \\
 & & & branch\_cond(Inst_L, Cond, j) : \\
 & & & \quad Ann \quad with([L+1] : \quad Ann[L] \wedge \neg\rho^i(Cond) \vee Ann[L+1]) \\
 & & & \qquad\quad with([j] : \qquad Ann[L] \wedge \rho^i(Cond) \vee Ann[j]) \\
 & & & 1 : Ann \\
 & & endcase &
\end{array}
$$

In the proof we use Ann^i to notify $t_i[Ann]$. Note that $t_i[L] = i$.

We inductively construct a sequence $s_0, \ldots, s_m$ of V-states, induced by the sequence $t_0, \ldots, t_{n+1}$. The first state is s_0. Next assume that we have already constructed the prefix $\sigma_j : s_0, \ldots, s_k; 0 \leq k < m$ corresponding to t_j. If $j > n$ we are done. Otherwise, we consider $Ann^j(j)$. If $Ann^j(j) = 0$ then we define $\sigma_{j+1} = \sigma_j$. If $Ann^j(j) = 1$, we define $s_{k+1} \Downarrow_{V-\{pc\}} = t_{j+1} \Downarrow_{V-\{pc\}}$, $s_{k+1}[pc] = N$ such that $N \geq j+1$, $Ann^{j+1}[N] = 1$ and $Ann^{j+1}[l] = 0$; forall l such that $j < l < N$

We use the notation ρ^{P^j} to denote the transition relation that corresponds to program P^j.

σ_{n+1} is a computation of P-

In order to show that the sequence σ_{n+1} constructed in the previous part is a computation of the program P, we'll prove, by induction on j, that for every $j : 0 < j \leq n+1$

1. The sequence $\sigma_j : s_0, \ldots, s_k$ is a computation of P^j, and $Ann^j(s_k[pc]) = 1$.

2. For all i such that $i > j$ and $i \neq s_k[pc]$, holds: $Ann^j[i] = 0$.

For the base case, $j = 0$, we have $\sigma_0 = s_0, Ann^0 = (1, 0, \ldots, 0), s_0 \models \Theta$ thus $s_0[pc] = 0$. Now we assume that the claim holds for j and for the sequence $\sigma_j : s_0, \ldots, s_k; 0 \leq k < m$. Let's consider the sequence σ_{j+1}.

Case 1: $Ann^{j+1}(j+1) = 0$. From the construction we get that $\sigma_j = \sigma_{j+1}$. According to lemma 11, σ_{j+1} is a computation prefix of program P^{j+1}. Also $s_k[pc] > j$ because σ_j is a computation of P^j(by induction hypothesis). Since $Ann^{j+1}(j+1) = 0$, we conclude that $s_k[pc] > j+1$ (from the construction), and σ_{j+1} is a computation of P^{j+1}. Substituting $Ann^{j+1}(j+1) = 0$ in ρ^A

shows that $Ann^{j+1} = Ann^j$. Consequently, part 2 of the induction hypothesis holds.

Case 2: Assume that $Ann^{j+1}(j+1) = 1$. According to the induction assumption $s_k[pc] = j+1$ and for all $i > j+1$ holds $Ann^j(i) = 0$. Consider the instruction $inst_{j+1}$.

Case 2.1: Let assume that $inst_{j+1}$: `x := E` is a regular instruction ($regular(inst_{j+1})$).
Substituting $inst_{j+1}$ in ρ^A, yields $Ann^{j+2} = A^{j+1} with([j+2] : 1)$. Thus, by construction, $s_{k+1}[pc] = j+2$. Induction hypothesis (2) implies that for all $i > j+2$ holds $Ann^{j+2}(i) = 0$, also $Ann^{j+2}(s_{k+1}[pc]) = 1$. So part two of the induction hypothesis holds. In addition,

$$\rho^{P^{j+1}} = \bigvee_{i=0}^{j+1} pc = i \wedge \rho_i. \tag{11.1}$$

Thus it is sufficient to show that $\langle s_k, s_{k+1}\rangle \models pc = j+1 \wedge (x' = E \wedge Pres(V - \{x\}) \wedge pc' = j+2$, which we get by extracting the disjunct for $j+1$, replacing ρ_{j+1} by $(x' = E \wedge Pres(V - \{x\}) \wedge pc' = pc+1)$ and substituting $[pc \longleftarrow j+1], [pc' \longleftarrow j+2]$ in equation 11.1. We consider t_{j+1}, t_{j+2}, and

$$\langle t_0 \Downarrow_{V-\{pc\}}, t_{j+1} \Downarrow_{V-\{pc\}}\rangle \models \rho^{j+1} ,$$
$$\langle t_0 \Downarrow_{V-\{pc\}}, t_{j+2} \Downarrow_{V-\{pc\}}\rangle \models \rho^{j+1} \setminus x \wedge x' = E \wedge Pres(V - \{x, pc\}) ,$$

from the definition of ρ^{j+2} for a regular instruction. We extract from the t states the s projection, substitute $s_{k+1}[pc] = j+2$ and based on lemma 13 we get that $\langle s_k, s_{k+1}\rangle \models (pc = j+1) \wedge (x' = E) \wedge Pres(V) \wedge (pc' = j+2)$.

Case 2.2: Assume that $branch\_cond(inst_{j+1})$ ($inst_{j+1}$: `branch <cond> <label>`,where the value of `<label>` is l). In case that $t_{j+1}(cond) = 1(true)$, The value of the annotations array at state t_{j+2} is $Ann^{j+2} = Ann^{j+1}$ $with$ $([l] : 1)$. Obviously part two of the induction hypothesis holds. According to the construction $s_{k+1}[pc] = l$, and $\rho^{j+2} = \rho^{j+1}$ (by definition of ANNOTATION), which implies that $t_{j+2} \Downarrow_V = t_{j+1} \Downarrow_V$,thus $s_k \Downarrow_{V-\{pc\}} = s_{k+1} \Downarrow_{V-\{pc\}}$. We now look at ρ_{j+1} and substitute $s_{k+1}[cond] \longleftarrow 1, s_k[pc] \longleftarrow j+1$ and $s_{k+1}[pc] \longleftarrow l$. We get $\rho_{j+1} : pc = j+1 \wedge pc' = l$. Thus $\langle s_k, s_{k+1}\rangle \models \rho^{P^{j+1}}$.

In case that $t_{j+1}(cond) = 0$, we get $Ann^{j+2} = Ann^{j+1}$ $with([j+2] : 1)$. $s_{k+1}[pc] = j+2$ holds by construction, and for every $i > j+2$ holds $Ann^{j+2}[i] = 0$ and $Ann^{j+1}[s_{k+1}(pc]) = 1$ This proves part 2 of the induction hypothesis. As in the case of $t_{j+1}(cond) = 1(true)$, we get that $s_k \Downarrow_{V-\{pc\}} =$

$s_{k+1} \Downarrow_{V-\{pc\}}$, and for the instruction transition relation ρ_{j+1}, we get ρ_{j+1} : $pc = j+1 \wedge pc' = j+2$, implying that $\langle s_k, s_{k+1} \rangle \models \rho_{j+1}$ and thus $\langle s_k, s_{k+1} \rangle \models \rho^{P^{j+1}}$.

Case 2.3: The case of $branch(inst_{j+1})$ (`branch <label>`)is the same as the previous case, when substituting the constant value $1(true)$ for `<cond>` thus previous reasoning is applicable ■

Proof. Direction two: In order to prove that for every computation of P the compressed computation produces a correct final state, we rely on the fact that machine code programs are decidable, and thus have only one computation for each initial state. We prove by contradiction.

Let $\sigma : S_0, \ldots, S_m$ be a computation of the program P. Let $\langle s_0, s \rangle \models \rho^C$, where $s_0 = S_0$ and ρ^C is the compressed transition relation constructed by the ANNOTATION algorithm. We will show that $S_m = s$. Assuming that $S_m \neq s$ leads to the conclusion (based on the proof of direction one) that there exists another computation of P, $\sigma' : S_0, \ldots, S_n$ such that $S_n = s$. This contradicts the fact that P has only one computation which starts from the initial state S_0. Therefore the assumption is wrong and $S_m = s$. ■

11.4 Producing a Compressed TS for C

In order to produce a compressed transition system for a synchronous C program, we use the abstract data type *Stack*, which is a stack of boolean expressions and supports the following operators: $push(Exp)$ (pushes a boolean expression on the stack), *pop* (pops a boolean expression from the stack) and $Exp : tos$ (returns the boolean expression which is on Top Of Stack).

Let $P = \{d_O\{d_L, S_0, ..., S_n\}\}$ be a syntactically correct synchronous C program, then the following algorithm constructs a compressed transition relation for P. The algorithm reads the statements, one by one.

1. Initially the stack contains one element and $tos = true$. ρ^0 has the trivial form $v'_1 = v_1 \wedge \cdots \wedge v'_k = v_k$.

2. Considering S_i

   (a) Let S_i be an *if* statement: `if` $(Cond)$ then:
   $(push(tos \wedge Cond))$, where $Cond$ is expressed in terms of the input variables and $\rho^{i+1} = \rho^i$.

(b) Let S_i be an *end* statement then:
an expression is popped off of the stack and $\rho^{i+1} = \rho^i$.

(c) Let S_i be an `if` $(Cond) - else$ statement then:
$push(\neg Cond \wedge tos)$ and $push(Cond \wedge tos)$, where $Cond$ is expressed in terms of the input variables and $\rho^{i+1} = \rho^i$.

(d) Let S_i be an *assignment* statement $S : x := Exp(V)$ then
if tos is the constant $true$, a simple root substitution is invoked for the assignment statement: $\rho^{i+1} = \rho^i \circ [x := Exp(V)]$, otherwise $\rho^{i+1} = \rho^i \ with \ ite(tos, \ Exp(V^i) \ , x)$.

3. If $i = n$ exit, and let $\rho^P = \rho^i$, otherwise $i := i + 1$.

Starting from $\rho^0 : Pres(V', \mathcal{V})$, the algorithm terminates with ρ^n, which is the compressed transition relation of P. We define the semantics of the synchronous C program, P to be expressed by the transition system $S^P = \{V, O, \Theta, \rho^P\}$ where V, O and Θ are defined by d_O.

Chapter 12

The MCVT Tool

12.1 General Architecture of the Tool

MCVT, machine-code validation tool, was developed to validate the translation of synchronous programs. The source programs are C programs, produced automatically from SCADE synchronous programs. The target program is binary optimized code, produced by the commercial compiler Diab-Data of WindRiver, and is intended to run on a PowerPC computer. MCVT accepts the source and target program and checks that the target is a correct translation of the source. If so, it prints "Valid", otherwise it prints "Invalid". The whole process is fully automatic, though it might be long.

In this section we describe the general architecture of MCVT. In the sections that follow more details are given.

- **Producing a compressed transition relation -** This is the main component. It contains the producers of the common semantics, for both the source (synchronous C) and target (synchronous machine-code) programs. It consists of the following six sub-components: C and machine code parsers, root-substitution module (a common package to handle both machine-code and C programs), the *Annotation* algorithm, a formula simplifier and a code-pattern recognizer.

- **Data abstraction mapping -** This mapping is deduced from the debug information encapsulated in the binary file. It is also used to

decompose the verification condition, so each variable is checked separately.

- **Decomposition -** After having the two compressed transition relations and the data abstraction mapping , a verification condition is produced and then, decomposed into one assertion for every variable (see section 9.1).

- **Identifying code patterns -** Special code sequences are simplified. In some cases the compiler uses a fixed sequence of machine instructions to implement a commonly used, predefined C statement. We built a set of rules which allows the substitution of higher level arithmetic operations for those specific instruction sequences. The validity of these rules can be formally proved. However, this work was excluded from this thesis (See section 12.4).

- **Adaptation for CVC -** CVC has some limitations that enforce strengthening of the verification conditions with assertions. For example, CVC does not support integer arithmetic. Thus MCVT further processes the created formulas, so as to conform with CVC's supported theories.

- **The verifier -** CVC, a Cooperating Validity Checker [SBD02], developed at Stanford University, is invoked to establish the validity of the verification conditions. CVC decides logical validity of quantifier-free formulas in classical first-order logic with equality, enriched with certain background theories. In the current release, the background theories are for

  - linear real arithmetic (operators are addition, subtraction, less-than, and multiplication and division by a constant)
  - arrays (operations are reading from an array and updating an array at a given index)
  - inductive data types (e.g., lists and trees; records and tuples are special cases)

12.2 Compact Expression Representation

Using a compressed transition system has the disadvantage of producing huge expressions in the transition relation. For example, the compressed transition relation of the following machine code program:

```
P :
0 : ro := v + 5;
1 : r1 := r0 * 10;
2 : v := r0 + r1;
```

is $\rho^P : r_0' = v + 5 \wedge r_1' = (v + 5) \cdot 10 \wedge v' = v + 5 + (v + 5) \cdot 10$. However, internally, each expression has only one copy and, a pointer to this copy is produced whenever the expression is used. Thus, the transition relation ρ^P of program P is represented by the compact tree that is shown in Fig. 12.1.

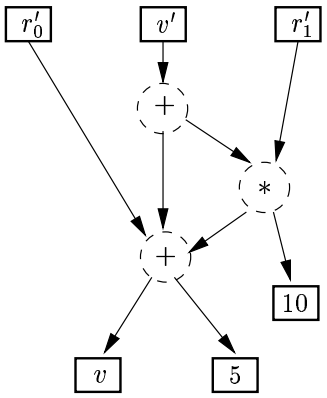

Figure 12.1: Compact expressions tree

12.3 The Formulas' Simplifier

The annotations that are produced for machine-code programs tend to be big, since each branch instruction adds a term. However, in many cases significant simplifications are possible. A common pattern in the machine-code is the following one:

```
l0:  branch <cond> l2;
l1:  assignment instructions...
     branch l3;
l2:  assignment instructions...
l3:
```

Such a pattern is the result of an $if - else$ statement in the source C code. Let $Ann(l_0) = true$. Then, $Ann(l) = \neg cond$, $Ann(l_2) = cond$ and $Ann(l_3) = cond \vee \neg cond = true$. In order to simplify the annotations, the condition expressions are handled symbolically. MCVT replaces every condition by a proposition symbol and simplifies the resulting formulas by resolution. The resolution is not used to validate the formula, but rather to extract an equivalent smaller one. A BDD simplifier could be used as well.

DNF formulas - Let $p_1, p_2, \ldots$ be a set of propositions. The set $\mathcal{L}$ of *literals* is

$$\mathcal{L} ::= p \mid \neg p \mid true \mid false.$$

The $\mathcal{C}$ set of *clauses is*

$$\mathcal{C} ::= l_1 \wedge l_2 \wedge \ldots \mid true \mid false, l_i \in \mathcal{L}.$$

The set $\mathcal{F}$ of DNF - *Disjunctive Normal Form* formulas is

$$\mathcal{F} ::= c_1 \vee c_2, \ldots \mid true \mid false, c_i \in \mathcal{C}.$$

DNF resolution [Rob65, BA01] - For clauses C, C' and a literal l, let $C = l \wedge C'$ be a clause, then $C - l = C'$. An empty clause is equivalent to $true$. If a formula contains a $true$ clause, the whole formula is equivalent to $true$. Let $F = C \vee F'$ be a DNF formula, then $F - C = F'$ and $F = F' + C$. An empty formula is equivalent to $false$. The resolution is based on the observation that if

$$C_1, C_2 \in F, l \in C_1, \neg l \in C_2, C_1 - l = C_2 - \neg l,$$

then

$$F \approx F - C_1 - C_2 + (C_1 - l).$$

12.4 The Code Pattern Recognizer

The optimizing compiler tries to choose the best machine-code sequence to perform a high-level language operation. In some cases, it is a sequence of bit manipulation instructions. These sequences are known patterns. We do not want to use the bit manipulation semantics because it complicates the produced transition relation. Therefore, MCVT recognizes these patterns, when they are used in the target machine-code and, replaces them with equivalent high-level operations. Following is one example of a pattern that is recognized by MCVT. Another example can be found in section 10.1.1.

Let the source C program contain the line: `A := (I=0);`. In order to avoid branch instructions which are expensive in terms of performance, the compiler produces the following instruction sequence:

r_0`:=cnlzw I;`	Count the Number of Leading Zeros of I and assign to r_0.
`A:=rlwnm` $r_0, 5, 1$	Rotate r_0 by 5, then extract the first bit.

The execution of these two instructions assigns `A` the value one if `I` equals zero and, zero otherwise. This instruction sequence is identified and replaced by its higher level semantics: $A' = ite(I = 0, 1, 0)$.

12.5 The Data Abstraction

The variables of the source C program are local and global ones. The target program variables are the machine registers, as well as the memory variables, which are referred to as memory addresses. The target system observable variables, are decided by the compiler memory allocation scheme and the calling conventions. At this stage of the research we assume that only the memory is observable and thus, only memory variables are observable. Also, no consistency check is done, to validate that the calling conventions are correctly implemented in the target code. Another assumption relates to the variables' type. We assume that all variables are of integer type and, that all integers are of the same size.

Registers of the target system, as well as local variables of the source system, are not part of the transition system, due to the use of compressed transition relation. Thus, in order to construct the data abstraction, it is

sufficient to map the memory variables of the source system, to their memory addresses. MCVT reads the debug information from the binary code and, uses it to produce the required mapping. Generally, the debug information of an optimized code is not necessarily correct. Yet, it is correct for memory variables at the entry and the exit points of the program. In any case, the soundness of the proof does not depend on the correctness of this information. That is, erroneous information could not lead to false positive conclusion.

12.6 Decomposing the Verification Conditions

Based on the method described in section 9.1 and, by using the data abstraction mapping, it is possible to decompose the verification conditions, so as to have one verification condition for each observable variable. We demonstrate this by an example.

Let the source transition relation be $A' = B + 1 \wedge B' = ite(A > 5, 1, 2)$, the target transition relation be

$$mem'_0 = mem_4 + 1 \wedge mem'_4 = ite(\neg(mem_0 \leq 5), 2, 1),$$

and the data abstraction mapping be the following: $mem_0 = A \wedge mem_4 = B$, then the decomposition module produces the following two verification conditions:

$$\begin{aligned} &mem'_0 = A' \wedge mem_0 = A \wedge mem'_4 = B' \wedge mem_4 = B \wedge \\ &mem'_0 = mem_4 + 1 \rightarrow A' = B + 1 \end{aligned} \tag{1}$$

$$\begin{aligned} &mem'_0 = A' \wedge mem_0 = A \wedge mem'_4 = B' \wedge mem_4 = B \wedge \\ &mem'_4 = ite(mem_0 \leq 5,\ 2,\ 1) \rightarrow B' = ite\ (A > 5, 1, 2) \end{aligned} \tag{2}$$

12.7 Adaptation to CVC

In certain cases CVC does not satisfy MCVT's needs. One example is the type of variables. *Simple types* of CVC are either *real* or *boolean*. For real variables i, j, the formula $i > j \leftrightarrow i \geq j + 1$, is not valid. However, it is valid for integer variables i, j and, the compiler may use this equivalence

to optimize the code. To handle this problem, MCVT adds the needed assertions (e.g. $(i > j) = (i \geq j + 1)$), whenever an equality of this kind is encountered.

Another problem is that CVC does not support multiplication commutativity, when both multiplication operands are variables. MCVT adds the needed assertions.

12.8 The Verifier Module (CVC)

Within the scope of this research, we had to choose a validity checker that is capable of automatically validating the verification conditions. The verification conditions that MCVT produces are formulas, which include: first order logic operators (without quantifiers), Presburger Arithmetic, uninterpreted functions and array operations. The following tools were explored:

1. STeP (Stanford) - The Stanford Temporal Prover [BBC+95].
2. Omega (University of Maryland) - A system for verifying Presburger formulas as well as a class of logical formulas and, for manipulating linear constraints over integer variables [KMP+].
3. CVT/C+tlv+tlvp (Weizmann) - A set of tools which provides a range minimizing module, an environment that supports model checking and, a validator for Presburger arithmetic [PSS98b], [PS96].
4. ICS (SRI International) - Integrated Canonizer and Solver. (We checked the alpha version) [FORS01].
5. CVC (Stanford) - Cooperative Validity Checker - Supports a combination of decision procedures for the validity of quantifier-free first-order formulas, with theories of extensional arrays, linear real arithmetic and uninterpreted functions [SBD02].

After a thorough examination of the above tools, CVC was found to best fit our needs, due to its reliability and the wide range of decision procedures it supports. It combines independent cooperating decision procedures for a wide set of theories, into a decision procedure for the combination of the theories, based on a version of the Nelson-Oppen [NO79] framework. Appendix

C demonstrates the application of MCVT on a small synchronous program and its translation. The C source program, the target machine-code, and one verification file (which is one verification condition ready for CVC) is listed there.

12.9 Case Studies

MCVT has validated the translation of a program developed for training, as part of the SafeAirII project. This program contains the code for a railroad crossing controller. It handles the passage of a train in a one-way railroad crossing. The system consists of a pair of reliable sensors that indicate train entering and exiting the crossing region, a signal for entering trains and a gate for blocking passage of cars from a side road. The verification took 200 seconds on a Linux workstation with i686 CPU.

Within the scope of the SafeAirII project, we have run MCVT on part of a jet engine application from the French company Hispano-Suiza. Appendix D lists detailed results of validating the translation of this program. The application consists of small modules, most of which were verified within a fraction of a second. However, as the module size increases, the verification time rises significantly. About 65% of the application, which totally contains more than 8000 lines of C code, was validated in less than 21 minutes on a Linux machine with i686 CPU. The rest of the application could not be checked, because of either syntax limitations of MCVT or MCVT run errors.

We are currently enhancing MCVT in order to decrease verification time and to better handle larger programs (see next section).

12.10 Future Research

Our research, including both the practical and the theoretical parts, may be further developed. The subset of synchronous C programs that we support may be extended to include the syntactic elements that are not currently implemented, e.g. case statements and function calls. The current partial implementation of the instruction set of the PowerPC processor may be completed to ensure that MCVT does not encounter unsupported machine instructions.

In some cases, synchronous programs may contain loops with a constant number of iterations, in order to handle elements of an array. Incorporating a solution of these cases into our framework is needed.

Another area which deserves further improvement is the validation runtime. In our work we use singleton transition systems and ignore basic blocks. We choose this method since it does not require any internal knowledge of the compiler and its optimizations. However, even though the singleton transition system is decomposed, the produced formulas size is significant. Using an improved version of CVC, CVC Lite [cvc], may decrease the validation time since this version is much faster. We may also decrease the size of the formulas by using the new extended set of basic-types (REAL, INT, subranges) that are supported by CVC Lite.

Further study about typing and type casting, as defined by the C language, is required. For example, one should further explore the formal semantics of different integer sizes, of usual arithmetic conversions, of type casting as well as of floating point constants accuracy. However, such future extensions of our framework should maintain its practical value within an industrial environment.

Chapter 13

Publications

The following are publications derived from the research presented in this thesis:

1. L. Zuck, A. Pnueli and R. Leviathan. Validation of optimizing compilers. Technical Report MCS01-12, Weizmann Institute of Science, 2001.[ZPL01]

   Includes a description of loop unrolling verifications presented in part I, chapter 6.2.

2. R. Leviathan and A. Pnueli. Validating Software Pipelining Optimizations. In Proceedings of the international conference on Compilers, architecture and synthesis for embedded systems (CASES02), ACM Press, 2002, pages 280-287, [LP02].

   Describes the results related to software pipelining validation presented in part I, chapter 7.

Bibliography

[AJLA95] V.N. Allan, R.B Jones, R.M. Lee, and S.J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):368–432, September 1995.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[ASU88] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1988.

[BA01] M. Ben-Ari. *Mathematical Logic for Computer Science.* springer Verlag London, 2001.

[BBC+95] N. Bjørner, I.A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, November 1995.

[COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995.

[cvc] Cvc lite home page. http://chicory.stanford.edu/CVCL.

[Dia] *DiabC/C++ Compiler for PowerPC, user's guide.*

[EdR+99] K. Engelhardt, W.P. de Roever, et al. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge University Press, 1999.

[Flo67] R.W. Floyd. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics*, 19:19–32, 1967.

[FORS01] J.C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. To be presented at CAV'2001, 2001.

[FS97] T. Fahringer and B. Scholz. Symbolic evaluation for parallelizing compilers. In *Proceedings of the 11th international conference on Supercomputing*, pages 261–268. ACM Press, 1997.

[GADT00] G. Gao, J.N. Amaral, J.C. Dehnert, and R.A. Towle. Tutorial on the sgi pro64 compiler infrastructure. In *PACT 2000: Int'l Conf. on Parallel Architectures and Compilation*, 2000.

[GZ99] G. Goos and W. Zimmermann. Verification of compilers. In Bernhard Steffen and Ernst Rdiger Olderog, editors, *Correct System Design*, volume 1710, pages 201–230. Springer-Verlag, Nov 1999.

[GZ00] G. Goos and W. Zimmermann. Verifying compilers and asms. In Yuri Gurevich, Philipp W. Kutter andMartin Odersky, and Lothar Thiele, editors, *Abstract State Machines, Theory and Applications*, volume 1912, pages 177–202. Springer, Apr 2000.

[GZG00] T. Gaul, W. Zimmermann, and W. Goerigk. Practical Construction of Correct Compiler Implementations by Runtime Result Verification. In *Proceedings of SCI'2000, International Conference on Information Systems Analysis and Synthesis*, Orlando, Florida, 2000.

[Huf93] R. Huff. Lifetime-sensitive modulu scheduling. In *Programming Language Design and Implementation. SIGPLAN*, 1993.

[Int] Intel. *Intel IA-64 Architecture Software Developer's Manual.*

[KMP+] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega library interface guide.

[KSR00] V. Kathail, M.S. Schlansker, and B.R. Rau. Hpl-pd architecture specification: Version 1.1. Technical report, HP - Compiler and Architecture Research, 2000.

[LE95] Jack L. Lo and Susan J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *Proceedings of the conference on Programming language design and implementation*, pages 151–162. ACM Press, 1995.

[LP02] R. Leviathan and A. Pnueli. Validating software pipelining optimizations. In S. S. Bhattacharyya, T. Mudge, W. Wolf, and A. Jerraya, editors, *Proc. of the international conference on Compilers, architecture, and synthesis for embedded systems.* ACM Press, 2002.

[LS84] J. Loeckx and K. Sieber. *The Foundation of Program Verification.* Wiley-Teubner, 1984.

[Man72] Z. Manna. *Mathematical theory of computation.* McGraw Hill, 1972.

[Man74] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[Muc97] S.S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, San Francisco, 1997.

[Nec00] G. Necula. Translation validation of an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI) 2000*, pages 83–95, 2000.

[NO79] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct 1979.

[PPC] *Book E- Enhanced PowerPC Architecture.*

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In R. Alur and T. Henzinger, editors, *R. Alur and T. Henzinger, editors,* Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96), *volume 1102 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 184–195, 1996.

[PSS98a] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT)- automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2(2):192–201, 1998.

[PSS98b] A. Pnueli, M. Siegel, and O. Shtrichman. Translation validation for synchronous languages. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proc. 25th Int. Colloq. Aut. Lang. Prog.*, volume 1443 of *Lect. Notes in Comp. Sci.*, pages 235–246. Springer-Verlag, 1998.

[PSS98c] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. Technical report, 1998.

[PSS99] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT)- automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2, 1999.

[PZP00] A. Pnueli, L. Zuck, and P. Pandya. Translation validation of optimizing compilers by computational induction. Technical report, Weizmann Institute of Science, 2000.

[RM00] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Run-Time Result Verification Workshop*, Trento, July 2000.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–49, Jan 1965.

[RST92] B.R. Rau, M.S. Schlansker, and P.P. Tirumalai. Code generation schemas for modulo scheduling loops. In *Proc. 25th annual international symposium on microarchitectur*, pages 158–169, 1992.

[SBD02] A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.

[SC9] *SCADE Language Reference Manual - Version 3.0.*

[SRM+94] M.S. Schlansker, B.R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik, and S.G. Abraham. Achieving high lelels of instruction-level parallelism with reduced hardware complexity. Technical report, HP Computer Research Center, 1994.

[TR0] Trimaran home page. www.trimaran.org.

[ZG97] W. Zimmermann and T. Gaul. On the Construction of Correct Compiler Back-Ends: An ASM-Approach. *j-jucs*, 3(5):504–567, may 1997.

[ZPFG02] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: a translation validator for optimizing compilers. In *Proc. of the workshop: Compiler Optimization Meets Compiler Verification (COCV'02), Satellite of ETAPS'02*, pages 6–22, Grenoble, April 2002.

[ZPG00] L. Zuck, A. Pnueli, and B. Goldberg. Translation validations of loop optimizations in optimizing compliers. Technical report, Weizmann Institute of Science, 2000.

[ZPL01] L. Zuck, A. Pnueli, and R. Leviathan. Validations of optimizing compliers. Technical report, Weizmann Institute of Science, 2001.

Appendix A

Loop Unrolling Validation- An Example

This appendix demonstrates a translation validation of loop unrolling. The method is not implemented. However, the detailed process is described here. A C source program, is translated to a block program and to PlayDoh assembly, by the TRIMARAN compiler. According to the AVALIDATE procedure, verification conditions are produced. The main optimization we want to demonstrate is loop unrolling. However, in this example we go all the way from C source to target assembly and thus, other optimizations are included as well. The verification conditions are produced by a special version of STEP and, are proved to be valid.

This example also demonstrates a limitation of our method. Our formal semantics treats all *read from memory* operations as non-destructive reads. This is not the case for real memory systems. For these, accessing the memory even for a read operation, may cause an access violation exception. The code in the example under discussion may cause, in some cases, read access to an illegal memory address.

A.1 Assembly Code Semantics

In this section a partial definition of the target system is given. We include only those topics which appear in the example.

Target system variables -

$$V = \{mem[N], pc, r_1...r_R, tr_1..tr_T, p_1...p_P\}$$

where:

mem	-An array of type *integer*.
pc	-A system control variable.
$r_i : i = 1..R$	-Integer registers.
$tr_i : i = 1..T$	-Target registers - registers that hold control values.
$p_i : i = 1..P$	-Boolean registers.
addr	-An array, indexed by *symbol-name*, where $addr[symbol\text{-}name] = Address$.

We equally use the function notation for array: $addr(symbol\text{-}name) = Address$.

Memory operations -

Operation	Syntax	Semantics
Load from memory	`ri = ld rj`	$r_i' = mem[r_j]$
Store to memory	`st ri rj`	$mem' = mem\ with([r_i] : r_j)$

A.2 The Source and Target Programs

Fig. A.1 shows a C program and its corresponding block program. The block program, which is our source program, is the intermediate representation of the C program that is produced by the TRIMARAN compiler. Fig. A.2 shows the optimized assembly-like program, which is the target program. The labels `b4` and `b5` were manually added, to mark the copies of the body block.

A.3 Optimization Methods Applied in the Example

In the above small example, a few optimization methods are applied by the TRIMARAN compiler. A summary of these techniques is given below. For further discussion on optimization methods, see [Muc97].

C Code

```
int M;
int a[100];
main() {
  int i;
  for(i = 0; i < 100 ; i ++){
    a[i] = a[i] + M;
  }
}
```

Source Block Program

```
extern int main();
int M;
int a[100];
{
  int i;
  LL_1 :
    i := 0;
    branch (!  i< 100) LL_3;
  LL_2 :
    a[i] := a[i] + M;
    i:=i+1 ;
    branch (i < 100) LL_2;
  LL_3 :
    return(0);
}
```

Figure A.1: The C program and its corresponding block program

- *Loop Inversion* - This optimization moves the loop-end test from before the body of the loop, to after it. It can be done in our example, since the loop body is guaranteed to be executed at least once.
- *Loop Unrolling* - Replaces the body of a loop by several copies of the body and, adjusts the loop-control code accordingly. In the current example, the main loop is unrolled by a factor of 3.
- *Strength reduction* - In general, strength reduction replaces expensive operations (e.g. a multiplication), by less expensive ones (e.g. an addition). In the example, strength reduction is applied on the address calculations of the elements of array `a` .
- *Dead Code Elimination* - As a result of the strength reduction and loop unrolling, the variable `i` is *dead* - it is not used on any path. Thus, the

```
b2:             enter
      r5        := addr(M)
      r2        := addr(a) + 2
      r3        := 2
      r4        := ld r5
b3:   r5        := r2 - 2
      r6        := r2 - 1
      r11       := r2 - 2
      tr2       := b6
      r7        := ld r2
      r8        := ld r5
      r9        := ld r6
      p2        := r3 >= 101
      r13       := r2 - 1
      tr3       := b6
      p3        := r3 >= 100
      r3        := r3 + 3
      tr4       := b3
      r14       := r7 + r4
      r10       := r8 + r4
      r12       := r9 + r4
      p4        := r3 < 102
      st        r11 r10
      branch    p2 tr2
b4:   st        r13 r12
      branch    p3 tr3
b5:   st        r2 r14
      r2        := r2 + 12
      branch    p4 tr4
b6:             Exit
```

Figure A.2: Optimized assembly program

compiler eliminates code that assigns values to `i`.

- *Instruction Scheduling* - At this optimization stage, operations are moved and grouped together to compose a large instruction word, so they can be executed in parallel.

A.4 Verification Process

We produce the corresponding transition systems manually, as well as the mapping and the annotations. All this information is fed into STeP, which first produces the relevant verification conditions and then proves them. Most of the verification conditions are proved automatically. Following are the source and the target transition systems, which are input to STe P :

A.4.1 Source Transition System

```
in N :int
local M, i : int
local a:  array [0..99] of int
local pi : int


Transition S12 Just:
        enable (pi = 1) /\ ( 0 < 100)
        assign pi := 2,
                i := 0

Transition S13 Just:
        enable (pi = 1) /\ ( 0 >= 100)
        assign pi := 3,
                i := 0
```

```
Transition S22 Just:
        enable pi = 2 /\ (i + 1) < 100
        assign  pi := 2,
                a[i] := a[i] + M,
                i := i + 1

Transition S23 Just:
        enable pi = 2 /\ (i + 1) >= 100
        assign  pi := 3,
                a[i] := a[i] + M,
                i := i + 1

Transition Sexit Just:
        enable pi = 3
```

A.4.2 Target Transition System

```
 in  addr_a : int where (addr_a + 400) <= N  /\ addr_a >= 0
 in addr_M :int where addr_M <= N /\ addr_M >=0

 local r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14 : int
 local tr1,tr2,tr3,tr4,tr5,tr6,tr7 : int
 local p1,p2,p3,p4 : bool
 local mem: array [1..N] of int
 local pc : int
 local r3_aux : int

Transition T2_3 Just:
        enable pc = 2
```

```
        assign
                pc := 3,
                r5 := addr_M,
                r2 := addr_a + 2*4,
                r3 := 2,
                r4 := mem[addr_M]

Transition T2_1 Just:
    enable (false)

Transition T3_4  Just:
        enable pc = 3 /\ !(r3 >= 101)
        assign
                pc := 4,
                r5 := r2 - 8,
                r6 := r2 - 4,
                r11 := r2 - 8,
                tr2 := 6
                r7 := mem[r2],
                r8 := mem[r2 - 8],
                r9 := mem[r2 - 4],
                p2 := (r3 >= 101),
                r13 := r2 - 4,
                tr3 := 6,
                p3 := (r3 >= 100),
                tr4 := 3,
                r14 := mem[r2] + r4,
                r10 := mem[r2-8] + r4,
                r12 := mem[r2-4] + r4,
```

```
                p4 := r3  + 3 < 102,
                mem[r2-8] := mem[r2-8] + r4,
                r3 := r3 + 3

Transition T3_6  Just:
        enable pc = 3 /\ r3 >= 101
        assign
                pc := 6,
                r5 := r2 - 8,
                r6 := r2 - 4,
                r11 := r2 - 8,
                tr2 := 6,
                r7 := mem[r2],
                r8 := mem[r2 - 8],
                r9 := mem[r2 - 4],
                p2 := (r3 >= 101),
                r13 := r2 - 4,
                tr3 := 6,
                p3 := (r3 >= 100),
                tr4 := 3,
                r14 := mem[r2] + r4,
                r10 := mem[r2-8] + r4,
                r12 := mem[r2-4] + r4,
                r3 := r3 + 3,
                p4 := r3  + 3 < 102,
                mem[r2-8] := mem[r2-8] + r4,
               r3_aux := 4

Transition T4_5  Just:
```

```
        enable   pc = 4 /\ !p3
        assign
                pc := 5,
                mem[r13] := r12

Transition T4_6  Just:
        enable  pc = 4 /\ p3
        assign
                pc := 6,
                mem[r13] := r12,
                r3_aux := 3

Transition T5_3  Just:
        enable  pc = 5 /\ p4
        assign
                pc := tr4,
                mem[r2] := r14,
                r2 := r2 + 12

Transition T5_6 Just:
        enable pc = 5 /\ !p4
        assign
                pc := 6,
                mem[r2] := r14,
                r2 := r2 + 12,
                r3_aux := 2

Transition Texit Just:
        enable pc = 6
```

A.4.3 Adding an Auxiliary Variable to the Target System

We add an auxiliary variable `r3_aux` to the target transition system. This variable has no effect on the system computation, but it enables the target system to keep track of the loop control variable `i`, which was eliminated by the optimizer.

A.4.4 Control Abstraction

In the case of loop unrolling, the control abstraction maps a few target cut-points to one source cut-point. The compiler produces three copies of the loop body, so the three values of the *pc* at the beginning of these loop bodies, are mapped to the one cut-point at the beginning of the source loop-body. Let

$$Dom(pc) = \{b2 = 2, b3 = 3, b4 = 4, b5 = 5, b6 = 6\},$$

$$Dom(\pi) = \{LL_1, LL_2, LL_3\},$$

then:

$$\kappa(b3) = \kappa(b4) = \kappa(b5) = LL_2 \wedge$$

$$\kappa(b2) = LL_1 \wedge$$

$$\kappa(b6) = LL_3$$

A.4.5 Data Abstraction

The data abstraction α is:

$$\{(pc = 2 \quad \rightarrow \quad i = r_3 - 2) \wedge$$

$$(pc = 3 \rightarrow i = r_3 - 2) \wedge$$

$$(pc = 5 \quad \rightarrow \quad i = r_3 - 3) \wedge$$

$$(pc = 4 \quad \rightarrow \quad i = r_3 - 4) \wedge$$

$$(pc = 6 \quad \rightarrow \quad i = r_3 - r_3 aux) \wedge$$

$$(\forall_{j=0..99} : a[j] = mem[addr(a) + 4 \times j]) \wedge$$

$$(mem[addr(M)] = M)\}$$

A.4.6 Invariants

The invariant θ_i corresponds to location $pc = i$.

$$\begin{array}{ll}
\theta_3 = & (r_2 = addr(a) + (i+2) \times 4) \wedge (M = r_4), \\
\theta_4 = & (r_2 = addr(a) + (i+1) \times 4) \wedge \\
& (r_{14} = mem[r_2] + M) \wedge \\
& (r_{12} = mem[r_{13}] + M) \wedge \\
& (r_{13} = addr(a) + i \times 4) \wedge \\
& (p_3 = ((r_3 - 3) >= 100)) \wedge \\
& (p_4 = (r_3 < 102) \wedge tr_4 = 3)), \\
\theta_5 = & (tr_4 = 3 \wedge (r_2 = (addr(a) + i \times 4)) \wedge (r_{14} = mem[r_2] + M) \wedge \\
& (p_4 = (r_3 < 102)) \wedge M = r_4)
\end{array}$$

A.4.7 Verification Conditions

The following is the verification condition produced for block `b3` *(pc=3)*, which is the first unrolled loop body:

$$\left.\begin{array}{l} r_2 = (addr(a) + (i+2) \times 4) \wedge (M = r_4) \end{array}\right\} \theta_3$$

$$\wedge$$

$$\left.\begin{array}{l} \wedge (i = r_3 - 2) \\ \wedge (\forall_{j:0..99} : a[j] = mem[addr(a) + j \times 4]) \\ \wedge (mem[addr(M)] = M) \end{array}\right\} \alpha$$

$$\wedge$$

$$\left.\begin{array}{l}(pc = 3) \wedge (\neg r_3 \geq 101) \wedge (pc' = 4) \\ \wedge(r'_5 = r_2 - 8) \\ \wedge(r'_6 = r_2 - 4) \wedge (r'_{11} = r_2 - 8) \\ \wedge(tr'_2 = 6) \wedge (r'_7 = mem[r_2]) \\ \wedge(r'_8 = mem[r_2 - 8]) \wedge (r'_9 = mem[r_2 - 4]) \\ \wedge(p'_2 = (r_3 >= 101)) \wedge (r'_{13} = r_2 - 4) \\ \wedge(tr'_3 = 6) \wedge (p'_3 = (r_3 >= 100)) \\ \wedge(tr'_4 = 3) \wedge (r'_{14} = mem[r_2] + r_4) \\ \wedge(r'_{10} = mem[r_2 - 8] + r_4) \wedge (r'_{12} = mem[r_2 - 4] + r_4) \\ \wedge(p'_4 = (r_3 + 3 < 102) \wedge (mem' = update(mem, mem[r_2 - 8] + r_4, r_2 - 8)) \\ \wedge(r'_3 = r_3 + 3)\end{array}\right\} \delta_{T_{3,4}}$$

$$\wedge$$

$$\left.\begin{array}{l}i' = r'_3 - 2 \\ \wedge\forall_{j:0..99} : a'[j] = mem'[addr(a) + j \times 4] \\ \wedge mem'[addr(M)] = M\end{array}\right\} \alpha'$$

$$\longrightarrow$$

$$\left.\begin{array}{l}(\pi = 2) \\ \wedge((i + 1) < 100) \\ \wedge(\pi' = 2) \\ \wedge(a'[i] = a[i] + M) \\ \wedge(\forall_{k=0..99} : \neg(k = i) \quad \rightarrow \quad a'[k] = a[k]) \\ \wedge(i' = i + 1)\end{array}\right\} \delta_{S_{2,2}}$$

$$\wedge$$

$$\left.\begin{array}{l}(M = r_4)\\ \wedge(r_2 = addr(a) + (i' + 1) \times 4)\\ \wedge(r'_{14} = mem'[r'_2] + M)\\ \wedge(r'_{12} = mem'[r'_{13}] + M)\\ \wedge(r'_{13} = addr(a) + i' \times 4)\\ \wedge(p'_3 = ((r'_3 - 3) \geq 100))\\ \wedge(p'_4 = (r'_3 < 102))\\ \wedge(tr'_4 = 3)\end{array}\right\} \theta_4$$

Appendix B

IA64 Pipelined Loop

$L1:$

$(p19)$	st	$[r3] = r35, 4$	//[Stage4]
$(p18)$	add	$r34 = 5, r37$	//[Stage3]
$(p18)$	nop		//[Stage3]
$(p16)$	ld	$r35 = [r2], 4$	//[Stage1]
$(p16)$	nop		//[Stage1]
	$br.ctop$	$L1;$	

Figure B.1: IA64 assembly code

The registers $r2, r3$ are general purpose registers, $r34, r35, r37$ are in the rotating register file and $p16, p18, p19$ are predicate registers. *br.ctop* is a special branch instruction that updates and checks the loop count, rotates the registers and update the predicate registers.

Appendix C

Translation Validation of a C Program

Fig. C.1 shows a C program and its optimized translation. The machine code produced, by the compiler, for $\_C\_- > I3 < 200 \&\& \_C\_- > I3 > 50$, contains a single unsigned comparison. The machine code produced for the assignment $\_C\_- > I4 = \_C\_- > I4 == 0$ is the sequence : `cntlzw` (count leading zeros), `rlwinm` (rotate with mask). MCVT recognizes the sequence and, treats it properly, even when the two instructions are not adjacent. In Fig. C.2 one of the input files to CVC is presented (without the variable declaration part).

C Program	**Machine-Code Program**		
`typedef struct {`	`ACS:`	`lwz`	`r12,0(r3)`
`int I3;`	`ACS+4:`	`lwz`	`r6,12(r3)`
`int I4;`	`ACS+8:`	`addi`	`r12,r12,-51`
`int out1;`	`ACS+12:`	`cmpli`	`0,r12,149`
`int gc; }`	`ACS+16:`	`addi`	`r5,r6,123`
`vars_rec;`	`ACS+20:`	`addi`	`r4,r6,567`
`int ACS ( vars_rec *_C_) {`	`ACS+24:`	`lwz`	`r6,4(r3)`
`  int c1; int c2;`	`ACS+28:`	`bc`	`4,0,ACS+48`
`  c1 = 123+_C_->gc;`	`ACS+32:`	`cmpi`	`0,r6,0`
`  c2 = 567+_C_->gc;`	`ACS+36:`	`bc`	`12,2,ACS+48`
`  if (((_C_->I3) < 200)&&`	`ACS+40:`	`stw`	`r5,8(r3)`
`     (50 < (_C_->I3)) &&`	`ACS+44:`	`b`	`ACS+52`
`     _C_->I4)`	`ACS+48:`	`stw`	`r4,8(r3)`
`      { _C_->out1 = c1; }`	`ACS+52:`	`cntlzw`	`r12,r6`
` else  {_C_->out1 = c2; }`	`ACS+56:`	`rlwinm`	`r12,r12,27,5,31`
` _C_- > I4=(_C_->I4==0);`	`ACS+60:`	`stw`	`r12,4(r3)`
`  return (1);`	`ACS+64:`	`addi`	`r3,r0,1`
`}`	`ACS+68:`	`bclr`	`20,0`

Figure C.1: C program and its translation to machine code

```
%Machine code conditions
ASSERT p1 = (((MEMORY0+-51)>=0) AND ((MEMORY0+-51)<149));
ASSERT p2 = (IF (p1) THEN
(MEMORY4=0) ELSE
((-1>=(MEMORY0+-51)) OR ((MEMORY0+-51)=149)) ENDIF) ;

%C code conditions
ASSERT pc1 = (((I3<200) AND (50<I3)) AND ( NOT I4=0));

%Target compressed transition for one variable
ASSERT PMEMORY8 = (IF (( NOT p1 OR (p1 AND p2))) THEN
(MEMORY12+567) ELSE
(IF ((p1 AND NOT p2)) THEN
(MEMORY12+123) ELSE
MEMORY8 ENDIF) ENDIF) ;

%Data mapping
ASSERT Pout1 = PMEMORY8;
ASSERT gc = MEMORY12;
ASSERT I3 = MEMORY0;
ASSERT Pgc = PMEMORY12;
ASSERT I4 = MEMORY4;
ASSERT PI3 = PMEMORY0;
ASSERT PI4 = PMEMORY4;
ASSERT out1 = MEMORY8;

%Integer assertions
ASSERT (((MEMORY0+-51)>=0)=((MEMORY0+-51)>(0-1)));
ASSERT (((MEMORY0+-51)<149)=(149>=((MEMORY0+-51)+1)));
ASSERT ((-1>=(MEMORY0+-51))=(-1>((MEMORY0+-51)-1)));

%Source compressed transition
QUERY Pout1 = (IF (pc1) THEN
(123+gc) ELSE
(567+gc) ENDIF) ;
```

Figure C.2: One CVC input file - verification condition for one variable

Appendix D

Experimental Results

As part of the SafeAirII project, we have run MCVT on part of a jet engine application from the French Company Hispano-Suiza. The tables presented in Fig. D.1 and in Fig. D.2 list the execution time of these verification tests.

Number of Lines	**Module Name**	**CPU Time** [seconds]
266	Est_Pos_ERe_ssm.c	1221.82
144	LogicalConstantFalse.c	0.09
144	LogicalConstantFalse1.c	0.07
144	LogicalConstantFalse10.c	0.07
144	LogicalConstantFalse1_1.c	0.06
144	LogicalConstantFalse2.c	0.09
144	LogicalConstantFalse3.c	0.08
144	LogicalConstantFalse5.c	0.07
144	LogicalConstantFalse6.c	0.08
144	LogicalConstantFalse7.c	0.08
144	LogicalConstantFalse8.c	0.08
144	LogicalConstantFalse9.c	0.05
144	LogicalConstantFalse_1.c	0.06
144	LogicalConstantFalse_2.c	0.06
144	LogicalConstantFalse_3.c	0.07
144	LogicalConstantTrue.c	0.05
144	LogicalConstantTrue1.c	0.08
144	LogicalConstantTrue_1.c	0.07
144	LogicalConstantTrue_2.c	0.07
145	NOT.c	0.10
145	NOT1.c	0.08
145	NOT2.c	0.11
155	S2S_Demux.c	0.18
149	S2S_Mux.c	0.11

Figure D.1: Run time of a jet engine translation validation (continued in Fig. D.2)

Number of Lines	**Module Name**	**CPU Time** [seconds]
155	S2S_Mux_1.c	0.18
149	S2S_Mux_2.c	0.10
145	S2S_Selector.c	0.05
145	S2S_Selector_1.c	0.08
145	S2S_Selector_2.c	0.08
148	S2S_Vect_3_2.c	0.16
148	S2S_Vect_3_2_1.c	0.19
148	S2S_Vect_3_3.c	0.21
221	Seq_Cmd_ER.c	0.58
220	Commande_ERe	-
2327	Command_electro_robinet	-
178	S2S_MPSwitch	-
145	SMLK_Terminator_2	-
8286	Total	1225.31

Figure D.2: Run time of a jet engine translation validation - continued