EFFICIENT VERIFICATION OF BIT-LEVEL PIPELINED MACHINES USING REFINEMENT

A Thesis
Presented to
The Academic Faculty

by

Sudarshan Kumar Srinivasan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2007

EFFICIENT VERIFICATION OF BIT-LEVEL PIPELINED MACHINES USING REFINEMENT

Approved by:

Professor Panagiotis Manolios, Adviser
College of Computing
Georgia Institute of Technology

Professor Abhijit Chatterjee
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Sung-Kyu Lim
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Ganesh Gopalakrishnan
School of Computing
University of Utah

Professor Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: August 2007

Dedicated to my late mother, Manjula Srinivasan

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my adviser, Panagiotis Manolios, for his exemplary guidance, support, and encouragement, which have been crucial for my research, career, and the successful completion of this thesis. He provided the impetus for my dissertation work. His vision, depth and breadth of knowledge, and critical feedback have been instrumental for my growth as a researcher. He also provided financial assistance for most of my Doctoral studies. I am extremely fortunate to have had the opportunity to work with him.

The members of my dissertation committee Abhijit Chatterjee, Sung-Kyu Lim, Ganesh Gopalakrishnan, and Sudhakar Yalamanchili spent valuable time and effort reading my dissertation and providing insightful comments and feedback. I am very grateful to them.

I would like to thank Vincent J. Mooney for giving me the opportunity to study at GeorgiaTech and providing financial assistance for my first year at GeorgiaTech. I would like to thank Jason Baumgartner and John O'Leary for being supportive of my research work and career goals. I would like to thank Olin Shivers for his useful comments and feedback on some of my research work. I also benefited from taking his class on Semantics of Programming Languages.

I would like to thank the members of the *formal* group at GeorgiaTech: Peter Dillinger, Roma Kane, Daron Vroon, and Yimin Zhang for their collaboration and encouragement. Peter and Daron also provided valuable feedback on some of the research presented in this dissertation and helped me in better understanding the internals of the ACL2 theorem proving system. I would also like to thank Krishnakumar Sundaresan for reading parts of this dissertation and providing useful feedback and comments.

Finally, I would like to thank my family for their strong support and love through trying times. My parents, brother, grandmother, and Ramya have all been wonderful and the greatest source of joy in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In this work, we want to defend the following thesis:

Using refinement and a combination of deductive reasoning and decision procedures enables the verification of bit-level pipelined machines in a highly automated, efficient, and scalable manner.

Functional verification is a critical problem facing the semiconductor industry as hardware designs are extremely complex and highly optimized, and as the cost of bugs in deployed systems can be colossal. Pipelining is a key optimization that appears extensively in hardware systems such as microprocessors, multicore systems, and cache coherence protocols. Verifying pipelined machines—models that describe the pipelined behavior of hardware designs—entails showing that these machines behave like their instruction set architectures (ISAs). Existing approaches for verifying bit-level pipelined machines are based on deductive reasoning and require extraordinary expert user effort, as the problem involves reasoning about the difference in time-scale between the pipeline and its ISA and the intricate control circuitry involved in optimized pipeline designs. More automatic approaches are based on the use of decision procedures, but are applicable only to very abstract, high-level models, known as term-level models.

We present a novel, highly automated, efficient, and scalable refinement-based approach for the verification of bit-level pipelined machines. The notion of refinement that we use is compositional and guarantees that pipelined machines satisfy the same safety and liveness properties as their instruction set architectures (ISAs). The high-level idea of the verification approach is to use a deductive reasoning engine, such as the ACL2 theorem proving system, to reduce the bit-level pipelined machine verification problem to a term-level problem. This drastically reduces the amount of expert user effort required when compared to approaches based on the use of deductive reasoning. The verification of term-level models is automated by providing techniques to express

correctness statements in a decidable fragment of first-order logic, which can be handled with existing decision procedures. The verification time required for term-level problems is reduced by optimizing parameters of the refinement framework such as refinement maps, which are functions that map pipelined machine states to instruction set architecture states. We have also developed a complete compositional reasoning framework that can be used to decompose correctness proofs into smaller manageable pieces leading to drastic reductions in verification times and a high-degree of scalability. The verification is discharged using the ACL2-SMT system, which we developed by combining the ACL2 theorem prover with a decision procedure.

The methods developed for term-level verification are evaluated using a large number of complex, highly pipelined machine models. The term-level models incorporate various features such as branch prediction, an instruction queue, an instruction cache, a data cache, and a write buffer. The effectiveness of our verification approach for bit-level pipelined machines is demonstrated using an Intel XScale inspired processor model that implements 593 instructions and has features such as branch prediction, precise exceptions, and predicated instruction execution.

CHAPTER I

INTRODUCTION

Hardware systems are ubiquitous and find applications ranging from personal computers and business solutions to safety-critical systems such as aircraft and automotive control systems, medical monitoring systems, systems for controlling nuclear reactors, *etc.* Ensuring the correct functioning of these systems is therefore of paramount importance as failure of deployed systems can lead to loss of life and exceedingly high economic costs. A well known example is the bug that was found in the floating point division (FDIV) unit of the Intel Pentium processor that cost Intel 475 million dollars. Estimates show that a similar bug in the current generation of Intel processors will cost Intel about 12 billion dollars [12].

Validation and verification techniques targeted at ensuring the functional correctness of hardware designs are key to finding bugs and developing reliable systems. But, functional validation and verification are critical problems facing the semiconductor industry as hardware designs are extremely complex and highly optimized. For example, the IBM Power5 chip is described using more than 1.5 million lines of VHDL code and has about 276 million transistors [84]. The challenge of verifying hardware systems has in fact been addressed by the International Technology Roadmap for Semiconductors (ITRS) 2004 report [36].

> *Verification has become the dominant cost in the design process. ... Without major breakthroughs, verification will be a non-scalable, show-stopping barrier to further progress in the semiconductor industry.*

One of the key optimizations used in hardware systems is pipelining, the idea being that the functionality of a system is partitioned into several stages, which can work in parallel resulting in an increase in the throughput of the pipelined implementation of the system. Although pipelining has been around for decades and is used extensively in hardware systems such as microprocessors, cache coherence protocols, and multicore systems, there are currently no efficient and scalable techniques that can check that these pipelines work correctly. Existing techniques for verifying

pipelined machines—machine models that describe the pipelined behavior of hardware systems—either consume excessive amount of time, effort, and resources or are not applicable at the Register Transfer Level (RTL), the level of abstraction at which commercial systems are functionally verified.

This thesis presents a novel, highly automated, efficient, and scalable approach based on formal methods for checking that RTL pipelined machine models work correctly. Our focus in this thesis is on verifying microprocessor pipelines, but the methods and techniques developed can be directly applied or easily extended to be applicable to other pipelined hardware systems.

Pipelined machine verification entails providing a mathematical proof that guarantees that a pipelined machine behaves like its instruction set architecture (ISA), which is a non-pipelined specification that describes the functionality of the design. While the idea of relating pipelined machines to their ISAs to prove correctness is not new, the novelty in our verification approach lies in the methods and techniques used to achieve this.

The approach uses a theory of refinement based on Well Founded Equivalence Bisimulations (WEBs) to show that bit-level pipelined machines correctly implement their ISAs. WEB refinement is a compositional notion and guarantees that the pipelined machine and its ISA have the same safety and liveness properties. Informally, proving safety properties provides assurance that nothing bad ever happens, while proving liveness properties provides assurance that something good happens eventually.

The proof obligations generated by our verification approach are checked using a combination of deductive reasoning and decision procedures. Deductive reasoning or theorem provers can be thought of as an integrated system of ad hoc proof techniques. Theorem provers typically have underlying logics that are very powerful and expressive, but are also undecidable. Therefore, theorem provers can be used to reason about pipelined machines at various levels of abstraction including at the RTL but require an extraordinary amount of expert user effort. In contrast, decision procedures are highly automated verification engines, but are limited in that they can be used to reason only about high-level abstractions of RTL/bit-level models, called term-level models.

The high-level idea of the verification approach is that since theorem provers are powerful enough to reason about pipelined machines at various levels of abstraction including at the term-level and the bit-level, a theorem prover is used to reduce the bit-level pipelined machine verification

problem to a term-level problem. The reduction is achieved by verifying the abstractions used to construct the term-level model from the bit-level model. A decision procedure is then used to reason about the pipeline at the term-level. We have also developed several refinement-based methods and techniques that allow us to verify term-level pipelined machine models in a highly automated, efficient, and scalable manner. Since the verification approach reduces the bit-level problem to a term-level problem, all of these techniques can be leveraged to solve the term-level problem efficiently. We believe that these methods will have a strong impact as they can be easily tailored to be effective in the design cycle of a commercial system.

The predominant method used in the industry for validating pipelined hardware systems such as microprocessor designs is based on simulation, which is very effective in finding a large number of shallow bugs. The primary drawback with simulation is that even though it consumes a large amount of time and resources, it is still far from being exhaustive. For example, at Intel, large teams of engineers using workstation clusters containing thousands of machines run simulations over the course of several years. Even so, they are only able to simulate about one minute of actual running time of the microprocessor [11, 12].

To overcome the limitations of simulation, the industry is starting to use semi-formal and formal methods [50, 7, 12] in conjunction with simulation-based techniques. Verification approaches based on formal methods essentially provide a mathematical proof of correctness for the system and are therefore very exhaustive. Intel's first use of formal verification on a large-scale was during the Pentium 4 design cycle and consisted of about 60 person years [12]. Formal methods were used to verify that the design satisfied various properties describing the expected behavior of the microprocessor and, to date, no bugs have been discovered in the parts of the design that were formally verified [12].

The two primary formal verification techniques that have been successfully used in the industry are combinational equivalence checking [37] and property-based verification [97, 79, 88, 15]. Combinational equivalence checking is typically used to compare and check equivalence of two designs described at the RTL/bit-level or at the gate-level, but, cannot be used to verify the functionality of a design. Combinational equivalence checking methods also require a one-to-one mapping of the latches in the two designs that are compared and therefore cannot be used to verify sequential

optimizations to the design such as pipelining.

Property-based verification is the predominant formal technique used in the industry to verify the functional correctness of hardware designs. The behavior of the design is specified using a number of properties and the design is checked to see if it satisfies these properties. There are primarily two limitations of property-based verification for verifying pipelined machines. First, a large number of design-dependent properties are required to describe the behavior of pipelined machines. Second, the properties themselves are complex and it is difficult to avoid erroneous properties and to ensure completeness of the verification of the design.

Due to the limitations of industry standard techniques for checking the correctness of pipelined machines, the area of pipelined machine verification has recently received a lot of interest from the research community.

1.1 Pipelined Machine Verification

In this section, we review previous work on pipelined machine verification to provide context for the contributions of this thesis. Surveys of previous work on specific topics can be found in the chapters that describe each of these contributions.

To avoid the limitations of industry standard methods for checking that pipelined machines work correctly, Burch and Dill introduced a notion of correctness based on commuting diagrams that much of the previous work in the area is based on [22]. The idea is to show that the pipelined machine behaves like its instruction set architecture (ISA), which is a non-pipelined specification that describes the functionality of the design. The Burch and Dill notion of correctness is based on the use of flushing-based abstraction functions used to relate pipelined machine states to ISA states. An abstraction function otherwise known as a refinement map can be thought of as a function that gives the ISA state corresponding to a given pipelined machine state. The idea with flushing is that a pipelined machine state is related to an ISA state by completing partially completed instructions without fetching any new instructions. Unfortunately, this notion is not as complete as we would like, *e.g.*, does not fully address liveness, and even when augmented with various liveness properties, it can still be satisfied by machines that deadlock [51]. Proving liveness guarantees that the pipelined machine will not deadlock, i.e., it will always make forward progress. In comparison, our methods

for checking the correctness of pipelined machines are based on WEB-refinement and account for both safety and liveness.

There are various approaches that are based on variations of the Burch and Dill notion of correctness to verify pipelined machines. These approaches can be primarily classified into approaches based on the use of theorem provers [43] and approaches that use decision procedures.

Approaches based on decision procedures are highly automatic, but can be used to verify only pipelined machine models described at the term-level, models that abstract away the data path and combinational circuit blocks such as the ALU. They are not applicable to bit-level designs and therefore have not had much impact in the industry. In contrast, our approach can be used to verify both term-level and bit-level designs. Below, we briefly describe various approaches that use decision procedures.

Burch and Dill [22] gave a decision procedure for the logic of Equality with Uninterpreted Functions. There is also recent work on decision procedures for the CLU logic [20], which is based on previous work on exploiting positive equality [19]. The decision procedure is implemented in UCLID, which has been used to verify out-of-order microprocessors [49].

Jones et al. [39] verify an out-of-order execution unit using incremental flushing using the SVC decision procedure. Their approach relates the implementation to an intermediate machine, where the scheduling logic is abstracted, which is then related to the ISA. In comparison, we can deal with any refinement map, we have a general theory for relating any number of intermediate machines, and we guarantee that all safety *and* liveness properties are preserved. Mishra and Dutt [72] use SVC to check the correct flow of instructions in a pipelined DLX model. An XScale processor model is also verified using a variation of the Burch and Dill approach [95]. There are approaches based on model-checking, eg., McMillan [69] uses compositional model-checking and symmetry reductions. Symbolic Trajectory Evaluation (STE) is used by Patankar et al. to verify a processor that is a hybrid between ARM7 and StrongARM [77]. Recent advances in decision procedures [26, 100] also drastically reduce the verification times of term-level pipelined machines.

Another popular approach for pipelined machine verification is based on the use of theorem provers. While such approaches are applicable to bit-level designs, they usually require a prohibitive amount of effort on the part of the expert user. An early, pioneering body of work on

the use of theorem proving for the verification of microprocessors is the CLI stack verification effort [33, 34, 13]. Another notable use of theorem proving in the context of hardware verification used ACL2 to reason about Motorola's CAP digital signal processor [16]. Examples of the use of theorem provers for pipelined machine verification include the work by Sawada and Hunt, who use an intermediate abstraction called Microarchitecture Execution Trace Table (MAETT) to verify some safety and liveness properties of complex pipelined machines [89, 90] using the ACL2 theorem prover. The MAETT stores information about each of the partially executed instructions in the pipeline. Instead of directly analyzing the entire microarchitecture, MAETT is used to show that each of the partially executed instructions execute correctly, which is then used to prove the Burch and Dill based commuting diagram.

Another example of a theorem proving approach is the work by Hosabettu et al., who use the notion of completion functions [31] to compute the abstraction function or the refinement map. A completion function specifies the effect of completing a partially executed instruction in the pipeline on the programmer visible components, which are the program counter, the register file, and the data memory. One completion function for each of the partially executed instructions in the pipeline is used to compute the abstraction function. The correctness proofs are carried out using the PVS theorem prover. Arons and Pnueli [9] have also used the PVS theorem prover to verify a machine with speculative instruction execution. They use an inductive proof to show that machines which differ only in the size of the retirement buffer are related; however, due to the complexity of the refinement maps involved, they conclude that a direct approach is far simpler than the inductive one. Kroning [48] verified data consistency of pipelined machine models using the PVS theorem prover. The models are synthesizable and are described very close to the gate-level. There is also work by Cohn [25], who used the HOL theorem prover to check the equivalence of two high-level specifications of the VIPER microprocessor.

The notion of correctness for pipelined machines that we use was first proposed by Manolios [51], and is based on WEB-refinement [52]. The first proofs of correctness for pipelined machines based on WEB-refinement were carried out using the ACL2 theorem proving system [43]. The advantage of using a theory of refinement over using the Burch and Dill notion of correctness —even when augmented with a "liveness" criterion— is that the Burch and Dill approach cannot

detect deadlock [51], whereas it follows directly from the WEB-refinement approach that deadlock (or any other liveness problem) will be detected.

1.2 Structure of this Dissertation

In this section, we describe the organization of the rest of this dissertation. In Chapter 2, we describe a simple three stage pipelined machine, and show how to model and verify it using approaches based on deductive reasoning and decision procedures. In the process, we also give an overview of the notion of correctness that we use for pipelined machines, which is based on refinement. The next seven chapters describe the contributions of this thesis.

- Chapter 3 introduces two methods for automating refinement proofs for term-level pipelined machines [57, 63]. A consequence of proving refinement is that it guarantees that pipelined machines satisfy the same safety and liveness properties as their instruction set architecture (ISA) machines. Checking liveness automatically was thought to be expensive, but using our approach, liveness can be proved for various complex pipelined machine models with an overhead cost of only about 25%.

- An important parameter of the refinement framework is the refinement map, which is a function that maps pipelined machine states to ISA states. A well known approach for defining this refinement map is based on commitment, the idea being that partially executed instructions in the pipeline latches of a pipelined machine state are invalidated and any effect that these instructions had on the programmer visible components is undone. In Chapter 4, we describe a method for verifying term-level pipelined machines using an optimization of the commitment refinement map [59]. This method provides a 30-fold improvement in verification times over previous approaches.

- Another approach for defining refinement maps is based on flushing, which can be thought of as the dual of commitment as partially executed instructions are forced to complete without fetching any new instructions, as opposed to being invalidated. In Chapter 5, we introduce an optimization of the flushing refinement map, called collapsed flushing [41]. We also introduce

a new, simpler, and easier-to-verify rank function, which is used for handling liveness. Empirical evaluations show that we obtain over an order-of-magnitude improvement in verification times when using collapsed flushing instead of standard flushing.

- The flushing and the commitment refinement maps can be fruitfully combined by applying the commitment refinement map to the latches in the front of the pipeline and the flushing refinement map to the latches in the end of the pipeline giving rise to a new class of refinement maps that we call intermediate refinement maps [60]. These refinement maps are described in Chapter 6. The result is that we are left with two verification problems, but on machines that are half as complex as the initial pipelined machine; since verification times are exponential in the size of the machines, this leads to drastic improvements in verification times. By combining an optimized commitment refinement map with collapsed flushing using intermediate refinement maps, we can monolithically verify very deep pipelines with 16 stages, which was not possible using previous approaches.

- Monolithic verification efforts based on optimized refinement maps provide drastic improvements in verification times, but, the verification times of monolithic methods increase exponentially with increase in the complexity of the design. We demonstrate the use of compositional reasoning based on WEB-refinement to verify complex term-level pipelined machines that provide a scalable approach for verifying pipelined machines and also gives exponential savings in verification times [58]; this work is described in Chapter 7. We found that a more important benefit of the compositional approach over current methods is that the counterexamples generated are much simpler and easier to analyze, as bugs are isolated to a particular step in the composition proof.

- Our verification approach for checking the correctness of bit-level pipelined machines leverages on all of the previous methods for reasoning about term-level models and also uses a combination of deductive reasoning and decision procedures. In Chapter 8, we describe a novel framework for integrating term-level decision procedures with the ACL2 theorem proving system. This combined system can then be used to discharge the proof obligations generated by our verification approach.

- The proof strategy, which is at the crux of our verification approach is described in Chapter 9. The strategy itself is based on refinement and heavily exploits the compositionality of refinement. We developed an initial approach for reasoning about bit-level pipelined machines [61, 62]. The strategy presented here is a simplified version of this initial approach and is more efficient. In this chapter, we also describe an application of our verification approach to check the correctness of an Intel XScale inspired complex processor model, most of which is defined at the bit-level.

Conclusions and a summary of the main contributions of this thesis appears in Chapter 10. We also describe future work in this chapter.

CHAPTER II

MODELING AND VERIFYING A THREE STAGE PIPELINED MACHINE

In this chapter, we describe a simple three stage pipelined machine and show how to model and verify it at the RTL and at a high-level of abstraction. This excercise allows us to give an overview of previous approaches for verifying pipelined machines and also allows us compare these methods. In the process, we also describe the notion of correctness that we use for pipelined machines, which is based on a theory of refinement.

The chapter is organized as follows. Section 2.1 describes in detail the three stage pipelined machine example, which we call 3PM, and the instruction set architecture that 3PM implements. Section 2.2 describes the notion of correctness that we use for pipelined machines, which is based on refinement. Section 2.3 describes how to model and verify 3PM at a high-level of abstraction and Section 2.4 describes the modeling and verification of 3PM at the bit-level. Conclusions are given in Section 2.5.

2.1 Pipelined Machine Example

A pipelined machine model can be thought of as a model that describes the pipelined behavior of a hardware design such as a microprocessor, as opposed to the instruction set architecture (ISA) that describes the functionality of the design. An ISA machine is as a single cycle implementation of an ISA. Most microprocessors that are currently being used in commercial applications implement their ISA in a pipelined manner. In a pipelined implementation, the functionality of the microprocessor (as described by its ISA) is split into several stages. The output of one stage is the input of the next stage. Each of these stages, known as pipelined stages can process a different instruction. Therefore the pipelined implementation can process multiple instructions simultaneously giving rise to large speedups over the ISA machine. In this section, we describe a simple three stage pipelined machine, which we use as a running example in subsequent chapters to illustrate various concepts that we have developed. The ISA that the pipelined machine implements is described first.

2.1.1 Instruction Set Architecture

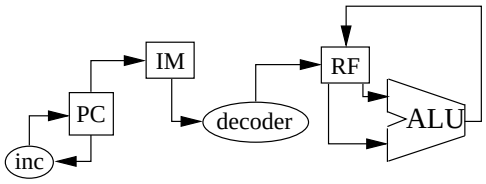


Figure 1: Instruction set architecture machine model.

The high-level organization of the ISA machine model is shown in Figure 1. We call this machine ISAS. The state components of ISAS include a program counter (*PC*), an instruction memory (*IM*), and a register file (*RF*). ISAS implements only one instruction, which is an add instruction.

The instruction format is as follows. An instruction has 2 source addresses and a destination address. An instruction is fetched from the instruction memory using the program counter as the reference. The fetched instruction is then decoded using three functions *src1*, *src1*, and *dest*, which take the instruction as input and are used to compute the first source address, the second source address, and the destination address, respectively. The two source addresses are then used as references to obtain the two source operands corresponding to the instruction from the register file. The sum of the two operands is computed using the *alu* function. The output of the *alu*, which is the result of the instruction is then written to the location in the register file corresponding to the destination address. During every cycle, the program counter is incremented using the *inc* function. Therefore, after ISAS has processed an instruction, its program counter points to the next instruction in the memory to be processed.

2.1.2 Three Stage Pipelined Machine

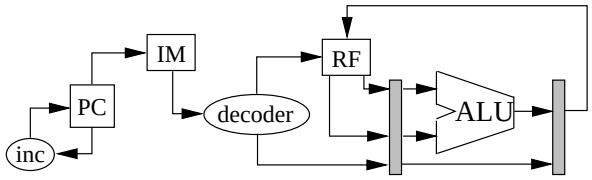


Figure 2: Three stage pipelined machine example (3PM).

We now describe an example of a simple three stage pipelined machine, which implements ISAS, the instruction set architecture example described in Section 2.1.1. We call this machine

3PM. The high-level organization of 3PM is shown in Figure 2. Unlike an ISA machine that executes only one instruction per cycle, the pipelined machine has three stages each of which are capable of processing an instruction every cycle. Therefore, 3PM can process upto a maximum of three instructions simultaneously, in each of its pipelined stages. The model has 3 stages which are separated by the pipeline latches shown as dark boxes in Figure 2. The stages are the fetch and decode stage (before the first pipeline latch), the execute stage (between the two pipeline latches), and the write back stage (after the second pipeline latch). The model implements only one instruction, which is an add instruction.

In the fetch and decode stage, the instruction is fetched from the instruction memory using the address as the program counter. Note that the instruction format of the instructions in the three stage pipelined machine (3PM) is the same as the instruction format for instructions in ISAS. Therefore, we can use the same functions as we did for the ISA to decode the instruction. The two sources addresses are then used to read the two source operands from the register file. These source operands are then stored in the first pipelined latch, along with the destination address corresponding to the instruction.

In the execute stage, the two source operands are read from the first pipelined latch, and their sum is computed using the same *alu* function that was used in ISAS. The result along with the destination address from the first pipeline latch is stored in the second pipeline latch. In the write back stage, the result and the destination address of the instruction are read form the second pipeline latch and this result is written into the register file using the destination address.

It is possible for an instruction in the first pipeline latch (say *i1*) to depend on an instruction in the second latch (say *i2*), *i.e.*, the result produced by *i2* is required by *i1*. In this situation, the program counter and the instruction in the first pipeline latch (*i1*) are stalled for one cycle to allow the instruction in the second pipeline latch (*i2*) to complete and update the register file.

2.2 Refinement

In this section, we describe the notion of correctness that we use for pipelined machines, which is based on a general theory of refinement developed by Manolios [52, 53]. Using a general notion of correctness is advantageous as we can clearly identify the parameters of our correctness framework

and then analyze as to how these parameters can be optimized using domain knowledge about pipelined machines. In fact, in the subsequent chapters where we report the contributions of this thesis, we describe several methods to optimize parameters of the refinement framework that lead to a high degree of efficiency.

Pipelined machine verification is an instance of the refinement problem: given an abstract specification, S, and a concrete specification, I, show that I refines (implements) S. In the context of pipelined machine verification, the idea is to show that `MA`, a machine modeled at the microarchitecture level, a low level description that includes the pipeline, refines `ISA`, a machine modeled at the instruction set architecture level. A refinement proof is relative to a *refinement map*, r, a function from `MA` states to `ISA` states. The refinement map, r, shows us how to view an `MA` state as an `ISA` state, *e.g.*, the refinement map has to hide the `MA` components (such as the pipeline) that do not appear in the `ISA`.

There are several ways in which refinement maps can be defined for pipelined machines. One well known approach is based on commitment, the idea being that partially executed instructions in the pipeline latches are invalidated, and their effect on the programmer state components—state elements of an ISA state such as the program counter, register file, and memories that are also part of a pipelined machine state—is undone. Another approach for defining refinement maps is based on flushing, which can be thought of as the dual of commitment, as partially executed instructions are forced to complete, without fetching any new instructions. The refinement maps used play a critical role in the efficiency of the verification methods used, and we study these maps closely and develop several optimization techniques for these maps. We now give the formal definitions for refinement.

The `ISA` and `MA` machines are arbitrary transition systems (TS). A TS, $\mathcal{M}$, is a triple $\langle S, \dashrightarrow, L\rangle$, consisting of a set of states, S, a left-total transition relation, $\dashrightarrow \subseteq S^2$, and a labeling function L whose domain is S and where $L.s$ (we sometimes use an infix dot to denote function application) corresponds to what is "visible" at state s.

Our notion of refinement is based on stuttering bisimulation [18]. When we say that `MA` refines `ISA`, we mean that in the disjoint union ($\uplus$) of the two systems, there is a stuttering bisimulation refinement (STB) that relates every pair of states w, s such that w is an `MA` state and $r(w) = s$.

Definition 1. *(STB Refinement) [52] Let $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$, $\mathcal{M}' = \langle S', \dashrightarrow', L'\rangle$, and $r : S \to S'$. We say that $\mathcal{M}$ is a STB refinement of $\mathcal{M}'$ with respect to refinement map r, written $\mathcal{M} \approx_r \mathcal{M}'$, if there exists a relation, B, such that $\langle \forall s \in S :: sBr.s\rangle$ and B is an STB on the TS $\langle S \uplus S', \dashrightarrow \uplus \dashrightarrow', \mathcal{L}\rangle$, where $\mathcal{L}.s = L'.s$ for s an S' state and $\mathcal{L}.s = L'(r.s)$ otherwise.*

Our notion of refinement is based on the following definition of stuttering bisimulation [18], where by $fp(\sigma, s)$ we mean that σ is a fullpath (infinite path) starting at s, and by $match(B, \sigma, \delta)$ we mean that the fullpaths σ and δ are equivalent sequences up to finite stuttering (repetition of states).

Definition 2. *$B \subseteq S \times S$ is a stuttering bisimulation (STB) on TS $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ iff B is an equivalence relation and for all s, w such that sBw:*

$$\text{(Stb1)} \qquad L.s = L.w$$

$$\text{(Stb2)} \qquad \langle \forall \sigma : fp(\sigma, s) : \langle \exists \delta : fp(\delta, w) : match(B, \sigma, \delta)\rangle\rangle$$

We note that stuttering bisimulation differs from weak bisimulation [71] in that weak bisimulation allows infinite stuttering. Stuttering is a common phenomenon when comparing systems at different levels of abstraction, *e.g.*, if the pipeline is empty, `MA` will require several steps to complete an instruction, whereas `ISA` completes an instruction during every step. Distinguishing between infinite and finite stuttering is important, because (among other things) we want to distinguish deadlock from stutter.

A major shortcoming of the above formulation of refinement is that it requires reasoning about infinite paths, something that is difficult to automate [75]. Manolios [52] developed Well Founded Equivalence Bisimulation (WEB) refinement, which is an equivalent formulation that requires only local reasoning, involving only `MA` states, the `ISA` states they map to under the refinement map, and their successor states. We now give the relevant definitions that are given in terms of general transition systems (TS).

Definition 3. *(WEB Refinement) Let $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$, $\mathcal{M}' = \langle S', \dashrightarrow', L'\rangle$, and $r : S \to S'$. We say that $\mathcal{M}$ is a WEB refinement of $\mathcal{M}'$ with respect to refinement map r, written $\mathcal{M} \approx_r \mathcal{M}'$, if there exists a relation, B, such that $\langle \forall s \in S :: sBr(s)\rangle$ and B is a WEB on the TS $\langle S \uplus S', \dashrightarrow \uplus \dashrightarrow', \mathcal{L}\rangle$, where $\mathcal{L}(s) = L'(s)$ for s an S' state and $\mathcal{L}(s) = L'(r(s))$ otherwise.*

In the above definition, it helps to think of $\mathcal{M}'$ as corresponding to ISA and $\mathcal{M}$ as corresponding to MA. Note that in the disjoint union ($\uplus$) of $\mathcal{M}$ and $\mathcal{M}'$, the label of every $\mathcal{M}$ state, s, matches the label of the corresponding $\mathcal{M}'$ state, $r(s)$. WEBs are defined next; the main property enjoyed by a WEB, say B, is that all states related by B have the same (up to stuttering) visible behaviors.

Definition 4. *$B \subseteq S \times S$ is a WEB on TS $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ iff:*

(1) B is an equivalence relation on S; and

(2) $\langle \forall s, w \in S :: sBw \;\Rightarrow\; L(s) = L(w) \rangle$; and

(3) There exist functions $erankl : S \times S \to \mathbb{N}, erankt : S \to W$,

such that $\langle W, \prec \rangle$ is well-founded, and

$\langle \forall s, u, w \in S :: sBw \;\wedge\; s \dashrightarrow u \;\Rightarrow$

(a) $\langle \exists v :: w \dashrightarrow v \;\wedge\; uBv \rangle \;\vee$

(b) $(uBw \;\wedge\; erankt(u) \prec erankt(s)) \;\vee$

(c) $\langle \exists v :: w \dashrightarrow v \;\wedge\; sBv \;\wedge\; erankl(v,u) < erankl(w,u) \rangle \rangle$

The third WEB condition says that given states s and w in the same class, such that s can step to u, u is either matched by a step from w, or u and w are in the same class and a rank function decreases (to guarantee that w is eventually forced to take a step), or some successor v of w is in the same class as s and a rank function decreases (to guarantee that u is eventually matched). In the following chapters, we provide several methods to define these rank functions that does not require deep understanding of the machine models. To prove that a relation is a WEB, reasoning about single steps of $\dashrightarrow$ suffices. It turns out that if MA is a refinement of ISA, then the two machines satisfy the same formulas expressible in the temporal logic CTL$^* \setminus$ X, over the state components visible at the instruction set architecture level.

There are several ways in which we can model and verify these pipelined machines using the refinement-based notion of correctness that we use. We can model and verify these machines at a high-level of abstraction or at the bit-level, and there are advantages and disadvantages to both approaches. These approaches can be classified into two categories, which are approaches based on the use of deductive reasoning, and approaches that use decision procedures. We describe these approaches in the subsequent sections.

2.3 Term-Level Modeling and Verification

One approach for checking the correctness of pipelined machines is to model these machines at a high-level of abstraction, resulting in what we call term-level models, and verify them using decision procedures. Term-level models are obtained by manually abstracting RTL/bit-level models, and they use numerous abstractions. The bit-vector data path is abstracted using integers (also known as terms in this context), and therefore the name term-level models. Combinational circuit blocks such as the ALU are abstracted using black box functions, which are functions that have no body. These models use numerous other abstractions for instruction and data memory, register file, branch prediction, decoding logic, *etc*. In this section, we describe how to model and verify the three stage pipelined machine (3PM) using the UCLID system.

UCLID is a tool for verifying properties of term-level models and has been used to check properties about our-of-order microprocessor models [49]. The UCLID specification language can be used to model pipelined machines at the term-level and specify properties about these models.

We now describe how we model ISAS at the term-level using UCLID. The UCLID specification corresponding to the ISA machine is shown in Figure 3.

The program counter is modeled as an integer variable (sPC in Figure 3). The instruction memory is modeled as a function that maps addresses to data values. For this example, we assume that the instruction memory (IM in Figure 3) is only read from and never updated or written to. Therefore, we can model IM as an uninterpreted function. The register file is modeled as a state variable that can hold function values (sRF in Figure 3). Therefore, we can both read from the register file and update the register file. Note that the *init* and *next* operators are used to specify an initial value and the transition relation, respectively, for a state element.

Typically using UCLID, combinational circuit blocks such as the ALU are modeled using Uninterpreted Functions (UFs). For the ISA machine, we model the function that increments the program counter, the three decoder functions (i.e., decoding of *src1*, *src2*, and *dest*), and the ALU using the uninterpreted functions *inc*, *src1*, *src2*, *dest*, and *alu*, respectively.

We now describe how the three stage pipelined machine is modeled using the UCLID specification language. The UCLID specification of the three stage pipelined machine is shown in Figure 4.

```
DEFINE

(* Instruction read from the instruction memory *)

        inst := IM(sPC);

(* Two source arguments read from the register file *)

        arg1 := sRF(src1(inst));
        arg2 := sRF(src2(inst));

(* Result computed using the alu *)

        result := alu(arg1, arg2);

ASSIGN

(* Program Counter *)

        init[sPC] := pc0;
        next[sPC] := inc(sPC);

(* Register File *)

        init[sRF] := rf0;
        next[sRF] := Lambda ( a ) .
            case
                (a = dest(inst)) : result;
                          default : sRF(a);
            esac;
```

Figure 3: Part of a UCLID specification that describes a simple instruction set architecture example, we call (ISAS).

Just as with the ISA, the program counter is modeled as the integer variable pPC, the instruction memory is modeled as an Uninterpreted Function IM, and the register file is modeled as the function variable pRF. The first source operand stored in the first pipeline latch is modeled using the integer variable *dearg1*. The second source operand and the destination address stored in the first pipeline latch are modeled as integer variables. The result in the second pipelined latch is modeled as the integer variable *ewresult*. The destination address stored in the second pipeline latch is also modeled using an integer variable. A valid bit is associated with each of the pipeline latches, which are modeled as a boolean variables. Just as with the ISA, the ALU, the three decoder functions, and the function that increments the program counter are modeled using Uninterpreted Functions *alu*,

```
DEFINE


(* Instruction read from the instruction memory *)
        inst := IM(pPC);

(* Two source arguments read from the register file *)
        arg1 := pRF(src1(inst));
        arg2 := pRF(src2(inst));


(* stalling logic *)
        stall := (devalid & ewvalid &
                  ((desrc1 = ewdest) | (desrc2 = ewdest)));


(* Result computed using the alu *)
        result := alu(dearg1, dearg2);

ASSIGN

(* Program Counter *)

        init[pPC] := pc0;
        next[pPC] := inc(pPC);

(* Pipeline latch 1 *)

        init[dearg1] := dearg10;
        next[dearg1] :=
            case
                stall : dearg1;
              default : arg1;
            esac;

        ...

(* Pipeline latch 2 *)

        init[ewresult] := ewresult0;
        next[ewresult] := result;

        ...

(* Register File *)

        init[pRF] := rf0;
        next[pRF] := Lambda ( a ) .
            case
                (a = ewdest) & ewvalid : ewresult;
                               default : pRF(a);
            esac;
```

Figure 4: Part of a UCLID specification that describes part of the three stage pipelined machine model example. The full UCLID description of the pipelined machine model is not shown due to space limitations.

src1, *src2*, *dest*, and *inc*, respectively.

Even though term-level models use numerous abstractions, they are useful as properties about models defined at the term-level can be reduced to decision problems in a restricted fragment of first-order logic, such as the logic of Counter arithmetic with restricted Lambda expressions and Uninterpreted functions (CLU) [20]. Efficient decision procedures such as UCLID [20] can be used to decide CLU formulas. By specifying the refinement based correctness criterion as a statement expressible in CLU, we can use UCLID to automatically check the correctness of pipelined machines (see Chapter 3 for more details). In fact, the UCLID decision procedure has been used extensively in the work described in this dissertation in developing efficient methods to automate the verification of term-level pipelined machines. Below we give a brief description of the CLU logic and the UCLID decision procedure.

The CLU logic contains the boolean connectives, uninterpreted functions (UFs), uninterpreted predicates (UPs), equality, counter arithmetic, ordering, and restricted lambda expressions. The basic CLU types are booleans and integers (also known as terms). UFs and UPs are essentially black box functions that have a name but do not have a function body. The only property satisfied by a UF or a UP is functional consistency, *i.e.*, if the inputs to two different applications of a function are equal, then their outputs are equal. A UF takes integer inputs and produces an integer output, whereas a UP takes integer inputs and produces a boolean output. Lambda expressions are restricted in that they can only take integer inputs. Therefore, it is not possible to express any form of recursion or iteration in the CLU logic. The only arithmetic operations on integers allowed are counter arithmetic, *i.e.*, integers can be incremented and decremented.

At the heart of UCLID is a decision procedure for formulas expressible in the CLU logic. The UCLID symbolic simulation engine unrolls the specification to generate a formula in CLU, which is then reduced to a SAT problem in CNF format. UCLID uses a number of sophisticated techniques such as efficient encoding schemes and positive equality in this reduction from CLU to CNF. A state-of-the-art SAT solver can then be used to check the SAT problem. If UCLID was able to prove that the property does not hold on the system, then it will generate a counter example, which is a satisfying assignment to all variables in the CLU formula that was generated by UCLID from the original verification problem. The counter example is very useful as it can be interpreted in context

to determine the reason the property failed to hold on the system.

Recently, there have been significant advances in decision procedures and initial experiments show that they will be able to automatically handle significantly harder pipelined machine verification problems than can be currently handled by UCLID. Examples of such decision procedures include DPLL(T) [26] and Yices [100].

While approaches based on decision procedures are automatic and efficient, there are several drawbacks. The term-level models lack a firm connection to the RTL, the level of abstraction at which commercial designs are functionally verified. As can be seen from the term-level model of 3PM, large pieces of the design such as the ALU are missing. Therefore, many errors are not caught [61]. Also, the models are not executable, which makes it very hard to debug. In the next section, we describe how to model and verify 3PM at the bit-level.

2.4 Bit-Level Modeling and Verification

Hardware designs of commercial systems are functionally verified at the RTL/bit-level. Approaches based on the use of deductive reasoning can be used to verify pipelined machines defined at the bit-level. In this section, we show how to model and verify the three stage pipelined machine example (3PM) using the ACL2 theorem proving system. Manolios also describes in detail the modeling and verification of a more complex pipelined machine example using refinement and ACL2 [55].

ACL2 stands for "A Computational Logic for Applicative Common Lisp." It is the name of a programming language, a first-order mathematical logic based on recursive functions with induction, and a mechanical theorem prover for that logic [43, 47, 42]. We choose to work with ACL2 because it is an industrial strength theorem prover that has successfully applied to model and verify commercial systems. Some examples of commercial applications of ACL2 include the verification of floating point units of the AMD-K5 processor [86], the AMD-K7 processor [85], and the IBM Power4^{TM} processor [91]. ACL2 has been used as part of the verification effort of an IBM secure co-processor [94] and an intrinsic partitioning mechanism in the AAMP7 avionics microprocessor [29]. ACL2 has also been used to analyze bit and cycle accurate models of the Motorola CAP, a digital signal processor [17],

The ACL2 programming language is an applicative (side-effect free or purely functional) subset

of Lisp. ACL2 is also a total programming language, meaning that ACL2 functions are defined for any input irrespective of its type. For example, consider the expression `(and 5 t)`, where `t` is the boolean value *true*. Executing `(and 5 t)` will not result in an error and will return `t`, as the `and` function treats `5` as `t`.

We now show how to model both ISAS and 3PM using ACL2 at the bit-level. Part of the ACL2 bit-level model of ISAS is shown in Figure 5. The entire model is not shown due to space limitations. The *step-isa* function corresponds to the operational semantics of ISAS and is defined using several other functions, some of which are shown. Bit-vectors are represented as lists of 1s and 0s. The *nth* function is used to get the nth bit of a list, and is used for manipulating bit-vectors. The *src1* and *src2* functions both take an instruction as input and are used to compute the source addresses. The *src1* function pulls out the first four bits of the instruction and constructs the first source address using these bits. The register file and the instruction memory are modeled an association lists (alists). The *s* function is used to update an alist and the *g* function is used to read the data from an alist corresponding to a given key. The *nextsrf* function corresponds to the transition relation of the register file. Other functions such as the *alu*, which describes the ALU are defined at the bit-level, but are not shown here. The *let* construct is used to bind lexically scoped local variables. *seq* is a macro for constructing and updating records. A field of a record can be read using the *g* function. A state of ISAS is modeled as a record with three fields, which are the program counter (*'pc*), the register file (*'rf*), and the instruction memory (*'imem*).

Part of the ACL2 bit-level model pf 3PM is shown in Figure 6. The entire model is not shown due to space limitations. The *step-ma* corresponds to the operational semantics of 3PM and is defined in a style similar to ISAS. Many of the functions used to define ISAS such as the decoder functions (*src1*, *src2*, *etc*) and the function that describes the ALU are also used in the 3PM model. A state of the 3PM includes the state components of ISAS and other components that correspond to the two pipeline latches.

The WEB-refinement theorem that relates 3PM and ISAS is expressible in the ACL2 logic, which may be thought of as first-order predicate calculus with equality, recursive function definitions, and mathematical induction. In the logic, the primitives of applicative Common Lisp are

```
(defun src1 (x)
  (list (nth 0 x)
        (nth 1 x)
        (nth 2 x)
        (nth 3 x)))

(defun src2 (x)
  (list (nth 4 x)
        (nth 5 x)
        (nth 6 x)
        (nth 7 x)))

...

(defun nextsrf (inst rf result)
  (s (dest inst) result rf))

(defun step-isa (isa)
  (let*
      ((pc (g 'pc isa))
       (rf (g 'rf isa))
       (imem (g 'imem isa))
       (inst (select pc imem))
       (arg1 (g (src1 inst) rf))
       (arg2 (g (src2 inst) rf))
       (result (alu arg1 arg2)))
    (seq nil
         'pc (pcadd pc)
         'rf (nextsrf inst rf result)
         'imem imem)))
```

Figure 5: Part of an ACL2 model of the instruction set architecture example (ISAS).

axiomatized, as are the basic data types, including natural numbers, integers, rationals, complex rationals, ordered pairs, symbols, characters, and strings. A principle of definition is also provided, by which the user can extend the axioms by the addition of equations defining new function symbols. Only terminating recursive definitions can be so admitted under the definitional principle.

We can now use the ACL2 theorem prover to prove that 3PM refines ISAS. Below we give a brief overview of the ACL2 theorem prover, which is essentially an integrated system of *ad hoc* proof techniques that include simplification, generalization, induction, and many others. The user interacts with the theorem prover my modeling an artifact such as a hardware system using the

```
(defun nextprf (ewvalid ewdest ewresult rf)
  (cond
   (ewvalid (s ewdest ewresult rf))
   (t rf)))

(defun step-ma (ma)
  (let*
      ((pc (g 'pc ma))
       (rf (g 'rf ma))
       (imem (g 'imem ma))
       (dearg1 (g 'dearg1 ma))
       (dearg2 (g 'dearg2 ma))
       ...
       (ewdest (g 'ewdest ma))
       (ewvalid (g 'ewvalid ma))
       (ewresult (g 'ewresult ma))
       (inst (g pc imem))
       ...
       (arg1 (g (src1 inst) rf))
       (arg2 (g (src2 inst) rf))
       (result (alu dearg1 dearg2)))
    (seq nil
         'pc (pcadd pc)
         'rf (nextprf ewvalid ewdest ewresult rf)
         'imem imem
         'dearg1 (cond (stall dearg1) (t arg1))
         ...
         'ewresult result
         ...
         'ewdest dedest
```

Figure 6: Part of an ACL2 model of a the three stage pipelined machine example (3PM).

ACL2 language. The user then invokes the theorem prover by posing a conjecture about the model. The user can provide hints to the prover to guide its proof search. The theorem prover then uses a sequence of proof mechanisms, which are used to try and prove the conjecture. One of the primary techniques used is simplification. The prover has a database of rules, which it uses to simplify/transform the conjecture. If the conjecture is proved, it is added to the rule database and used in subsequent proof attempts. Otherwise, the prover produces a failed proof, which can be studied by the user to determine why the proof failed.

To check properties about complex models, the user has to first prove a number of rules that can

be stored in the rule database and used by the prover to reason about various parts of the model. Using a theorem prover such as ACL2 to prove properties about reasonably complex models is a laborious process and can typically require an enormous amount of effort on the part of the expert user.

In fact, Manolios has proved WEB-refinement theorems for various three stage pipelined machines using ACL2. Some of these models were described at the RTL and included interrupts. Some models were deterministic, while others were nondeterministic. Using the *records* book that had many useful rules about s and g functions, Manolios was able to prove refinement theorems using both flushing and commitment refinement maps automatically.

We tried to use a similar approach to verify a 5 stage term-level DLX pipelines machine, but found that the deductive reasoning approach did not scale well. The proof took fifteen and a half *days* for ACL2 [56] to complete, but required only 3 seconds using UCLID. To reduce the time required for ACL2 to complete the proof, one would have to start looking very closely at the internals of the pipeline and prove low-level invariants. In fact, there have been several efforts to verify complex pipelined machines using theorem provers. But, experts have found this to be a laborious process requiring an extraordinary amount of effort. For example, Sawada and Hunt [89, 90] have used ACL2 to verify complex pipelined machines models using ACL2 and required to prove a large number of low-level invariants about the internals of the design.

There is also recent progress in the development of decision procedures that can reason at the bit-level, an example of which is the Bit-level Analysis Tool (BAT) [67, 66]. BAT uses a state-of-the-art memory abstraction technique [65] and an efficient method for generating SAT problems from high-level circuit representations [68]. BAT has been used to verify a 32-bit 5 stage pipelined machines in approximately 2 minutes. The only drawback with such approaches is that the they do not scale well as they are limited by the complexity thresh hold of the decision procedure used.

2.5 Conclusions

We described a simple three stage pipelined machine model and showed how to model and verify it using two different approaches. The first approach was based on the use of decision procedures.

While the verification using this approach was efficient and automatic, the approach was only applicable to very high-level abstract models, which did not have a firm connection with the RTL. In contrast, using theorem provers such as ACL2, we could model and verify the three stage pipelined machine at the bit-level. But, we showed that approaches based on theorem provers did not scale well to more complex models.

In this dissertation, we develop a verification approach for bit-level pipelined machines that is based on using a combination of theorem proving and decision procedures. We show that in fact, using our approach, we can very complex processor models with only about 0.9 the amount of effort required to abstract and verify RTL models using UCLID. Note that using UCLID, the RTL models are manually abstracted, and there is no real connection between the abstract UCLID models and the original RTL model.

CHAPTER III

AUTOMATING SAFETY AND LIVENESS

In this chapter, we introduce a method for automatically verifying that a pipelined machine refines its instruction set architecture (ISA). The notion of refinement we use is based on Well founded Equivalence Bisimulations (WEBs) and guarantees that the pipelined machine and its ISA have the same safety and liveness properties. A consequence is that the pipelined machine satisfies exactly the same $\text{CTL}^* \setminus \mathsf{X}$ properties satisfied by its ISA (see Section 2.2 for a more detailed description of WEB-refinement).

The pipelined machine models we consider are abstract *term-level* models. Such models abstract away the datapath using integers, abstract away combinational circuit blocks (such as the ALU) using uninterpreted functions, and employ numerous other abstractions. The use of term-level models allows us to focus on the pipeline while ignoring other aspects of microprocessor designs. This helps make the verification problem tractable because there are powerful tools capable of automatically analyzing term-level models [20].

Automation of refinement proofs for term-level pipelined machine models is achieved in the following ways. First, we use domain knowledge about pipelined machines to strengthen the WEB refinement theorem to a statement expressible in the logic of counter arithmetic with lambda expressions and uninterpreted functions (CLU), a decidable logic. Second, we show how to define the refinement maps and rank functions required to state the refinement theorem. Refinement maps are functions that map pipelined machine states to ISA states and rank functions map pipelined machine states to natural numbers. As our machines are modeled at the term-level and our refinement-based correctness statements are expressible in CLU, we can use UCLID to automatically check the correctness statements [20].

We provide experimental results for 17 pipelined machine models of varying complexity with features such as precise exceptions, branch prediction, and interrupts. Our results show that our approach is computationally efficient, as verification times for WEB-refinement proofs are only

25% longer than the verification times for the standard Burch and Dill type proofs [22], which do not address liveness.

The rest of this chapter is organized as follows. In Section 3.1, we show how to reduce the WEB-refinement theorem to a statement expressible in CLU. In Section 3.2, we describe methods for automatic verification of term-level pipelined machines. In Section 3.3, we describe several complex pipelined machine models used to evaluate our methods. In Section 3.4, we report verification times and statistics for 34 pipelined machine models, some based on the flushing approach and some on the commitment approach. We compare the time taken to prove safety alone with the time taken to prove both safety and liveness. We also compare the flushing and the commitment approaches. Related work and Conclusions appear in Sections 3.5 and 3.6, respectively.

3.1 Core Theorem

The general statement of WEB refinement is not expressible in a decidable fragment of first-order logic, such as CLU. Therefore, to achieve automation of refinement proofs, we use domain knowledge about pipelined machines to strengthen the WEB refinement theorem to a statement expressible in CLU, called the *core theorem*. We start by defining the equivalence classes of a WEB, B, to consist of one ISA state and all the pipelined machine states that map to the ISA state under the refinement map, r. Now, condition 2 of the WEB definition (see Section 2.2) clearly holds. Our ISA and pipelined machines are deterministic (actually they are nondeterministic, but we use oracle variables to make them deterministic [53]), thus, after some symbolic manipulation, we can strengthen condition 3 of the WEB definition to the following *core theorem*, where *rank* is a function that maps states of the pipelined machine into the natural numbers.

$$\begin{aligned} \langle \forall w \in \texttt{MA} :: \quad & s = r(w) \ \wedge \ u = \texttt{ISA-step}(s) \ \wedge \\ & v = \texttt{MA-step}(w) \ \wedge \ u \neq r(v) \\ \Longrightarrow \quad & s = r(v) \ \wedge \ \mathit{rank}(v) < \mathit{rank}(w) \rangle \end{aligned}$$

In the formula above that is also depicted as a diagram shown in Figure 7, s and u are ISA states, and w and v are MA states; `ISA-step` is a function corresponding to stepping the ISA machine once and `MA-step` is a function corresponding to stepping the MA machine once. The core theorem says

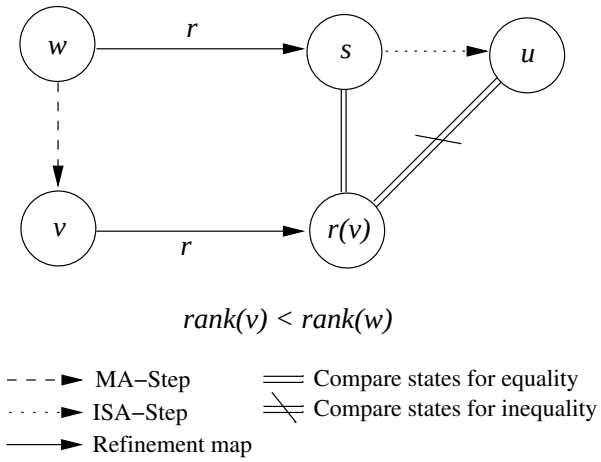


Figure 7: Diagram shows the core theorem that can be expressed in CLU logic.

that if w refines s, u is obtained by stepping s, v is obtained by stepping w, and v does not refine u, then v refines s and the *rank* of v is less than the *rank* of w. The proof obligation relating s and v is the safety component, and the proof obligation that $rank(v) < rank(w)$ is the liveness component. We use two types of refinement maps and provide a general method for defining *rank* functions in both cases. The details appear in Section 3.2, after we describe the pipelined machine models.

3.2 Verification of Pipelined Machines

Stating the core theorem described in Section 3.1 involves defining refinement maps and rank functions. We use two refinement maps, flushing and commitment. In this section, we describe these refinement maps and their associated rank functions.

As stated earlier, refinement maps are functions that map pipelined machine states to ISA states. ISA states contain the programmer visible components, including the program counter, the instruction memory, the data memory, and the register file. For the machines with exceptions, the programmer visible components also include the exception program counter and the exception flag. Pipelined machine states contain all the programmer visible components, and also include the pipeline registers.

3.2.1 Flushing Refinement Map

The flushing refinement map is defined using a flush operation that "pushes" instructions in the pipeline forward without fetching any new instructions [22]. A pipelined machine state is flushed

by applying a sufficient number of flush operations successively so that all the partially executed instructions in the pipeline are forced to complete without fetching any new instructions. The flushing refinement map is defined by flushing a pipelined machine state and projecting out the programmer visible components, resulting in an ISA state.

A pipelined machine model can be easily instrumented to enable such flushing by introducing an external input signal *flush*. A regular step is one in which the pipelined machine model is stepped with *flush* set to FALSE, in which case the pipelined machine behaves the same way as the uninstrumented machine. During a flushing step, *flush* is set to TRUE, and the machine is stepped without fetching a new instruction. In this case, the program counter is not incremented but can be updated by existing instructions in the pipeline (such as a branch instruction that mispredicts), and a bubble is introduced in the first stage of the pipeline. The maximum number of flushing steps, n, required to flush a pipelined machine state depends on the machine under consideration. For example, we require a maximum of 8 flushing steps to flush the pipelined machine model with 7 stages, as the most recently fetched instruction in the pipeline (the instruction in the pipeline register after the first fetch stage) can stall at most two times before it completes. When n flushing steps are applied to a pipelined machine, it reaches a flushed state, a state in which all its pipeline registers are invalid, meaning that the pipeline registers do not hold any valid instructions.

To check the safety component of the refinement theorem for the pipelined machine, we start from an arbitrary pipelined machine state w, and apply n flushing steps to reach w_f, the flushed state corresponding to w. Projecting out the programmer visible components from w_f results in the ISA state s. Next, we apply a regular step to the pipelined machine in state w to get v. Applying n flushing steps to v results in the flushed state v_f. Projecting out the programmer visible components in v_f results in the ISA state $r(v)$. ISA state u is obtained by stepping the ISA machine in state s. Now, the safety property based on the "core theorem" can be defined using states s, u, and $r(v)$. It turns out that for a single-issue pipelined machine, the safety proof of the core WEB theorem is similar to the Burch and Dill approach [22].

We now describe how flushing works for a simple three stage pipelined machine model (3PM). The model itself is described in Section 2.1.2. The three stage pipelined machine has two pipeline latches. If we consider any state, it can have a maximum of two partially executed instructions in

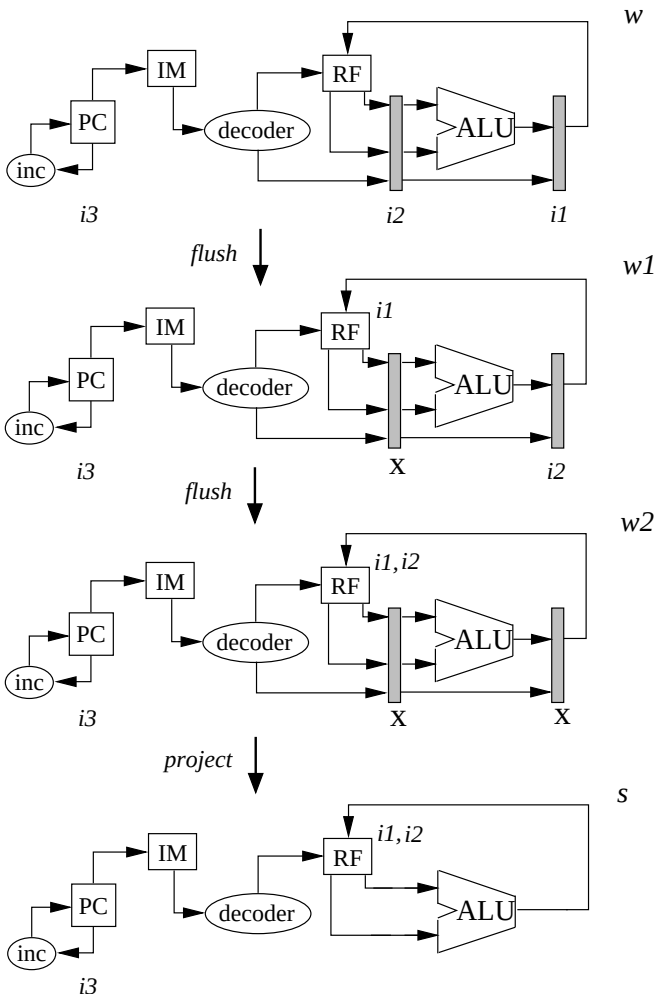


Figure 8: The figure depicts the flushing refinement map for state w of the three stage pipelined machine, 3PM. In state w shown in this figure, we assume that instruction $i2$ does not depend on instruction $i1$.

the pipeline latches. Therefore, it might seem that one can flush any pipelined machine state using two flush operations. But, notice that if the instruction in the first pipeline latch (say $i2$) depends on the instruction in the second pipeline latch (say $i1$), then $i1$ is stalled for one step. Therefore, a maximum of 3 flush operations are required to flush any state of the three stage pipelined machine.

In Figure 8 we depict how the pipelined machine state w is flushed. In further discussions, when we say that a pipeline latch or register is valid, we mean that the pipeline latch or register holds a valid instruction. Similarly, an invalid pipeline latch does not hold a valid instruction. The flush operation is shown as *flush* in the figure. The *project* operation in the figure indicates the projection of the programmer visible components of a pipelined machine state onto an ISA state. w has two

partially executed instructions in the pipeline latches, $i1$ and $i2$, where $i1$ is the older instruction and $i2$ does not depend on $i1$. The program counter is pointing to instruction $i3$ in the instruction memory. If we flush w for one step, then instruction $i1$ completes and updates the register file, and instruction $i2$ moves to second pipeline latch. Since no new instruction is fetched, the first pipeline latch becomes invalid (indicated using a x in the figure). The resulting pipelined machine state is $w1$. If we flush $w1$, we reach state $w2$, where $i2$ also completes and updates the register file. State $w2$ is a flushed state as all the pipelined latches are invalid. Projecting out the programmer visible components from state $w2$ gives the ISA state s, corresponding to the pipelined machine state w.

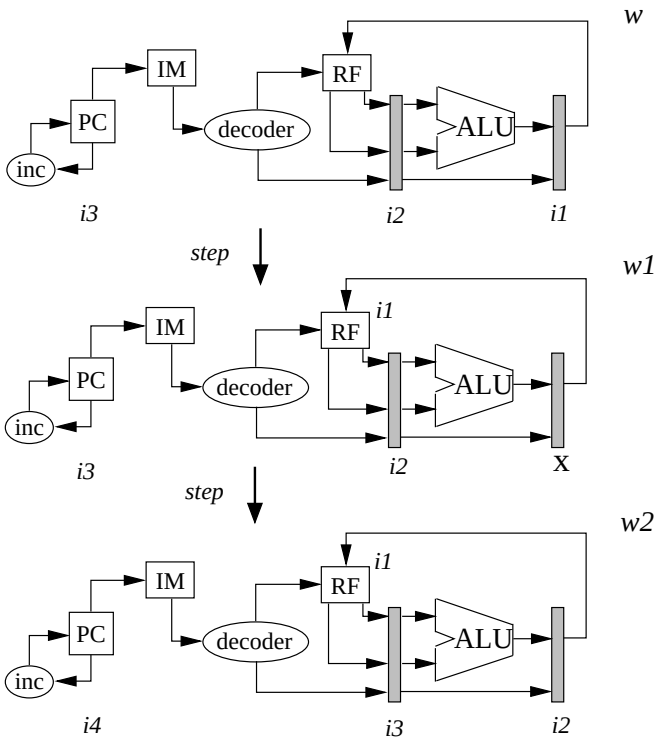
3.2.2 Liveness

The liveness component of the refinement theorem is checked by comparing the *rank*s of w and v. For the flushing refinement map, we define the *rank* of a pipelined machine state to be the number of pipelined machine steps required to fetch an instruction that eventually completes. An initial attempt at automatically computing the rank of a state is as follows. Starting with a state, say p_0, we take n steps, leading to states $p_1, \ldots, p_n$. We also apply the refinement map to each of the p states, leading to the sequence of states $q_0, \ldots, q_n$. The rank of p is the smallest value of i such that $q_i \neq q_0$. Unfortunately, defining *rank* in this way requires a larger number of symbolic simulation steps than UCLID can handle.

We now introduce another method for defining *rank*. This new method is the one we actually use and it is much more amenable to analysis. Starting with a state, say p_0, we take k steps, where k is the number of steps required for the data in the first pipeline register of p_0 to reach the last pipeline register (of p_k). We then keep stepping p_k until we reach a state p_l such that the last pipeline register of p_l is valid. The *rank* of p_0 is then $l - k$; that is, the *rank* of a state is the number of steps required for a new instruction to reach the end of the pipeline, after all previous instructions have finished.

As a final remark, note that even if the rank function is erroneously defined, no unsoundness can result. This is because the core theorem guarantees that a WEB-refinement exists if *any* rank function makes it true. The practical implication is that erroneous rank definitions are caught during verification.

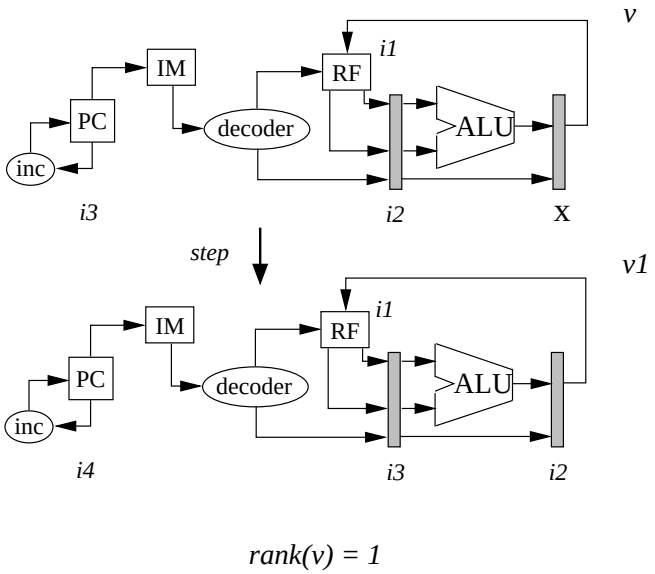
In Figures 9 and 10, we show how the rank of two pipelined machine states w and v of a simple



rank(w) = 2

Figure 9: Figure shows the computation of the rank function for a concrete pipelined machine state *w*. In this state, the instruction in the first pipelined latch (*i*2) depends on the instruction in the second pipelined latch (*i*1).

three stage pipelined machine model, 3PM (described in Section 2.1.2), are computed. In both the figures, *step* indicates a step of 3PM. For the state *w*, since the instruction in the first pipeline latch *i*2 depends on *i*1, *i*2 is stalled for one cycle before it can make progress. Therefore, a new instruction is fetched only after two steps of 3PM starting at state *w*. Since 3PM does not have any mechanisms for invalidating, all instructions that are fetched will complete. Therefore, the rank of state *w* is two. Stepping 3PM in state *v* on the other hand results in an instruction fetch, as the second pipeline latch in state *v* is invalid and *i*2 will not be stalled and will make progress. Therefore, the rank of *v* is one. Note that *v* is actually the state obtained by stepping *w*. When *w* is stepped, it stalls and does not make any forward progress with respect to the ISA, therefore the value of rank decreases from two to one.



rank(v) = 1

Figure 10: Figure shows the computation of the rank function for a concrete pipelined machine state *v*. Note that state *v* is obtained by stepping state *w* shown in Figure 9

3.2.3 Commitment Refinement Map

Given a pipelined machined state, the commitment refinement map returns the ISA state obtained by invalidating all partially executed instructions in the pipeline, undoing any effect they had on the programmer-visible components, and projecting out the programmer-visible components [51].

The commitment refinement map can be easily implemented using history variables, variables that record the history of the programmer visible components. We keep track of the values of the programmer visible components before they were updated by each of the partially executed instructions in the pipeline. For example, in the 7 stage machine models, we keep track of the last 6 values of the program counter. The commitment refinement map can now be defined by invalidating the partially executed instructions and setting the programmer visible components to their values before they were updated by the oldest instruction in the pipeline. Note that no symbolic simulations are required.

Figure 11 shows how the commitment refinement map is computed for a state of the three stage pipelined machine, called 3PM. A description of 3PM can be found in Section 2.1.2. The *project* operation in the figure indicates the projection of the programmer visible components of a pipelined machine state onto an ISA state. The *pull-back* operation in the figure indicates rolling back the

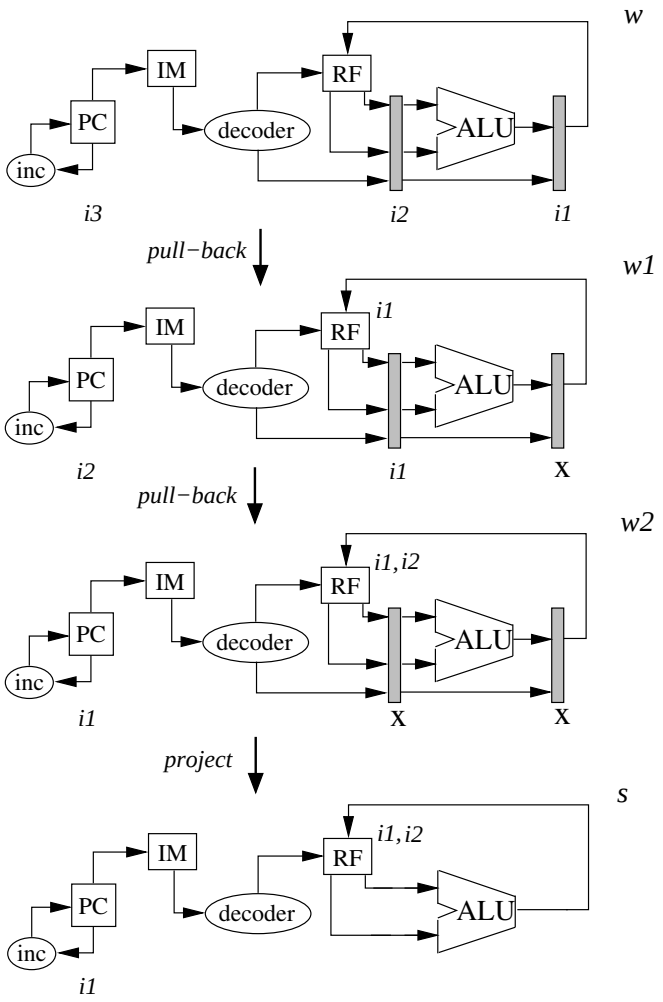


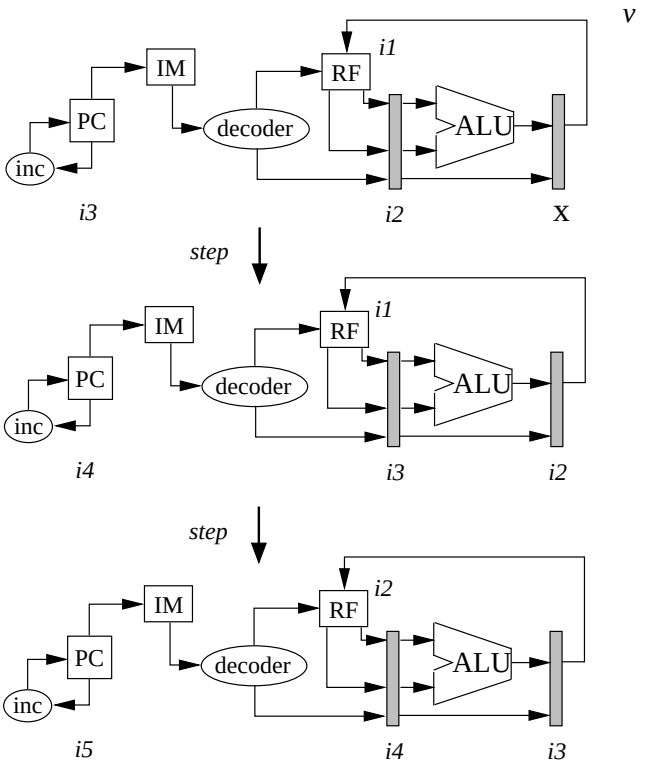
Figure 11: The figure depicts how the commitment refinement map is computed for state *w* of the three stage pipelined machine, 3PM.

pipelined machine for one step. Note that we use the *pull-back* operation to describe the intuition behind the commitment refinement map. As stated earlier, when computing the commitment refinement map, we do not roll the pipelined machine backward. Instead we use history variables to implement the commitment refinement map.

The pipelined machine state *w*, shown in Figure 11 has two partially executed instructions $i1$ and $i2$, and the program counter is pointing to instruction $i3$ in the instruction memory. The ISA state corresponding to *w* can be computed by rolling back state *w* for two steps. This results in a 3PM state $w2$ where the effect of instructions $i2$ and $i1$ on the programmer visible components is undone, the two pipeline latches are invalid, and the program counter is pointing to instruction $i1$. Projecting out the programmer visible components in the resulting state, gives the ISA state *s*

corresponding to *w*.

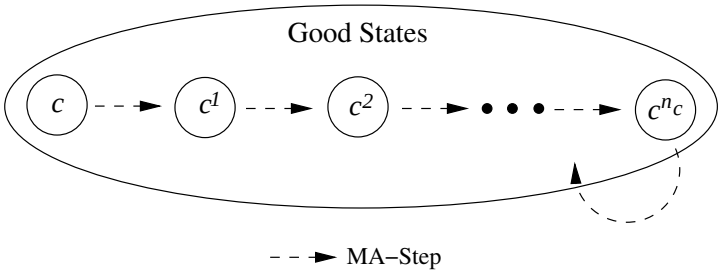
For the commitment refinement map, we define the *rank* of a pipelined machine state to be the number of steps required to commit an instruction. The first instruction that gets committed is the oldest. In addition, for the machines we consider, the flow of an instruction through the pipeline is only affected by older instructions in the pipeline. Therefore, the number of steps required to commit the oldest instruction is essentially the number of pipeline registers between that instruction and the end of the pipeline, which is how we define *rank*.



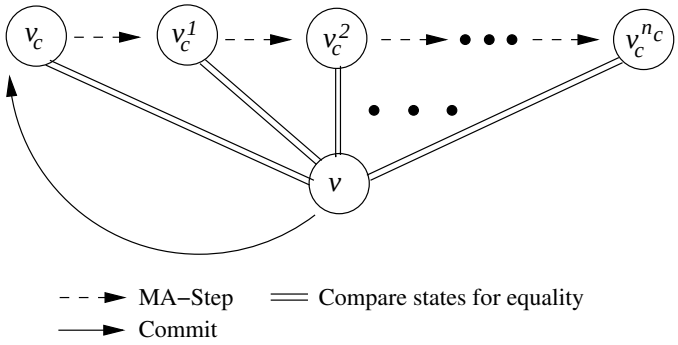
rank(v) = 2

Figure 12: The figure shows the computation of the rank function corresponding to the commitment refinement map for a pipelined machine state *v*.

Figure 12 depicts the computation of the rank corresponding to the commitment refinement map for a state of 3PM, namely *v*. Note that in the figure *step* corresponds to a step of the pipelined machine. The instruction next to the register file (RF) indicates the last committed instruction. As can be seen from the figure, the second pipeline latch is invalid, and the first pipeline latch contains instruction *i*2. It takes two steps for *i*2 to complete, and therefore, the rank of state *v* is two.



(a) The LFP invariant states that a state is good if it can be reached from a committed state, c, within u steps, where u is a machine specific upper bound. States reached after more than u steps must be reachable within u steps from some other committed state.



(b) To check that an MA state v is good, commit v to get x and check if v is equal to one of the states obtained by stepping x from 0 to u steps, where u is a machine specific upper bound.

Figure 13: The Least Fixpoint (LFP) Invariant

To use the commitment refinement map, we require an invariant that characterizes the set of reachable pipelined machine states. To see why, consider a state, w, of the three stage pipelined machine (3PM) that has only one instruction in the pipeline, but that instruction does not match any instruction in the instruction memory. Committing and projecting the programmer visible components in w results in state s, however w and s will not have the same infinite executions up to stuttering because w will eventually execute its instruction, which will differ from the instruction s executes.

We now show how to define the required invariant. First we define the notion of a committed state, which is a pipelined state in which all the pipeline registers are invalid. A pipelined state is "good" if it is reachable from a committed state. The set of good states is an invariant that we call

the Least Fixpoint (LFP) invariant, as computing this set involves a least fixpoint computation (see Figure 13(a)). To check that a pipelined machine state, w, is good, we start by computing c, the committed state corresponding to w. State w is good if it is equal to any of the states obtained by stepping c for some number of steps up to u, an upper bound depending on the pipelined machine design. For example, for a machine model with 7 stages, the upper bound is 6, which is the largest number of steps required for a new instruction to travel through and reach the end of the pipeline.

To prove that the LFP invariant really is an invariant requires that we show that the good states are closed under the pipelined machine transition relation. The invariant proof is trivially true for the good MA states that are less than u steps from a committed state, as their successors are within u steps from a committed state. Therefore, all we need to show is that the successor of any state that is u steps from a committed state is good, as depicted in Figure 13(b).

Having established the LFP invariant, we prove the refinement theorem using case analysis. A good state is either an arbitrary committed state or a state reachable in $1,\ldots,u$ steps from an arbitrary committed state. We then prove the "core theorem" for each of these possibilities.

3.2.4 Remarks

The core theorem is easily expressible in the CLU logic, as the successor function can be used to directly define the rank functions. However, we can do without the successor function since the *rank* of a state is always less than the number of registers in the pipeline. This means that our approach is applicable even with tools that only support propositional logic, equality, uninterpreted functions, and memories, but we find that defining the rank explicitly is clearer.

3.3 Benchmarks

We automatically verify a number of pipelined machine models that are obtained by extending a base model with various features. The base model has 6 pipeline stages including an instruction fetch (IF1), an instruction decode (ID), an execute (EX), a 2-cycle memory access (M1 and M2) and a write back (WB). The models implement five instruction types including register-register, register-immediate, load, store, and branch. The branch and store instructions complete out of order with respect to the ALU instructions. This base model is extended with a pipelined fetch stage, branch prediction, ALU exceptions, interrupts, and an instruction queue. Figure 14 shows

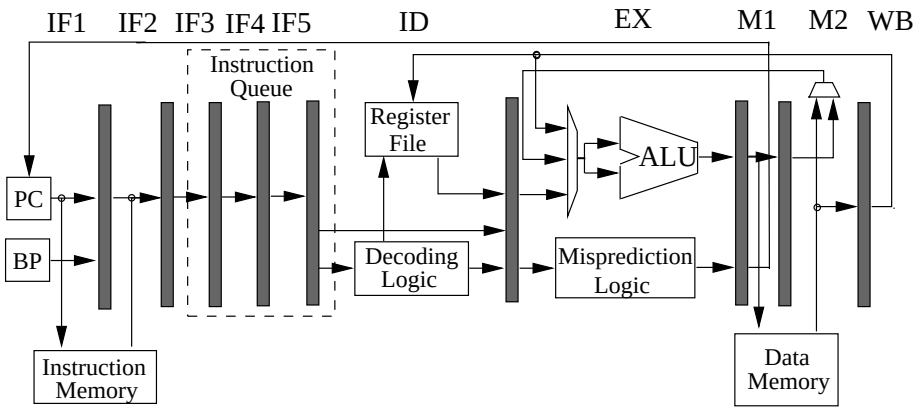


Figure 14: High-level organization of 10 stage pipeline machine.

the high-level organization of a complex 10 stage pipelined machine model with branch prediction, pipelined fetch stage and a 3-stage instruction queue. The 7 stage models are inspired by the Intel XScale architecture and the other pipelined machine models are obtained by extending these 7 stage models. Some of our modeling techniques such as those used for branch prediction and exceptions are based on [99].

We now describe how we model the pipelined machines at the term-level. Combinational circuit blocks such as the ALU are abstracted using Uninterpreted Functions (UFs). The register file is modeled using using restricted lambda expressions. The read and write accesses to the data memory in the pipelined machines models we consider are in-order. Therefore, we use a term variable to model the data memory and UFs to model the read and write functions of the data memory. Since the instruction memory is never updated, we model it using a UF. Counter arithmetic, implemented using ordering, and interpreted functions successor and predecessor, is used to define rank functions.

Interrupts are modeled with a term variable *INPState* that stores the state associated with the generation of the interrupt, a UF *NextINPState* that takes *INPState* as input and produces the next interrupt state, and UP *IsInterrupt* that also takes *INPState* as input and produces a boolean value that indicates if an interrupt is raised. Interrupts are detected in the M1 stage and invalidate all instructions in pipeline latches before the M1 stage including the instruction that caused the interrupt. We use temporal abstraction to model the behavior of interrupts, as in our model the result of an interrupt can be seen in one step of the machine. An interrupt modifies the data memory arbitrarily

to model the result of running an interrupt handler and sets the PC to the program counter corresponding to the first instruction that was invalidated by the interrupt. An arbitrary modification to the data memory is implemented using an uninterpreted function that takes the current data memory as input and returns the modified data memory.

We use two approaches to abstract branch predictors. In the first approach, a branch predictor is abstracted with a term variable *BPState* that corresponds to the current state of the branch predictor. Also, two UFs *NextBPState* and *PredictTarget*, and a UP *PredictDirection* are used, all of which have one input, *BPState*. The output of *NextBPState*, *PredictDirection*, and *PredictTarget* is the next state of the branch predictor, a prediction on the direction, and a prediction on the target of the branch, respectively. We call this the general branch prediction abstraction scheme. In the second approach, the branch predictor is abstracted using a non-deterministic input that corresponds to the state of the branch predictor. The predictions on the direction and target of the branch are determined using the UF *PredictTarget*, and the UP *PredictDirection*, respectively. We call this the non-deterministic branch prediction abstraction scheme. The actual direction and target of the branch are determined in EX. Mispredictions are corrected in M1. What is verified is the circuit that implements the misprediction logic.

ALU exceptions are modeled with a UP that takes the same input as the ALU, and outputs a predicate indicating if an exception is raised. M1 handles ALU exceptions in the following way. In case of an ALU exception, all instructions in pipeline latches before the M1 stage are invalidated, the program counter is updated with the address corresponding to the ALU exception handler routine, and the PC of the excepting instruction is stored in the Exception Program Counter (EPC). A return-from-exception instruction is also implemented that restores the PC with the EPC.

To show how ISA and pipelined machines can be modeled at the term-level, we describe in detail, two simple examples of term-level models. The examples are based on ISAS (a simple ISA that is described in Section 2.1.1) and 3PM (a simple three stage pipelined machine that implements ISAS and is described in Section 2.1.2).

3.4 Results

In this section, we describe the results obtained from automatically verifying safety and liveness for 17 pipelined machine models using both flushing and commitment refinement maps. In summary, we find that verification times increase 17% when proving safety and liveness over the time required to prove just safety. Interestingly, when using the commitment refinement map, there is no increase in verification times, but when using flushing verification times increase by about 23%. Finally, commitment takes less time overall and scales better than flushing.

The verification times and the statistics for the boolean correctness formulas are shown in Table 1 and Table 2, respectively. We report the number of CNF variables and clauses and the verification time for both the safety proofs and the safety and liveness proofs, *i.e.*, for the proofs of the core theorem. The total verification time reported includes the time taken by Siege and UCLID, thus the time taken by UCLID can be obtained by subtracting the Siege column from the Total column. Siege uses a random number generator, which leads to (sometimes large) variations in the execution times obtained from multiple runs of the same input; thus, in order to make reasonable comparisons, every Siege entry is the average over 10 runs. We also report the standard deviation of the 10 runs for every Siege entry in the safety and liveness proofs. The experiments were performed using the UCLID system (version 1.0) and the Siege SAT solver (variant 4) and run on a 3.06GHz Intel Xeon machine with an L2 cache size of 512 KB.

We use the following naming convention for the pipelined machine models and the verification problems. A model name begins with either "c" or "f," indicating that the commitment or flushing refinement map is used, respectively. Following is a number that indicates the number of pipeline stages. This is followed by the optional letters "b", "n", "e", and "p", indicating the presence of the general branch prediction abstraction scheme, the nondeterministic branch prediction abstraction scheme, exceptions, and interrupts, respectively.

The overhead cost of liveness, computed by subtracting the sum of the "Safety Verification Times Total" column from the sum of the "Safety and Liveness Verification Times Total" column and dividing by the latter is 17%; notice that for the commitment approach it is -1.6%, whereas it is 23% for the flushing approach. Since the liveness and safety theorems share considerable structure

Table 1: Verification Times

Verification Problem	Safety		Safety and Liveness		
	Verification Times (secs)		Verification Times (secs)		
	Siege	Total	Siege	Stdev	Total
c6	28	30	28	5.4	30
c6n	160	165	170	35.5	175
c6b	121	124	119	16.0	122
c7	25	27	26	3.8	28
c7n	226	230	234	43.4	237
c7b	201	204	222	52.6	225
c7be	199	203	213	25.3	217
c7bep	239	243	260	73.7	264
c8	27	29	31	5.8	33
c8n	770	776	758	66.9	764
c8b	560	564	493	83.8	497
c9	30	32	29	6.1	31
c9n	1,455	1,462	1,517	238.0	1,524
c9b	982	987	934	169.2	939
c10	33	36	31	7.1	34
c10n	2,901	2,910	2,641	358.2	2,650
c10b	1,675	1,681	1,774	423.2	1,780
f6	10	12	10	2.1	14
f6n	10	13	12	2.7	17
f6b	11	14	15	3.6	20
f7	134	138	135	6.8	142
f7n	109	114	136	14.9	145
f7b	124	129	139	24.5	148
f7be	134	139	157	23.2	167
f7bep	120	126	171	20.9	182
f8	821	828	697	31.8	709
f8n	597	605	716	49.5	731
f8b	594	602	699	55.1	715
f9	2,574	2,584	2,309	100.1	2,328
f9n	1,998	2,010	2,546	117.7	2,570
f9b	2,089	2,101	2,333	124.7	2,357
f10	5,407	5,422	6,385	506.9	6,411
f10n	4,229	4,247	6,726	365.8	6,761
f10b	4,039	4,057	6,540	584.9	6,575

Table 2: CNF Statistics

Verification Problem	Safety		Safety and Liveness	
	CNF Vars	CNF Clauses	CNF Vars	CNF Clauses
c6	12,817	37,876	12,334	36,442
c6n	37,718	111,790	37,147	110,077
c6b	22,410	66,223	21,850	64,558
c7	13,728	40,609	13,296	39,328
c7n	28,007	82,978	27,500	81,472
c7b	26,785	79,294	26,058	77,128
c7be	26,766	79,186	26,264	77,695
c7bep	26,806	79,291	26,615	78,733
c8	14,528	43,009	14,100	41,740
c8n	54,252	161,260	53,697	159,595
c8b	32,569	96,526	31,914	94,576
c9	15,648	46,369	15,214	45,082
c9n	63,101	187,771	62,536	186,076
c9b	37,539	111,376	36,757	109,045
c10	17,526	52,003	17,121	50,803
c10n	73,727	219,613	73,163	217,921
c10b	44,287	131,560	43,517	129,265
f6	13,429	39,694	28,256	83,725
f6n	16,462	48,529	37,452	110,920
f6b	17,135	50,548	37,002	109,570
f7	28,477	84,535	53,165	158,182
f7n	33,160	98,212	70,667	210,172
f7b	33,674	99,754	70,985	211,126
f7be	35,961	106,516	74,702	222,145
f7bep	37,599	111,406	81,759	243,259
f8	47,551	141,538	95,092	283,465
f8n	56,790	168,742	121,499	361,954
f8b	58,180	172,912	121,645	362,392
f9	70,295	209,551	144,045	429,973
f9n	87,650	261,001	185,149	552,412
f9b	87,278	259,885	183,371	547,078
f10	111,631	333,124	198,375	592,660
f10n	129,085	384,793	255,780	763,861
f10b	129,957	387,409	256,272	765,337

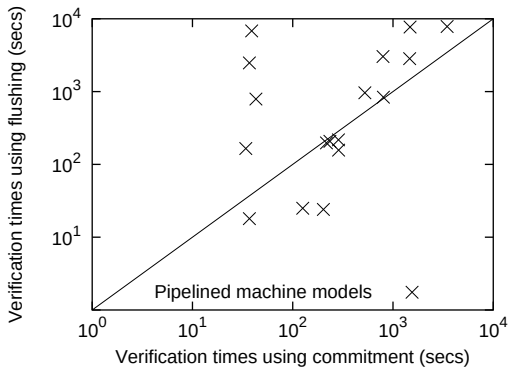


Figure 15: Comparison of commitment and flushing based on verification times.

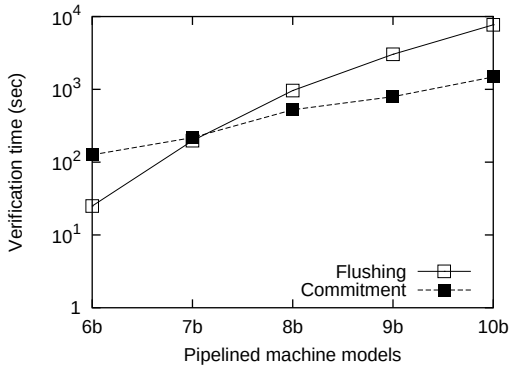


Figure 16: Variation in verification times with increase in the length of the pipeline for commitment and flushing.

(*e.g.*, the machine models), the SAT solvers are able to prove the conjunction of the two theorems in time comparable to the time required to prove just one of the theorems. In fact, in some cases the verification times for safety and liveness are slightly less than the verification times for safety alone (as is the case with some of the commitment problems), indicating that the heuristics of the SAT solver are able to effectively exploit the shared structure.

3.4.1 Commitment vs. Flushing

Figure 15 is a scatter plot that compares commitment and flushing using 17 pipelined machine models. Notice that both the *x* and *y* axis have a logarithmic scale. As can be seen from the figure, commitment does better than flushing on most of the models, especially on models with longer pipelines. This is depicted more clearly in Figure 16, which shows the variation in verification times with increase in the length of the pipeline for both flushing and commitment. Note that the

y-axis in Figure 16 has a logarithmic scale.

A crucial factor in understanding the results is the notion of *symbolic distance* of a problem, which is the maximum number of nested symbolic simulations required to state the refinement theorem for the problem. The complexity of the verification problem and the size of the CNF formulas generated increase as the symbolic distance increases. The intuition is just that larger symbolic distances mean that we have to reason about longer traces.

The symbolic distance required for flushing is greater than that required for commitment because the rank function for commitment is trivial, whereas the rank function for flushing is quite complicated. In fact, it is responsible for the larger symbolic distance required for the flushing refinement map.

The differences in verification times for the commitment approach when including a branch prediction abstraction scheme (for example c7 and c7b) can be understood by noting that we compute the strongest invariant and introducing branch mispredicts leads to an irregular set of good states. Since exceptions and interrupts are very similar to branch mispredicts, introducing these features does not affect verification times much. We would also like to note that a very large portion of the verification time for the commitment approach is spent in proving the LFP invariant [59].

When deciding between the commitment and flushing approaches, consider the following. First, flushing verification times are more sensitive to the depth of the pipeline than commitment verification times. Second, there are ways of optimizing both methods that should be considered [59, 60, 41]. Where possible, we prefer the commitment approach because its refinement map seems clearer and it has various advantages in the context of nondeterministic machines [51].

3.5 Related Work

We now review previous work on pipelined machine verification. The correctness of pipelined machines is a subject that has received much attention. Early, seminal work in this area is due to Burch and Dill, who introduced a notion of correctness based on commuting diagrams, we call the BD correctness criterion [22]. Aagaard et al., [3] [4] have provided a survey of the various notions of correctness for pipelined machines, most of which are variations of the BD correctness criterion.

Unfortunately, the BD notion is not as complete as we would like, *e.g.*, does not address liveness

and thus does not guarantees that the pipelined machine is free of deadlock and livelock. To overcome this limitation, a variation of the BD correctness criterion augmented with a liveness property was proposed by Sawada and was used to verify some very complex processor models using the ACL2 theorem proving system [89]. This strengthened notion of correctness is still not complete, as it is possible to mechanically prove that certain pipelined machines that can deadlock satisfy this notion of correctness [51]. Another approach that handles both safety and liveness is described in [98] and is also a variation of the BD correctness criterion. Liveness is proved by showing that the pipelined machine makes forward progress after a finite number of steps. The reported overhead of proving liveness for single-issue pipelines is about 80%, compared with 17% using our approach. Another difference is that our approach is based on proving refinement, which has certain advantages. For example, a consequence of our proofs is that the pipelined machine satisfies the same $\mathrm{CTL}^* \setminus \mathsf{X}$ properties as its ISA. Another advantage is that refinement is a compositional notion, which can be exploited to verify complex pipelined machines that cannot be handled using automatic monolithic approaches [58].

Automatic verification of term-level pipelined machines has directly benefited from advances in decision procedures. An early decision procedure for the logic of Equality with Uninterpreted Functions was due to Burch and Dill [22]. Another, more efficient, decision procedure for the same logic was given in [19], and that work was further extended in [20], where a decision procedure for the CLU logic was given. The decision procedure is implemented in UCLID, which has been used to verify out-of-order microprocessors [49]. Also worth mentioning is the SVC decision procedure, which was used to check the correct flow of instructions in a pipelined DLX model [72]. Jones et al. [39] used SVC to verify an out-of-order execution unit using incremental flushing.

3.6 Conclusions

We have presented a method to automatically verify safety and liveness properties of complex pipelined machine models based on WEB-refinement. Automation is achieved in two steps. First, we strengthen the WEB-refinement theorem so that it is expressible in a the CLU logic. Second, we provide a simple recipe to define the rank function that does not require any deep understanding of the machine models. We analyze two approaches based on commitment and flushing to define the

refinement map using extensive experimentation with 17 pipelined machine models. We find that commitment approach scales better than the flushing approach with increase in the length of the pipeline. Also, the cost of liveness when compared with the cost of proving both safety and liveness is about 23% for the flushing approach and is negligible for the commitment approach.

CHAPTER IV

OPTIMIZING COMMITMENT REFINEMENT MAPS

Refinement proofs for pipelined machines depend on a critical parameter: the refinement map, a function that relates pipelined machine states to instruction set architecture states. Chapter 3 describes methods to automatically check refinement proofs for term-level pipelined machines by providing recipes to define refinement maps using high-level information about the pipelined design. These methods for defining refinement maps are based on commitment [51, 52] and flushing [22]. The idea with commitment is that partially completed instructions are invalidated and the programmer visible components are rolled back to correspond with the last committed instruction. Flushing is a kind of dual of commitment, where partially completed instructions are made to complete without fetching any new instructions.

Refinement maps used for pipelined machine verification such as those based on flushing and commitment have a drastic impact on the verification times as these maps tend to be complex functions. The reason for the complexity of refinement maps is that they are in essence constructed by stepping the pipelined machine multiple times. As a result, the computational complexity of refinement proofs for even reasonably complex processor models grow easily beyond the complexity threshold of decision procedures such as UCLID.

In this chapter, an automatic technique for defining commitment refinement maps is introduced that provides over a 30-fold improvement in verification times over both the previous method for defining commitment-based refinement maps and the standard method for defining flushing-based refinement maps. Refinement maps based on commitment requires the use of invariants. Extensive profiling using 42 pipelined machine models shows that proving the invariant accounts for almost all of the verification time required when using the standard commitment-based refinement maps. Based on this observation, a new invariant is introduced called the Greatest Fixpoint invariant that can be used for commitment-based refinement proofs. Not only does the new invariant lead to an average speedup factor of about 30, but it is also *much* simpler to define, leading to a decrease both

in the human effort required to verify pipelined machines and in the code size.

The rest of the chapter is organized as follows. In Section 4.1, the pipelined machine models and verification benchmarks that are used for the experiments are described. In Section 4.2, the Greatest Fixpoint (GFP) invariant is introduced. Section 4.3 shows results obtained by applying the commitment refinement map using the GFP invariant on our 42 pipelined machine models. Finally, related work is presented in Section 4.4, and conclusions in Section 4.5.

4.1 Pipelined Machine Models and Benchmarks

For our experiments, we have created 42 pipelined machine models of varying complexity that include and extend the models described in Section 3.3. We start with a base processor model and extend it with features such as a pipelined fetch stage, a 3-stage instruction queue, two different ways of abstracting branch predictors, an instruction cache, a data cache, and a write buffer. The base processor model is a 6 stage pipelined machine with the following stages: instruction fetch (IF), instruction decode (ID), execute (EX), data memory access (M1 and M2), and write back (WB). We implemented ALU instructions, register-register and register-immediate addressing modes, loads, stores, and branch instructions. We assign names to the pipelined machine models that are consistent with the names in the "Processor" column of Table 3. The model names start with a number indicating the number of stages followed optionally by the letters "I", "D", "W", "B" and "N" indicating the presence of an instruction cache, data cache, write buffer, branch prediction abstraction scheme 1, and branch prediction abstraction scheme 2, respectively. By applying different refinement maps to the pipelined machine models, we get in all 210 benchmarks (5 verification problems for each pipelined machine model).

The basic features of the pipelined machines are modeled in a style similar to [57], which in turn are similar to [95]. The models are described at the term-level. Word-level values are abstracted using terms or integers and much of the combinational circuit blocks that are common between the pipelined machine and its ISA are abstracted using Uninterpreted Functions (functions that only satisfy the property of functional consistency). We use restricted lambda expressions to model memories. The caches and write buffer are modeled as described below.

We model a direct mapped instruction and data cache. The instruction cache is modeled using

three memory elements `ICache-Valid`, `ICache-Tag`, and `ICache-Block` that take the index as input and return a predicate indicating if the entry in the instruction cache is valid, the tag, and the data block, respectively. Three uninterpreted functions *GetIndex*, *GetTag*, and *GetBlockOffset* take the program counter as input, and are used to obtain the index, tag, and the block offset, respectively. Another uninterpreted function *SelectWord* is used to extract the instruction from the data block. The instruction memory is modeled as a lambda expression that takes 2 arguments, an index and a tag, and returns a block of data. This way of modeling the instruction memory allows us to relate the contents of the instruction memory with the instruction cache contents. We require an invariant about the instruction cache that valid entries in the cache are consistent with the instruction memory. We also prove that the instruction cache invariant is inductive, *i.e.*, we prove that if the invariant holds for an arbitrary pipelined machine state, w, then it holds for v, where v is obtained by stepping w once.

The data cache is direct mapped and the way we model it is similar to the way we model the instruction cache. Writes are write-through and update both the data cache and memory. Also, an invariant stating that all the valid entries in the data cache are consistent with the data memory is required.

The write buffer is implemented as a queue and has 4 entries. Each entry has a data part, an address part, and a valid bit. Store instructions do not update the data memory directly, but write to the tail of the write buffer queue. The head of the write buffer queue is read and used to update the data memory. Reads from the data memory have to take into account the valid entries in the write buffer, as the write buffer has the most recent data values. Among the write buffer entries, priority is given to the entries closer to the tail. We require an inductive invariant for the write buffer that states that the combined state of the write buffer and the data memory is consistent with a data memory that is updated directly, without using a write buffer.

4.2 Greatest Fixpoint Invariant

We introduce a new inductive invariant that can be used with commitment-based refinement maps; we call it the Greatest Fixpoint (GFP) invariant. In this section, we define the GFP invariant and compare, based on proof times and the ease of implementation, the commitment approach that uses

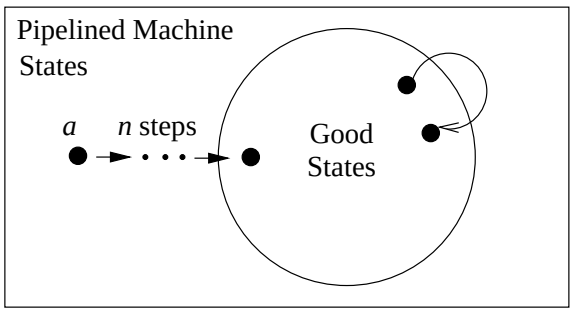


Figure 17: The Greatest Fixpoint (GFP) invariant characterizes the set of states that can be reached in n steps from some pipelined machine state.

the GFP invariant, with the flushing approach and the commitment approach that uses the Least Fixpoint (LFP) invariant.

The definition of the the Greatest Fixpoint (GFP) invariant is straightforward.

Definition 5. gfp.w *iff* $\langle \exists a \in S :: a \dashrightarrow_n w \rangle$

In the above definition, S is the set of all pipelined machine states, $\dashrightarrow$ is the transition relation, and $\dashrightarrow_n$ is the n-fold composition of $\dashrightarrow$ (*i.e.*, it relates u to v if v can be reached in n steps from u). The definition states that a pipelined machine state w is in the invariant if it can be reached from some state in n steps. Reasonable values of n depend on the pipelined machine in question and should be selected to correspond to the minimum number of steps required to replace all the partially executed instructions in the pipeline with instructions fetched from the instruction memory. For the pipelined machine models that we consider, n is the number of steps required to flush the machine.

The reason we call this invariant the greatest fixpoint invariant is that we have the following lemma.

Lemma 1.

$\langle \forall k \in \mathbb{N} :: \langle \exists a \in S :: a \dashrightarrow_{k+1} w \rangle \Rightarrow \langle \exists a \in S :: a \dashrightarrow_k w \rangle$

Therefore, for the sequence of sets $S_0, \ldots, S_n$, where $S_i = \{w \in S :: \langle \exists a \in S :: a \dashrightarrow_i w \rangle\}$, we have $S_0 \supseteq S_1 \supseteq \cdots \supseteq S_n$.

The GFP invariant is depicted in Figure 17. If an arbitrary pipelined machine state is stepped for the number of steps required to flush the pipeline, then all the partially executed instructions in

the pipeline are made to complete and update the programmer visible components, and the pipeline latches are filled with new instructions. For the machines we consider, the flow of an instruction in the pipeline depends only on older instructions in the pipeline. Therefore, all the instructions in pipeline latches of the original arbitrary state are guaranteed to complete and be replaced by new instructions from the instruction memory. The new partially executed instructions in the pipeline latches of the resulting state will be consistent, thereby avoiding the problems inherent in the commitment approach. Note that if the initial arbitrary state is an illegal state and causes the machine to deadlock, a counterexample will be generated as we are checking for liveness. The user would then have to add more invariants to exclude such states. That GFP is an invariant follows from the following lemma.

Lemma 2. $\langle \forall w, v \in S :: (\text{gfp}.w \wedge w \dashrightarrow v) \Rightarrow \text{gfp}.v \rangle$

The lemma is true by definition. Recall that checking the standard invariant used for the commitment approach is where most of the verification time is spent, but by the above lemma, no such check is required for the GFP invariant approach. As we show in the next section, this leads to drastically faster verification times.

The commitment refinement map using the GFP invariant is defined as follows. A pipelined machine state satisfying the GFP invariant is committed by invalidating the partially executed instructions in the pipeline and rolling back the programmer visible components (program counter, instruction and data memory, and register file) so that they correspond with the last committed instruction. The programmer visible components are then projected out, resulting in an ISA state. The rank function is the same as the one used for the LFP commitment approach (*i.e.*, it is the length, in latches, from the end of the pipeline to a valid latch).

4.3 Results and Analysis

We used flushing, the commitment approach with the LFP invariant, and the commitment approach with the GFP invariant to verify the 42 pipelined machine models described in Section 4.1. For all experimental results presented in this chapter, we used the UCLID decision procedure (version 1.0) coupled with the Siege SAT solver [87] (variant 4), using a 3.06 GHz Intel Xeon, with an L2 cache size of 512 KB. The results should be interpreted taking into consideration the following

two factors. First, the Siege SAT solver uses a random number generator and large variations in the running times are possible, *e.g.*, in previous work we noticed that the standard deviation of the Siege running times can be significant [57]. Second, the machines we used for the experiments are part of a public cluster. While we tried to use idle machines, the running times we obtained could have been slightly influenced by other jobs running on the machines.

Table 3 shows the verification times and related statistics for the various pipelined machine models. The names in the "Processor" column start with a number indicating the number of stages followed optionally by the letters "I", "D", "W", "B", and "N" indicating the presence of an instruction cache, data cache, write buffer, branch prediction abstraction scheme 1, and branch prediction abstraction scheme 2, respectively. Branch prediction abstraction schemes 1 and 2 refer to two different ways of abstracting branch predictors. For all the three approaches we report the time taken by both UCLID and Siege to complete the refinement proof. For the commitment approach based on the LFP invariant, we also report the time taken by both UCLID and Siege for the invariant proof and the total time for both the refinement proof and the invariant proof. A "Fail" entry indicates that Siege failed on the problem (by immediately reporting that the problem is too complex and quiting).

Figure 18 shows the running times for the refinement proof and the invariant proof for the LFP approach and the refinement proof for the GFP approach, as the complexity of the pipelined machine models increases. An interesting observation is that more than 98% of the total proof time for the LFP approach is spent in proving the invariant. This motivates the use of the Greatest Fixpoint (GFP) invariant, which is computationally less expensive.

Figures 19, 20, and 21 are scatter plots with log scales for both axes and compare the use of commitment (LFP) vs. flushing, commitment (GFP) vs. flushing, and commitment (GFP) vs. commitment (LFP), respectively. The comparison is based on running times for verifying 42 pipelined machine benchmarks. From Figure 19, it can be seen that flushing and commitment based on LFP have similar performance characteristics on the models that flushing completes. But, flushing fails to produce a result on 9 of the more complex benchmarks. Commitment (LFP) scales better than flushing, but the verification times for the more complex benchmarks reach over 250,000 seconds. Figure 20 shows that the commitment based on GFP does better than flushing on all the 42 benchmarks.

Benchmark	Flushing		Commitment (LFP)				Commitment (GFP)	
	CNF Vars	Ref. Time (s)	Inv. Time (s)	Ref. Time (s)	Total CNF Vars	Total Time (s)	CNF Vars	Ref. Time (s)
6	28,256	20	23	1	12,334	24	9,498	3
6I	48,917	22	172	4	36,498	176	16,955	5
6ID	114,124	202	414	20	75,405	434	29,089	10
6IDW	159,620	458	438	35	80,434	473	34,107	16
7	53,165	168	24	1	13,296	25	17,528	13
7IDW	263,022	1,012	723	95	105,313	818	55,182	40
8	95,092	630	47	1	14,100	48	27,107	43
8IDW	393,719	2,995	1,054	156	131,364	1,210	96,710	147
9	144,045	1,394	24	1	15,214	25	39,346	100
9IDW	526,651	Fail	1,754	98	161,759	1,852	125,585	265
10	198,375	3,841	30	1	17,121	31	55,763	164
10I	293,862	4,752	1,325	8	82,795	1,333	91,416	384
10ID	580,355	Fail	2,685	126	195,562	2,811	159,638	540
10IDW	690,598	Fail	2,885	88	197,258	2,973	174,122	536
6B	37,002	14	89	1	21,850	90	13,495	4
6BI	63,824	23	1,423	11	51,114	1,434	23,891	8
6BID	137,935	283	2,968	167	100,406	3,135	40,371	17
6BIDW	191,101	439	2,851	229	105,639	3,080	45,567	25
7B	70,985	216	232	1	26,058	233	25,676	22
7BIDW	311,425	1,627	5,537	579	144,441	6,116	76,820	70
8B	121,645	733	701	1	31,914	702	40,559	150
8BIDW	424,604	5,376	30,438	1,043	177,741	31,481	122,550	304
9B	183,371	1,737	686	2	36,757	688	59,110	674
9BIDW	628,179	Fail	58,029	876	230,500	58,905	173,479	1,145
10B	256,272	4,563	1,555	2	43,517	1,557	81,569	1,756
10BI	371,249	4,706	73,710	299	126,785	74,009	136,545	1,780
10BID	695,833	Fail	160,523	926	276,289	161,449	221,420	4,431
10BIDW	824,633	Fail	233,928	1,193	278,137	235,121	237,485	6,039
6N	37,452	19	101	1	37,147	102	12,631	4
6NI	63,563	23	878	8	95,821	886	23,229	8
6NID	137,885	282	3,599	51	161,995	3,650	40,132	18
6NIDW	190,399	428	3,472	267	163,763	3,739	45,259	23
7N	70,667	188	240	1	27,500	241	23,936	14
7NIDW	310,434	1,679	11,103	307	162,225	11,410	75,496	73
8N	121,499	499	794	1	53,697	795	39,165	140
8NIDW	424,124	5,968	34,423	433	259,031	34,856	12,1170	270
9N	185,149	2,027	970	2	62,536	972	54,631	447
9NIDW	626,884	Fail	75,453	417	350,587	75,870	170,918	899
10N	255,780	4,910	2,136	2	73,163	2,138	75,676	1,938
10NI	368,888	4,544	51,514	493	224,692	52,007	131,642	2,101
10NID	698,555	Fail	225,636	4,455	414,530	230,091	217,725	4,229
10NIDW	824,210	Fail	286,285	3,479	416,378	289,764	233,852	6,155

Table 3: Verification statistics for the flushing approach, the commitment approach using the Least Fixpoint invariant, and the commitment approach using the Greatest Fixpoint invariant for various pipelined machines.

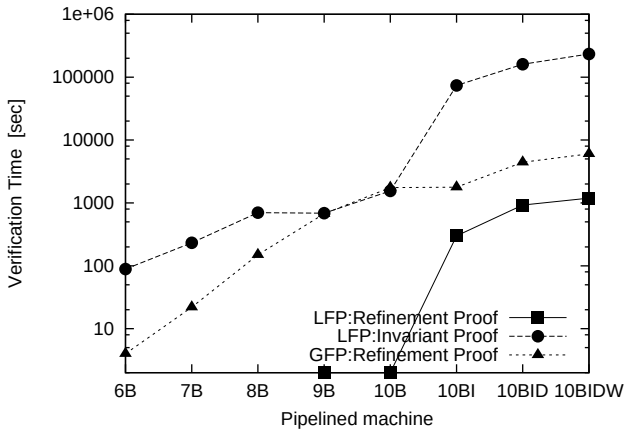


Figure 18: The invariant and refinement proof times for the LFP commitment approach and the refinement proof times for the GFP commitment approach for pipelined machine models with increasing complexity.

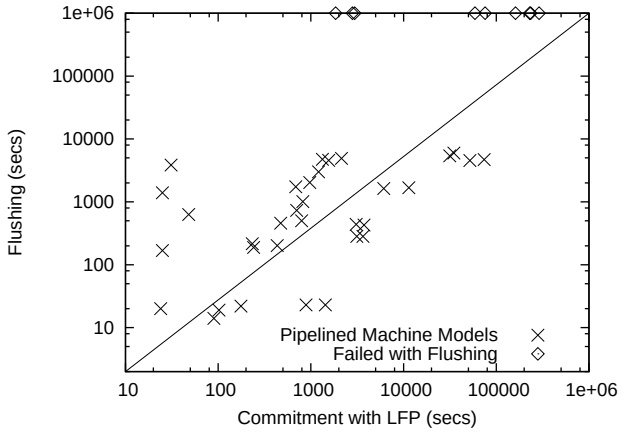


Figure 19: A comparison of the verification times required for our benchmark problems between commitment using the LFP invariant and flushing.

From Figure 21, it can be seen that commitment based on the GFP invariant does better than commitment based on the LFP invariant for most of the benchmarks. We note that the time required for the refinement proofs of the two commitment approaches differs. From Figure 18 and Table 3, we see that the time for the refinement proofs for the GFP approach is much higher. The difference can be explained by noting that once the invariants are proved, the sets of states satisfying the invariants are defined as the set of states reachable from an initial state after some number of steps. The maximum number of such steps required for the two approaches depends on the number of steps required for a newly fetched instruction to reach the end of the pipeline. For the LFP approach, the initial state is a committed state (this is an advantage of using the LFP invariant), meaning that all the pipeline latches are initially invalid. Therefore, the flow of the first newly fetched instruction

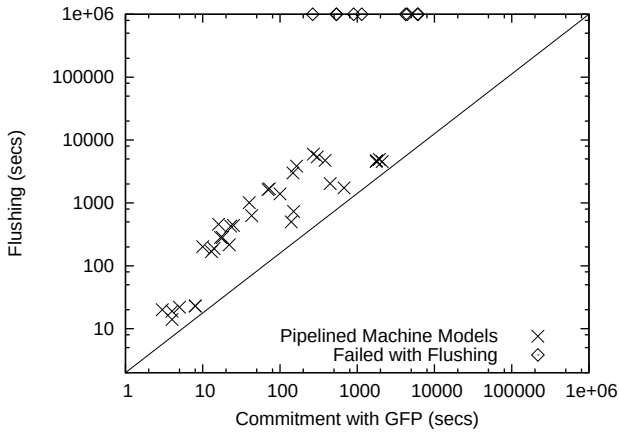


Figure 20: A comparison of verification times required for our benchmark problems between commitment using the GFP invariant and flushing.

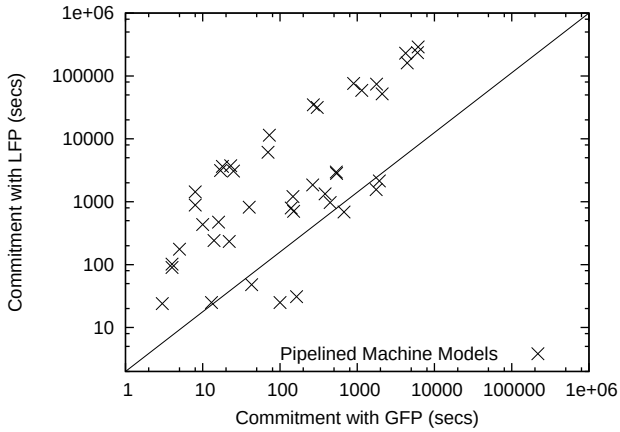


Figure 21: A comparison of verification times required for our benchmark problems between commitment using the GFP invariant and commitment using the LFP invariant.

is uninhibited (*e.g.*, it cannot stall) and depends only on the length of the pipeline. For example, for the 10 stage pipeline models, starting from a committed state, 9 steps of the pipelined machine are required for a newly fetched instruction to reach the end of the pipeline. In contrast, when using the GFP approach, the initial state is an arbitrary state and the flow of the first newly fetched instruction in the pipeline depends on the older instructions in the pipeline. For example, if the first newly fetched instruction is data dependent on older instructions in the pipeline, it will stall. For the 10 stage pipeline, starting from an arbitrary state, a maximum of 14 steps of the pipelined machine is required for a newly fetched instruction to reach the end of the pipeline. As a rule of thumb, verification times increase exponentially with the number of symbolic simulation steps required, therefore, the refinement proof times for the GFP approach is much higher than for the

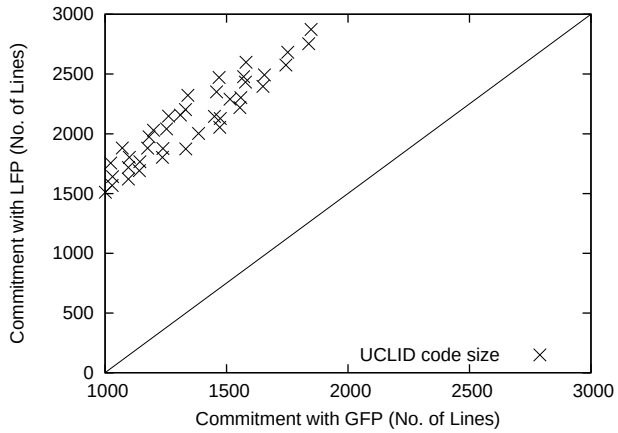


Figure 22: A comparison of the size of the UCLID specifications required for our benchmark problems between commitment using the GFP invariant and commitment using the LFP invariant.

LFP approach. Of course, if we look at the total time required for verification, the GFP approach is the clear winner because it does not require us to prove an invariant.

One further important observation is worth making and that is that the GFP approach is much easier to implement than the LFP approach, because (in contrast to the LFP approach) we do not require an extra invariant proof. Such proofs require that we symbolically simulate a pipelined machine state in two different ways and check that that results are equal. Figure 22 is a scatter plot that compares the size of the UCLID specifications for the two commitment based refinement maps for each of the 42 pipelined machine models. The UCLID specifications consist of the machine model and the refinement map. As can be seen from the figure, the UCLID specifications that use the LFP approach are much larger than those that use the GFP approach. Further, once the machine models are defined, implementing the commitment refinement map based on the GFP approach required about a couple of hours while the implementation of the LFP approach took more than twice as long.

4.4 Related Work

We now describe previous work related to the use of commitment refinement maps for verifying pipelined machines. The use of the commitment approach for relating pipelined machine states to ISA states was introduced by Manolios [51, 52]. In this work, the ACL2 theorem proving system [43, 47] was used to prove the correctness of pipelined machines based on WEB-refinement,

where the commitment approach based on the Least Fixpoint (LFP) invariant was used. Our approach extends these methods to check the correctness of pipelined machines in a more automatic and efficient manner.

Ray and Hunt [82] also use WEB-refinement to verify pipelined machines. Their approach is based on relating pipelined machine states to ISA states by computing witness states. If MA is a pipelined machine state in which $i1$ is poised to complete, then there must have been a state (called the witness state), where $i1$ was poised to enter. The refinement map for a given pipelined machine state MA is computed by flushing its witness state. The resulting refinement map is equivalent to commitment. They use the ACL2 theorem proving system to discharge the refinement proofs. Our commitment-based methods for verifying pipelined machines are more automatic and efficient.

Another approach is based on the concept of synchronization-at-retirement [5]. If instruction $i1$ in a given pipelined machine state MA is poised to complete, it is used to compare single steps of the implementation and the specification. The abstraction function, otherwise known as the refinement map, used to achieve this is equivalent to the commitment refinement map. Their approach uses a variation of the Burch and Dill notion of correctness, which does not account for liveness. We use commitment refinement maps to prove WEB-refinement that accounts for liveness.

4.5 Conclusion

A new method for automatically verifying pipelined machines using commitment-based refinement maps is introduced and is based a greatest fixed-point characterization of the commitment invariant. We defined 210 benchmark verification problems and 42 processor models, which we used to compare our method with previous approaches. Our results clearly show that our new method is easier to define and automate, and gives rise to more than a 35-fold reduction in verification times over the standard approach to verifying commitment-based refinement maps. We noticed a similar improvement over flushing, although the standard flushing approach was not able to complete the verification of 9 of the 42 flushing benchmarks. We also showed that further improvements in verification times are possible by using the recently introduced notion of intermediate refinement maps. The verification engines we used are the UCLID decision procedure and the Siege SAT solver.

CHAPTER V

COLLAPSED FLUSHING

Refinement maps are an important parameter of the refinement framework. As we have shown in Chapter 4, the computational complexity of the refinement map used can have a drastic impact on verification times for checking the correctness of pipelined machines. In this chapter, we introduce another efficient method for automating refinement proofs for term-level pipelined machine models. The method is based on a variation of the flushing refinement map and is called collapsed flushing. Our extensive empirical evaluation shows that collapsed flushing leads to drastically faster verification times than is possible with standard flushing. We also introduce new rank function for refinement proofs based on flushing that is easier to define and allows us to implement collapsed flushing efficiently.

Automation of refinement proofs based on collapsed flushing is achieved by providing a recipe for defining refinement maps and rank functions. It is enough to prove the core theorem described in Section 3.1 to show that a pipelined machine refines its ISA. The core theorem formulated using a refinement map based on collapsed flushing is expressible in the logic of Counter arithmetic with Lambda expressions and Uninterpreted functions (CLU), which is a decidable logic [20]. CLU formulas can be checked using the UCLID decision procedure [49]. Therefore, we use the UCLID decision procedure to automatically check the core theorem for term-level pipelined machines formulated using collapsed flushing.

The rest of the chapter is organized as follows. For a description of the theory of refinement based on WEBs, the reader is referred to Section 2.2 of this dissertation. The standard and collapsed flushing refinement maps are described in Section 5.1. In Section 5.2, empirical evaluations are used to compare collapsed flushing with standard flushing and commitment. Related work is discussed in Section 5.3, and we conclude in Section 5.4.

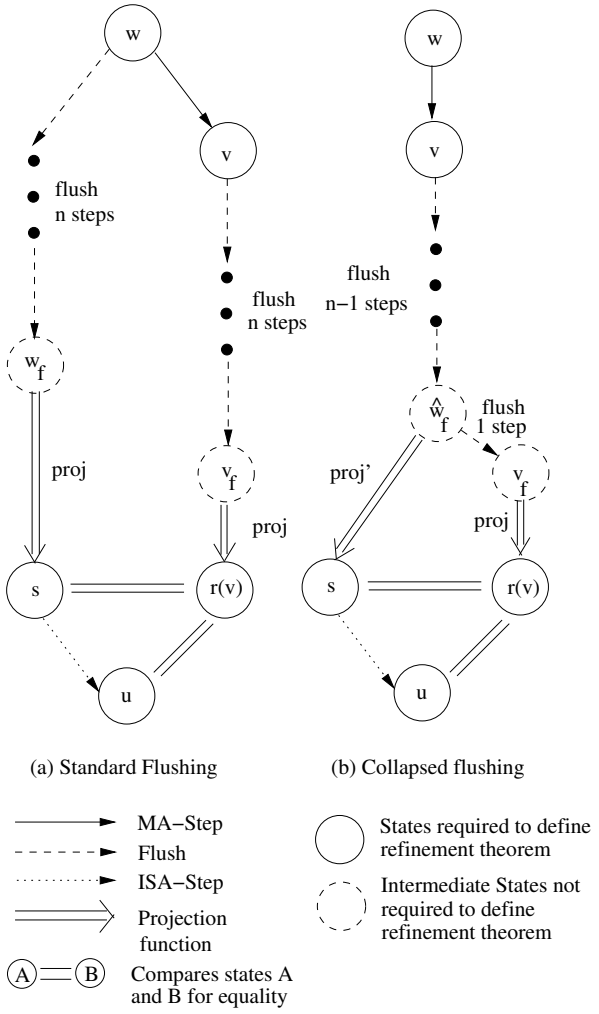


Figure 23: Implementation of standard and collapsed flushing refinement maps.

5.1 Collapsed Flushing

In this section, we describe collapsed flushing, an implementation of the flushing refinement map that leads to faster verification times over previous methods. The use of flushing as a refinement map was proposed by Burch and Dill [22]. As mentioned previously, flushing can be thought of as the dual of commitment, as partially executed instructions in the pipeline are completed (without fetching any new instructions) instead of being invalidated.

In Figure 23(a) we represent the refinement theorem based on standard flushing as a graph we call the *refinement graph*. The nodes of the graph are variables whose names match the ones given in Section 3.1; the edges correspond to symbolic simulation steps, flushing steps, or projections.

Pipelined machine state w is flushed for n steps, resulting in flushed state w_f, where n is the number of steps required to invalidate all of w's pipeline latches. The ISA state returned by the flushing refinement map is s, the state obtained by projecting out the ISA components of w_f. State u is obtained by stepping state s, and state v is obtained by stepping w. Flushing state v gives us state v_f, and projecting out the ISA components gives us state $r(v)$. The safety component of the refinement theorem compares $r(v)$ with s and u. The liveness component depends on the ranks of w and v. Thus, the refinement theorem depends only on states w, v, s, u, and $r(v)$, nodes depicted with a solid circle in Figure 23.

The verification times for the flushing method depend on two factors. The first factor is number of symbolic simulation steps required to reach u from w. We call this factor the *flushing length*. If n is the number of symbolic simulation steps required to flush the pipelined machine, then the flushing length is $n+1$, as can be seen from the refinement graph in Figure 23(a). The number of steps required to flush the pipelined machine depends on the pipelined machine under consideration, and it is an inherent parameter of the flushing refinement map. Therefore, there is no way to reduce the flushing length without abandoning the use of the flushing refinement map.

The second factor is the *state distance*, the number of symbolic simulation steps separating u from $r(v)$. This is approximately the length of the shortest path between u and $r(v)$ in the refinement graph, when viewed as an undirected graph. As can be seen from Figure 23(a), the state distance for standard flushing is $2n+2$. The intuition as to why this metric is related to verification times is that the statements $u = r(v)$ and $s = r(v)$ are quite complex, each requiring about $2n$ symbolic simulation steps to express.

We now describe collapsed flushing, depicted in Figure 23(b). The insight is that we can reduce the state distance from $2n+2$ to 2. Consider the state $\hat{w}_f$, obtained by stepping w once, to obtain v, and then flushing v for $n-1$ steps. If no new instruction is fetched during the initial step (as is the case during a stall or a branch mispredict) then $\hat{w}_f$ is exactly w_f. Otherwise, we can obtain w_f from $\hat{w}_f$ by using history variables to factor out any effect that the instruction fetched from the transition to v has on the programmer visible components. That is, w_f can be obtained by slightly modifying the process of computing v_f. This allows us to collapse the two flushing computations arising in the implementation of the standard flushing refinement map into one, which is why we

name this method collapsed flushing. Figure 23(b) shows that the state distance is 2, improving upon the $2n+2$ value for standard flushing. We validate these intuitions in Section 5.2, where we show empirically that collapsed flushing leads to much faster verification times and scales better than standard flushing.

History variables are used with collapsed flushing as follows. If a new instruction is fetched during the transition from w to v, a tag is attached to it that follows it through the pipeline. In addition, every programmer visible component in the pipelined machine has a history variable associated with it. While non-history variables are updated normally, history variables are updated only by instructions that are not tagged. Thus, the history variables in $\hat{w}_f$ contain the values we would have obtained had the step from w to v been a flush step, which allows us to determine w_f, which in turn is used to obtain state s.

In the standard flushing method, the rank of a pipelined machine state is defined as the number of steps required to fetch an instruction that eventually completes. For the collapsed flushing method, we use an alternate rank function that can be easily implemented and that leads to faster verification times. The new rank function is defined as follows. Let n be the maximum number of steps required to flush any pipelined machine state. The rank of a pipelined machine state is the number of steps required to flush the state if that number is less than or equal to n. Otherwise, the rank is n. To compute this for v, we simply determine how many flushing steps are needed before all pipelined latches are invalid. Since we step w before flushing it (see Figure 23(b)), the rank of w is the number of steps required before all pipelined latches are either invalid or contain a tagged instruction (only the step from w to v can lead to a tagged instruction).

In Figures 24 and 25, we show how the new flushing rank of two states of 3PM (a simple three stage pipelined machine), w and v are computed. A description of 3PM can be found in Section 2.1.2. For the state w, since the instruction in the first pipeline latch $i2$ depends on $i1$, $i2$ is stalled for one cycle before it can make progress. Therefore the rank of state w is three. State v is flushed in two steps and therefore the rank of v is two. Note that v is actually the state obtained by stepping w. When w is stepped, it does not make any forward progress with respect to the ISA, therefore then rank decreases from three to two.

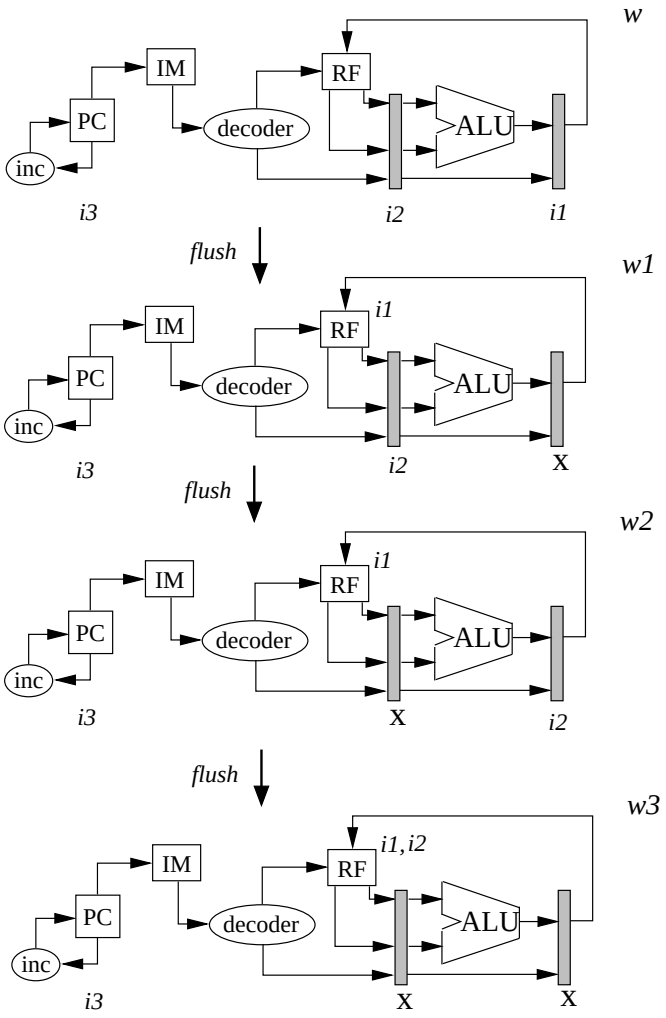
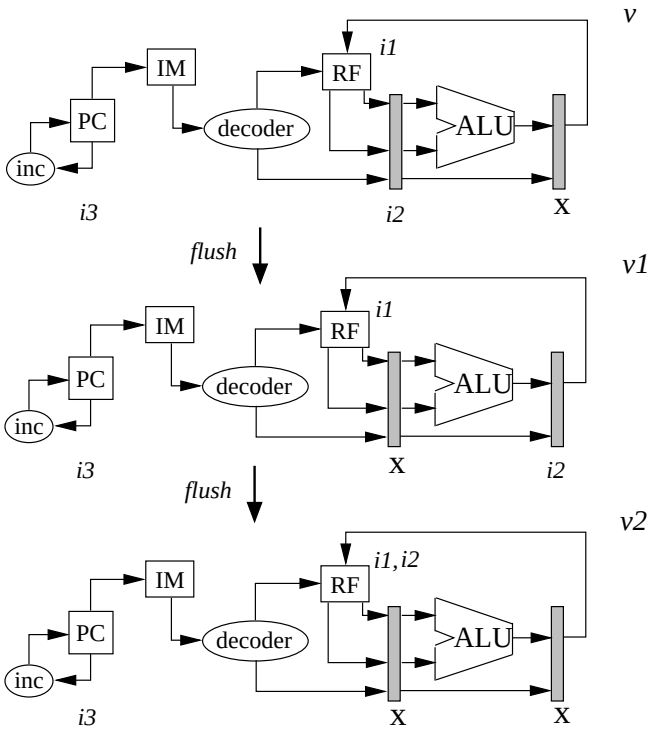


Figure 24: Figure shows the computation of the new flushing rank function for a state of 3PM, w. In this state, the instruction in the first pipelined latch ($i2$) depends on the instruction in the second pipelined latch ($i1$).



rank(v) = no. of steps required to flush v = 2

Figure 25: Figure shows the computation of the new flushing rank function for a state of 3PM, *v*. Note that state *v* is obtained by stepping state *w* shown in Figure 24

5.2 Experimental Results

In this section, we present our empirical evaluation of collapsed flushing, which is based on an extensive set of experiments. To summarize, we found that using collapsed flushing gives an order-of-magnitude improvement in verification time when compared with standard flushing. We also show that the CNF files generated when using collapsed flushing are much smaller than when using standard flushing. Both observations validate our analysis of collapsed flushing in Section 5.1.

For the experiments, we used the pipelined machine models described in Section 4.1. These models contain branch prediction mechanisms, instruction caches, data caches, write buffers, and instruction queues. They were formally verified using the UCLID decision procedure (Version 1.0) along with the Siege SAT Solver [87] (variant 4), using a 3.06 GHz Intel Xeon with an L2 cache size of 512 KB.

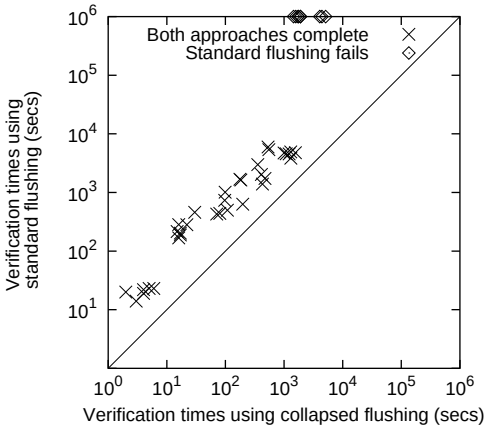


Figure 26: A comparison of standard and collapsed flushing based on verification times.

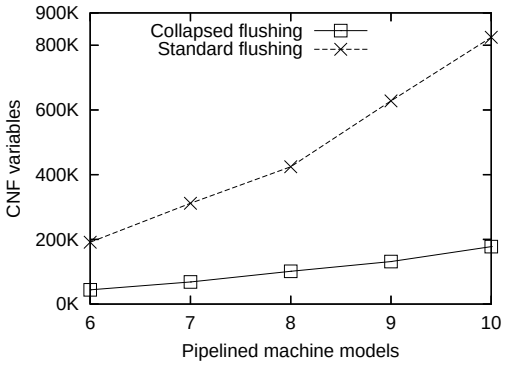


Figure 27: A comparison of standard and collapsed flushing based on the number of CNF variables generated.

Figure 26 compares collapsed flushing with standard flushing using a benchmark suite consisting of 42 pipelined machine models, where the number of stages ranges from 6 to 10. Notice that both the *x* and *y* axes use a logarithmic scale. When using standard flushing, Siege fails on 9 of the benchmarks by reporting that the problem is too complex to handle and immediately quiting; this is denoted in the figure as "Standard flushing fails." However, when collapsed flushing is used, Siege can handle all of the benchmark problems.

Our analysis in Section 5.1 shows that the complexity of pipelined machine verification problems is greatly reduced when standard flushing is replaced by collapsed flushing, because of the differences in state distance. As a metric of the complexity of these problems, we use the number of

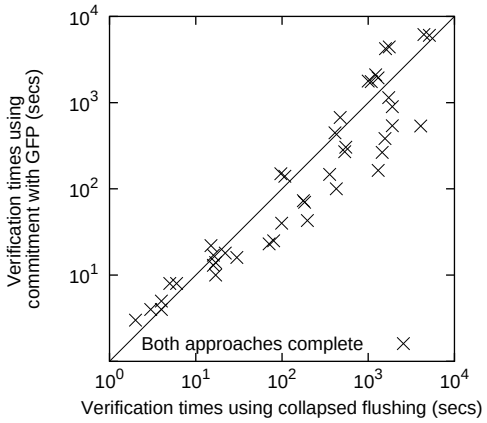


Figure 28: A comparison of verification times for collapsed flushing and GFP-based commitment.

CNF variables generated. In Figure 27, we plot the number of CNF variables generated when verifying pipelined machines of varying length for both standard and collapsed flushing. Recall, that for the machine models we consider, the number of flushing steps required to define either standard or collapsed flushing is the same and is directly proportional to the length of the pipeline. From the figure, it can be seen that as the length of the pipeline increases, the CNF variables generated for standard flushing rapidly increase, whereas the increase for collapsed flushing is more modest. The reason for this, as explained in Section 5.1, is that the state distance for standard flushing depends linearly on the number of steps required to flush the machine, but remains a constant for collapsed flushing. Therefore, collapsed flushing scales much better than standard flushing as the length of the pipeline increases, and it can even handle problems that standard flushing cannot.

In Figure 28, we compare collapsed flushing with Greatest Fixpoint (GFP) invariant based commitment on the same 42 pipelined machine models used in Figure 26. As can be seen from the scatter plot, the two approaches are comparable.

5.3 Related Work

We briefly review previous work on pipelined machine verification that is related to flushing refinement maps. Burch and Dill [22] showed how to compute flushing refinement maps automatically. Several variants of flushing have been previously considered. One example is controlled flushing, an implementation of the flushing refinement map that uses a fixed stalling and flushing pattern, leading to simpler formulas and faster verification times [21]. A second example is incremental flushing,

which uses an inductive argument, making it difficult to apply, *e.g.*, the authors conclude that the effort required to deductively justify the proof decompositions offsets the benefits obtained [40]. Also note that neither of these approaches deals with liveness.

5.4 Conclusion

We have introduced collapsed flushing, a new refinement map based on flushing that results in about an order-of-magnitude improvement in verification times over standard flushing. We also presented a new, simpler, and easier-to-verify rank function, which is used for handling liveness. The utility of collapsed flushing was empirically validated with an extensive set of experiments on a benchmark suite containing a large number of pipelined machines.

CHAPTER VI

INTERMEDIATE REFINEMENT MAPS

In this chapter, we describe a new class of refinement maps that can provide several orders of magnitude improvements in verification times over the standard flushing-based refinement maps. Our refinement maps are based on flushing and commitment, two well-known refinement maps. The idea with flushing is that partially completed instructions are made to complete without fetching any new instructions. The idea with commitment is that partially completed instructions are invalidated and the programmer visible components are rolled back to correspond with the last committed instruction. Our refinement maps use the commitment approach on the latches at the front of the pipeline and the flushing approach on the latches at the end of the pipeline. This essentially decomposes the verification problem into two smaller problems, each half as complex as the original problem. However, since verification times grow exponentially in the size of the problem, this leads to drastic verification time improvements.

We also show how to combine collapsed flushing (an optimization of the flushing refinement map described in Chapter 5) with GFP based commitment (an optimization of the commitment refinement map described in Chapter 4). We show that the resulting refinement map can be used to efficiently reason about complex machine models with deep pipelines. This is an important problem, as recent state-of-the-art microprocessor designs have very deep pipelines, *e.g.*, Intel's hyper-pipelined technology appearing in the Pentium 4 processor has a pipeline with 31 stages [35].

The chapter is organized as follows. A description of the notion of correctness that we use for pipelined machines that is based on Well Founded Equivalence Bisimulation (WEB) refinement can be found in Chapter 2, in Section 2.2. In Section 6.1, we present experimental data measuring verification time and related statistics for the standard refinement maps. In Section 6.2, we propose our new intermediate refinement maps and analyze their performance experimentally. In Section 6.4, we show how to combine collapsed flushing with GFP based commitment using intermediate refinement maps. In Section 6.3, we show how to combine standard flushing with GFP

Processor	CNF Vars	Verification Times (sec)	
		UCLID	Total
C6	12,328	2	36
C6I	31,347	5	61
C6ID	52,077	9	105
C6IDW	75,494	13	164
C7	13,290	2	31
C7IDW	101,065	22	264
C8	14,094	2	32
C8IDW	127,637	24	407
C9	15,208	3	24
C9IDW	159,441	31	582
C10	17,115	3	33
C10I	76,418	21	1,826
C10ID	128,102	26	2,038
C10IDW	195,159	45	2,388
F6	40,083	6	19
F6I	66,843	11	25
F6ID	110,181	20	72
F6IDW	120,343	23	97
F7	53,441	9	137
F7IDW	218,572	79	400
F8	95,456	15	597
F8IDW	316,217	115	1,812
F9	143,954	24	2,163
F9IDW	452,124	181	7,711
F10	198,222	34	5,481
F10I	291,492	58	6,689
F10ID	572,063	151	Fail
F10IDW	605,734	170	Fail

Table 4: Verification statistics for various pipelined machine models.

based commitment using intermediate refinement maps. In Section 6.4, we show how to combine collapsed flushing with GFP based commitment using intermediate refinement maps. Conclusions appear in Section 6.5. Note that each of the sections also provide results and conclusions about the methods described in that section.

6.1 Refinement Maps

Burch and Dill proposed the use of flushing to automatically define the refinement map used to establish the correctness of pipelined machines [22]. The idea with flushing is that partially executed instructions in the pipeline latches are made to complete and update the programmer visible

components, without fetching any new instructions. The programmer visible components for the pipelined machine models we consider include the program counter, the register file and the data memory. Once the pipeline is flushed, all the pipeline latches are invalid and the resulting state is an instruction set architecture (ISA) state. The Burch and Dill approach did not consider liveness, but in our context a rank function is needed and we define it as the number of steps required to fetch and complete a new instruction. Note that the presence of branch prediction makes this a non-trivial function.

The commitment approach can be thought of as the dual of flushing, as partially executed instructions are invalidated instead of being flushed, and the programmer visible components are rolled back to correspond to the last committed instruction. We use history variables [6] to simplify the definition of this refinement map. Also, we need an inductive invariant that we call the Least Fixpoint (LFP) invariant, which states that the contents of the latches have to be consistent with memory. The rank function for the commitment approach is defined as the number of steps required to commit an instruction. This is a trivial function, as it is essentially the number of consecutive invalid latches starting at the end of the pipeline.

For the experiments, we used the pipelined machine models described in Section 4.1. We used both the flushing and the commitment approach to verify these pipelined machines models. For all experimental results presented in this paper, we used the UCLID decision procedure (version 1.0) coupled with the siege SAT solver [87] (variant 4), using a 3.06 GHz Intel Xeon, with an L2 cache size of 512 KB.

Table 4 shows the verification times and related statistics for the various processor models. The names in the "Processor" column start with a "C" or "F", indicating whether commitment or flushing is used. Next, a number indicating the number of stages is given. Finally and optionally, the letters "I", "D", and "W" indicate the presence of an instruction cache, data cache, and write buffer, respectively. The Siege running times can be obtained by subtracting the total time from the UCLID time. A "Fail" entry indicates that Siege failed on the problem (by immediately reporting that the problem is too complex and quiting).

Figure 29 shows how verification times vary as we first increase the length of the pipeline and then add an instruction cache, a data cache, and a write buffer. First, note that there are important

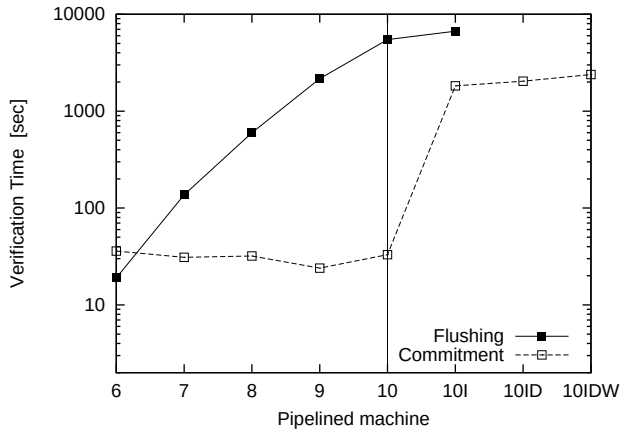


Figure 29: Verification times obtained by first increasing the length of the pipeline and then adding an instruction cache, a data cache, and a write buffer.

differences between the flushing and commitment approaches. As a function of the length of the pipeline, verification times based on flushing grow exponentially, whereas verification times based on commitment are much more stable, *e.g.*, for machine 10, the commitment refinement map leads to verification times that are about 166 times faster than verification times using flushing.

A key observation from the results in Table 4 and Figure 29 is the general rule of thumb that verification times grow exponentially as the number of stages in the pipeline (its length) increases or as the number of state variables per stage increases (the pipeline width), as happens when we add the instruction and data caches and write buffer. These results are not so surprising when we consider that the number of symbolic simulation steps required by the flushing approach depends on the length of the pipeline, and the invariant required for the commitment approach also depends on the complexity (width) of the pipeline. These observations give rise to the idea of an intermediate refinement map that uses flushing to deal with the width and commitment to deal with the depth. We explore this idea in more detail in the next section.

6.2 Intermediate Refinement Maps

In this section, we propose the idea of intermediate refinement maps that partially flush and partially commit. Flushing a stage of the pipeline implies that we also flush later stages; similarly, committing a stage of a pipeline implies that we commit previous stages. Therefore, the intermediate refinement maps are obtained by selecting a stage of the pipeline, we call the reference point, and committing all stages up to and including it and flushing all later stages. Since the intermediate refinement maps

are just based on commitment and flushing, they are quite easy to define.

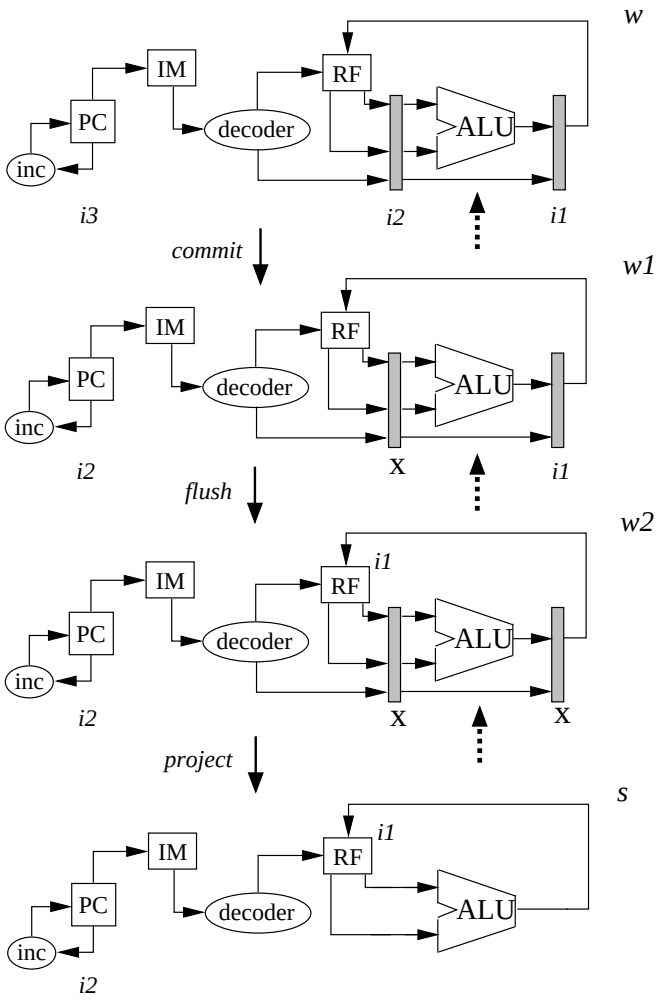


Figure 30: The figure depicts the computation of an IR for state *w* of the three stage pipelined machine, 3PM.

Figure 30 shows the computation of an intermediate refinement map (IR) for a simple three stage pipelined machine (3PM). A detailed description of this machine is provided in Section 2.1.2. The IR for this machine is defined by choosing the reference point between the two pipeline latches. Therefore, the first pipeline latch is committed and the second pipeline latch is flushed. Note that if we choose the reference point to be before the first pipeline latch, then the resulting IR is in fact flushing. Similarly, if we choose the reference point to be after the second pipeline latch, the resulting IR is commitment.

The *commit* operation in Figure 30 refers to committing all the pipeline latches before the reference point, which is the first pipeline latch only for 3PM. The *flush* operation in the figure refers to

Refinement Map	CNF Vars	Verification Times (sec)	
		UCLID	Total
IR0	729,285	153	Fail
IR1	507,790	94	34,913
IR2	382,970	62	11,631
IR3	276,001	44	5,553
IR4	183,236	28	3,626
IR5	100,746	14	7,451
IR9	253,461	34	234,440

Table 5: Verification statistics for a 10-stage pipeline machine with branch prediction, an instruction cache, a data cache, and a write buffer using various refinement maps.

flushing all the pipeline latches after the reference point, which is only the second pipeline latch for 3PM. As can be seen from the figure, instruction $i2$ in the first pipeline latch is pulled back. Then, instruction $i1$ is flushed and the programmer visible components are projected out from the resulting state giving the ISA state s, corresponding to the 3PM state w.

We now define a set of intermediate refinement maps for 10NIDW, a 10-stage machine that has an instruction queue, an instruction cache, a data cache, a write buffer, and a branch predictor that makes arbitrary choices. Note that 10NIDW is more complex than 10IDW (see Table 4), which only has a simple branch predictor that always predicts taken. The refinement maps are IR0, ..., IR9, where IRi commits the first i latches of the pipeline and flushes the remaining latches. For example, IR0, IR5, and IR9 correspond to pure flushing, committing all latches before the decode stage, and pure commitment, respectively.

Since we are proving that the pipelined machines satisfy the same safety and *liveness* properties as their corresponding instruction set architecture models, we also have to define rank functions. For refinement map IRi we define the rank function `rank`i to return a pair of natural numbers: the first is the commitment component, computed by $\texttt{rank}i_{\texttt{c}}$, and the second is the flushing component, computed by $\texttt{rank}i_{\texttt{f}}$. These two functions are essentially the standard rank functions used for flushing and commitment [57] (described in Chapter 3): $\texttt{rank}i_{\texttt{c}}$ returns the number of steps required for a new instruction to reach latch $i+1$, the first flushed latch, while $\texttt{rank}i_{\texttt{f}}$ returns the number of steps required to fetch an instruction that eventually completes for a machine that consists of the latches after latch i, *i.e.*, the flushed latches. The less-than ordering on `rank`i is defined as the lexicographic ordering with priority given to $\texttt{rank}i_{\texttt{f}}$.

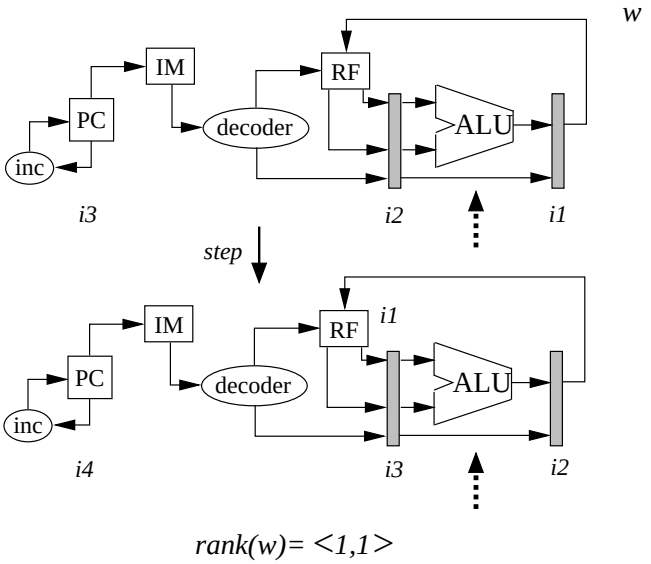


Figure 31: The figure depicts the computation of rank corresponding IR for state w of the three stage pipelined machine, 3PM.

Figure 31 shows the computation of rank corresponding to the intermediate refinement map for a state of the three stage pipelined machine (3PM). We assume that instruction $i2$ does not depend on instruction $i1$, and therefore, stepping the pipelined machine in state w results in instruction $i2$ moving to the flush latch, which is the second pipeline latch in this example. Thus, the value of $\texttt{rank}i_{\texttt{f}}$ is one as it required one step to fetch an instruction that eventually completes. Also, the value of $\texttt{rank}i_{\texttt{c}}$ is one as it required only one step for a new instruction to reach latch $i+1$.

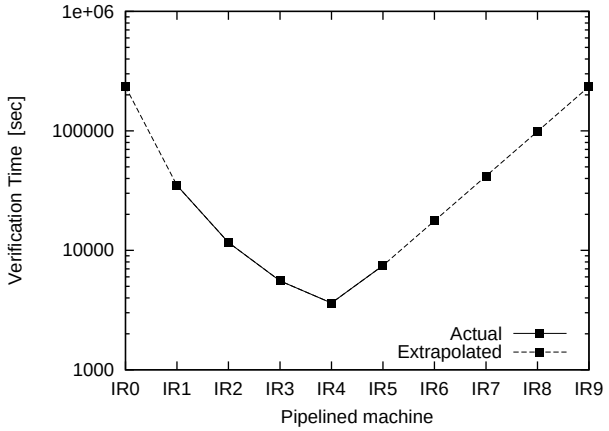


Figure 32: Verification times for a 10-stage processor model with an instruction cache, a data cache, and a write buffer using various refinement maps.

Table 5 and Figure 32 show the verification times we obtain as we apply the various intermediate refinement maps to 10NIDW. Note that the Y-axis in Figure 32 uses a logarithmic scale. The refinement map for IR9 is the standard commitment map and leads to a verification time of 234,440 seconds. The refinement map for IR0 is the standard flushing map and Siege is not able to handle the SAT instance generated by UCLID for IR0. Thus, in Figure 32, the verification time for IR0 is extrapolated, using Table 4 as a guide, and is shown as a dotted line. As i increases from 0 to 4, the verification times for IRi decrease at an exponential rate, with IR4 being about 64 times faster than IR0. After this point, verification times increase, as the time for IR5 shows.

There are several factors that account for the verification time improvements. First, by using the commitment approach for the latches up to the instruction queue, we effectively reduce the depth of the pipeline, thereby avoiding the exponential penalty incurred by flushing on deep pipelines, as witnessed in Figures 29 and 32. In addition, by using flushing for the wide, later part of the pipeline, the LFP invariant required by the commitment approach is greatly simplified in terms of the complexity of the resulting formulas; the conceptual complexity is about the same. Since about 90% of the verification effort required by the commitment approach is for proving the LFP invariant, the savings are substantial. Second, the use of intermediate refinement maps essentially gives rise to two verification problems: one for the part of the machine up to the selected stage and the other for the rest of the machine. By selecting a stage in the middle, the machines are about half as complex as the initial pipelined machine, but since verification times are exponential in the size of the machines, this leads to exponential improvements in verification times. This explains the U-shaped graph in Figure 32, which takes its minimal value at IR4 and then increases at IR5.

From the results we obtained in this and the previous section, we now give some simple guidelines for choosing refinement maps optimized for reduced verification times. From the above considerations, this amounts to deciding up to which stage to use commitment. We suggest choosing the stage closest to the middle of the pipeline for which the complexity of the formula corresponding to the LFP invariant is simple. The reason we suggest a latch somewhere in the middle is that this effectively leads to two verification problems, one of which is based on flushing and one on commitment, but as Table 4 and Figure 29 show, verification times are exponential in the length of the pipeline; thus, such a decomposition leads to drastic improvements in verification times. The

reason why we suggest that thought be given to the complexity of the formula for the LFP invariant is that the verification time for the commitment approach is dominated by cost of establishing this invariant.

Finally, we make two further observations. First, most pipelines have a structure similar to the ones we use in this paper; thus, we expect our techniques to be widely applicable. Second, we have found that our approach simplifies the verification effort because if the resulting SAT instance is satisfiable, it is easy to determine if the problem lies with the commitment part of the proof or with the flushing part of the proof. This allows one to more readily identify errors than if a pure flushing or pure commitment approach is used, as the counterexamples will involve the whole pipeline, and will therefore contain many irrelevant details.

6.3 Using GFP with Intermediate Refinement Maps

The verification approach based on the use of commitment refinement maps requires the use of an invariant that characterizes the set of reachable states. We developed an alternate invariant, called the Greatest Fixpoint (GFP) invariant, the use of which leads to drastic improvements in verification times [59] (see Chapter 4 for details). In this section, we describe the combination of the GFP based commitment approach with standard flushing using intermediate refinement maps (IRs).

The approach is very similar to the previous approach for defining intermediate refinement maps, the main difference being that the GFP invariant is used to characterize the set of reachable states. This is achieved by stepping the pipelined machine for i steps, where i is the number of steps required to replace the instructions in the pipeline latches being committed with new instructions from memory. The rank function for the intermediate refinement map defined using the GFP invariant is the same as the rank function for the intermediate refinement map defined using the LFP invariant.

For any given pipelined machine, many intermediate refinement maps can be defined by selecting different stages in the pipeline as the reference point. The fastest verification times are obtained when selecting a reference point that is close to the middle of the pipeline. We implemented the intermediate refinement map IR4 that commits the first 4 pipeline latches and flushes all other latches, for the most complex processor model with branch prediction scheme 1 (10BIDW) using the GFP

based commitment approach.

The experiments were conducted using the same experimental set up (tools and machines) described in Section 6.2. We found that the verification time for 10BIDW using IR4 defined with the commitment approach based on LFP invariant and the GFP invariant to be 3,500 seconds and 550 seconds, respectively. Note that with using only the commitment approach (LFP), the verification time for 10BIDW is 235,121 seconds. Thus, the GFP invariant approach can be fruitfully combined with intermediate refinement maps to get verification times that are about 6 times faster than the previous approach, for the most complex processor model (10BIDW).

6.4 Using Collapsed Flushing and GFP with Intermediate Refinement Maps

We describe how to define the intermediate refinement map (IR) obtained by combining GFP-based commitment with collapsed flushing [41], a variant of the flushing refinement map that leads to drastic improvements in verification times over standard flushing. See Chapter 5 for a detailed description of collapsed flushing. In the following discussion, we refer to the pipeline latches that are committed and flushed as the commit latches and the flush latches, respectively. We require an invariant for the commit latches and it is based on the GFP invariant: starting from an arbitrary state, we step the machine for the number of steps required to flush the commit latches. The flush latches are also stepped, as the commit latches depend on the flush latches, but once this process is finished, we assign arbitrary values to the flush latches. This defines the IR invariant.

Now, let w be an arbitrary state satisfying the IR invariant. We proceed by essentially applying the collapsed flushing refinement map. The states $\hat{w_f}$ and v_f are computed by applying $n-1$ and n flushing steps to the flush latches, where, n is the number of steps required to flush the flush latches. During the flushing sequence, the commit latches are modified only when there is a branch mispredict. Committing the commit latches and applying the corresponding projection functions to $\hat{w_f}$ and v_f results in ISA states s and $r(v)$, respectively. Just as before, u is obtained by stepping the ISA machine from state s.

A major benefit of collapsed flushing can be seen when it is combined with commitment (GFP) to define intermediate refinement maps (IRs) as shown in Figure 33, where we compare IRs defined using commitment (GFP) and collapsed flushing (CIRs) with IRs defined using commitment (GFP)

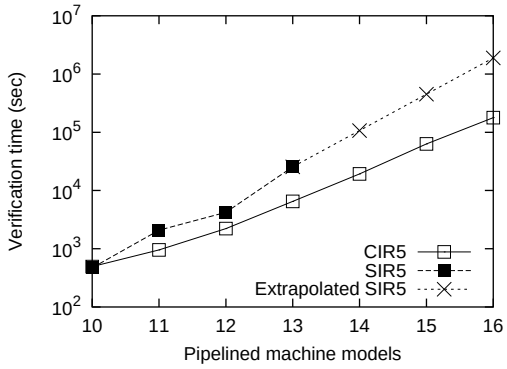


Figure 33: A comparison of verification times for CIR5 and SIR5, defined using collapsed and standard flushing, respectively.

and standard flushing (SIRs). IRs are effective for handling problems that are beyond the scope of pure flushing or commitment refinement maps, such as deep pipelines. For the experiments, we use IR5, which is the IR obtained by committing the first 5 pipeline latches and flushing all other pipeline latches. The *x*-axis shows pipelined machine models obtained by increasing the number of stages from 10 to 16 for a machine containing features such as a branch prediction mechanism, instruction and data caches, and write buffers. Notice that the *y*-axis is a logarithmic scale. From the figure, it can be seen that SIR5 is not able to handle machine models with pipelines that have more than 13 stages. For these models, we have extrapolated the verification times using the average slope of the SIR5 models that could be verified. CIR5 scales better as we increase the number of pipeline stages and is able to handle pipelines with 16 stages (and beyond). We note that some modern microprocessors have very deep pipelines, *e.g.*, Intel's Pentium 4 processor, with hyper-pipelined technology, has 31 stages [35].

6.5 Conclusion

We have introduced a new class of refinement maps for pipelined machine verification, and using verification tools UCLID and Siege have shown that one can attain several orders of magnitude improvements in verification times over the standard flushing-based refinement maps, even enabling the verification of machines that are too complex to otherwise automatically verify. The refinement maps allow us to use the commitment approach on the latches at the front of the pipeline and the

flushing approach on the latches at the end of the pipeline. The result is that we are left with two verification problems, but on machines that are half as complex as the initial pipelined machine; since verification times are exponential in the size of the machines, this leads to drastic improvements in verification times. We give a simple recipe for defining such refinement maps and for defining the rank functions needed to establish liveness.

CHAPTER VII

COMPOSITIONAL REASONING

We present a complete compositional framework [58] based on Well-founded Equivalence Bisimulation (WEB) refinement that can be used to reason about complex pipelined machines that are not easily handled using efficient refinement maps. Optimized and intermediate refinement maps provide drastic improvements in verification times over previous approaches, but do not scale well with increase in the complexity of the designs, and therefore, cannot be used to directly verify industrial designs. The compositional framework we introduce takes us a step closer, as it allows us to substantially extend the complexity of the pipelined machine designs that can be verified.

Using our framework we can verify in under 20 seconds a complex pipelined machine defined at the term-level that UCLID cannot directly handle with the flushing refinement map. This machine is quite complicated and to make its definition a manageable process, we defined a series of machines starting with the base processor model `M6`, a 6 stage pipelined machine, which we extended first with a pipelined fetch stage, then with an instruction queue holding up to 3 instructions, then with a direct mapped instruction cache, then a direct mapped data cache, and, finally, a write buffer, to obtain `M10IDW`. Unfortunately, proving that `M10IDW` refines `ISA`, the instruction set architecture, is beyond the capabilities of UCLID. Our compositional framework allows us to verify the machine the same way we defined it, one feature at a time, which leads to a manageable process. Each stage of the proof essentially entails establishing a WEB-refinement proof, which means that, relative to a refinement map and up to stuttering, the two machines have exactly the same infinite behaviors.

We introduce compositional proof rules that guarantee that this sequence of refinement proofs implies that the final pipelined machine has the same behaviors as the instruction set architecture. In terms of temporal logic, we have that the machines satisfy exactly the same $\mathrm{CTL}^* \setminus \mathsf{X}$ properties expressible at the instruction set architecture level. Our overall proof strategy is highly-automated as the proof obligations required by our compositional framework can be automatically handled using SAT-based decision procedures. For the term-level verification, we use the UCLID decision

procedure [20, 49], and the Siege [87] SAT solver to check the CNF problems generated by UCLID.

A major advantage, perhaps even more important than the increased performance, of our compositional framework over monolithic approaches is that counterexamples are shorter and clearer, which greatly simplifies debugging. Suppose that modifications are made to the design and in the process a bug is introduced. Compositional verification allows us to focus in on where the bug first appears and the counterexample generated is with respect to a specific refinement stage, *i.e.*, the counterexample is at exactly the right level of abstraction required to easily understand and correct the problem. For example, if the bug does not involve the cache, then neither does the counterexample, whereas in a monolithic approach, there is no way to know if the cache was involved; thus, as the verification engineer is trying to understand the counterexample, she is forced to manually rule out the possibility that the cache contributed to the error. By using our compositional approach, the engineer can bridge the abstraction gap on her own terms and at a rate that makes sense given available tools and the development process.

Why hasn't something like this been done before? Well, consider carrying out this proof using the standard Burch and Dill notion of correctness. The problem is that, while it is clear how to prove that a pipelined machine refines an instruction set architecture, how does one prove that one pipelined machine refines another? If we use flushing, we have to flush both machines, but then it would be easier to just verify against the instruction set architecture directly. Our main contribution is to show how to do this using state-of-the-art tools for both safety and liveness (the Burch and Dill approach only provides safety [51]), and with the use of any refinement map, not just flushing.

Can we really obtain the benefits of composition without paying a price? Actually, we often have to provide invariants. But, invariants are needed to verify complex designs anyway. For example, to verify a write-through cache, we need the invariant that the valid cache entries are consistent with memory. The invariants we used were straight-forward, requiring a few hours of thought; in contrast, defining the refinement maps can easily take days. If one uses a hierarchical, refinement-based approach to design, then the invariants should be known, as they allow for the separation of concerns that enables different engineers to implement different parts of the system independently. Therefore, composition can fit nicely into the design cycle, which is also compositional. Finally, there seems to be industrial interest in taking a fresh look at refinement-based methodologies.

The rest of this chapter is organized as follows. In Section 7.1, we give an overview of the compositional theory of refinement. Section 7.2, we describe the modeling and monolithic verification of the processor models. In Section 7.3, we describe the compositional techniques we developed for pipelined machine verification. Related work and Conclusions appear in Sections 7.4 and 7.5, respectively

7.1 Refinement

An overview of the theory of refinement used to show that pipelined machines behave like their instruction set architecture is given in Section 2.2. The emphasis here is on exploiting the compositionality of Well-Founded Equivalence Bisimulation (WEB) refinement. Refinement is a compositional notion as shown in the theorem below.

Theorem 3. *(Composition) If* $\mathcal{M} \approx_r \mathcal{M}'$ *and* $\mathcal{M}' \approx_q \mathcal{M}''$ *then* $\mathcal{M} \approx_{r;q} \mathcal{M}''$.

Above, $\mathcal{M} \approx_r \mathcal{M}'$ denotes that $\mathcal{M}$ is a WEB refinement of $\mathcal{M}'$; and $r;q$ denotes composition, *i.e.*, $(r;q)(s) = q(r.s)$.

From the above theorem we can derive several other composition results; for example:

Theorem 4. *(Composition)*

$$\frac{\texttt{MA} \approx_r \cdots \approx_q \texttt{ISA} \qquad \texttt{ISA} \parallel P \vdash \varphi}{\texttt{MA} \parallel P \vdash \varphi}$$

The above theorem states that to prove $\texttt{MA} \parallel P \vdash \varphi$ (that MA, the pipelined machine, executing program P satisfies property φ, a property over the ISA visible state), it suffices to prove $\texttt{MA} \approx \texttt{ISA}$ and $\texttt{ISA} \parallel P \vdash \varphi$: that MA refines ISA (which can be done using a sequence of refinement proofs) and that ISA, executing P, satisfies φ. In this form, the above rule exactly matches the compositional proof rules in [24]. What makes such a rule useful is that it can lead to drastically faster verification times, as we show in this chapter. It will turn out that the verification times depend much more on the semantic difference between models than on their complexity, *e.g.*, verifying that a complex pipelined machine, MA, refines a similar complex pipelined machine can take a fraction of a second,

even though current tools may not be able to verify that MA refines (the much simpler) instruction set architecture.

7.2 Processor Modeling and Monolithic Verification

In this section, we define a complex pipelined machine and describe how to model and verify it using UCLID. The machine is quite complicated and to make its definition a manageable process, we defined a series of machines starting with the base processor model M6, a 6 stage pipelined machine with the following stages: instruction fetch (IF), instruction decode (ID), execute (EX), data memory access (M1 and M2), and write back (WB). M6 has the following instruction types: branches, loads, stores, and ALU instructions. The addressing modes include register-register and register-immediate. M6 also has a simple branch prediction scheme that always predicts that the branch is taken. Once M6 was designed and verified, we extended it with a pipelined fetch stage to obtain M7; then we added an instruction queue holding up to 3 instructions, giving rise to machines M8, M9, and M10. Finally we added a direct mapped instruction cache, a direct mapped data cache, and a write buffer, giving rise to machines M10I, M10ID, and M10IDW.

Unfortunately, as we have seen before, M10IDW is too complex to directly verify with UCLID using the flushing refinement map. Use of more optimized refinement maps can lead to faster verification times, but with more complexity, the verification times increase exponentially.

Wouldn't it be great if we could use the same approach to verifying M10IDW that we used to design it? Recall that since M10IDW was too complicated to design directly we defined a sequence of intermediate machines instead. This allowed us to add features one at a time, making the design a manageable process. Why not verify M10IDW in the same way? For example, when proving M7 refines ISA, why can't we use the already established result that M6 refines ISA to simplify the proof? In the next section, we show how to do this.

7.3 Compositional Verification

In this section we develop techniques that allow us to prove that M10IDW refines ISA in a compositional manner, by proving that M10IDW refines M10ID, which refines M10I, ..., which refines M6, which refines ISA. We present a sound and complete method for proving such theorems, where most of the reasoning is local, *i.e.*, restricted to pairs of machines. By applying our techniques, we

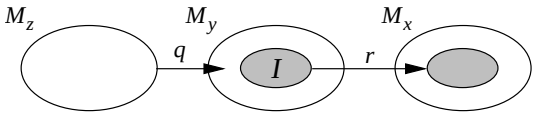


Figure 34: Invariant mismatch.

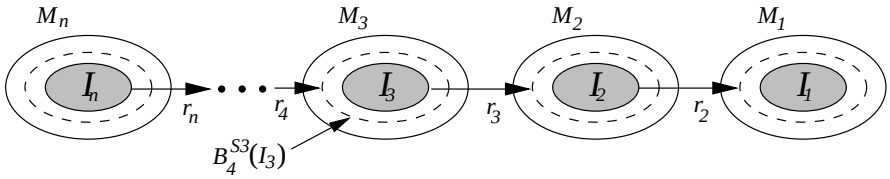


Figure 35: Local composition rule.

transform the problem of verifying that `M10IDW` refines `ISA` from one that UCLID cannot handle to one that takes less than 20 seconds.

A UCLID specification gives rise to a transition system $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$, where s is a state ($s \in S$) iff it maps the variables appearing in the UCLID specification to values of the right type. The transition relation $\dashrightarrow$ is similarly defined over S. An *inductive invariant*, I, is a subset of S that is closed under the transition relation ($s \in I$ implies $\dashrightarrow(\{s\}) \subseteq I$). Put another way, I is an inductive invariant if $\mathcal{M}' = \langle I, \dashrightarrow|_I, L|_I \rangle$, which we sometimes denote $\mathcal{M}|_I$, is a transition system (*i.e.*, the restriction of $\dashrightarrow$ to I is a subset of I^2). It is sometimes useful to identify a subset of S, $Init(\mathcal{M})$, as "initial." If B is a relation, we define $B^X(Y)$ to be $B(Y) \cap X$. We start with two basic observations.

Lemma 5. *If $\mathcal{M} = \langle S, \dashrightarrow, L \rangle, \mathcal{M}' = \langle S', \dashrightarrow', L' \rangle$ are TS's, and $\mathcal{M} \approx_r \mathcal{M}'$ with witness B, then: (a) if I is an inductive invariant of $\mathcal{M}$, then $I' = B^{S'}(I)$ is an inductive invariant of $\mathcal{M}'$ and $\mathcal{M}|_I \approx_{r|_I} \mathcal{M}'|_{I'}$, and (b) if I' is an inductive invariant of $\mathcal{M}'$, then $I = B^S(I')$ is an inductive invariant of $\mathcal{M}$ and $\mathcal{M}|_I \approx_{r|_I} \mathcal{M}'|_{I'}$.*

Proof: For the proof of (a), let $s \in I'$ and let $s \dashrightarrow' w$; we show that $w \in I'$. By the definition

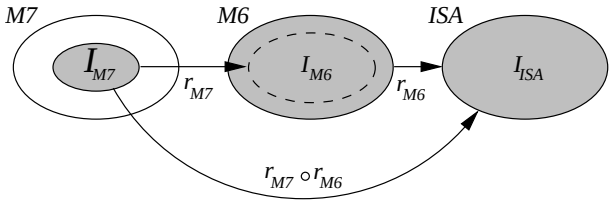


Figure 36: Incompleteness of local composition rule.

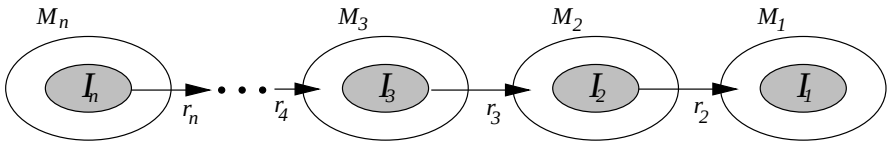


Figure 37: Global composition rule.

Refinement Proof	CNF		Verification Times (sec)		
	Vars	Clauses	UCLID	Siege	Total
M6	28,256	83,725	8	10	18
M7	53,165	158,182	15	150	165
M8	95,092	283,465	25	766	791
M9	144,045	429,973	41	2,436	2,477
M10	198,375	592,660	55	6,762	6,817
M10I	293,862	876,820	92	8,641	8,733
M10ID	580,355	1,730,704	244	FAILED	NA
M10IDW	690,598	2,060,557	297	FAILED	NA

Table 6: Verification times and CNF statistics for the various pipeline machine models.

of I', there is some $u \in I$ such that uBs. Since s and u are stuttering bisimilar, w can be matched by a state reachable from u, say by v, but since I is an invariant, $v \in I$, therefore I' is an inductive invariant of $\mathcal{M}'$, and $\mathcal{M}|_I \approx_{r_I} \mathcal{M}'|_{I'}$ with witness $B \cap (I \cup I')^2$. The proof of (b) is similar. ■

We will make use of the following corollary of Lemma 5, since it applies to all of our examples in this chapter.

Corollary 6. *If in Lemma 5 the equivalence class, under B, of every $s \in I$ has exactly one element from S', we can replace $B^{S'}(I)$ by $r(I)$ and $B^S(I')$ by $r^{-1}(I')$.*

Let's consider applying what we have so far to show that M10IDW refines ISA. Since we consider all states in ISA to be initial, this means that our refinement map has to be surjective. Recall that we are after a compositional proof, so we will prove a sequence of theorems. Let us say that one of these theorems shows that My refines Mx, which implies that: (a) we have an inductive invariant, I, of My, giving rise to $\texttt{My}|_I$, and (b) $\texttt{My}|_I$ refines Mx, say with refinement map r.

To prove that Mz refines Mx, we only need to prove that Mz refines My, say with refinement map q, as we can then appeal to the composition theorem and the theorem that My refines Mx. When one tries to do this in practice, the following problem arises: we need an invariant on Mz whose image under q is I, but defining such an invariant can be quite difficult, requiring much trial and error. (For example, this arises when proving that M9 refines M7, as we will shortly see.) As we show with the

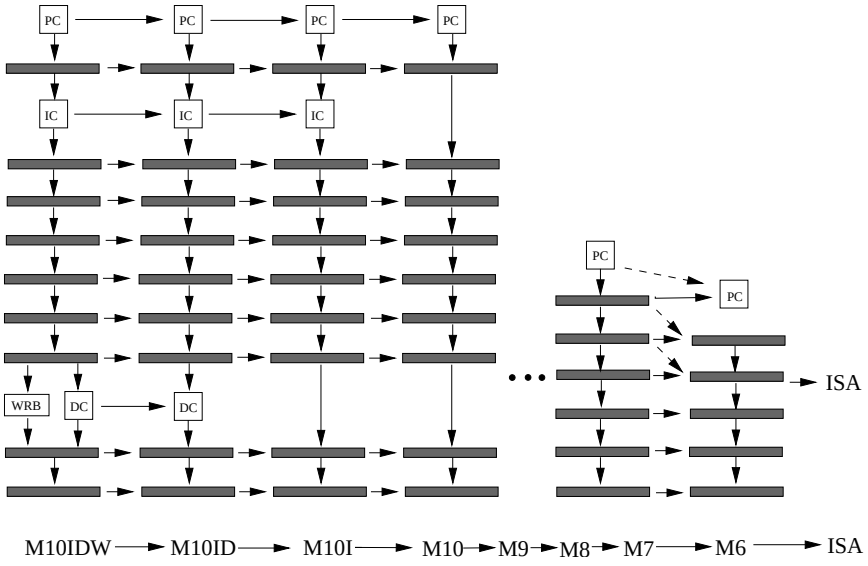


Figure 38: Refinement maps for the compositional verification of `M10IDW`.

following proof rule, it is in fact enough if the image of the invariant under q is a superset of I.

In the sequel, if Z is a set, then $Z^{=}$ and $Z^{\equiv}$ denote the identity relation on Z and the reflexive, symmetric, transitive closure on Z, respectively.

Theorem 7. *Suppose that for all $k \in [1..n]$, I_k is an inductive invariant of TS $\mathcal{M}_k = \langle S_k, \dashrightarrow_k, L_k \rangle$. Suppose also that for all $k \in [2..n]$, $(\mathcal{M}_k)|_{I_k} \approx_{r_k} \mathcal{M}_{k-1}$ with witness B_k and $I_{k-1} \subseteq I'_{k-1}$, where $I'_{k-1} = B_k^{S_{k-1}}(I_k)$. Then, there exists an inductive invariant $I \subseteq I_n$ such that $(\mathcal{M}_n)|_I \approx_R (\mathcal{M}_1)|_{I_1}$ with witness B and $\langle \forall s \in I_1 :: \langle \exists u \in I :: sBu \rangle \rangle$, where $R = (r_n; r_{n-1}; \cdots; r_2)|_I$ and $B = (I^{=}; B_n; B_{n-1}; \cdots; B_2; I_1^{=})^{\equiv}$.*

Proof: The proof is by induction on n, where the base case ($n = 2$) follows from Lemma 5. For the induction step, we have by the induction hypothesis, I', an inductive invariant of $\mathcal{M}_{n-1}$, such that $I' \subseteq I_{n-1}$, $(\mathcal{M}_{n-1})|_{I'} \approx_{R'} (\mathcal{M}_1)|_{I_1}$, and $\langle \forall s \in I_1 :: \langle \exists u \in I' :: sB'u \rangle \rangle$, where $R' = (r_{n-1}; r_{n-2}; \cdots; r_2)|_{I_{n-1}}$ and $B' = (I^{=}; B_{n-1}; B_{n-2}; \cdots; B_2; I_1^{=})^{\equiv}$. Now, letting $I = B_n^{S_n}(I')$, we see that $I \subseteq I_n$ and I is an inductive invariant, such that $(\mathcal{M}_n)|_I \approx_{r_n|_I} (\mathcal{M}_{n-1})|_{I'}$ (by Lemma 5). By Theorem 3, $(\mathcal{M}_n)|_I \approx_R (\mathcal{M}_1)|_{I_1}$. Finally, let $s \in I_1$; by the induction hypothesis, there is a $w \in I'$ such that $sB'w$. Now, let $u \in B_n^{I_n}(\{w\})$, which is non-empty, but since $w \in I'$ and I is an inductive invariant, $u \in I$. ■

The proof rule embodied in Theorem 7 is completely local —every proof obligation involves at most two TS's— and should be used where applicable. Unfortunately, it is *incomplete*: it is possible that there is an inductive invariant $I \subseteq I_n$ such that $(\mathcal{M}_n)|_I \approx_R (\mathcal{M}_1)|_{I_1}$, but we cannot prove it with

the above proof rule. (This situation arises when proving that M7 refines ISA, as explained later.) In such cases, the following complete proof rule should be used.

Theorem 8. *Suppose that for all $k \in [1..n]$, I_k is an inductive invariant of TS $\mathcal{M}_k = \langle S_k, \dashrightarrow_k, L_k \rangle$. Suppose also that for all $k \in [2..n]$, $(\mathcal{M}_k)|_{I_k} \approx_{r_k} \mathcal{M}_{k-1}$ with witness B_k. Then, there exists an inductive invariant $I \subseteq I_n$ such that $(\mathcal{M}_n)|_I \approx_R (\mathcal{M}_1)|_{I_1}$ with witness B and $\langle \forall s \in I_1 :: \langle \exists u \in I :: sBu \rangle \rangle$ iff $B_2^{S_1}(\cdots B_{n-1}^{S_{n-2}}(B_n^{S_{n-1}}(I_n))\cdots) \subseteq I_1$, where $R = (r_n; r_{n-1}; \cdots; r_2)|_I$ and $B = (I^{=}; B_n; B_{n-1}; \cdots; B_2; I_1^{=})^{\equiv}$.*

Proof: Let $I = B_2^{S_1}(\cdots B_{n-1}^{I_{n-2}}(B_n^{I_{n-1}}(I_n))\cdots), I' = B_n^{I_n}(\cdots B_3^{I_3}(B_2^{I_2}(I))\cdots)$. For the proof from left to right, we show that I, I' are inductive invariants, that $I \supseteq I_1, I' \subseteq I_n$, and $(\mathcal{M}_n)|_{I'} \approx_R \mathcal{M}_1|_I$. For the induction step, we can use the conclusion of the induction hypothesis because $B_2^{S_1}(\cdots B_{n-1}^{S_{n-2}}(I_{n-1})\cdots) \supseteq I_1$ (since $B_{n-1}^{S_{n-2}}(B_n^{S_{n-1}}(I_n)) \subseteq B_{n-1}^{S_{n-2}}(I_{n-1})$). We now have that $B_n^{S_{n-1}}(I_n)$ and I_{n-1} are inductive invariants, thus so is $B_n^{I_{n-1}}(I_n)$, which is not empty as $B_{n-1}^{S_{n-2}}(B_n^{I_{n-1}}(I_n)) = B_{n-1}^{S_{n-2}}(B_n^{S_{n-1}}(I_n))$. Using the induction hypothesis, we get that I, I' are inductive invariants, that $I \supseteq I_1, I' \subseteq I_n$, and $(\mathcal{M}_n)|_{I'} \approx_R \mathcal{M}_1|_I$. The rest of the proof is similar to the proof of Theorem 7. ■

Notice that Theorem 8 gives us much more flexibility than Theorem 7, because the relationship between I'_k and I_k can be arbitrary. Also, if (as is the case in our applications) the equivalence class of every $s \in I_i$, under B_i, has exactly one element from S_{i-1}, then the global condition amounts to showing that any state $s \in I_1$ can be reached by starting in some state in I_n and applying the following sequence of refinement maps: $r_n, r_{n-1}, \ldots, r_2$. For pipelined machines, this turns out to be easy to show because applying this sequence of refinement maps to pipelined machines whose non-ISA components are invalid amounts to projecting out the ISA-visible components; thus, every state in ISA is reachable.

We have now developed all of the theory required to verify M10IDW. An overview of the process is shown in Figure 38. Our proof scripts are available upon request and the few invariants required took us less than a day to define. In addition, the rank functions required are much easier to define than in the monolithic case, and there is a simple recipe for doing this described elsewhere [57]. The verification times and related statistics are given in Table 7. The names in the "Refinement Proof" column indicate which refinement proof the row corresponds to. The models are expressed in the UCLID language, and are translated to CNF formulas using the UCLID tool.

Figure 39 depicts the verification times required for both the direct and the composition methods for each of the processor models. As can be seen from Figure 39, if we compare the verification times required by the direct method versus our compositional method, then we see that the verification cost increases exponentially (the *y*-axis uses a logarithmic scale) for the direct approach for each new feature/pipeline stage, whereas, for the compositional approach, the verification cost is almost a constant. The data reported for the compositional proofs includes the total time required, including the time required for the proof of invariants, and everything else required by our proof rule. Notice that the SAT solver Siege failed to produce a result when applying the direct approach to M10ID, whereas with the compositional approach, the proof of M10IDW required less than 20 seconds.

We now explain the refinement proofs shown in Figure 38 in more detail. First, we discuss how to deal with deep pipelines. Second, we show how to handle caches and write buffers. Finally, we discuss counterexamples.

7.3.1 Deep Pipelines

The first five refinement proofs in Table 7, which together show that MA10 refines ISA, are described next. We use $I_{\mathtt{M}}$ to denote the invariant on machine M and $r_{\mathtt{M}}$ to denote the refinement map from machine M. (The range is uniquely determined by Table 7.) Recall that $I_{\mathtt{ISA}}$ is the set of all ISA states. The proof of M6-ISA is a straightforward direct proof using flushing as the refinement map, thus $I_{\mathtt{M6}}$ is the set of all M6 states.

Our first refinement proof involving two pipelined machines relates M7 to M6 using refinement map $r_{\mathtt{M7}}$ (see Figure 38). We now describe $r_{\mathtt{M7}}$ and merely note that the refinement maps for the other proofs are similar. We name pipeline latches based on the pipeline stage names surrounding them, *e.g.*, the pipeline latch between IF1 and ID in the 6 stage machine is IF1_ID.

The only essential difference between M7 and M6 is that when a branch mispredict occurs, the number of cycles required for M7 to recover is four, while M6 only needs three cycles. To deal with this stuttering, we define three invariants on M7; essentially, they state that a branch mispredict results in four consecutive bubbles in the pipeline. The invariants are 1) if IF1_IF2 is invalid, then IF2_ID, ID_EX, and EX_M1 are invalid; 2) if IF1_IF2 is valid and IF2_ID is invalid, then ID_EX,

Refinement Proof	CNF		Verification Times (sec)		
	Vars	Clauses	UCLID	Siege	Total
M6-ISA	28,256	83,725	8.00	10.00	18.00
M7-M6	1,116	3,124	0.39	0.06	0.45
M8-M7	479	1,291	0.24	0.01	0.25
M9-M8	380	1,045	0.21	0.01	0.22
M10-M9	433	1,201	0.29	0.01	0.30
M10I-M10	213	562	0.08	0.01	0.09
M10ID-M10I	469	1,210	0.15	0.01	0.16
M10IDW-M10ID	837	2,149	0.23	0.03	0.26

Table 7: Verification times and CNF statistics for the compositional verification problems.

EX_M1, and M1_M2 are invalid; and 3) if both IF1_IF2 and IF2_ID are valid, and ID_EX and EX_M1 are invalid, then M1_M2 and M2_WB are invalid.

The definition of the refinement map $r_{\mathtt{M7}}$ consists of three cases. In all the cases, the pipeline latches EX_M1, M1_M2, M2_WB, the register file, the instruction memory, and the data memory in M7 get mapped to their counterparts in M6. Case 1 occurs if in M7, IF1_IF2 is invalid, IF1_IF2 is valid and IF2_ID is invalid, or IF1_IF2 and IF2_ID are valid and ID_EX and EX_M1 are invalid. In this case, the program counter, IF1_IF2, and IF2_ID in M7 get mapped to the program counter, IF1_ID, and ID_EX in M6, and the rank is 1. Case 2 occurs when IF1_IF2, IF2_ID, and ID_EX in M7 are valid and EX_M1, M1_M2, and M2_WB are invalid. This is the result of a stuttering step by M7. The rank is 0 and we map the program counter associated with the instruction in IF1_IF2 of M7 to the program counter in M6, while IF2_ID and ID_EX in M7 are mapped to IF1_ID and ID_EX in M6. Otherwise, the mapping of states is the same as in case 2, except that the rank is 0.

To prove compositionally that M7 refines ISA requires the use of Theorem 8. To see why, note that the use of Theorem 7 requires that $r_{\mathtt{M7}}(I_{\mathtt{M7}}) \supseteq I_{\mathtt{M6}}$, which is not true, as I_6 is the set of all M6 states. However, $I_{\mathtt{M7}}$ satisfies the property that $r_{\mathtt{M6}}(r_{\mathtt{M7}}(I_{\mathtt{M7}})) \supseteq I_{\mathtt{ISA}}$, and therefore we can use Theorem 8. To prove this using UCLID, we define a witness function, f, that given an ISA state returns the M7 state with the same programmer visible components, but all of whose pipeline latches are invalid. It is now enough to show that for every state s in $I_{\mathtt{ISA}}$, we have that $r_{\mathtt{M6}}(r_{\mathtt{M7}}(f(s))) = s$ and that $f(s) \in I_{\mathtt{M7}}$.

For the rest of the deep pipeline proofs, it turns out that we can use the simpler Theorem 7. For example, in case of the M8-M7 proof, we have to show that $r_{\mathtt{M8}}(I_{\mathtt{M8}}) \supseteq I_{\mathtt{M7}}$, which we do by defining

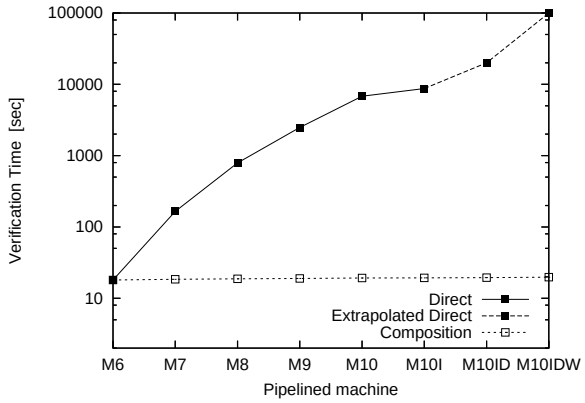


Figure 39: Comparison of direct and compositional approaches.

a suitable witness function that maps states in $I_{\texttt{M7}}$ to $I_{\texttt{M8}}$ and then proceed as above.

7.3.2 Instruction Caches, Data Caches, and Write Buffers

We now show how to verify the instruction cache, the data cache, and the write buffer. This corresponds to the last three refinement proofs in Table 7. For all of these proofs, we use the proof rule given in Theorem 7. Since we have seen how to apply the theorem in the previous section, here we only describe the refinement map, invariants, and witness function for each of the proofs.

The state components of `M10I` and `M10` are identical except for the instruction cache. Thus, the refinement map just ignores the instruction cache and is the identity mapping for all other state components. Since two machines do not stutter with respect to one another, we can in fact prove a bisimulation. This means that the WEB-refinement proof can be reduced further, as no rank function is needed. The only invariant required is that the valid instruction cache entries are consistent with the instruction memory.

The data cache is direct mapped and is similar to the instruction cache. The proof of `M10ID-M10I` is similar to the proof of `M10I-M10`. The refinement map ignores the data cache and retains all the other state components, including the instruction cache. Also, an invariant similar to the one used for the instruction cache is required stating that all valid entries in the data cache are consistent with the data memory.

`M10IDW` differs from `M10ID` only in that it contains a write buffer. These two machines do not stutter with respect to each other; thus, we can prove a bisimulation result, as before. The

refinement map is obtained by first updating the data memory with the valid entries in the write buffer, and projecting out the remaining state elements (including the instruction and data cache states). We prove the invariant that the combined state of the write buffer and the data memory is consistent with the state of the data memory of a machine that does not have a write buffer.

Finally, the witness function from $I_{\texttt{M10}}$ to $I_{\texttt{M10I}}$ just adds an instruction cache, all of whose elements are invalid to an $I_{\texttt{M10}}$ state. The witness functions for the other proofs are similarly defined.

7.3.3 Counterexamples

The most tedious and time-intensive part of the verification effort is often debugging and understanding counterexamples. Since the compositional approach reduces the verification problem into simpler subproblems, the debugging process is much simplified. This is because one can isolate the cause of failure simply by noting which stage of the composition proof fails. This is impossible to do when verifying the complex processor in a monolithic fashion and it is difficult to overstate the importance of this aspect of our work, as the differences in the complexity of the error traces can be quite drastic.

As a concrete example of how compositional verification simplifies the debugging task, we note that when we tried to verify a buggy variant of the instruction cache —there was a bug because when determining whether a cache hit has occurred, the design did not check the validity of the cache block— we found that the counter example generated by UCLID for the direct approach was 4,429 lines long while the counter example generated from the composition step was 390 lines long. Obviously, the shorter counterexample was much simpler to understand and, consequently, fixing the bug was much easier. All the bugs we encountered were similarly much easier to check in the compositional framework and this aspect of compositional verification may well be more important than the improvement we obtained in verification times.

7.4 Related Work

We now describe previous approaches for verifying pipelined machines based on proof strategies that use decomposition techniques to achieve scalability. Approaches based on the use of theorem provers are inherently compositional and scalable in nature, but require an extraordinary amount of effort. In contrast, our approach generates only a very small number of high-level proof obligations,

which are then automatically discharged using a decision procedure.

Jones et al. [39, 40] verify an out-of-order execution unit using incremental flushing. Their approach relates the implementation to an intermediate machine, where the scheduling logic is abstracted, which is then related to the ISA. In comparison, we can deal with any refinement map, we have a general theory with a complete rule for relating any number of intermediate machines, and we guarantee that all safety *and* liveness properties are preserved. They also state that the amount of effort required to deductively justify the proof decompositions offsets the advantages obtained using the decomposition. In our approach, the individual refinement proofs can be chained together as WEB-refinement is a compositional notion.

Jhala and McMillan [38] describe an approach for showing that processor models behave like their instruction set architecture models using compositional model checking. The proof methodology is to decompose the correctness criterion into a number of temporal properties that can then be automatically verified by model checking. The decomposition is performed using the SMV proof assistant [70]. They apply this approach to verify an abstract microprocessor model that has many features such as branch prediction, speculative execution, and out-of-order execution. In their models, combinational circuit blocks such as the ALU are abstracted using Uninterpreted Functions. They do not quantitatively describe the amount of expert user effort required for the proofs, but state that their approach is "considerably more laborious" than model checking finite state machines. In contrast, we required only about one week of expert user effort for the compositional refinement proof. Also, our approach accounts for liveness, which is not taken into account in their approach.

7.5 Conclusions

We presented a complete compositional framework based on refinement for proving that pipelined machine models satisfy the same safety and liveness properties as their corresponding instruction set architecture models. This allowed us to obtain exponential savings in verification times over previous monolithic approaches, and, in fact, we were able to easily verify models that decision procedures such as UCLID cannot directly handle. We also showed how compositional reasoning based on refinement can be integrated into the design cycle and how this leads to faster verification times, shorter and clearer counterexamples, and enhanced design understanding.

CHAPTER VIII

INTEGRATING DEDUCTIVE REASONING WITH DECISION PROCEDURES

Verification engines for checking that pipelined machines work correctly can be roughly classified into two categories: term-level decision procedures and deductive reasoning engines or theorem provers. Term-level decision procedures are automatic, but can only be used for problems expressible in restricted logics. In Chapters 3, 4, 5, 6, and 7, we have described several refinement-based methods for the verification of term-level pipelined machines in a highly automated, efficient, and scalable manner. Unfortunately, due to their limitation to restricted logics, term-level decision procedures as the name suggests can only be used to reason about pipelined machines defined at the term-level. The restriction to term-level models has severely limited the applicability of approaches based on decision procedures because to be industrially applicable, we need a firm connection to the RTL level, something that abstract term-level models do not provide.

In contrast, theorem provers such as ACL2 [43, 47, 42] have very expressive languages and associated logics and can be used to reason about pipelined machines at various levels of abstraction including at the term-level and at the bit-level. We choose to work with ACL2 as it is one of the most automated program verification systems available and has been used in a number of commercial applications (see Section 2.4 for more details). Even so, ACL2 requires extensive user guidance.

What we have is two verification systems for checking properties about hardware systems such as pipelined machines with radically different pragmatics. One engine is general purpose; the other is restricted, and by virtue of its restrictions, can exploit a specialized implementation to provide extreme speedups (by which we mean *several* orders of magnitude). If we could only design transformations that reduce ACL2 conjectures to problems that can be checked using decision procedures where there is semantic overlap—transformations we had some formal reason to trust—then we could have the best of both worlds: we could work in the general-purpose setting, yet use these transformations to hand off sub-problems to an efficient, but restricted decision procedure.

To this end, we developed ACL2-SMT, a framework for coarse-grained integration of SMT

solvers—decision procedures for decidable fragments of first-order logic—with ACL2. SMT solvers are decision procedures that can be used to decide formulas in a restricted fragment of first-order logic described in the Satisfiability Modulo Theories Library (SMT-LIB) language, which has become the standard input format for decision procedures. This is very similar to the CNF format, which is the standard input format used for SAT solvers. We choose to integrate ACL2 with SMT solvers that can decide formulas in the closed quantifier free logic of linear integer arithmetic, integer arrays, and uninterpreted functions and predicates (QF_AUfLia). The reason being that refinement-based properties of term-level pipelined machines are expressible in the QF_AUfLia logic and also because there are many decision procedures that can check QF_AUfLia formulas in the SMT-LIB language. This makes our framework general in that now any decision procedure that can check QF_AUfLia formulas in SMT format can be made to process ACL2 formulas using our framework.

Using this framework, an SMT solver is integrated with ACL2 as a simplification engine that can be used to process ACL2 clauses. Therefore, we call it the SMT clause processor. The integration involves a translation mechanism that converts ACL2 formulas to SMT problems in QF_AUfLia, which can then be checked using an SMT solver.

One of our primary motivations for integrating ACL2 with SMT solvers is for the verification of bit-level pipelined machines. The ACL2-SMT system is a crucial component of our approach for checking that bit-level pipelined machines work correctly. In the next chapter where this verification approach is described in detail, we also describe a large case study where we proved the correctness of executable pipelined machines defined at the bit-level using the ACL2-SMT system. The proof established that the pipelined machine is stuttering bisimilar, under a suitable refinement map, with the instruction set architecture. Such a proof is beyond the scope a term-level decision procedure since its logic is too weak to even state it, and would require considerably more human effort to check using just ACL2.

Another advantage of using the combined system obtained by integrating an SMT solver with ACL2 is easier debugging. Term-level models are not executable. One cannot run the models on

some inputs and get results, and therefore one cannot run tests, which is very useful for debugging. ACL2 models are efficiently executable and has been used as simulation platforms for prefabrication requirements testing. By developing a concrete model in ACL2 corresponding to the term-level model being verified, one can execute counterexamples on the concrete model generated from the verification of the abstract term-level model making it easier to debug.

The rest of the chapter is organized as follows. In Section 8.1, we give a high-level overview of our framework for integrating SMT solvers with ACL2 and discuss our design choices. In Section 8.3, we give a brief description of the semantics of ACL2, focusing on the subset of ACL2 formulas that can be translated to QF_AUfLia formulas. In Section 8.2, we describe the syntax and the semantics of QF_AUfLia. In Section 8.2 we describe in detail the SMT clause processor, an integration of an SMT solver with ACL2. Related work is described in Section 8.6 and we conclude in Section 8.7.

8.1 Integration Strategy

In this section, we describe a framework for the integration of term-level SMT solvers with ACL2. While much of the previous work on integrating decision procedures into heuristic theorem provers has focused on fine-grained integration, such integration is difficult to implement and maintain [14]. It requires the cooperation of the various proof heuristics employed and can take a long time to optimize. In addition, from the user's point of view, it is difficult to understand exactly what can be automatically handled. In combining SMT solvers with ACL2, we decided on a coarse-grained integration, meaning that the user has to explicitly call the SMT solver. The main reason for our choice is that there is a complexity mismatch between the formulas we want to automatically verify and the formulas that the proof heuristics used in ACL2 can easily handle: the formulas involved are so complicated, that it becomes very hard for the ACL2 proof heuristics to analyze them.

The coarse-grained integration of an SMT solver with ACL2 is achieved using the clause processor mechanism, which is an interface provided by ACL2 that allows the use of external reasoning tools to simplify formulas during an ACL2 proof attempt [44]. Note that integrating an SMT solver as a clause processor leads to a coarse-grained integration as the SMT solver has to be invoked explicitly by the user by providing a hint to ACL2 indicating that a particular goal be simplified using

the solver. Using this interface, an external tool can be integrated with ACL2 as a tool verified by the ACL2 theorem prover or as an unverified but trusted tool. Since SMT solvers are external tools not implemented in the ACL2 programming language, we cannot verify that they work correctly using the ACL2 theorem prover. Even otherwise, correctness proofs for SMT solvers would be very difficult given the complexity of the implementations of these solvers. Therefore, we choose to integrate SMT solvers as unverified but trusted clause processors. Note that ACL2 allows the use of an unverified tool only with a trust tag, which is used to indicate that the validity of the formulas proved using the external tool depends on the correctness of that tool.

Our approach for integrating an SMT solver with ACL2 is as follows. Remember that our goal is to use a solver to reason about term-level models in ACL2. Therefore, the first step is to identify a decidable fragment of first-order logic in which properties about term-level models are expressible. We chose QF_AUfLia as this logic allows us to conveniently express system-level refinement-based properties for term-level pipelined machine models. Also, there exist many efficient decision procedures that can handle formulas in QF_AUfLia, such as the Yices [100] system and the Barcelogic for SMT solver [10]. We then identified a subset of the ACL2 logic such that formulas in this subset can be directly translated to formulas in QF_AUfLia.

To use an SMT solver to reason about ACL2 formulas, we developed a translation mechanism from ACL2 to SMT that works as follows. Given an ACL2 formula say f, the translation mechanism reduces f by unrolling functions, expanding macros, and performing other simplifications resulting in formula f^{smt}. If the resulting reduced formula f^{smt} is not in the subset of ACL2 that can be translated to a QF_AUfLia formula, we give an error and abort. Otherwise, f^{smt} is translated to a formula say g in the SMT-LIB language. We then call an SMT solver to check the validity of g. If the solver can prove that g is valid, then it implies the validity of f, provided of course that we trust the solver and our translation mechanism. Otherwise, the solver generates a counter example, which is mapped back to ACL2 and can be used to analyze the reason why f is not valid.

Using this framework, we have integrated the Yices solver to work as an ACL2 clause processor. We call this the SMT clause processor. Note that this approach can be easily extended to work with other SMT solvers if a parser for the output of that solver is available.

bool-exp ::= `true` | `false`
| *bool-var*
| `(if_then_else` *bool-exp bool-exp bool-exp*`)`
| `(equal` *int-exp int-exp*`)` | `(<` *int-exp int-exp*`)`
| `(`*pred-symb int-exp ...int-exp*`)`

int-exp ::= *int-const* | *int-var*
| `(ite` *bool-exp int-exp int-exp*`)`
| `(+` *int-exp int-exp*`)` | `(-` *int-exp int-exp*`)`
| `(`*func-symb int-exp ...int-exp*`)`
| `(select` *iarray-exp int-exp*`)`

iarray-exp ::= *iarray-var*
| `(store` *iarray-exp int-exp int-exp*`)`

The top-level expression is a boolean expression

Figure 40: QF_AUfLia syntax

8.2 Syntax and Semantics of QF_AUfLia

We now describe the syntax and semantics of QF_AUfLia, a decidable fragment of first order logic that includes linear integer arithmetic, integer arrays, and uninterpreted functions and predicates.

QF_AUfLia manipulates of world of integers, booleans, integer arrays, and functions over these basic values. Boolean expressions, also known as formulas, yield `true` or `false`. A QF_AUfLia "function" is a function from multiple integers to an integer, while a "predicate" is a function from multiple integers to a boolean. The QF_AUfLia syntax is summarized in Figure 40.

A boolean expression can be the constants `true` or `false`, a boolean variable, or the application of the `if_then_else` operator on sub-formulas. They can also be formed by comparing two integer expressions with = or <, or by the application of a predicate symbol representing integer-expression arguments, where a predicate symbol represents an uninterpreted predicate (UP).

An integer expression can be a variable; an application of the if/then/else operator `ITE`, which selects one of its two integer sub-terms based on the value of its controlling boolean formula; an application of addition (+) or subtraction (-); or application of the `select` function which is used to get the value from an array corresponding to a given index; or application of an uninterpreted function (UF) symbol.

An integer array expression can be a variable or an application of the `store` operation, which is

used to update an array.

Note that QF_AUfLia allows other operations. For example, boolean formulas can also be constructed using conjunction, disjunction, and negation of sub-formulas. But, we only show those operations that are used in the translation from ACL2 to SMT.

Figure 41 provides a simple semantics for QF_AUfLia. Just as with the ACL2 semantics, the semantics for QF_AUfLia is given by a meaning function $[\![exp]\!]^u_\beta$, which gives the meaning of some QF_AUfLia term *exp* given some *environment* provided by β, a function mapping integer and boolean variables to values, and *u*, a function providing the meaning for the uninterpreted functions and predicates.

The SMT-LIB language for describing QF_AUfLia formulas also has the `let` construct, which can be used to bind lexically scoped local variables.

8.3 ACL2 Syntax and Semantics

Before we delve into the details of the SMT clause processor, we give a brief description of the syntax and semantics of a subset of the ACL2 that can be translated to QF_AUfLia. We call this subset Arrays, Linear integer arithmetic, and Uninterpreted functions and predicate symbols (*ALU*). The syntax and semantics of *ALU* are shown in Figures 43 and 43, respectively. A full discussion of the ACL2 logic can be found in [45].

The semantics is given by a meaning function $[\![exp]\!]^u_\beta$, which gives the meaning of some ACL2 term *exp* given some *environment* provided by β, a function mapping integer and boolean variables to values, and *u*, a function providing the meaning for the uninterpreted functions and predicates.

The reason we chose *ALU* is that it can be used to model pipelined machines at the term-level and that can also be used to express system-level refinement-based properties for these models. This subset consists ACL2 formulas constructed using Uninterpreted Functions (UFs), Uninterpreted Predicates (UPs), constrained functions `store` and `select` for array operations, and ACL2 primitives `if`, `<`, `equal`, `binary-+` and `unary--` over the boolean, integer, and integer array domains. The integer and boolean domains are defined in ACL2 using the `integerp` and `booleanp` functions. We define integer arrays as associative lists, which are lists of key and value pairs, where both the key and the value are integers. We define the function `integer-arrayp` to identify integer

u :function symbols of arity $n \rightarrow$ functions of arity n,
predicate symbols of arity $n \rightarrow$ predicates of arity n

β :assignment for variables

lp :looks up an integer array for the value corresponding to a given index

$$
\begin{aligned}
[\![\textit{bool-var}]\!]^u_\beta &= \beta\,\textit{bool-var} \\
[\![\texttt{true}]\!]^u_\beta &= \textit{true} \\
[\![\texttt{false}]\!]^u_\beta &= \textit{false} \\
[\![\texttt{(if\_then\_else}\ \textit{bool-exp}_1\ \textit{bool-exp}_2\ \textit{bool-exp}_3\texttt{)}]\!]^u_\beta &= \begin{cases} [\![\textit{bool-exp}_2]\!]^u_\beta & [\![\textit{bool-exp}_1]\!]^u_\beta, \\ [\![\textit{bool-exp}_3]\!]^u_\beta & \text{otherwise.} \end{cases} \\
[\![\texttt{(=}\ \textit{int-exp}_1\ \textit{int-exp}_2\texttt{)}]\!]^u_\beta &= [\![\textit{int-exp}_1]\!]^u_\beta = [\![\textit{int-exp}_2]\!]^u_\beta \\
[\![\texttt{(<}\ \textit{int-exp}_1 \textit{int-exp2}\texttt{)}]\!]^u_\beta &= [\![\textit{int-exp}_1]\!]^u_\beta < [\![\textit{int-exp}_2]\!]^u_\beta \\
[\![\texttt{(}\textit{pred-symb}\ \textit{int-exp}_1 \ldots \textit{int-exp}_n\texttt{)}]\!]^u_\beta &= [\![\textit{pred-symb}]\!]^u_\beta([\![\textit{int-exp}_1]\!]^u_\beta \ldots [\![\textit{int-exp}_n]\!]^u_\beta) \\
[\![\textit{int-var}]\!]^u_\beta &= \beta\,\textit{int-var} \\
[\![\texttt{(ite}\ \textit{bool-exp}\ \textit{int-exp}_1\ \textit{int-exp}_2\texttt{)}]\!]^u_\beta &= \begin{cases} [\![\textit{int-exp}_1]\!]^u_\beta & [\![\textit{bool-exp}]\!]^u_\beta, \\ [\![\textit{int-exp}_2]\!]^u_\beta & \text{otherwise.} \end{cases} \\
[\![\texttt{(+}\ \textit{int-exp}_1\ \textit{int-exp}_2\texttt{)}]\!]^u_\beta &= [\![\textit{int-exp}_1]\!]^u_\beta + [\![\textit{int-exp}_2]\!]^u_\beta \\
[\![\texttt{(-}\ \textit{int-exp}_1\ \textit{int-exp}_2\texttt{)}]\!]^u_\beta &= [\![\textit{int-exp}_1]\!]^u_\beta - [\![\textit{int-exp}_2]\!]^u_\beta \\
[\![\texttt{(}\textit{func-symb}\ \textit{int-exp}_1 \ldots \textit{int-exp}_n\texttt{)}]\!]^u_\beta &= [\![\textit{func-symb}]\!]^u_\beta([\![\textit{int-exp}_1]\!]^u_\beta \ldots [\![\textit{int-exp}_n]\!]^u_\beta) \\
[\![\textit{pred-symb}]\!]^u_\beta &= u\,\textit{pred-symb} \\
[\![\textit{func-symb}]\!]^u_\beta &= u\,\textit{func-symb} \\
[\![\textit{iarray-var}]\!]^u_\beta &= \beta\,\textit{iarray-var}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{(select (store}\ \textit{iarray-exp}\ \textit{int-exp}_1 \\
&\qquad \textit{int-exp}_2\texttt{)}\ \textit{int-exp}_3\texttt{)}]\!]^u_\beta = \\
&\begin{cases} [\![\textit{int-exp}_2]\!]^u_\beta & [\![\textit{int-exp}_1]\!]^u_\beta = [\![\textit{int-exp}_3]\!]^u_\beta, \\ [\![\texttt{(select}\ \textit{int-exp}_3\ \textit{iarray-exp}\texttt{)}]\!]^u_\beta & \text{otherwise.} \end{cases}
\end{aligned}
$$

e is valid iff $\forall \beta, u\ .\ [\![e]\!]^u_\beta = \textit{true}$

Figure 41: QF_AUfLia semantics

bool-exp ::= `T` | `NIL`
| *bool-var*
| `(if` *bool-exp bool-exp bool-exp*`)`
| `(equal` *int-exp int-exp*`)` | `(<` *int-exp int-exp*`)`
| `(`*pred-symb int-exp ...int-exp*`)`

int-exp ::= *int-const* | *int-var*
| `(if` *bool-exp int-exp int-exp*`)`
| `(binary-+` *int-exp int-exp*`)` | `(unary--` *int-exp*`)`
| `(`*func-symb int-exp ...int-exp*`)`
| `(select` *int-exp iarray-exp*`)`

iarray-exp ::= *iarray-var*
| `(store` *int-exp int-exp iarray-exp*`)`

The top-level expression is a boolean expression

Figure 42: *ALU* syntax

arrays.

Boolean connectives such as conjunction, disjunction, and negation in are defined in ACL2 using `if` expressions. Therefore, all the boolean connectives can be reduced to `if` expressions by expanding functions and macros.

Uninterpreted functions and Uninterpreted predicates (UPs) are used extensively in term-level models to abstract combinational circuit blocks such as the ALU. UFs can be thought of as functions from integers to integers with a name but no intensional definition. UFs satisfy only the property of functional consistency, *i.e.*, if the inputs to two different instances of a UF are equal, then it implies that the outputs are also equal. UPs are similar, but are taken to have boolean range rather than integer range.

UFs and UPs are modeled using encapsulated functions. Encapsulation allows the introduction of constrained functions. The definition of the function is hidden by the encapsulation, and only some properties are exported to the ACL2 world. An example of an UF in ACL2 is shown in Figure 44. The ternary UF `alu` that abstracts away the actual computation of a processor ALU is defined as a function that ignores its inputs and returns the integer value 1. The function definition is hidden using an `encapsulate` form. This form prevents the ACL2 reasoning system from making deductions about `alu` based on its functional definition. For ACL2's purposes, the only properties

u :function symbols of arity $n \rightarrow$ functions of arity n,
predicate symbols of arity $n \rightarrow$ predicates of arity n
β :assignment for variables

$$
\begin{aligned}
[\![\textit{bool-var}]\!]^u_\beta &= \beta\,\textit{bool-var} \\
[\![\texttt{T}]\!]^u_\beta &= \textit{true} \\
[\![\texttt{NIL}]\!]^u_\beta &= \textit{false} \\
[\![\texttt{(if } \textit{bool-exp}_1 \ \textit{bool-exp}_2 \ \textit{bool-exp}_3\texttt{)}]\!]^u_\beta &= \begin{cases} [\![\textit{bool-exp}_2]\!]^u_\beta & [\![\textit{bool-exp}_1]\!]^u_\beta, \\ [\![\textit{bool-exp}_3]\!]^u_\beta & \text{otherwise.} \end{cases} \\
[\![\texttt{(equal } \textit{int-exp}_1 \ \textit{int-exp}_2\texttt{)}]\!]^u_\beta &= [\![\textit{int-exp}_1]\!]^u_\beta = [\![\textit{int-exp}_2]\!]^u_\beta \\
[\![\texttt{(<} \ \textit{int-exp}_1 \textit{int-exp2}\texttt{)}]\!]^u_\beta &= [\![\textit{int-exp}_1]\!]^u_\beta < [\![\textit{int-exp}_2]\!]^u_\beta \\
[\![\texttt{(}\textit{pred-symb} \ \textit{int-exp}_1 \ldots \textit{int-exp}_n\texttt{)}]\!]^u_\beta &= [\![\textit{pred-symb}]\!]^u_\beta([\![\textit{int-exp}_1]\!]^u_\beta \ldots [\![\textit{int-exp}_n]\!]^u_\beta) \\
[\![\textit{int-var}]\!]^u_\beta &= \beta\,\textit{int-var} \\
[\![\texttt{(if } \textit{bool-exp} \ \textit{int-exp}_1 \ \textit{int-exp}_2\texttt{)}]\!]^u_\beta &= \begin{cases} [\![\textit{int-exp}_1]\!]^u_\beta & [\![\textit{bool-exp}]\!]^u_\beta, \\ [\![\textit{int-exp}_2]\!]^u_\beta & \text{otherwise.} \end{cases} \\
[\![\texttt{(binary-+} \ \textit{int-exp}_1 \ \textit{int-exp}_2\texttt{)}]\!]^u_\beta &= [\![\textit{int-exp}_1]\!]^u_\beta + [\![\textit{int-exp}_2]\!]^u_\beta \\
[\![\texttt{(unary--} \ \textit{int-exp}\texttt{)}]\!]^u_\beta &= -[\![\textit{int-exp}]\!]^u_\beta \\
[\![\texttt{(}\textit{func-symb} \ \textit{int-exp}_1 \ldots \textit{int-exp}_n\texttt{)}]\!]^u_\beta &= [\![\textit{func-symb}]\!]^u_\beta([\![\textit{int-exp}_1]\!]^u_\beta \ldots [\![\textit{int-exp}_n]\!]^u_\beta) \\
[\![\textit{pred-symb}]\!]^u_\beta &= u\ \textit{pred-symb} \\
[\![\textit{func-symb}]\!]^u_\beta &= u\ \textit{func-symb} \\
[\![\textit{iarray-var}]\!]^u_\beta &= \beta\,\textit{iarray-var}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{(select } \textit{int-exp}_1 \ \texttt{(store } \textit{int-exp}_2 \\
&\qquad \textit{int-exp}_3 \ \ \textit{iarray-exp}\texttt{))}]\!]^u_\beta = \\
&\qquad \begin{cases} [\![\textit{int-exp}_3]\!]^u_\beta & [\![\textit{int-exp}_1]\!]^u_\beta = [\![\textit{int-exp}_2]\!]^u_\beta, \\ [\![\texttt{(select } \textit{int-exp}_1 \ \ \textit{iarray-exp}\texttt{)}]\!]^u_\beta & \text{otherwise.} \end{cases}
\end{aligned}
$$

e is valid iff $\forall \beta, u\ .\ [\![e]\!]^u_\beta = \textit{true}$

Figure 43: *ALU* semantics

```
(encapsulate
  ((alu (x y z) t))
  (local (defun alu (x y z)
           (declare (ignore x)
                    (ignore y)
                    (ignore z))
           1))
  (defthm alu-type
    (implies (and (integerp a)
                  (integerp b)
                  (integerp c))
      (integerp (alu a b c)))))
```

Figure 44: An uninterpreted function represented in ACL2

satisfied by `alu` are the ones defined by theorem `alu-type`: if all three arguments to `alu` are of integer type, then the output is also of type integer. Function `alu` also satisfies the property of functional consistency that is satisfied by all functions in ACL2. UP's can be defined in a similar way.

We define two constrained functions `select` and `select` that can be used to update and access integer arrays. The `store` takes an index, a value, and an integer array as input. The `select` takes an index and an integer array as input. Both the `store` and `select` functions satisfy a property relating their input and output types. The only other property satisfied by these functions is shown in Figure 43. Note that we require that in the ACL2 formulas that we can handle, `store` operations always appear inside of a `select`. If a `select` is applied directly to an integer array variable, it is treated as an uninterpreted function.

ACL2 functions are total, meaning that they are defined for all types. For example, the `and` function returns the conjunction of its inputs if the inputs are boolean, and returns the second argument, if the inputs are integers. For the ACL2 expressions that we consider, the input types of the functions are restricted to be either integers, booleans, or integer arrays. Since ACL2 functions are total, we require that the type restriction be enforced using a top-level hypothesis that assigns an integer, a boolean, or an integer array type for all the free variables in the ACL2 expression.

bool-var	$\rightarrow$	*bool-var*
`T`	$\rightarrow$	`true`
`NIL`	$\rightarrow$	`false`
(*pred-symbol int-expr*$_1$ ... *int-expr*$_n$)	$\rightarrow$	(*pred-symbol int-expr*$_1$ ... *int-expr*$_n$)
(`if` *bool-expr*$_1$ *bool-expr*$_2$ *bool-expr*$_3$)	$\rightarrow$	(`if_then_else` *bool-expr*$_1$ *bool-expr*$_2$ *bool-expr*$_3$)
int-var	$\rightarrow$	*int-var*
int-const	$\rightarrow$	*int-const*
(`if` *bool-expr int-expr*$_1$ *int-expr*$_2$)	$\rightarrow$	(`ITE` *bool-expr int-expr*$_1$ *int-expr*$_2$)
(`equal` *int-expr*$_1$ *int-expr*$_2$)	$\rightarrow$	(`=` *int-expr*$_1$ *int-expr*$_2$)
(`<` *int-expr*$_1$ *int-expr*$_2$)	$\rightarrow$	(`<` *int-expr*$_1$ *int-expr*$_2$)
(`binary-+` *int-expr*$_1$ *int-expr*$_2$)	$\rightarrow$	(`+` *int-expr*$_1$ *int-expr*$_2$)
(`unary--` *int-expr*)	$\rightarrow$	(`-` *int-expr*)
(*func-symbol int-expr*$_1$ ... *int-expr*$_n$)	$\rightarrow$	(*func-symbol int-expr*$_1$ ... *int-expr*$_n$)
(`select` *int-expr int-array-expr*)	$\rightarrow$	(`select` *int-array-expr int-expr*)
(`store` *int-expr*$_1$ *int-expr*$_2$ *int-array-expr*)	$\rightarrow$	(`store` *int-array-expr int-expr*$_1$ *int-expr*$_2$)

Figure 45: Mapping from ACL2 to QF_AUfLia

8.4 SMT Clause Processor

We now describe the SMT clause processor that can be used to automatically check the validity of a subset of ACL2 formulas. As described earlier, the SMT clause processor employs a translation mechanism that converts an ACL2 expression to a QF_AUfLia formula in the SMT-LIB language, which can then be checked using an SMT solver. The high-level approach of the translation mechanism is to reduce ACL2 clauses to formulas constructed using a limited set of ACL2 operators for which a mapping to QF_AUfLia exists. This mapping is defined first, which is then used to describe the translation mechanism.

8.4.1 Mapping from ACL2 to QF_AUfLia

QF_AUfLia is a simple, restricted logic relative to ACL2; We map ACL2 formulas to QF_AUfLia by carefully defining the domains in ACL2. One of the main issues we have to handle is that QF_AUfLia operations obey a statically monomorphic type discipline, while ACL2 functions are total over the entire domain of values. For example, the `if` function in ACL2 can be applied to values from any domain, whereas this is not possible in QF_AUfLia. Therefore, we translate only ACL2 formulas that have a top-level hypothesis that assigns a type to all the free variables in the formula. The mapping is shown in Figure 45.

We map booleans in ACL2 to the the QF_AUfLia boolean domain. The constants `T` and `NIL` are mapped to `true` and `false`. The ACL2 function `booleanp` is a recognizer for booleans, returning true when its argument is either `T` or `NIL`. ACL2 integers are mapped to QF_AUfLia terms or integers. The function `integerp` is, likewise, the ACL2 recognizer for integer values. Integer arrays recognized by the `integer-arrayp` function are mapped to arrays in QF_AUfLia.

The ACL2 `if` construct applied to a boolean and two integer expressions (where the boolean expression is the controlling formula of the `if`) is mapped to the `ite` operator, and the ACL2 `if` construct applied to three boolean expressions is mapped to the `if_then_else` construct. The ACL2 binary addition operator is mapped to the `+` operator and the ACL2 unary subtraction operator is mapped to `-` operator applied to one argument. The ACL2 equal (`equal`) and less-than (`<`) operators are mapped to their counterparts `=` and `<`, respectively, in QF_AUfLia. The ACL2 functions `select` and `store` are mapped to the `select` and `store` operators.

In QF_AUfLia, all variables have a type, but ACL2 is untyped. When an ACL2 formula gets translated to QF_AUfLia, the ACL2 formula is of the form `(implies h f)`, where f is the ACL2 expression that is translated to a formula in QF_AUfLia, and h is a top-level hypothesis that assigns a type to all the free variables in f.

Theorem 9. *If an ACL2 formula f is valid, then its mapping into QF_AUfLia, $m(f)$ is valid.*

Proof: The theorem follows from the semantics of a subset of ACL2 and the semantics of QF_AUfLia defined in Sections 8.3 and 8.2, respectively, using structural induction. The ACL2 constants `T` and `NIL` have the same semantics as the QF_AUfLia constants `true` and `false`. The ACL2 `if` operator when restricted to boolean arguments has the same semantics of the QF_AUfLia `if_then_else` operator. The equality of linear integer arithmetic and array semantics can be clearly seen from Figures 43 and 41, which describe the semantics of an ACL2 subset and the QF_AUfLia semantics. Finally, constrained functions in ACL2 implemented using encapsulation, which are essentially uninterpreted functions are mapped to uninterpreted functions. ■

8.5 Translation Mechanism

We now describe the translation mechanism that converts ACL2 clauses to QF_AUfLia formulas in the SMT-LIB language. One approach would be to ask the ACL2 user to guarantee that the input

clause to the SMT clause processor is in a form that can be directly mapped to a formula in the SMT-LIB language. Such an approach would entail generating input clauses at a very low-level that could result in an explosion in the size of these clauses. Also, a lot of effort would be required on the part of the user to control ACL2 to generate these low-level clauses. Instead, our approach is to allow the user to provide ACL2 clauses, which can then be reduced to an expression in the subset of ACL2 that we can handle. We perform this reduction by unfolding functions, expanding macros, and expanding records defined using the ACL2 records library. It is possible that we are unable to translate the given input clause to an expression in the subset of the ACL2 logic that can be translated to QF_AUfLia. In such a situation, the user might have to reduce the input clause using ACL2 to an expression which can then be dealt with by the SMT clause processor. This approach drastically improves the amount of effort required on the part of the ACL2 user.

The high-level description of the algorithm that processes ACL2 clauses using an SMT solver is given below. We assume that the input ACL2 clause c is of the form $(\rightarrow h\, f)$, where h is a top-level hypothesis that assigns types to all the free variables in f.

1. *Compute free variables of the input formula f to create free-vars-f.*
2. *Determine the type of each of the free variables in f from the top-level hypothesis h. Include the type information for each of the variables in free-vars-f. If a variable is not assigned a type in h or if its type is not integerp, booleanp, or integer-arrayp, output an error and abort.*
3. *Negate the original formula f and expand all unconstrained functions in the negation of f according to their definitions. Call the resulting formula f^f.*
4. *Expand all the macros in f^f to create f^m.*
5. *Determine all the uninterpreted functions (UFs) and uninterpreted predicates (UPs) used in f^m to get a list of UFs and a list of UPs, ufs-of-f ufs-of-f and ufs-of-f, respectively.*
6. *Construct an initial environment using free-vars-f, ufs-of-f, and ups-of-f.*
7. *Translate f^m to a formula in the SMT-LIB language using the mapping from ACL2 to SMT. Note that record variables are not expanded out in this translation. Call the resulting formula f^r.*

8. *Expand all record variables in f^r to get f^{smt}.*

9. *Check f^{smt} using an SMT solver and return the result back to ACL2.*

We now describe each of the steps in the above algorithm in more detail. In implementing the SMT clause processor, we made use of previously defined functions that were used to build ACL2.

In the first step, the free variables of the original formula f are computed. The second step is used to determine the type of all the free variables from the hypothesis. As stated earlier, ACL2 functions are total in that the inputs to these functions can be of any type. Whereas, SMT operations obey a statically monomorphic type discipline. Therefore, we check that the top-level hypothesis h assigns a type to all the free variables of f. Also, QF_AUfLia supports only three domains, which are the integers, booleans, and integer arrays. Therefore, we also check if the type assigned to the free variables of f is any one of these three domains.

Proving that a negated version of the original formula f is unsatisfiable is equivalent to proving that f is valid. Therefore, the original formula f is negated in step 3 of the algorithm, the reason being that to prove the validity of f, the SMT solver can be used to check if after some reductions and translations, the negated version of f is unsatisfiable. In step 3, all the unconstrained functions in the negated version of f are also unrolled, resulting in formula f^f. Note that constrained functions cannot be unrolled as they are essentially black box functions that satisfy some properties but do not have a body.

All the macros in formula f^f are expanded out in step 4 to create f^m. The formula f^m is built using only functions `s` and `g` from the *records* library, constrained functions `select` and `store` for array operations, UFs, UPs, closed lambdas, and ACL2 primitives `if`, `<`, `equal`, `binary-+` and `unary--`. We do not expand lambda expressions as this could result in an explosion in the size of the resulting formula. Instead, we translate lambda expressions to SMT `let` expressions, which are used to bind lexically scoped local variables.

In step 5, we determine the uninterpreted functions (UFs) and uninterpreted predicates (UPs) used in the original formula. UFs and UPs are implemented in ACL2 using constrained functions. UFs have a type constraint that states that if the inputs of a UF are integers then the output is an integer. Similarly, UPs have a type constraint that states that if the inputs of a UP are integers then

the output is a boolean.

We therefore determine that a function is uninterpreted, if it is defined as a constrained function. This information can be ascertained from the ACL2 world, a property list used to maintain a data base of rules, that stores properties about various events such as a function definitions. The formula f^m, which is obtained after expanding out all unconstrained functions and macros in the original formula is traversed to construct the list of constrained functions. We then distinguish UFs and UPs by checking the type constraint associated with a UF or a UP (which can also be obtained by querying the ACL2 world) to determine if the output of the function is an integer or a boolean.

Before we translate f^m to an SMT problem, we run the *expander*, which is an ACL2 library that can be used to simplify ACL2 terms or clauses, on f^m. The *expander* can be thought of as a simplification engine and used as a preprocessor, but its application is not always effective. For example, if deciding the original formula requires considerable amount of propositional reasoning, it might be preferable not to use the *expander* as an SMT solver would be far more efficient at propositional reasoning than the *expander*. Therefore, we provide a flag, which when set uses the *expander* as a preprocessor, but does not do so otherwise.

To translate f^m to a formula in the SMT-LIB language, we use an environment that keeps track of the types of all the free variables in f, the types of newly introduced variables, and the UFs and UPs in f. The environment is a list of 5 elements. The first, second, and fifth elements are lists of boolean, integer, and integer array variables, respectively. The third element is a list of the UPs of f and the fourth element is a list of UFs of f. The fourth element is a list of record variables and also includes type information for each of the record variables, such as the fields of the record and the type of each field. In step 6, the initial value of the environment is created using the list of free variables of f (*free-vars-f*), the list of UFs (*ufs-of-f*), and the list of UPs (*ups-of-f*).

Step 6 take as input f^m and the initial value of the environment and translates f^m to a formula in the SMT-LIB language f^r, using the mapping from ACL2 to SMT. Note that lambdas can be used to introduce new record variables, but the types of these variables are not yet known. To expand these newly introduced record variables, type information is required. Therefore, in step 6, we do not expand out record variables. Instead, while translating f^m to f^r, the types of these newly introduced record variables is determined, and f^r is annotated with this type information. In step 7, the record

variables in f^r are expanded out resulting in the SMT-LIB language f^{smt}.

In step 8, the SMT solver is used to check if f^{smt} is unsatisfiable. If a counterexample is found, it can be directly mapped back to ACL2, which can then be analyzed by the user to determine why f the original formula f is not valid.

8.5.1 Correctness

The correctness of the translation mechanism that converts ACL2 clauses to QF_AUfLia formulas in the SMT-LIB language is given below.

Theorem 10. *If an ACL2 clause c can be translated to a QF_AUfLia formula f^{smt} using the translation mechanism from* ALU *to QF_AUfLia, then c is valid if and only if f^{smt} is unsatisfiable.*

Proof: In the above theorem, c is the input ACL2 clause and is of the form $(\rightarrow \ h \ f)$, where h is a top-level hypothesis that assigns types to all the free variables in f. f^{smt} is the QF_AUfLia formula in the SMT-LIB language obtained by translating c. The translation mechanism can be thought of as performing two high-level transformations. The first transformation is expanding out unconstrained function applications according to their definitions. This transformation is justified by the definitional principle of ACL2 (see [43] for details). The second transformation converts the formula obtained after expanding out all functions and macros, f^m to a QF_AUfLia formula using the mapping from ACL2 to QF_AUfLia. ACL2 operations are total, whereas, QF_AUfLia operations obey a statically monomorphic type discipline. The mapping from ACL2 to QF_AUfLia is justified as shown in Theorem 9, only if the ACL2 operations that are mapped have the arguments of the right type. We ensure this by checking that the top-level hypothesis h assigns either integer, boolean, or an integer array type to all the free variables in f. We also use the SMT solver Yices to type check the resulting SMT formula. Note that f is initially negated and then translated. Therefore, we check if f^{smt} is unsatisfiable to prove the validity of c. ■

8.5.2 Instruction Set Architecture Example

An example is described that demonstrates how the translation mechanism from ACL2 to SMT works. The example uses the term-level model in ACL2 of a simple instruction set architecture (ISA) machine. The ISA machine itself is described in Section 2.1.1.

```
(defun nextsrf (inst rf result)
  (store (dest inst) result rf))

(defun step-isa (isa)
  (let
      ((pc (g 'pc isa))
       (rf (g 'rf isa))
       (imem (select 'imem isa)))
    (let
        ((inst (g pc imem)))
      (let
          ((arg1 (select (src1 inst) rf))
           (arg2 (select (src2 inst) rf)))
        (let
            ((result (alu arg1 arg2)))
          (let
              ((isa-new (seq nil
                             'pc (pcadd pc)
                             'rf (nextsrf inst rf result)
                             'imem imem)))
            isa-new))))))
```

Figure 46: Term-level ACL2 model of a ISAS.

The ACL2 code for the term-level model of the ISA machine is shown in Figure 46. The model is defined using the top-level function `step-isa`, which describes the operational semantics of the ISA machine.

Figure 47 shows a simple property `isa-pc` of the ISA machine that we check using the SMT clause processor. The construct `defthm` is a directive that asks ACL2 to try and prove that the formula given inside of the directive is valid. Note that a hint provided along with the formula asks ACL2 to prove that the formula is valid using the SMT clause processor. Without this hint, ACL2 will try the proof directly. The property checked is that after a step of the ISA machine, the program counter in the resulting ISA state is incremented using the `pcadd` function. Note that the top-level hypothesis of `isa-pc` assigns a type to the only free variable in `isa-pc`, `isa`. As seen from the top-level hypothesis, the `isa` is a record variable with three fields, `'pc`, which is an integer and corresponds to the program counter; `'rf`, which is an integer array and corresponds to the register file; `'imem`, which is an integer array and corresponds to the instruction memory.

The formula corresponding to `isa-pc` after step 6 of the translation mechanism is shown in

```
(defthm isa-pc
  (implies
   (and
    (integerp (g 'pc isa))
    (integer-arrayp (g 'rf isa))
    (integer-arrayp (g 'imem isa)))
   (equal
    (g 'pc (step-isa isa))
    (pcadd (g 'pc isa))))
  :hints (("Goal"
           :clause-processor
           (smt-clause-processor clause nil state))))
```

Figure 47: Command to the ACL2 theorem prover to check a simple property about the ISA machine model. The property states that the program counter is incremented after every step of the ISA machine. We call this property isa-pc.

```
(if (equal
     (g 'pc
        ((lambda (pc rf imem)
           ((lambda (inst imem pc rf)
              ((lambda (arg1 arg2 pc inst rf imem)
                 ((lambda (result imem rf inst pc)
                    ((lambda (isa-new) isa-new)
                     (s 'pc (pcadd pc)
                        (s 'rf
                           (store (dest inst) result rf))
                           (s 'imem imem 'nil)))))
                  (alu arg1 arg2)
                  imem rf inst pc))
               (select (src1 inst) rf)
               (select (src2 inst) rf)
               pc inst rf imem))
            (select pc imem)
            imem pc rf))
         (g 'pc isa)
         (g 'rf isa)
         (g 'imem isa)))
     (pcadd (g 'pc isa)))
    nil t)
```

Figure 48: Expression corresponding to isa-pc obtained after step 6 the translation mechanism.

```
(nil
 (smt1_isa_pc)
 nil
 ((pcadd . 1)
  (src2 . 1)
  (dest . 1)
  (src1 . 1)
  (alu . 2))
 (smt1_isa_imem smt1_isa_rf)
 ((isa ('pc int)
       ('rf int-array)
       ('imem int-array))))
```

Figure 49: An initial snapshot of the environment.

Figure 48. Note that this formula is a negated version of `isa-pc`, and all the unconstrained functions and macros have been expanded out. The lambdas are a result of expanding the ACL2 `let` macros used to bind lexically scoped local variables.

The initial value of the environment is shown in Figure 49. The first element of the environment is empty denoted by `nil` as there are no free variables in the original formula that have a boolean type. The sixth element shows the free record variable `isa`, the fields of `isa`, and the types of each of these fields. Since the SMT-LIB language does not support records, the translation mechanism expands out these record variables by creating new variables for each of the fields of a record variable. For example, the `isa` variable has three fields and the variables `smt1_isa_pc`, `smt1_isa_rf`, and `smt1_isa_imem` have been introduced corresponding to the three fields of `isa`. While introducing a new variable, we use `smt1` as a precursor to name of that variable so as to avoid name clashes with already existing variable names. In order for this to work, we require the original input formula to not have any variable names starting with `smt1`. Note that these newly introduced variables are also contained in the environment.

The formula corresponding to isa-pc obtained after step 7 of the translation mechanism is shown in Figure 50. As can be seen from the figure, the formula is in the SMT-LIB language except for record variables, which have not been expanded out yet. An example is the `isa-new` record variable, which is assigned the value of the ISA state obtained after stepping the ISA machine. Also, the `let` expressions are annotated with type information. This information can be used in step 8 of the

```
(if_then_else
 (= (let int (pc smt1_isa_pc)
      (let int-array (rf smt1_isa_rf)
        (let int-array (imem smt1_isa_imem)
          (let int (inst (select imem pc))
            (let int (arg1 (select rf (src1 inst)))
              (let int (arg2 (select rf (src2 inst)))
                (let int (result (alu arg1 arg2))
                  (let (('pc int)
                        ('rf int-array)
                        ('imem int-array))
                    (isa-new (s 'pc
                                (pcadd pc)
                                  (s 'rf (store rf (dest inst) result)
                                     (s 'imem imem nil))))
                      smt1_isa-new_pc))))))))
    (pcadd smt1_isa_pc))
 false true)
```

Figure 50: Expression corresponding to isa-pc obtained after step 7 of the translation mechanism.

translation to expand record variables.

The formula obtained after step 8 of the translation mechanism is shown in Figure 51. The formula is now in the SMT-LIB language and can be checked using an SMT solver. Note that type declarations have also been included that declares a type for all the free variables, UFs, and UPs in the original formula. We checked the *isa-pc* property using ACL2 and using the SMT clause processor integrated with ACL2. Since the model is simple and the property checked is trivial, in both cases, the property was proved in zero seconds.

8.6 Related Work

In this section, we review related work on integrating automated verification engines such as decision procedures with deductive reasoning. We had previously developed a method for integrating the UCLID decision procedure with the ACL2 theorem proving system [61, 62], which was the first such effort. We called the combined system ACLU (ACL2+UCLID). The design choices used to develop the ACL2-SMT framework are to a large extent influenced by our experiences gained in the development and application of the ACLU system. The ACL2-SMT framework can be used to easily integrate ACL2 with SMT solvers that can check formulas in QF_AUfLia. Note that the

```
(benchmark acl2_smt
:extrafuns ((smt1_isa_pc Int))
:extrafuns ((smt1_isa_imem Array))
:extrafuns ((smt1_isa_rf Array))
:extrafuns ((pcadd Int Int))
:extrafuns ((src2 Int Int))
:extrafuns ((dest Int Int))
:extrafuns ((src1 Int Int))
:extrafuns ((alu Int Int Int))
:formula
 (if_then_else
  (= (let (pc smt1_isa_pc)
       (let (rf smt1_isa_rf)
         (let (imem smt1_isa_imem)
           (let (inst (select imem pc))
             (let (arg1 (select rf (src1 inst)))
               (let (arg2 (select rf (src2 inst)))
                 (let (result (alu arg1 arg2))
                   (let (smt1_isa-new_pc (pcadd pc))
                     (let (smt1_isa-new_rf
                            (store rf (dest inst) result)
                          (let (smt1_isa-new_imem imem)
                            smt1_isa-new_pc)))))))))))
     (pcadd smt1_isa_pc))
  false true))
```

Figure 51: Expression corresponding to isa-pc obtained after step 8 of the translation mechanism. This expression is given as input to the SMT solver.

number and efficiency of such solvers is increasing. In contrast, the ACLU system is limited to the use of only the UCLID decision procedure. Also, using the ACL2-SMT framework, we have been able to verify bit-level pipelined machines with much less expert user effort than was required when using the ACLU system (see Chapter 9 for details).

Another recent approach developed by Reeber and Hunt uses ACL2's clause processor mechanism to integrate a SAT-based decision procedure with ACL2 [83]. They identify a decidable subset of ACL2 formulas called Subclass of Unrollable List Formulas (SULFA). They have developed a decision procedure that translates ACL2 formulas in SULFA to a propositional problem in CNF format. The user has to make an explicit call to use the decision procedure to check a given ACL2 conjecture. If the conjecture is in SULFA and if they cannot prove its validity, a counter example is

returned. Note that the SULFA approach is targeted at specifying and checking properties about bit-level models in ACL2. In comparison, the ACL2-SMT framework is used to enable the automatic checking of properties about term-level models.

Several other automated verification engines have also been integrated with ACL2. Sawada and Reeber [92] have connected ACL2 with SixthSense [73], an automated commercial formal verification tool. Ray, Matthews, and Tuttle have connected ACL2 with the SMV model checker [81]. Model checkers and SAT solvers have been connected with the PVS theorem prover [80, 93]. There are several other efforts in integrating external tools with theorem provers such as Isabelle [74], HOL4 [28], *etc*.

8.7 Conclusion

We have developed a framework for integrating term-level SMT solvers with ACL2 theorem-proving system in order to establish the correctness of programming systems that are beyond the scope of SMT solvers and that would require heroic human effort to verify using just ACL2. The integration required the design of transformations between terms in two different systems. The ACL2 language is general purpose, while QF_AUfLia formulas described in the SMT-LIB language are restricted and decidable. However, by virtue of its restrictions, specialized decision procedures can be exploited to provide extreme speedups, which can be several orders of magnitude. By integrating ACL2 with SMT solvers, we have the best of both worlds: our ACL2-SMT system allows us to work in a general-purpose setting, yet using the SMT clause processor, we can obtain the efficiency of SMT solvers. We have also developed a concrete system that integrates the Yices SMT solver with ACL2 using the SMT clause processor framework. Also, we note that very little effort is required to use this framework to integrate other SMT solvers with ACL2.

CHAPTER IX

BIT-LEVEL VERIFICATION

We now describe our verification approach for checking that pipelined machines defined at the bit-level work correctly. Our approach is based on using a combination of deductive reasoning and decision procedures. As we stated earlier, approaches that use theorem provers such as ACL2 [43, 42] can be used to verify bit-level pipelined machine models but require significant expert user effort.

Approaches based on decision procedures are highly automated and efficient. When verifying term-level models, we found that the UCLID decision procedure is orders of magnitude faster than ACL2. For example, the verification of a simple five-stage DLX pipelined machine defined at the term-level took three seconds with UCLID, but took fifteen and a half *days* with ACL2 [56]. But, the application of term-level decision procedures is restricted to the verification of term-level models, models that abstract away the datapath, implement a small subset of the instruction set, require the use of numerous abstractions, and are far from executable.

Our main contribution is to show how to attain a high degree of automation when verifying pipelined machines defined at the bit-level. The high-level idea of the approach is that deductive reasoning, using the ACL2 theorem proving system, is used to reduce the correctness theorem for an executable, bit-level pipelined machine to a theorem about a term-level model, which can then be automatically discharged using decision procedures. Note that all of the refinement-based methods and techniques described in Chapters 3, 4, 5, 6, and 7 for the verification of term-level pipelined machines in a highly automated, efficient, and scalable manner can be exploited in this verification approach when reasoning about the pipeline at the term-level. We demonstrate our approach using ACL2-SMT (described in Chapter 8), an extension of the ACL2 theorem proving system obtained by integrating an SMT solver such as the Yices decision procedures with the ACL2 and using the combined system to verify a complex seven-stage processor model defined mostly at the bit-level.

The significant gap between term-level models and executable, bit- and cycle- accurate models

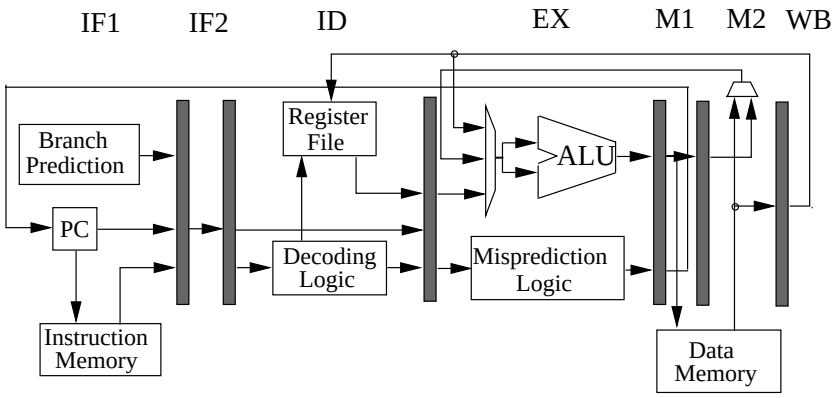


Figure 52: High-level organization of bit-level interface processor model

is one of the main problems in the area of pipelined machine verification. One way to view our approach is that it bridges this gap by verifying all the abstractions used in term-level models.

The rest of the chapter is organized as follows. In Section 9.1, we describe the seven-stage pipelined machine model, most of which is defined at the bit-level. Section 9.2 describes in detail the proof methodology, which is at the crux of our verification approach. Section 9.3 gives the verification statistics of the proof in terms of the running time and expert user effort required. We describe related work in Section 9.4 and conclude in Section 9.5.

9.1 Processor Model

We demonstrate our verification approach using a complex executable seven-stage pipelined machine model, most of which is defined at the bit-level. The high-level organization of the pipeline is inspired by the Intel XScale architecture [23] and is shown in Figure 52. The model has seven pipeline stages including a 2-cycle fetch, a decode, an execute, a 2-cycle memory access, and a write back. The model has various features such as branch prediction, precise exceptions, and predicated instruction execution.

The model is described at the bit-level except for the instruction and data memories, the register file, and combinational circuit blocks such as the ALU; these blocks have bit-level interfaces *i.e.*, their inputs and outputs are bit-vectors, but they are not necessarily defined at the bit-level internally. For example, the ALU in our machine takes bit-vector inputs, converts the inputs to integers, performs the appropriate ALU operation on these integers, and converts the result to a bit-vector,

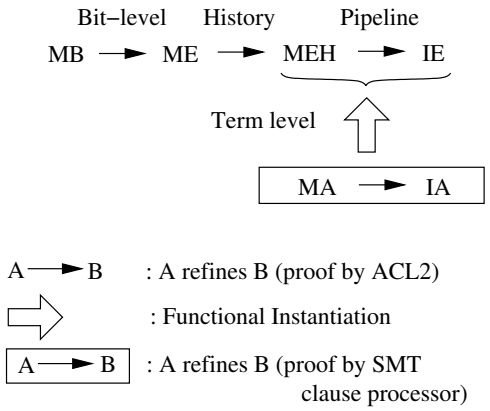


Figure 53: Proof outline that uses ACL2 and UCLID to show that MB refines IE.

which is the output of the ALU unit. Therefore, we use the term "bit-level interface" to describe the model. In the bit-level interface model the instruction decoder, control logic, and data path logic operate on bit-vectors. The model is described using the ACL2 programming language, and unlike term-level models it is executable. Instructions are 32 bits in length and the model has 16 registers. The size of the data path is a parameter that can be set to any integer value greater than one, and the verification times of our correctness proofs do not vary with the size of the data path.

The model implements 16 types of ALU instructions, a return from exception instruction, and various branch, jump, load, and store instructions. Our model has both register-register and register-immediate addressing modes and our model supports predicated instruction execution, *i.e.*, some instructions have an associated condition that depends on the processor status flags. The instructions are allowed to complete and update the programmer visible components such as the program counter, the data memory, and the register file only if the condition associated with the instruction is true. Each of the ALU, branch, load, and store instructions can be executed using 16 different conditions. ALU, load, and store instructions can also use immediate values. In all, the model implements 593 instructions.

9.2 Proof Methodology

The proof methodology which is at the crux of our verification approach for checking that bit-level pipelined machines work correctly is shown in Figure 53. The goal is to prove that the bit-level interface pipelined machine model MB refines its instruction set architecture IE. If we were to attempt this proof directly using a theorem prover, it would require an extraordinary amount of expert

user effort. Instead, we define a series of transformations, that converts MB to IE. We prove each of these transformations are correct using WEB-refinement. Since refinement is a compositional notion, we chain each of these individual refinement proofs together to get that MB refines IE.

Why are these transformations any better than carrying out the proof directly using a theorem prover? As stated earlier, the idea is to use a theorem prover to reduce the bit-level verification problem to a term-level problem, which can then be automatically solved using a term-level decision procedure. The transformations are aimed at transforming the bit-level model to a term-level model, and then a decision procedure is used to reason about the pipeline at the term-level.

The first transformation is used to move from bit-vectors to integers. We do this by proving (with ACL2) that MB refines ME, an executable pipelined machine that is similar to MB, but which operates on integers, not bit-vectors. The second transformation augments ME with history variables, which are required to define many refinement maps that relate pipelined machine states to ISA states, such as those based on commitment. The resulting executable model is MEH, which is very similar to ME, except that it has additional state elements that record history information.

The pipeline is dealt with next, when MEH is shown to refine IE. Both ACL2 and the Yices SMT solver that is integrated with ACL2 are used for this refinement proof. The proof that MEH refines IE cannot be directly handled with a term-level decision procedure, *e.g.*, the models use arithmetic operations on integers that are not expressible in QF_AUfLia. Therefore, several abstractions are employed, resulting in machines MA and IA which abstract MEP and IEP, respectively. MA and IA are term-level models and we prove that MA refines IA by using the Yices SMT solver which we have integrated with ACL2.

We now describe in detail aspects of the refinement proof. Note that the various models are very large and given the limited space, it is not possible to fully describe these models.

9.2.1 Reasoning about Bit-Level Interface Designs

In this section we describe the first transformation that converts the bit-level pipelined machine model MB to ME, a pipelined machine operating on integers. The transformation is proved correct using refinement. The refinement proof that relates MB and ME is carried out exclusively using ACL2 and is parameterized with respect to the word size, *i.e.*, our proof remains the same regardless

of the word size of the machines involved. Since MB and ME do not stutter with respect to each other, we prove that the two systems are bisimilar.

The refinement map from MB to ME converts unsigned and signed bit-vectors in MB to naturals and integers, respectively. For the proof, we developed a bit-vector library in ACL2. For example, we defined and developed a theory of rules for functions to convert bit-vectors to numbers and vice-versa. The functions include *n-ubv* (which converts naturals to unsigned bit-vectors), *ubv-n* (which converts unsigned bit-vectors to naturals), *i-sbv* (which converts integers to signed bit-vectors), and *sbv-i* (which converts signed bit-vectors to integers). The library required about four days for an expert ACL2 user to develop. For the refinement proof, we required theorems such as the following.

1. $natp(a) \wedge natp(n) \wedge len(\text{n-ubv}(a)) \leq n$

   $\Rightarrow \quad \text{ubv-n}(\text{extend-n}(\text{n-ubv}(a), n)) = a$

2. $integerp(a) \wedge natp(n) \wedge len(\text{i-sbv}(a)) \leq n$

   $\Rightarrow \quad \text{sbv-i}(\text{sign-extend-n}(\text{i-sbv}(a), n)) = a$

3. $bvp(x) \wedge natp(a) \wedge (a < len(x)) \quad \Rightarrow \quad bitp(nth(a, x))$

4. $bvp(a) \wedge bvp(b) \wedge (len(a) = len(b))$

   $\Rightarrow \quad (\text{ubv-n}(a) = \text{ubv-n}(b)) \Leftrightarrow (a = b)$

5. $bvp(a) \wedge bvp(b) \wedge (len(a) = len(b))$

   $\Rightarrow \quad (\text{sbv-i}(a) = \text{sbv-i}(b)) \Leftrightarrow (a = b)$

In the above theorems, $len(x)$ is the length of the bit-vector x, $natp(a)$ denotes that a is a natural number, $integerp(a)$ denotes that a is an integer, $bvp(a)$ denotes that a is a bit-vector, $bitp(a)$ denotes that a is a bit, $nth(n, x)$ corresponds to the n^{th} element of list x, $\text{extend-n}(b, n)$ extends the unsigned bit-vector b to a length of n, and $\text{sign-extend-n}(b, n)$ sign extends the signed bit-vector b to a length of n. Theorems 1, 2, 4, and 5 are used to reason about the refinement map and Theorem 3 is useful for reasoning about the instruction decoder, which generates control signals from the bit-vector corresponding to instructions. The theorems described above were essential for our proof, but our proofs required many other theorems all of which are included in our bit-vector library. Also, the bit-vector library was developed based on what was required for the refinement proof, and can be

easily extended with more bit-vector operations and rules.

9.2.2 Augmenting Executable Models with History Information

The second transformation augments the executable pipelined machine model ME with additional state that records history information, resulting in the machine model MEH. Remember that the goal is to move towards a term-level pipelined machine model that can then be used to reason about the pipeline. Proving that the pipeline works correctly requires defining refinement maps that relate pipelined machine states to ISA states. Many such efficient refinement maps require the use of history information. A class of such refinement maps are those that are based on commitment, which we use to prove the correctness of the 7 stage pipeline.

Using the commitment refinement map, a pipelined machine state is related to an instruction set architecture state by invalidating all the partially executed instructions in the pipeline and rolling back the programmer-visible components so that they correspond with the last committed instruction. While it is hard to roll back the pipelined machine, one can determine the values of the programmer visible components corresponding to the last committed instruction using history information.

The history information is recorded using history variables, which are state elements that record previous values of the state components of a pipelined machine. These history variables are used to only to define the refinement map and are not to interfere with the normal operation of the pipelined machine. However, there is no guarantee that this is the case with MEH. For example, it is quite possible that when instrumenting ME with history variables, the user allowed the value of a history variable to be used in updating a state component of MEH. Therefore, proving that MEH is correct does not imply the correctness of ME, and so we have to show that ME refines MEH.

Since MEH has additional state components, it is easier to define a refinement map from MEH to ME, and prove that MEH refines ME. As ME and MEH do not stutter with respect to each other, proving that MEH refines ME guarantees that the two systems are bisimilar and also that ME refines MEH. We prove that MEH refines ME by defining a refinement that relates MEH states to ME states. The refinement map directly maps the state components of MEH to ME excluding history variables, which are ignored. The proof can be easily carried out using ACL2.

9.2.3 Relating Executable Models and Term-Level Models

In this section, we give an overview of the proof that MEH refines IE. This refinement step deals with the pipeline and uses the Yices SMT solver. However, in order to use Yices, we have to show a relationship between executable machines and term-level machines. The difficulty is in mechanically verifying the various abstractions employed, which are used to deal with memories, branch prediction, instruction classes, etc. Below we describe two abstraction techniques that are very hard to mechanically verify. Both these abstraction techniques– one for memories and the other for branch predictors– are widely used in term-level modeling.

Memories at the term-level can be modeled using array semantics, which can be expressed in QF_AUfLia. However, in cases where reads and writes are in order—*e.g.*, this is the case for the data memory of our machine—memory can be modeled as an integer variable using two UFs, one to read and one to write. This modeling style leads to faster verification times than the approach using array semantics [49]. However, it is much more difficult to use if the abstraction has to be mechanically verified. To mechanically verify this abstraction, we have to encode the memory state as an integer and define the read and write operations for this encoding of the memory, in order to obtain our executable model. This is possible using Gödel encoding scheme, as shown below.

$$((a_1 \ . \ d_1) \ (a_2 \ . \ d_2) \ \ldots \ (a_n \ . \ d_n)) \ \rightarrow \ p_1^{p_3^{a_1+1} p_4^{a_2+1} \cdots p_{n+2}^{a_n+1}} \ p_2^{p_3^{d_1+1} p_4^{d_2+1} \cdots p_{n+2}^{d_n+1}}$$

In the above equation, the data memory is an alist whose address elements are $a_1, a_2, \ldots, a_n$ and whose data elements are $d_1, d_2, \ldots, d_n$. The i^{th} prime is denoted p_i. Any finite memory can now be represented as a single integer, but there are several problems with the above approach. For example, the theorem proving effort required to show that this scheme works is non-trivial, *e.g.*, it requires that we prove the prime decomposition theorem. In addition, the above encoding scheme cannot be used for infinite memories, as there is no bijection between the set of infinite memories and the natural numbers. Therefore, we find that the time savings attained by abstracting the data memory with an integer are not worth the added theorem proving effort required to justify this abstraction.

Branch predictors at the term-level can also be modeled using an integer variable that represents

the state of the branch predictor and three UFs that take the branch predictor state as input and return the next state of the branch predictor, a prediction for the branch direction, and a prediction for the branch target [49]. To show that the above correctly abstracts an executable implementation, for example a Branch Target Buffer (BTB), we are required to model the environment of the BTB using an integer. However, since the next state of the BTB depends on the entire processor state, we have to encode the state of the processor with one integer. We can do this using Gödel encoding schemes, as described above, provided the memories are finite, but the effort required would be considerable. Therefore, we use an alternate abstraction, where we simply model the branch predictor choices using non-determinism. Justifying this abstraction is straightforward, thus the ACL2 verification effort is drastically simplified. In addition, the UCLID verification times are comparable to the verification times required by the standard approach.

A final abstraction that we briefly mention concerns the instruction set. Term-level models only have one instruction per instruction class, whereas the executable models have the full instruction set. This turned out to be surprisingly easy to deal with because the term-level models abstract the instructions by using uninterpreted functions (UFs) that take the opcode as argument and collapse the instructions corresponding to various values of the opcode to one instruction. When we instantiate the term-level model, we replace the uninterpreted functions that take the opcode as input with functions that check the value of the opcode and perform the appropriate operation. For example, the term-level model only has one ALU operation, but the executable model first checks the opcode to determine whether it is an add or a subtract etc., and then performs the appropriate operation.

Executable models have other advantages. We can use them to debug designs more easily. For example, using Yices counterexamples one can determine the pipelined machine state that leads to a bug, but it might be more difficult to determine what the actual bug is. Using our executable models, we can execute the pipelined machine from the concrete state obtained from the counter example to track the bug in the design. Note that counterexamples generated by Yices correspond to counterexamples for the refinement relation between ACL2 models MA and IA.

Executable models also allow proofs of properties that might not be theorems in the abstract models. In addition, while the refinement proof established that the pipelined machine model (MB) behaves like the instruction set architecture model (IE), how do we know that the instruction set

architecture model (IE) is correct? Executable models allows us to run test programs. In our case, while executing a simple program, we found three bugs in the instruction set architecture model (IE), which are described below.

- Instructions are 32 bits with the least significant bit and the most significant bit corresponding to the 0^{th} bit and the 31^{st} bit of the instruction, respectively. The bug was in the functions that implement the instruction decoder, which were reading the 32-bit instruction in the reverse order. For example, if a decoder function was supposed to read the 5^{th} bit, it was instead reading the 26^{th} bit (31 - 5) of the instruction.

- The register file is updated by both ALU and load instructions. A decoder function that takes the instruction as input is used to determine if that instruction updates the register file. The function was buggy in that it did not signal that the register file should be updated if the input was a load instruction.

- The processor has 4 flags that are used to store various properties of the result obtained from the previous instruction. For example, if the previous instruction was an ALU instruction whose result was zero, then the Z flag is set. The *update_nzcv* function that takes the result and the previous value of the flags is used to update the processor flags. The *update_nzcv* was buggy in that the two input arguments to the function were swapped.

Since the decoder functions and the *update_nzcv* function are abstracted using uninterpreted functions (UFs) in the term-level models, none of the above bugs could have been caught during the verification of the term-level models using Yices.

The bugs described above bring up an important aspect of automatic term-level verification using decision procedures such as Yices. Recall that in order to use such methods, one must abstract away the ALU, the decoding logic, etc. using UFs shared by both the MA and ISA. While these abstractions drastically reduce the complexity of the verification problem, they also lead to ISA models that are structurally similar to MA models. MA models tend to have next state relations for each of the components of the MA machine and this way of specifying the MA model makes sense because they are inherently parallel machines whose every component is continuously updated. ISA

models defined at the term-level tend to have the same structure as their corresponding MA models. This is what allows them to share the same UFs as their MA models, but it also is what makes it easy to mask the kinds of errors reported above. Notice, however, that ISA models are inherently sequential and, conceptually, the simplest way to define them is to just have a big case statement that checks the type of the next instruction and executes code corresponding to the semantics of this instruction. If we define ISA models in this way, we have a much better chance of catching errors, as the semantic gap between the MA and ISA models is now larger. Using our approach, we can in fact define such an ISA machine and can prove that it is refined by IE.

9.2.4 Abstract Models

MA and IA are term-level models, and we are finally at the point where we can invoke the Yices SMT solver, which is optimized to automatically and efficiently reason about such models. MA, IA, and the refinement theorem that relates these models are processed using the SMT clause processor described in Chapter 8. The clause processor translates the formula expressing the refinement relation between MA and IA to a formula in QF_AUfLia, which is then checked using the Yices SMT solver. If the formula is found to be valid, then the formula that relates MA and IA is admitted as a theorem in ACL2. Otherwise, the counterexample generated by Yices is mapped back to ACL2 and can be utilized by the user to debug the models.

9.3 Verification Statistics

The verification times for the proofs and the expert user effort required in terms of man-weeks for each intermediate step in the proof methodology is shown in Table 8. In the "Proof Step" column in the table, A $\rightarrow$ B means that system A refines system B. For all the proof steps, except MA $\rightarrow$ IA, we used the ACL2 theorem proving system (version 3.2). For MA $\rightarrow$ IA, we used the integration of the Yices decision procedure (version 1.0.9) with ACL2. All the experiments were run on a 1.2 GHz Intel Pentium 3 machine, with a cache size of 512 KB. The user effort required for the proof steps is an estimate of the effort that would be required for an expert user of an SMT solver such as Yices and the ACL2 theorem proving system to apply this verification approach to verify another pipelined machine design of similar complexity. The times reported above do not include the time required to learn ACL2 and do not include the time required for the integration, which took several

Proof Step	Proof Time (secs)	User Effort (man-days)
MB → ME	30.40	7
ME → MEH	25.18	1
MA → IA	5.74	7
MEH → IE	3.30	10

Table 8: Verification times and expert user effort required for the refinement proofs.

months.

As can be seen from the table, the correctness proof for the XScale inspired processor model required about 25 days of expert user effort. Based on our experiences in using UCLID, we would have required about 30 days of user effort to abstract and verify the processor model at the term-level using UCLID. This amount of user effort is required for two reasons. First, the limitations of the UCLID specification language makes it difficult to model the pipelined machines and state correctness theorems. Second, since term-level model are not executable, debugging becomes much harder. Automation is really a measure of the amount of user effort required [54]. Therefore, we believe that our verification approach is highly automated as it requires only about 0.9 the amount of effort required to verify the processor model using UCLID.

9.4 Related Work

We now describe previous work on the verification of bit-level designs of microprocessor models. We had previously developed a refinement-based proof methodology for checking the correctness of bit-level pipelined machines that was based on using a combination of the UCLID decision procedure with the ACL2 theorem proving system [61, 62]. The refinement-based proof methodology described in this chapter is a more efficient and simplified version of this previous approach. The current approach required less than a month of expert user effort, whereas the previous approach required close to four months of user effort.

An early, pioneering body of work on the use of theorem proving for the verification of microprocessors is the CLI stack work [33, 34, 13]. Another early approach by Srivas and Bickford was based on the use of skewed abstraction functions [96]. A notable use of theorem proving in the context of hardware verification used ACL2 to reason about Motorola's CAP digital signal processor [16]. Sawada and Hunt have used the ACL2 theorem proving system to verify the FM9801

Microarchitecture. Their work is based on computing an intermediate abstraction of the pipelined machine state called MAETT that keeps track of completed and in-flight instructions. Using the MAETT abstraction, they check that each of the instructions in the pipeline executes correctly. Hosabettu et al., [30, 31] use the PVS theorem prover to verify pipelined processors. Their work is based on the use of completion functions that specifies the effect of completing an instruction in the pipeline on the programmer visible components. The abstraction function is computed by using a composition of completion functions, one for every partially executed instruction in the pipeline. Arons and Pnueli [9] have also used the PVS theorem prover to verify a machine with speculative instruction execution. They use an inductive proof to show that machines which differ only in the size of the retirement buffer are related; however, due to the complexity of the refinement maps involved, they conclude that a direct approach is far simpler than the inductive one. In [48], data consistency and liveness of pipelined machine models is verified using the PVS theorem prover. The models are synthesizable and are described very close to the gate-level.

There is also recent progress in the development of bit-level decision procedures that have been used to directly verify pipelined machines defined at the bit-level. One such approach is based on the Bit-level Analysis Tool (BAT) [67, 66], a decision procedure for solving quantifier-free formulas over the extensional theory of fixed-size bit-vectors and fixed-size bit-vector arrays (memories). BAT has been used to verify a 32-bit 5 stage pipelined machine in approximately 2 minutes [65]. Key features of BAT that enable the verification of complex systems such as pipelined machines are a fully automatic and efficient algorithm for abstracting bit-level memories [65] and a novel method for generating CNF (Conjunctive Normal Form) from a high-level circuit representation [68].

Another approach for verifying bit-level designs is based on Counter Example Guided Abstraction and Refinement (CEGAR) is introduced in [8] and implemented in the Reveal system. This approach automatically abstracts Verilog models and properties to the logic of Counter arithmetic, with restricted Lambda expressions and Uninterpreted functions (CLU) logic, which can then be checked efficiently using a decision procedure. If a spurious counter example is generated, the abstract model is refined using minimal unsatisfiable subset (MUS) extraction. The Reveal system has been used to check parts of the safety property based on the Burch and Dill commuting diagram for a 4-bit pipelined machine [22]. In contrast, while our approach is not fully automatic, it can be

used to handle much more complex pipelined machines. Also our verification times are independent of the data path width, whereas the verification times when using bit-level decision procedures increases as the size of the data path width increases.

9.5 Conclusions

We have shown how to verify executable pipelined machine models with bit-level interfaces using our integration of SMT solvers with the ACL2 theorem proving system. This has allowed us to overcome the major limitation of approaches based on decision procedures, namely that they only work for abstract term-level models and do not provide a firm connection with RTL models. Theorem proving approaches can reason about RTL-level designs, but tend to require heroic human effort. With our approach, the proof required only minutes of CPU time and the human theorem proving effort required was modest. Our proofs are based on WEB-refinement, a theory of refinement that is compositional and preserves both safety and liveness properties.

CHAPTER X

CONCLUSIONS

In this dissertation, we have developed a verification approach based on refinement and a combination of deductive reasoning and decision procedures for checking the correctness of bit-level pipelined machines. We showed that using our approach, pipelined machines can be verified in a highly automated, efficient, and scalable manner. The high-level idea of the approach is as follows. A deductive reasoning engine such as the ACL2 theorem proving system is used to reduce the bit-level pipelined machine verification problem to a term-level problem. We have developed several methods to automatically reason about pipelines at the term-level.

Automation was achieved by reducing the correctness criterion to a statement expressible in a decidable fragment of first-order logic that can be handled using existing decision procedures. We also provided recipes to define refinement maps and rank functions using only high-level information about the design. Checking liveness automatically was thought to be expensive, but using our approach liveness could be proved for complex pipelined machines with an overhead cost of only about 25%.

Using empirical evaluation, we showed that refinement maps used for correctness proofs can have a drastic impact on verification times. We developed several automatic and efficient methods to verify term-level pipelined machines using variations of flushing and commitment, two well-known methods for defining refinement maps. We also developed a method for combining flushing and commitment to obtain several orders of magnitude improvement in verification times over previous approaches. All of these methods can be used in our verification approach to reason about the pipeline at the term-level. We have also used extensive experiments to evaluate and compare these methods with previous approaches. For our experiments, we used a large number of pipelined machine models that have anywhere between 6 and 16 stages and also incorporate features such as branch prediction, precise exceptions, interrupts, instruction queues, instruction cache, data cache, and write buffers.

Even with the use of these efficient methods, term-level verification problems are often beyond the complexity threshold of decision procedures. To overcome this limitation, we developed a complete compositional reasoning framework based on refinement that allows us to break up correctness proofs into smaller pieces. We demonstrated that using this framework, term-level pipelined machines can be verified in a highly efficient and scalable manner. Another important advantage of our framework is that counterexamples generated are simpler, making it easier to debug the design.

The proof obligations generated by our verification approach are discharged using the ACL2-SMT system, which we developed by integrating a decision procedure with the ACL2 theorem proving system. We demonstrated the effectiveness of our verification approach by using it to check the correctness of an Intel XScale inspired processor model, most of which is defined at the bit-level. The model implements 593 instructions and has features such as predicated instruction execution, precise exceptions, and branch prediction. We were able to verify the model with less than a month of expert user effort and only a few minutes of CPU time. Using previous approaches based on deductive reasoning would have required an extraordinary amount of expert user effort to construct a correctness proof for the processor model. Using our approach, we could verify the processor model with only about 0.9 the amount of effort required to abstract and verify RTL models using UCLID.

10.1 Future Work

We see a number of directions for future work, and we now outline some of these possibilities.

The techniques developed in this thesis have been evaluated using academic models inspired by commercial designs. The next goal we plan to pursue is to apply these methods to verify a microprocessor core that has commercial applications. In fact, we plan to verify the Plasma CPU [78], an Intellectual Property (IP) core that has been used in academic and commercial projects. The RTL design of the Plasma CPU is described in VHDL—an industry standard hardware description language—and can be downloaded from the Opencores webpage [76]. Most of the MIPS I user mode instructions are supported, which include multiply and divide instructions. Precise exceptions and interrupts are implemented. A memory controller is also incorporated that is used to interface with a unified memory that stores both instructions and data.

The Plasma CPU design brings up several verification challenges. One such example is the memory model. In the academic models we verified, the memory load and store operations are assumed to complete in one cycle. Whereas, memory load and store operations in the Plasma CPU can take an arbitrary number of cycles to complete. Another challenge is that the decoder unit is much more complex in comparison to the models we have verified. Dealing with this complexity might require the use of clever abstractions. Also, multiply and divide instructions multicycle and can take upto 32 cycles to complete.

Another direction of future work is the verification of Cache Coherence Protocols (CCPs). Previous works on verifying CCPs are targeted towards checking that high-level models of these protocols are correct [27]. These high-level models are very abstract and hide many of the implementation details. For example, the implementations of these protocols are heavily pipelined and it is beyond the scope of automated tools to apply state-of-the-art verification techniques for CCPs to these low-level implementations. High-level CCP models can be thought of as ISAs and low-level CCP models can be thought of as pipelined machines. Therefore, an interesting approach for verifying CCP designs would be to prove that the low-level implementation refines its high-level model by extending the verification approach for bit-level pipelined machines described in this thesis. The high-level CCP model can then be verified using existing methods.

Another aspect of CCPs designs is that many of the pipelines have simple behaviors, but, can have a large number of stages. Such pipelines also appear in a number of other hardware designs. This brings up the following question. Can we fully automate the verification of bit-level pipelines with a limited set of behaviors? One possible approach would be to use compositional reasoning based on refinement, which we have described in Chapter 7 to be both efficient and scalable, but, requires some user effort to define the intermediate models, and the refinement maps and rank functions that relate these models. But, if the pipeline has only a limited number of behaviors, it might be possible to automatically construct a compositional proof by developing methods that can analyze the design and determine the various components required for the compositional proof.

The approach described above can be made applicable to RTL designs by leveraging recent

progress in decision procedures for bit-level reasoning, an example of which is the Bit-level Analysis Tool (BAT) [67]. BAT implements a state-of-the-art decision procedure for solving quantifier-free formulas over the extensional theory of fixed-size bit-vectors and fixed-size bit-vector arrays (memories). Preliminary results [64] show that using refinement-based compositional reasoning with bit-level decision procedures such as BAT can be used to reason about complex pipelined machines in a highly-automated and efficient manner.

Our focus in this dissertation has been the verification of in-order pipelines. But, state-of-the-art microprocessor pipelines have many more complex behaviors such as out-of-order, superscalar, and VLIW execution, pipelines that implement variable latency instructions, *etc*. We plan to explore and develop refinement-based methods for designs with more complex pipeline behaviors. Note that some of these behaviors such as our-of-order execution of instructions can be verified using WEB refinement as the instructions complete in order. Whereas, other behaviors such as superscalar execution, where the implementation can complete multiple instructions in a single cycle will require extensions to the theory of refinement that we currently use.

REFERENCES

[1] *2005 International Conference on Computer-Aided Design (ICCAD'05), November 6-10, 2005, San Jose, CA, USA*, IEEE Computer Society, 2005.

[2] *Formal Methods in Computer-Aided Design, 6th International Confrence, FMCAD 2006, San Jose, California, USA, November 12-16, 2006, Proceedings*, IEEE Computer Society, 2006.

[3] AAGAARD, M., COOK, B., DAY, N. A., and JONES, R. B., "A framework for microprocessor correctness statements.," in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)* (MARGARIA, T. and MELHAM, T. F., eds.), vol. 2144 of *Lecture Notes in Computer Science*, (Livingston, Scotland), pp. 433–448, Springer, 2001.

[4] AAGAARD, M., COOK, B., DAY, N. A., and JONES, R. B., "A framework for superscalar microprocessor correctness statements.," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 3, pp. 298–312, 2003.

[5] AAGAARD, M., DAY, N. A., and JONES, R. B., "Synchronization-at-retirement for pipeline verification.," in Hu and Martin [32], pp. 113–127.

[6] ABADI, M. and LAMPORT, L., "The existence of refinement mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, 1991.

[7] ABADIR, M. S., ALBIN, K., HAVLICEK, J., KRISHNAMURTHY, N., and MARTIN, A. K., "Formal verification successes at Motorola.," *Formal Methods in System Design*, vol. 22, no. 2, pp. 117–123, 2003.

[8] ANDRAUS, Z. S., LIFFITON, M. H., and SAKALLAH, K. A., "Refinement strategies for verification methods based on datapath abstraction.," in *Asia South Pacific Design Automation (ASP-DAC'06)* (HIROSE, F., ed.), pp. 19–24, IEEE, 2006.

[9] ARONS, T. and PNUELI, A., "A comparison of two verification methods for speculative instruction execution," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, vol. 1785 of *Lecture Notes in Computer Science*, pp. 487–502, Springer-Verlag, March 2000.

[10] "Barcelogic for SMT," 2007. See URL `http://www.lsi.upc.edu/ oliveras/-bclt-main.html`.

[11] BENTLEY, B., "Validating the Intel Pentium 4 microprocessor," in *38th Design Automation Conference*, pp. 253–255, 2001.

[12] BENTLEY, B., "Validating a modern microprocessor," 2005. See URL `http://-www.cav2005.inf.ed.ac.uk/bentley_CAV_07_08_2005.ppt`.

[13] BEVIER, W. R., HUNT, JR., W. A., MOORE, J. S., and YOUNG, W. D., "An approach to systems verification," *Journal of Automated Reasoning*, vol. 5, pp. 411–428, December 1989.

[14] BOYER, R. S. and MOORE, J. S., "Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic," in *Machine intelligence 11*, pp. 83–124, Oxford University Press, Inc., 1988.

[15] BRINKSMA, E. and LARSEN, K. G., eds., *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, vol. 2404 of *Lecture Notes in Computer Science*, Springer, 2002.

[16] BROCK, B. and HUNT, JR., W. A., "Formally specifying and mechanically verifying programs for the Motorola complex arithmetic processor DSP," in *1997 IEEE International Conference on Computer Design*, pp. 31–36, IEEE Computer Society, Oct. 1997.

[17] BROCK, B., KAUFMANN, M., and MOORE, J. S., "ACL2 theorems about commercial microprocessors," in *Formal Methods in Computer-Aided Design (FMCAD'96)* (SRIVAS, M. and CAMILLERI, A., eds.), pp. 275–293, Springer-Verlag, 1996.

[18] BROWNE, M., CLARKE, E. M., and GRUMBERG, O., "Characterizing finite Kripke structures in propositional temporal logic," *Theoretical Computer Science*, vol. 59, 1988.

[19] BRYANT, R. E., GERMAN, S., and VELEV, M. N., "Exploiting positive equality in a logic of equality with uninterpreted functions," in *Computer-Aided Verification–CAV '99* (HALBWACHS, N. and PELED, D., eds.), vol. 1633 of *LNCS*, pp. 470–482, Springer-Verlag, 1999.

[20] BRYANT, R. E., LAHIRI, S. K., and SESHIA, S., "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions," in *Computer-Aided Verification–CAV 2002* (BRINKSMA, E. and LARSEN, K., eds.), vol. 2404 of *LNCS*, pp. 78–92, Springer-Verlag, 2002.

[21] BURCH, J. R., "Techniques for verifying superscalar microprocessors," in *Design Automation Conference (DAC '96)*, (Las Vegas, Nevada), pp. 552–557, ACM Press, June 1996.

[22] BURCH, J. R. and DILL, D. L., "Automatic verification of pipelined microprocessor control," in *Computer-Aided Verification (CAV '94)*, vol. 818 of *LNCS*, pp. 68–80, Springer-Verlag, 1994.

[23] CLARK, L., HOFFMAN, E., MILLER, J., BIYANI, M., LIAO, Y., STRAZDUS, S., M.MORROW, VELARDE, K., and YARCH, M., "An embedded 32-bit microprocessor core for low-power and high-performance applications," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 11, pp. 1599–1608, 2001.

[24] CLARKE, E. M., GRUMBERG, O., and PELED, D., *Model Checking*. MIT Press, 1999.

[25] COHN, A., "A proof of correctness of the VIPER microprocessor:the first level," technical report, University of Cambridge, Cambridge Laboratory, 1987.

[26] GANZINGER, H., HAGEN, G., NIEUWENHUIS, R., OLIVERAS, A., and TINELLI, C., "DPLL(T): Fast decision procedures," in *Computer Aided Verification–CAV'04* (ALUR, R. and PELED, D., eds.), vol. 3114 of *LNCS*, pp. 175–188, Springer, 2004.

[27] GERMAN, S. M., "Formal design of cache memory protocols in ibm.," *Formal Methods in System Design*, vol. 22, no. 2, pp. 133–141, 2003.

[28] GORDON, M. J. C., "Programming combinations of deduction and bdd-based symbolic calculation," *LMS Journal of Computation and Mathematics 5*, pp. 56–76, 2002.

[29] GREVE, D., RICHARDS, R., and WILDING, M., "A summary of intrinsic partitioning verification," in *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2004.

[30] HOSABETTU, R., SRIVAS, M., and GOPALAKRISHNAN, G., "Decomposing the proof of correctness of a pileplined microprocessors," in *Computer-Aided Verification – CAV '98* (HU, A. J. and VARDI, M. Y., eds.), vol. 1427 of *LNCS*, Springer-Verlag, 1998.

[31] HOSABETTU, R., SRIVAS, M., and GOPALAKRISHNAN, G., "Proof of correctness of a processor with reorder buffer using the completion functions approach," in *Computer-Aided Verification–CAV '99* (HALBWACHS, N. and PELED, D., eds.), vol. 1633 of *LNCS*, Springer-Verlag, 1999.

[32] HU, A. J. and MARTIN, A. K., eds., *Formal Methods in Computer-Aided Design, 5th International Confrence, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, vol. 3312 of *Lecture Notes in Computer Science*, Springer, 2004.

[33] HUNT, JR., W. A., "Microprocessor design verification," *Journal of Automated Reasoning*, vol. 5, no. 4, pp. 429–460, 1989.

[34] HUNT, JR., W. A., *FM8501: A Verified Microprocessor*, vol. 795. Springer-Verlag, 1994.

[35] "Intel Pentium 4 Processor - Product Overview," 2005. See URL `http://www.intel.-com/design/pentium4/prodbref/`.

[36] "International technology roadmap for semiconductors," 2004. See URL `http://-www.itrs.net/Links/2004Update/2004Update.htm`.

[37] JHA, S., LU, Y., MINEA, M., and CLARKE, E. M., "Equivalence checking using abstractbdds.," in *ICCD*, pp. 332–337, 1997.

[38] JHALA, R. and MCMILLAN, K. L., "Microarchitecture verification by compositional model checking.," in *International Conference on Computer Aided Verification (CAV'01)* (BERRY, G., COMON, H., and FINKEL, A., eds.), vol. 2102 of *Lecture Notes in Computer Science*, pp. 396–410, Springer, 2001.

[39] JONES, R., SKAKKEBÆK, J., and DILL, D., "Reducing manual abstraction in formal verification of out-of-order execution," in *Formal Methods in Computer-Aided Design (FMCAD)* (GOPALAKRISHNAN, G. and WINDLEY, P., eds.), vol. 1522 of *Lecture Notes in Computer Science*, pp. 2–17, Springer-Verlag, November 1998.

[40] JONES, R. B., SKAKKEBÆK, J. U., and DILL, D. L., "Formal verification of out-of-order execution with incremental flushing," *Formal Methods in System Design, Special Issue on Microprocessor Verification*, vol. 20, pp. 139–158, Mar. 2002.

[41] KANE, R., MANOLIOS, P., and SRINIVASAN, S. K., "Monolithic verification of deep pipelines with collapsed flushing.," in *Design, Automation and Test in Europe, (DATE'06)* (GIELEN, G. G. E., ed.), pp. 1234–1239, European Design and Automation Association, Leuven, Belgium, 2006.

[42] KAUFMANN, M., MANOLIOS, P., and MOORE, J. S., eds., *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[43] KAUFMANN, M., MANOLIOS, P., and MOORE, J. S., *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.

[44] KAUFMANN, M., MOORE, J. S., RAY, S., and REEBER, E., "Integrating external deduction tools with acl2," in *Proceedings of the 6th International Workshop on the Implementation of Logics (IWIL'06)*, pp. 7–26, 2006.

[45] KAUFMANN, M. and MOORE, J. S., "A precise description of the ACL2 logic," tech. rep., Department of Computer Sciences, University of Texas at Austin, 1997. See URL `http://www.cs.utexas.edu/users/moore/publications/acl2-papers.-html#Foundations`.

[46] KAUFMANN, M. and MOORE, J. S., eds., *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004)*, November 2004. See URL `http://-www.cs.utexas.edu/users/moore/acl2/workshop-2004/`.

[47] KAUFMANN, M. and MOORE, J. S., "ACL2 homepage.," July, 2007. See URL `http://-www.cs.utexas.edu/users/moore/acl2`.

[48] KRONING, D., *Formal Verification of Pipelined Microprocessors*. PhD thesis, Universität des Saarlandes, 2001.

[49] LAHIRI, S., SESHIA, S., and BRYANT, R., "Modeling and verification of out-of-order microprocessors using UCLID," in *Formal Methods in Computer-Aided Design (FMCAD'02)*, vol. 2517 of *LNCS*, pp. 142–159, Springer-Verlag, 2002.

[50] LUDDEN, J. M., ROESNER, W., HEILING, G. M., REYSA, J. R., JACKSON, J. R., CHU, B.-L., BEHM, M. L., BAUMGARTNER, J., PETERSON, R. D., ABDULHAFIZ, J., BUCY, W. E., KLAUS, J. H., KLEMA, D. J., LE, T. N., LEWIS, F. D., MILLING, P. E., MCCONVILLE, L. A., NELSON, B. S., PARUTHI, V., POUARZ, T. W., ROMONOSKY, A. D., STUECHELI, J., THOMPSON, K. D., VICTOR, D. W., and WILE, B., "Functional verification of the POWER4 microprocessor and POWER4 multiprocessor system.," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 53–76, 2002.

[51] MANOLIOS, P., "Correctness of pipelined machines," in *Formal Methods in Computer-Aided Design–FMCAD 2000* (HUNT, JR., W. A. and JOHNSON, S. D., eds.), vol. 1954 of *LNCS*, pp. 161–178, Springer-Verlag, 2000.

[52] MANOLIOS, P., *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, August 2001. See URL `http://www.cc.gatech.edu/~manolios/-publications.html`.

[53] MANOLIOS, P., "A compositional theory of refinement for branching time," in *12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003* (GEIST, D. and TRONCI, E., eds.), vol. 2860 of *LNCS*, pp. 304–318, Springer-Verlag, 2003.

[54] MANOLIOS, P., "The challenge of hardware-software co-verification," in *IFIP Working Conference on Verified Software: Theories, Tools, Experiments*, 2005.

[55] MANOLIOS, P., "Refinement and theorem proving," in *School on Formal Methods for the Design of Computer, Communication, and Software Systems: Hardware Verification*, Lecture Notes in Computer Science, Springer Verlag, 2006.

[56] MANOLIOS, P. and SRINIVASAN, S., "A suite of hard ACL2 theorems arising in refinement-based processor verification," in Kaufmann and Moore [46]. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/`.

[57] MANOLIOS, P. and SRINIVASAN, S. K., "Automatic verification of safety and liveness for xscale-like processor models using web refinements.," in *Design, Automation and Test in Europe (DATE'04)*, pp. 168–175, IEEE Computer Society, 2004.

[58] MANOLIOS, P. and SRINIVASAN, S. K., "A complete compositional reasoning framework for the efficient verification of pipelined machines.," in *International Conference on Computer-Aided Design (ICCAD'05)* [1], pp. 863–870.

[59] MANOLIOS, P. and SRINIVASAN, S. K., "A computationally ef∼cient method based on commitment re∼nement maps for verifying pipelined machines.," in *Formal Methods and Models for Co-Design (MEMOCODE'05)*, pp. 188–197, IEEE, 2005.

[60] MANOLIOS, P. and SRINIVASAN, S. K., "Refinement maps for efficient verification of processor models.," in *Design, Automation and Test in Europe (DATE'05)*, pp. 1304–1309, IEEE Computer Society, 2005.

[61] MANOLIOS, P. and SRINIVASAN, S. K., "Verification of executable pipelined machines with bit-level interfaces.," in *International Conference on Computer-Aided Design (ICCAD'05)* [1], pp. 855–862.

[62] MANOLIOS, P. and SRINIVASAN, S. K., "A framework for verifying bit-level pipelined machines based on automated deduction and decision procedures.," *J. Autom. Reasoning*, vol. 37, no. 1-2, pp. 93–116, 2006.

[63] MANOLIOS, P. and SRINIVASAN, S. K., "Automatic verification of safety and liveness for pipelined machines using web refinement.," *ACM Transactions on Design Automation of Electronic Systems*, 2007. Accepted to appear.

[64] MANOLIOS, P. and SRINIVASAN, S. K., "A refinement-based compositional reasoning framework for pipelined machine verification.," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2007. Accepted to appear.

[65] MANOLIOS, P., SRINIVASAN, S. K., and VROON, D., "Automatic memory reductions for rtl model verification.," in *International Conference on Computer-Aided Design (ICCAD'06)* (HASSOUN, S., ed.), pp. 786–793, ACM, 2006.

[66] MANOLIOS, P., SRINIVASAN, S. K., and VROON, D., "BAT: The Bit-level Analysis Tool," 2006. Available from `http://www.cc.gatech.edu/~manolios/bat/`.

[67] MANOLIOS, P., SRINIVASAN, S. K., and VROON, D., "BAT: The bit-level analysis tool," in *International Conference Computer Aided Verification (CAV'07)*, 2007.

[68] MANOLIOS, P. and VROON, D., "Efficient circuit to CNF conversion," in *International Conference on Theory and Applications of Satisfiability Testing*, 2007.

[69] MCMILLAN, K. L., "Verification of an implementation of Tomasulo's algorithm by compositional model checking," in *Computer Aided Verification (CAV '98)* (HU, A. J. and VARDI, M. Y., eds.), vol. 1427 of *LNCS*, pp. 110–121, Springer-Verlag, 1998.

[70] MCMILLAN, K. L., "A methodology for hardware verification using compositional model checking.," *Sci. Comput. Program.*, vol. 37, no. 1-3, pp. 279–309, 2000.

[71] MILNER, R., *Communication and Concurrency*. Prentice-Hall, 1990.

[72] MISHRA, P. and DUTT, N., "Modeling and verification of pipelined embedded processors in the presence of hazards and exceptions," in *IFIP WCC 2002 Stream 7 on Distributed and Parallel Embedded Systems (DIPES'02)*, 2002.

[73] MONY, H., BAUMGARTNER, J., PARUTHI, V., KANZELMAN, R., and KUEHLMANN, A., "Scalable automated verification via expert-system guided transformations.," in Hu and Martin [32], pp. 159–173.

[74] MÜLLER, O. and NIPKOW, T., "Combining model checking and deduction for i/o-automata.," in *Tools and Algorithms for Construction and Analysis of Systems (TACAS'95)* (BRINKSMA, E., CLEAVELAND, R., LARSEN, K. G., MARGARIA, T., and STEFFEN, B., eds.), vol. 1019 of *Lecture Notes in Computer Science*, pp. 1–16, Springer, 1995.

[75] NAMJOSHI, K. S., "A simple characterization of stuttering bisimulation," in *17th Conference on Foundations of Software Technology and Theoretical Computer Science*, vol. 1346 of *LNCS*, pp. 284–296, 1997.

[76] Opencores. July, 2007. See URL `http://www.opencores.org`.

[77] PATANKAR, V. A., JAIN, A., and BRYANT, R. E., "Formal verification of an ARM processor," in *Twelfth International Conference On VLSI Design*, pp. 282–287, 1999.

[78] Plasma - most MIPS I(TM) opcodes: Overview. July, 2007. See URL `http://-www.opencores.org/projects.cgi/web/mips/overview`.

[79] QIANG, Q., SAAB, D. G., and ABRAHAM, J. A., "Checking nested properties using bounded model checking and sequential atpg.," in *VLSI Design*, pp. 225–230, IEEE Computer Society, 2006.

[80] RAJAN, S., SHANKAR, N., and SRIVAS, M. K., "An integration of model checking with automated proof checking.," in *International Conference on Computer Aided Verification (CAV'95)* (WOLPER, P., ed.), vol. 939 of *Lecture Notes in Computer Science*, pp. 84–97, Springer, 1995.

[81] RAY, S., MATTHEWS, J., and TUTTLE, M., "Certifying compositional model checking algorithms in acl2.," in *4th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2'03)*, 2003.

[82] RAY, S. and JR., W. A. H., "Deductive verification of pipelined machines using first-order quantification.," in *International Conference on Computer Aided Verification (CAV'04)* (ALUR, R. and PELED, D., eds.), vol. 3114 of *Lecture Notes in Computer Science*, pp. 31–43, Springer, 2004.

[83] REEBER, E. and JR., W. A. H., "A sat-based decision procedure for the subclass of unrollable list formulas in acl2 (sulfa).," in *International Joint Conference on Automated Reasoning (IJCAR'06)* (FURBACH, U. and SHANKAR, N., eds.), vol. 4130 of *Lecture Notes in Computer Science*, pp. 453–467, Springer, 2006.

[84] ROESNER, W., "Ecological niche or survival gear? - improving an industrial simulation methodology with formal methods.," in *Formal Methods in Computer-Aided Design (FMCAD'06)* [2].

[85] RUSSINOFF, D. M., "A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions," *London Mathematical Society Journal of Computation and Mathematics*, vol. 1, pp. 148–200, December 1998.

[86] RUSSINOFF, D. M., "A mechanically checked proof of correctness of the AMD-K5 floating-point square root microcode," *Formal Methods in System Design*, vol. 14, pp. 75–125, 1999.

[87] RYAN, L., "Siege homepage.," July, 2007. See URL `http://www.cs.sfu.ca/~loryan/personal`.

[88] SAAB, D. G., ABRAHAM, J. A., and VEDULA, V. M., "Formal verification using bounded model checking: Sat versus sequential atpg engines.," in *VLSI Design*, pp. 243–248, IEEE Computer Society, 2003.

[89] SAWADA, J., *Formal Verification of an Advanced Pipelined Machine*. PhD thesis, University of Texas at Austin, Dec. 1999. See URL `http://www.cs.utexas.edu/users/sawada/-dissertation/`.

[90] SAWADA, J., "Verification of a simple pipelined machine model," in Kaufmann *et al.* [42], pp. 137–150.

[91] SAWADA, J., "Formal verification of divide and square root algorithms using series calculation," in *Proceedings of the ACL2 Workshop 2002* (KAUFMANN, M. and MOORE, J. S., eds.), 2002.

[92] SAWADA, J. and REEBER, E., "Acl2six: A hint used to integrate a theorem prover and an automated verification tool.," in *Formal Methods in Computer-Aided Design (FMCAD'06)* [2], pp. 161–170.

[93] SHANKAR, N., "Using decision procedures with a higher-order logic.," in *International Conference Theorem Proving in Higher Order Logics (TPHOLs'01)* (BOULTON, R. J. and JACKSON, P. B., eds.), vol. 2152 of *Lecture Notes in Computer Science*, pp. 5–26, Springer, 2001.

[94] SMITH, S., PEREZ, R., WEINGART, S., and AUSTEL, V., "Validating a high-performance, programmable secure coprocessor," in *22nd National Information Systems Security Conference*, Oct. 1999.

[95] SRINIVASAN, S. K. and VELEV, M. N., "Formal verification of an intel xscale processor model with scoreboarding, specialized execution pipelines, and impress data-memory exceptions.," in *Formal Methods and Models for Co-Design (MEMOCODE'03)*, pp. 65–74, IEEE Computer Society, 2003.

[96] SRIVAS, M. and BICKFORD, M., "Formal verification of a pipelined microprocessor," *IEEE Software*, pp. 52–64, Sept. 1990.

[97] VASUDEVAN, S., EMERSON, E. A., and ABRAHAM, J. A., "Efficient model checking of hardware using conditioned slicing.," *Electr. Notes Theor. Comput. Sci.*, vol. 128, no. 6, pp. 279–294, 2005.

[98] VELEV, M. N., "Using positive equality to prove liveness for pipelined microprocessors.," in *Asia and South Pacific Design Automation Conference (ASPDAC'04)* (IMAI, M., ed.), (Yokohama, Japan), pp. 316–321, IEEE, 2004.

[99] VELEV, M. N. and BRYANT, R. E., "Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction," in *Proceedings of the 37th conference on Design Automation*, pp. 112–117, ACM Press, 2000.

[100] "Yices homepage," 2007. See URL `http://fm.csl.sri.com/yices`.

PUBLICATIONS

Journal Articles

1. Panagiotis Manolios and Sudarshan K. Srinivasan. Automatic Verification of Safety and Liveness for Pipelined Machines Using WEB Refinement, 18 pages. *Accepted pending minor revisions, which we have made and submitted. ACM Transactions on Design Automation of Electronic Systems*.

2. Panagiotis Manolios and Sudarshan K. Srinivasan. A Framework for Verifying Bit-Level Pipelined Machines Based on Automated Deduction and Decision Procedures, 26 pages. *Journal of Automated Reasoning (accepted to appear)*.

3. Jun Cheol Park, Vincent Mooney and Sudarshan K. Srinivasan. Combining data remapping and voltage/frequency scaling of second level memory for energy reduction in embedded systems, 6 pages. *Microelectronics Journal*, 2003.

Submitted Journal Articles

4. Panagiotis Manolios and Sudarshan K. Srinivasan. A Refinement-Based Compositional Reasoning Framework for Pipelined Machine Verification, 13 pages. *Submitted to IEEE Transactions on VLSI Systems*.

Conference Papers

5. Panagiotis Manolios, Sudarshan K. Srinivasan, and Daron Vroon. Automatic Memory Reductions for RTL-Level Verification. *ACM-IEEE International Conference on Computer Aided Design (ICCAD)*, IEEE Computer Society, 2006. (24% acceptance rate, 130 out of 537; the acceptance rate for long papers, like ours, is lower)

6. Roma Kane, Panagiotis Manolios, and Sudarshan K. Srinivasan. Monolithic Verification of Deep Pipelines with Collapsed Flushing. *ACM-IEEE Design Automation and Test in Europe*

(DATE), IEEE Computer Society, pages 1234–1239, 2006. (27% acceptance rate, 233 out of 834; the acceptance rate for long papers, like ours, is lower)

7. Panagiotis Manolios and Sudarshan K. Srinivasan. Verification of Executable Pipelined Machines with Bit-Level Interfaces. *ACM-IEEE International Conference on Computer Aided Design (ICCAD)*, IEEE Computer Society, pages 855–862, 2005. (23% acceptance rate, 128 out of 540; the acceptance rate for long papers, like ours, is lower)

8. Panagiotis Manolios and Sudarshan K. Srinivasan. A Complete Compositional Reasoning Framework for the Efficient Verification of Pipelined Machines. *ACM-IEEE International Conference on Computer Aided Design (ICCAD)*, IEEE Computer Society, pages 863-870, 2005. (23% acceptance rate, 128 out of 540; the acceptance rate for long papers, like ours, is lower)

9. Panagiotis Manolios and Sudarshan K. Srinivasan. A Parameterized Benchmark Suite of Hard Pipelined-Machine-Verification Problems. *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 363-366, 2005.

10. Panagiotis Manolios and Sudarshan K. Srinivasan. A Computationally Efficient Method Based on Commitment Refinement Maps for Verifying Pipelined Machines. *ACM-IEEE Formal Methods and Models for Codesign (MEMOCODE)*, IEEE,pages 188–197, 2005. (36% acceptance rate; 17 out of 47 papers)

11. Panagiotis Manolios and Sudarshan K. Srinivasan. Refinement Maps for Efficient Verification of Processor Models. *ACM-IEEE Design Automation and Test in Europe (DATE)*, IEEE Computer Society, pages 1304-1309, 2005. (21% acceptance rate, 176 out of 825; the acceptance rate for long papers, like ours, is lower)

12. Panagiotis Manolios and Sudarshan K. Srinivasan. Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB-Refinements. *ACM-IEEE Design Automation and Test in Europe (DATE)*, pages 168-175, 2004. (17% acceptance rate for long papers)

13. Sudarshan K. Srinivasan, and Miroslav N. Velev. Formal Verification of an Intel XScale Processor Model with Scoreboarding, Specialized Execution Pipelines, and Imprecise Data-Memory Exceptions. *ACM-IEEE Formal Methods and Models for Codesign (MEMOCODE)*, IEEE, pages 65–74, 2003.

Refereed Workshop Papers

14. Panagiotis Manolios and Sudarshan K. Srinivasan. A Suite of Hard ACL2 Theorems Arising in Refinement-Based Processor Verification. Fifth *International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2)*, 2004.

15. Sudarshan K. Srinivasan, Jun Cheol Park, and Vincent Mooney.Combining Data Remapping and Voltage/Frequency Scaling of Second Level Memory for Energy Reduction in Embedded Systems. *Proceedings of the International Workshop on Embedded System Codesign (ESCODES)*, 2002.

Technical Reports

16. Panagiotis Manolios and Sudarshan K. Srinivasan. Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB-Refinements. CERCS TR GIT-CERCS-03-17, 2003.

VITA

Sudarshan Srinivasan is a Ph.D. candidate in the School of Electrical and Computer Engineering at the Georgia Institute of Technology. He received an M.S. in Electrical and Computer Engineering from the Georgia Institute of Technology in 2003 and a B.E. in Electrical and Electronics Engineering from the University of Madras in 2001. His research interests are in Formal Verification, Hardware Validation, Computer Architecture, and Computer-Aided Design of Digital Systems. His current research focus is in the development and application of Formal Verification methods to hardware systems.