Automatic Verification of Safety and Liveness for Pipelined Machines Using WEB Refinement

PANAGIOTIS MANOLIOS
Northeastern University
and
SUDARSHAN K. SRINIVASAN
North Dakota State University

We show how to automatically verify that complex pipelined machine models satisfy the same safety and liveness properties as their instruction-set architecture (ISA) models by using well-founded equivalence bisimulation (WEB) refinement. We show how to reduce WEB-refinement proof obligations to formulas expressible in the decidable logic of counter arithmetic with lambda expressions and uninterpreted functions (CLU). This allows us to automate the verification of the pipelined machine models by using the UCLID decision procedure to transform CLU formulas to Boolean satisfiability problems. To relate pipelined machine states to ISA states, we use the commitment and flushing refinement maps. We evaluate our work using 17 pipelined machine models that contain various features, including deep pipelines, precise exceptions, branch prediction, interrupts, and instruction queues. Our experimental results show that the overhead of proving liveness, obtained by comparing the cost of proving both safety and liveness with the cost of only proving safety, is about 17%, but depends on the refinement map used; for example, the liveness overhead is 23% when flushing is used and is negligible when commitment is used.

Categories and Subject Descriptors: J.6 [**Computer Applications**]: Computer-Aided Engineering

General Terms: Design, Reliability

Additional Key Words and Phrases: Refinement maps, flushing, commitment, SAT, pipelined machines, verification, refinement, bisimulation, liveness

ACM Reference Format:

Manolios, P. and Srinivasan, S. K. 2008. Automatic verification of safety and liveness for pipelined machines using WEB refinement. ACM Trans. Des. Autom. Electron. Syst. 13, 3, Article 45 (July 2008), 19 pages, DOI = 10.1145/1367045.1367054 http://doi.acm.org/10.1145/1367045.1367054

This research was funded in part by NSF grants CCF-0429924, IIS-0417413 and CCF-043887, and by ND EPSCoR through NSF grant EPS-0447679.

Authors' addresses: P. Manolios, College of Computer and Information Science, Northwestern University, Boston, MA 02115; S. K. Srinivasan, Department of Electrical and Computer Engineering, North Dakota State University, Fargo, ND 58105; email: sudarshan.srinivasan@ndsu.edu.

1. INTRODUCTION

Functional validation and verification are critical problems in the microprocessor industry because the cost of fixing defective products is prohibitively high and because processor designs are extremely complex and highly optimized. For example, in 1994 a bug in the floating-point division (FDIV) unit of the Intel Pentium processor cost Intel 475 million dollars. Estimates show that a similar bug in the current generation of Intel processors would cost Intel about 12 billion dollars [Bentley 2005].

Currently, the predominant method for validating microprocessor designs is simulation. 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 [Bentley 2005, 2001]. To overcome the limitations of simulation, the industry is starting to use semiformal and formal methods, examples of which include efforts at Intel [Bentley 2005], IBM [Ludden et al. 2002], and Motorola [Abadir et al. 2003]. 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. 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 those parts of the design that were formally verified [Bentley 2005].

In this article, we focus on the verification of pipelining, one of the main techniques used to increase microprocessor performance. We present an automatic, refinement-based methodology for verifying that abstract pipelined machine models satisfy the same safety and liveness properties as their instruction-set architectures. The work presented here extends a previous conference paper [Manolios and Srinivasan 2004] and technical report [Manolios and Srinivasan 2003] by including a detailed description of the techniques developed and a more thorough experimental analysis.

The pipelined machine models we consider are abstract *term-level* models. Such models abstract away the datapath using (unbounded) 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 design. This helps make the verification problem tractable because there are powerful tools capable of automatically analyzing term-level models [Bryant et al. 2002].

The notion of correctness we use is based on well-founded equivalence bisimulation (WEB) refinement. WEB refinement guarantees that the pipelined machine and its ISA (instruction-set architecture) satisfy the same safety and liveness properties. A consequence is that the pipelined machine satisfies exactly the same $\mathrm{CTL}^* \setminus \mathsf{X}$ properties satisfied by its ISA. Manolios [2000] introduced WEB refinement and showed how to apply it to verify simple three-stage pipelined machines using the ACL2 theorem-proving system [Kaufmann et al. 2000a, 2000b]. The main contribution of this work is to show how to automate the verification of term-level pipelined machine models using WEB refinement.

The use of refinement allows us to avoid the two main limitations of using property-based verification for establishing the correctness of 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. To avoid these problems, Burch and Dill introduced a notion of correctness based on the commuting diagrams upon which much of the previous work in the area is based [Burch and Dill 1994]. Unfortunately, this notion is not as complete as we would like; for example, it does not fully address liveness and even when augmented with various liveness properties, it can still be satisfied by machines that deadlock [Manolios 2000].

We present a method for automatically verifying that a pipelined machine refines its instruction-set architecture (ISA). Automation 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 term-level and our refinement-based correctness statements are expressible in CLU, we can use UCLID, a decision procedure for CLU logic, to automatically check the correctness statements [Bryant et al. 2002]. UCLID translates CLU formulas to SAT problems which can then be checked using SAT solvers. We use the Siege SAT solver [Ryan] because we have found it the most effective SAT solver for our problems.

The rest of the article is organized as follows. In Section 2, we provide an overview of WEB refinement. In Section 3, we give a brief description of the pipelined machine models used for our experiments, and in Section 4, we describe the verification of these models using refinement maps that are based on flushing and commitment. In Section 5, we report and analyze results obtained from verifying 17 pipelined machine models, using both flushing and commitment refinement maps. We compare the two refinement maps, and also compare the cost of safety proofs with the cost of safety and liveness proofs. The UCLID specifications and corresponding SAT problems in conjunctive-normal form (CNF) used for our experiments are available online [Manolios and Srinivasan 2005c]. We describe related work in Section 6 and conclude in Section 7.

2. REFINEMENT

We present a theory of refinement that can be used to establish that MA, a machine modeled at microarchitecture level (i.e., a low-level description that includes the pipeline), correctly implements the ISA, a machine modeled at instruction-set architecture level. We accomplish this by first defining a *refinement map* r, a function from MA states to ISA states; think of r as showing us how to view an MA state as an ISA state. We then prove a *stuttering*

bisimulation refinement: For every pair of states w, s such that w is an MA state and $s = r(w)$, for every infinite path σ starting at s, there is a "matching" infinite path δ starting at w, and conversely. That σ and δ match implies that applying r to the states in δ results in a sequence that can be obtained from σ by repeating, but only finitely often, some of σ's states, as MA may require several steps before matching a single step of ISA.

As presented earlier, our approach requires reasoning about infinite paths, which is difficult to automate. WEB refinement is an equivalent formulation, but requires only local reasoning [Manolios 2001]. One of the key ideas is the use of rank functions, which map MA states to the natural numbers, to ensure that there is only finite stuttering. We now give the relevant definitions, which are given in terms of general transition systems (TS). Specifically, a TS $\mathcal{M}$ is a triple $\langle S, \dashrightarrow, L\rangle$ consisting of a set of states S, a transition relation $\dashrightarrow$, and a labeling function L with domain S, where $L(s)$ is what is visible at s.

Definition 2.1 (*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 preceding 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 2.2. $B \subseteq S \times S$ is a WEB on TS $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ iff:

(1) B is an equivalence relation on S;
(2) $\langle \forall s, w \in S :: sBw \Rightarrow L(s) = L(w)\rangle$; and
(3) there exist functions $rankl : S \times S \to \mathbb{N}$, $rankt : S \to W$, such that $\langle W, \lessdot\rangle$ 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) $(uBw \ \wedge\ rankt(u) \lessdot rankt(s)) \ \vee$
(c) $\langle \exists v :: w \dashrightarrow v \ \wedge\ sBv \ \wedge\ rankl(v, u) < rankl(w, u)\rangle\rangle$.

The third WEB condition says that, given states s and w in the same class such that s can step to u, the 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). 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* \ X, over the state components visible at instruction-set-architecture level.

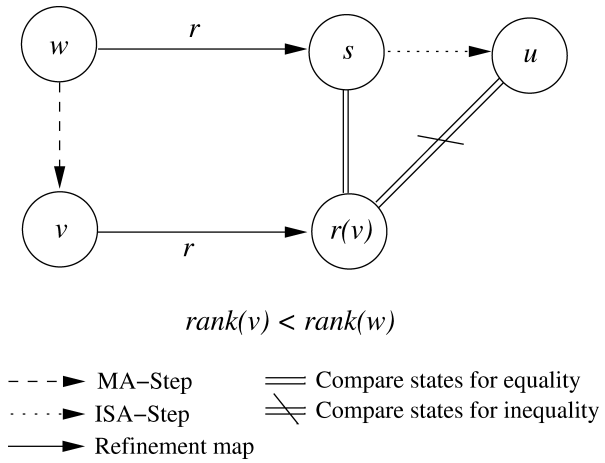


Fig. 1. Diagram showing the core theorem that can be expressed in CLU logic.

The idea now is to strengthen the WEB-refinement proof obligation such that we obtain a CLU-expressible statement that holds for the examples we consider. We start by defining the equivalence classes of B to consist of one ISA state and all those MA states that map to the ISA state under r. Now, condition 2 of the WEB definition clearly holds. Also, since the ISA does not stutter with respect to the MA, we can ignore the second disjunct of the third condition in the WEB definition. Our ISA and MA machines are deterministic (actually there is some nondeterminism in MA, but we use oracle variables to transform MA into a deterministic machine [Manolios 2003]); 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 MA into the natural numbers.

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

In the preceding formula and corresponding diagram in Figure 1, w and v are MA states, and s and u are ISA states; `MA-step` is a function corresponding to stepping the MA machine once and `ISA-step` is a function corresponding to stepping the ISA machine once. The core theorem says that if w refines s, and 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. Note that for the machines we consider, w uniquely determines s, u, and v. Also, the proof obligation relating s and v can be thought of as the safety component, and the proof obligation $rank(v) < rank(w)$ can be thought of as the liveness component.

In the sequel, we use two types of refinement maps and provide a general method for defining rank functions in both cases. The details appear in Section 4, after we describe the pipelined machine models.

We end this section by noting that the question of correctness is a basic concern in pipelined machine verification and by outlining the two advantages of

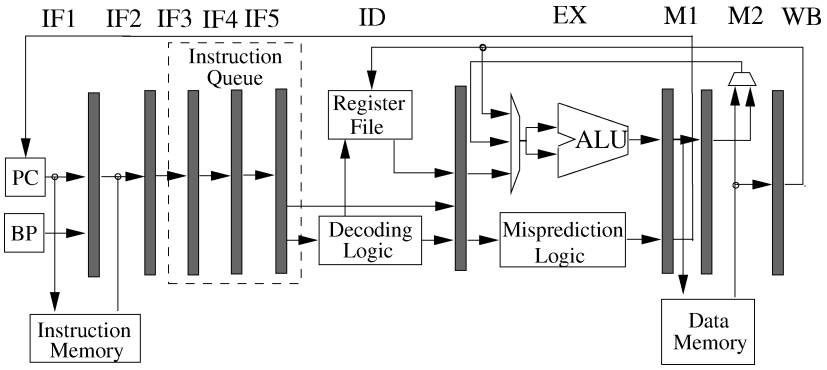


Fig. 2. High-level organization of the ten-stage pipelined machine model with a pipelined fetch stage, branch prediction, and an instruction queue.

our approach over variants of the widely used Burch and Dill notion of correctness. First, pipelined machines are complicated enough that certain types of error may avoid detection; for example, it does not necessarily follow from a Burch and Dill correctness proof, even when augmented with theorems that researchers thought would establish liveness, that the pipelined machine cannot deadlock [Manolios 2000]. In contrast, after we prove a WEB refinement, it follows directly from the metatheory that deadlock is not possible. If the deadlock arises only in certain rare corner cases, then the bug can also easily avoid detection from simulation. Second, by using a theory of refinement, we can strengthen the proof obligations in ways that make automation possible without risking inconsistencies. It was this line of reasoning that led to the work described in this article, the first method we know of for automatically proving both safety and liveness properties for pipelined machines. Note, however, that one cannot prove a WEB refinement for pipelined machines that can commit multiple instructions in a single cycle, as the infinite executions of such a pipelined machine and its ISA will not match.

3. PIPELINED MACHINE MODELS

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 two-cycle memory access (M1 and M2), and a writeback (WB). The models implement four instruction types, including ALU, 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 2 shows the high-level organization of a complex ten-stage pipelined machine model with branch prediction, a pipelined fetch stage, and a three-stage instruction queue. The seven-stage models are inspired by the Intel XScale architecture, and the other pipelined machine models are obtained by extending these seven-stage models. Some of our modeling techniques, such as those used

for branch prediction and exceptions, are based on Velev and Bryant [2000]. Modeling issues are not the point of this article; nevertheless, a brief overview of CLU and the processor models we use is helpful for understanding the rest of the work.

The CLU logic [Bryant et al. 2002] is a decidable logic that is limited in its expressive power, but powerful enough to describe various features of microprocessors at an abstract level. The CLU logic contains the Boolean connectives, uninterpreted functions (UFs) and predicates (UPs), equality, counter arithmetic, ordering, and restricted lambda expressions. The basic CLU types are Booleans and terms. Terms can be thought of as integers and are used to abstract word-level values. The only property satisfied by UFs and UPs is functional consistency, that is, given equal inputs, the outputs are also equal. UFs and UPs can be used to abstract combinational circuit blocks; for example, the ALU is abstracted away using a UF that takes the operands and opcode as input and returns a term, the result. The register file is modeled using restricted lambda expressions. The restrictions on lambda expressions do not allow one to define recursive definitions in CLU. The read and write accesses to data memory in the pipelined machine 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 (specifically, the interpreted function's 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 whether an interrupt is raised. Interrupts are detected in the M1 stage and result in the invalidation of all younger instructions, 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 instruction-memory address 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 the 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, the 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 nondeterministic input that corresponds to the state of the branch predictor. The predictions on the direction

and target of the branch are determined using the UP *PredictDirection* and the UF *PredictTarget*, respectively. We call this the nondeterministic-branch-prediction abstraction scheme. The actual direction and target of a 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 whether an exception is raised. M1 handles ALU exceptions in the following way. In case of an ALU exception, all younger instructions 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.

4. VERIFICATION OF PIPELINED MACHINE MODELS

Stating the core theorem described in Section 2 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, instruction memory, data memory, and register file. For 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.

4.1 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 [Burch and Dill 1994]. A pipelined machine state is flushed by applying a sufficient number of flush operations successively so that all 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 (e.g., 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 eight flushing steps to flush the pipelined machine model with seven stages, as the youngest 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.

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 $r(v)$. The 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 [Burch and Dill 1994].

The liveness component of the refinement theorem is checked by comparing the ranks 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 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 result is that erroneous rank definitions are caught during verification.

4.2 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 [Manolios 2000].

The commitment refinement map can be easily implemented by recording some of the history of the programmer-visible components using history variables (variables that store past values of state components). We track the values of the programmer-visible components before they were updated by each of the partially executed instructions in the pipeline. For example, in seven-stage machine models, we track the last six values of the program counter. The commitment refinement map invalidates the partially executed instructions and assigns to the programmer-visible components the values they had before they were updated by the oldest instruction in the pipeline. Note that no symbolic simulations are required.

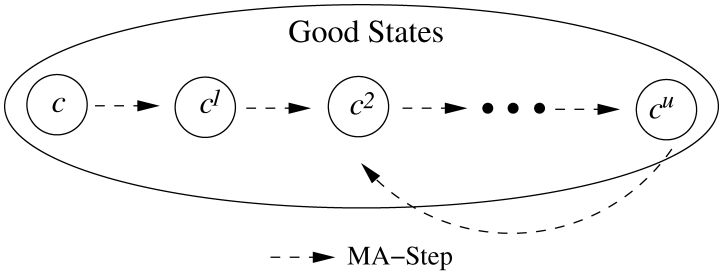
For the commitment refinement map, we define the rank of a pipelined machine state to be the number of steps required to commit an instruction. The first instruction that gets committed is the oldest. In addition, for the machines we consider, the flow of an instruction through the pipeline is only affected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

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 seven-stage pipelined machine that has only one instruction in the pipeline, but this 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.

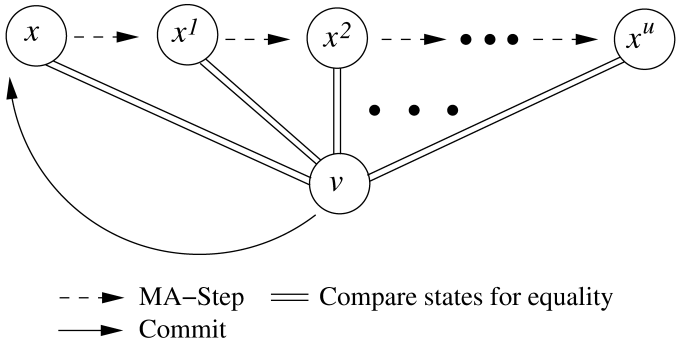
We now show how to define the required invariant. First we define the notion of a committed state, which is a pipelined state in which all the pipeline registers are invalid. A pipelined state is "good" if it is reachable from a committed state. The set of good states is an invariant that we call the least-fixpoint (LFP) invariant, as computing this set involves a least-fixpoint computation (see Figure 3(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 the machine model with seven stages, the upper bound is six, 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 we show the good states to be closed under the pipelined machine-transition relation. The invariant proof is trivially true for those 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 show is that the successor of any state that is u steps from a committed state is good, as depicted in Figure 3(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



(a) the LFP invariant proclaims 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 whether 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.

Fig. 3. The least-fixpoint (LFP) invariant.

state reachable in $1, \ldots, u$ steps from an arbitrary committed state. We then prove the core theorem for each of these possibilities.

4.3 Remarks

The flushing approach does not usually require use of an invariant. Why not? This is because inconsistent states such as the one mentioned before are flushed away; that is, inconsistent states are related to ISA states. Deciding between using the commitment approach or the flushing approach depends on whether this aspect of flushing is acceptable.

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 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. Finally, the UCLID tool generates a concrete counterexample if it finds a bug.

5. 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 the flushing refinement map verification times increase by about 23%. Finally, commitment takes less time overall and scales better than flushing.

5.1 Benchmark Suite

The verification times and statistics for the Boolean correctness formulas are shown in Table I. We report the number of CNF variables and clauses and the verification time for both the safety proofs and the safety and liveness proofs, that is, 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 ten runs. We also report the standard deviation of the ten 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 3.06 GHz Intel Xeon machine with an L2 cache size of 512KB.

We use the following naming convention for the pipelined machine models and verification problems. A model name begins with 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. A verification problem begins with either "c" or "f", indicating that the commitment or flushing refinement map is used, respectively, followed by a model name.

The overhead cost of liveness, computed by subtracting the sum of the "Safety Verification Times Total" column from that 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 (e.g., the machine models), SAT solvers are able to prove the conjunction of the two theorems in time comparable to that required to prove just one of the theorems. In fact, in some cases the verification times for safety and liveness are slightly less than those for safety alone (as with some of the commitment problems), indicating that the heuristics of the SAT solver are able to effectively exploit the shared structure.

Table I. Pipelined Machine-Verification Times and Statistics

Verification Problem	Safety				Safety and Liveness				
	CNF Vars	CNF Clauses	Verification Times (secs)		CNF Vars	CNF Clauses	Verification Times (secs)		
			Siege	Total			Siege	Stdev	Total
c6	12,817	37,876	28	30	12,334	36,442	28	5.4	30
f6	13,429	39,694	10	12	28,256	83,725	10	2.1	14
c6n	37,718	111,790	160	165	37,147	110,077	170	35.5	175
f6n	16,462	48,529	10	13	37,452	110,920	12	2.7	17
c6b	22,410	66,223	121	124	21,850	64,558	119	16.0	122
f6b	17,135	50,548	11	14	37,002	109,570	15	3.6	20
c7	13,728	40,609	25	27	13,296	39,328	26	3.8	28
f7	28,477	84,535	134	138	53,165	158,182	135	6.8	142
c7n	28,007	82,978	226	230	27,500	81,472	234	43.4	237
f7n	33,160	98,212	109	114	70,667	210,172	136	14.9	145
c7b	26,785	79,294	201	204	26,058	77,128	222	52.6	225
f7b	33,674	99,754	124	129	70,985	211,126	139	24.5	148
c7be	26,766	79,186	199	203	26,264	77,695	213	25.3	217
f7be	35,961	106,516	134	139	74,702	222,145	157	23.2	167
c7bep	26,806	79,291	239	243	26,615	78,733	260	73.7	264
f7bep	37,599	111,406	120	126	81,759	243,259	171	20.9	182
c8	14,528	43,009	27	29	14,100	41,740	31	5.8	33
f8	47,551	141,538	821	828	95,092	283,465	697	31.8	709
c8n	54,252	161,260	770	776	53,697	159,595	758	66.9	764
f8n	56,790	168,742	597	605	121,499	361,954	716	49.5	731
c8b	32,569	96,526	560	564	31,914	94,576	493	83.8	497
f8b	58,180	172,912	594	602	121,645	362,392	699	55.1	715
c9	15,648	46,369	30	32	15,214	45,082	29	6.1	31
f9	70,295	209,551	2,574	2,584	144,045	429,973	2,309	100.1	2,328
c9n	63,101	187,771	1,455	1,462	62,536	186,076	1,517	238.0	1,524
f9n	87,650	261,001	1,998	2,010	185,149	552,412	2,546	117.7	2,570
c9b	37,539	111,376	982	987	36,757	109,045	934	169.2	939
f9b	87,278	259,885	2,089	2,101	183,371	547,078	2,333	124.7	2,357
c10	17,526	52,003	33	36	17,121	50,803	31	7.1	34
f10	111,631	333,124	5,407	5,422	198,375	592,660	6,385	506.9	6,411
c10n	73,727	219,613	2,901	2,910	73,163	217,921	2,641	358.2	2,650
f10n	129,085	384,793	4,229	4,247	255,780	763,861	6,726	365.8	6,761
c10b	44,287	131,560	1,675	1,681	43,517	129,265	1,774	423.2	1,780
f10b	129,957	387,409	4,039	4,057	256,272	765,337	6,540	584.9	6,575

5.2 Commitment vs. Flushing

Figure 4 is a scatter plot that compares commitment and flushing using 17 pipelined machine models. Notice that both the x and y axes have logarithmic scale. As can be seen from the figure, commitment does better than flushing on most of the models, especially on those with longer pipelines. This is depicted more clearly in Figure 5, which shows the variation in verification time as the length of pipeline increases, for both flushing and commitment. Note that the y axis in Figure 5 has 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

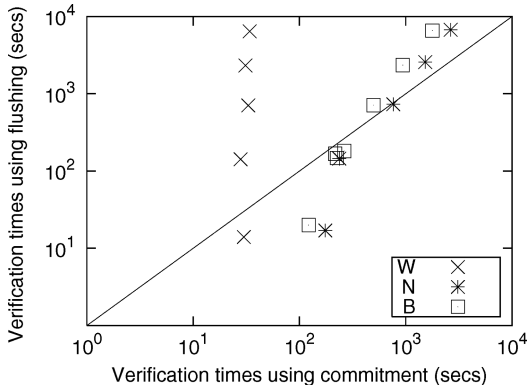


Fig. 4. Comparison of commitment and flushing based on verification times. The keys "W", "N", and "B" in the figure indicate models without a branch-prediction abstraction scheme, models with the nondeterministic-branch-prediction abstraction scheme, and models with the general-branch-prediction abstraction scheme, respectively.

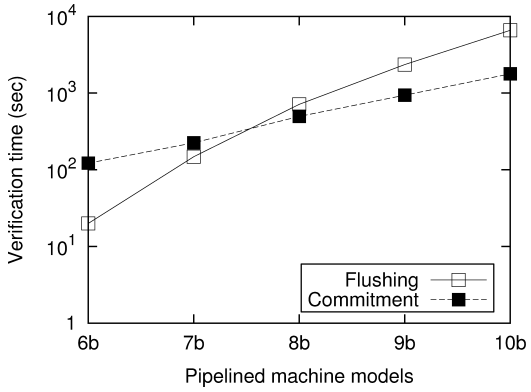


Fig. 5. Graph of verification times for commitment and flushing as the length of pipeline is increased.

the verification problem and size of the CNF formulas generated both 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, the latter is responsible for the larger symbolic distance required for the flushing refinement map.

The differences in verification time for the commitment approach when including a branch-prediction abstraction scheme (e.g., 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 much affect verification times. 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 [Manolios and Srinivasan 2005b].

When deciding between the commitment and flushing approaches, consider the following. First, flushing-verification times are more sensitive to the depth of pipeline than are commitment-verification times. Second, there are ways of optimizing both methods that should be considered [Manolios and Srinivasan 2005b, 2005d; Kane et al. 2006]. Where possible, we prefer the commitment approach because its refinement map seems clearer and it has various advantages in the context of nondeterministic machines [Manolios 2000].

6. 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 work in this area is by Burch and Dill, who introduced a notion of correctness based on commuting diagrams that we call the BD correctness criterion [Burch and Dill 1994]. Aagaard et al. [2003, 2001] 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. For instance, it does not address liveness and thus does not guarantee 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 used to verify some very complex processor models, using the ACL2 theorem-proving system [Sawada 1999]. This strengthened notion of correctness is still not complete, as it is possible to mechanically prove that certain pipelined machines which can deadlock nevertheless satisfy this notion of correctness [Manolios 2000]. Velev proposed another approach that handles both safety and liveness and is also a variation of the BD correctness criterion [Velev 2004]. 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 that can be exploited to verify complex pipelined machines, which cannot be handled using automatic monolithic approaches [Manolios and Srinivasan 2005a].

Automatic verification of term-level pipelined machines has directly benefited from advances in decision procedures. Burch and Dill [1994] showed how to use a decision procedure for the logic of equality with uninterpreted functions to automatically verify term-level pipelined processor models. An efficient decision procedure for the same logic was given by Bryant et al. [1999], and that work was further extended by Bryant et al. [2002], 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 [Lahiri et al. 2002]. Also worth mentioning is the SVC decision procedure which was used to check

the correct flow of instructions in a pipelined DLX model [Mishra and Dutt 2002]. Jones et al. [1998] employed SVC to verify an out-of-order execution unit, using incremental flushing. 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) [Ganzinger et al. 2004] and Yices [de Moura 2006].

There are also approaches based on the use of theorem provers for pipelined machine verification. A very early approach by Srivas and Bickford was based on the use of skewed abstraction functions [Srivas and Bickford 1990]. Sawada [1999] and Sawada and Hunt [2002] have used the ACL2 theorem prover [Kaufmann et al. 2000a, 2000b] and an intermediate abstraction called MAETT to verify very complex pipelined machines. Another example of a theorem-proving approach is the work by Hosabettu et al., who use the notion of completion functions [Hosabettu et al. 1999] to compute the abstraction function or refinement map. The correctness proofs are carried out using the PVS theorem prover [Owre et al. 2001]. Arons and Pnueli [2000] 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 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. Kroening [2001] verified the data consistency of pipelined machine models using the PVS theorem prover. The models are synthesizable and described very close to the gate level.

Other approaches to the pipelined machine verification problem include work based on model checking; for example, McMillan uses compositional model checking and symmetry reduction [McMillan 1998]. Symbolic trajectory evaluation (STE) is used by Patankar et al. to verify a processor that is a hybrid between ARM7 and StrongARM [Patankar et al. 1999]. Abstract state machines are used to prove the correctness of refinement steps that transform a nonpipelined ARM processor into a pipelined implementation [Huggins and Campenhout 1998].

7. CONCLUSIONS AND FUTURE WORK

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 the CLU logic. Second, we provide simple and general methods for defining commitment and flushing refinement maps and their associated rank functions. We conducted a thorough, extensive experimental validation of our techniques that involved the use of seventeen pipelined machine models containing various features, including branch prediction, precise exceptions, interrupts, instruction queues, and deep pipelines. Our experimental results show that proving safety and liveness leads to an increase of about 17% over the verification time required to prove just safety. Interestingly, the choice of refinement map used leads to significant differences in running times:

The use of flushing leads to a 23% increase in verification time, whereas the use of commitment does not lead to any increase in verification time. Also, the commitment refinement map required less time overall and scaled better than the flushing refinement map.

For future work, we plan to apply the refinement-based verification paradigm to executable pipelined machine models defined at bit level. We propose to do this by using a combination of decision procedures and deductive reasoning.

REFERENCES

AAGAARD, M., COOK, B., DAY, N. A., AND JONES, R. B. 2001. A framework for microprocessor correctness statements. In *Proceedings of the Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, Livingston, UK. T. Margaria and T. F. Melham, eds. Lecture Notes in Computer Science, vol. 2144. Springer, 433–448.

AAGAARD, M., COOK, B., DAY, N. A., AND JONES, R. B. 2003. A framework for superscalar microprocessor correctness statements. *Int. J. Softw. Tools Technol. Transfer 4*, 3, 298–312.

ABADIR, M. S., ALBIN, K., HAVLICEK, J., KRISHNAMURTHY, N., AND MARTIN, A. K. 2003. Formal verification successes at Motorola. *Formal Meth. Syst. Des. 22,* 2, 117–123.

ARONS, T. AND PNUELI, A. 2000. A comparison of two verification methods for speculative instruction execution. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Berlin. Lecture Notes in Computer Science, vol. 1785. Springer, 487–502.

BENTLEY, B. 2001. Validating the Intel Pentium 4 microprocessor. In *Proceedings of the ACM Design Automation Conference (DAC)*. ACM, 244–248.

BENTLEY, B. 2005. Validating a modern microprocessor. `http://www.cav2005.inf.ed.ac.uk/-bentley_CAV_07_08_2005.ppt`.

BRYANT, R. E., GERMAN, S., AND VELEV, M. N. 1999. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proceedings of the Computer-Aided Verification (CAV)*, Trento, Italy. N. Halbwachs and D. Peled, eds. Lecture Notes in Computer Science, vol. 1633. Springer, 470–482.

BRYANT, R. E., LAHIRI, S. K., AND SESHIA, S. A. 2002. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proceedings of the Computer-Aided Verification (CAV)*, E. Brinksma and K. G. Larsen, eds. Lecture Notes in Computer Science, vol. 2404. Springer, 78–92.

BURCH, J. R. AND DILL, D. L. 1994. Automatic verification of pipelined microprocessor control. In *Proceedings of the Computer-Aided Verification (CAV)*, D. L. Dill, ed. Lecture Notes in Computer Science, vol. 818. Springer, 68–80.

DE MOURA, L. 2006. Yices homepage. `http://fm.csl.sri.com/yices`.

GANZINGER, H., HAGEN, G., NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. 2004. DPLL(T): Fast decision procedures. In *Proceedings of the Computer-Aided Verification (CAV)*, Boston, MA. R. Alur and D. Peled, eds. Lecture Notes in Computer Science, vol. 3114. Springer, 175–188.

HOSABETTU, R., SRIVAS, M., AND GOPALAKRISHNAN, G. 1999. Proof of correctness of a processor with reorder buffer using the completion functions approach. In *Proceedings of the Computer-Aided Verification (CAV)*, Trento, Italy. N. Halbwachs and D. Peled, eds. Lecture Notes in Computer Science, vol. 1633. Springer, 686–698.

HUGGINS, J. K. AND CAMPENHOUT, D. V. 1998. Specification and verification of pipelining in the ARM2 RISC microprocessor. *ACM Trans. Des. Autom. Electron. Syst. 3,* 4, 563–580.

JONES, R., SKAKKEBÆK, J., AND DILL, D. 1998. Reducing manual abstraction in formal verification of out-of-order execution. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD)*, Palo Alto, CA. G. Gopalakrishnan and P. Windley, eds. Lecture Notes in Computer Science, vol. 1522. Springer, 2–17.

KANE, R., MANOLIOS, P., AND SRINIVASAN, S. K. 2006. Monolithic verification of deep pipelines with collapsed flushing. In *Proceedings of the Design Automation and Test in Europe (DATE)*, Leuven, Belgium. G. G. E. Gielen, ed. European Design and Automation Association, 1234–1239.

KAUFMANN, M., MANOLIOS, P., AND MOORE, J. S., Eds. 2000a. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic.

KAUFMANN, M., MANOLIOS, P., AND MOORE, J. S. 2000b. *Computer-Aided Reasoning: An Approach*. Kluwer Academic.

KROENING, D. 2001. Formal verification of pipelined microprocessors. Ph.D. thesis, Universität des Saarlandes.

LAHIRI, S., SESHIA, S., AND BRYANT, R. 2002. Modeling and verification of out-of-order microprocessors using UCLID. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD)*, Portland, OR. Lecture Notes in Computer Science, vol. 2517. Springer, 142–159.

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. 2002. Functional verification of the POWER4 microprocessor and POWER4 multiprocessor system. *IBM J. Res. Devel. 46*, 1, 53–76.

MANOLIOS, P. 2000. Correctness of pipelined machines. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD)*, W. A. H., Jr. and S. D. Johnson, eds. Lecture Notes in Computer Science, vol. 1954. Springer, 161–178.

MANOLIOS, P. 2001. Mechanical verification of reactive systems. Ph.D. thesis, University of Texas at Austin. `http://www.cc.gatech.edu/~manolios/publications.html`.

MANOLIOS, P. 2003. A compositional theory of refinement for branching time. In *Proceedings of the 12th IFIP WG 10.5 Advanced Research Working Conference (CHARME)*, D. Geist and E. Tronci, eds. Lecture Notes in Computer Science, vol. 2860. Springer, 304–318.

MANOLIOS, P. AND SRINIVASAN, S. K. 2003. Automatic verification of safety and liveness for XScale-like processor models using WEB refinements. Tech. Rep. GIT-CERCS-03-17, Georgia Institute of Technology, College of Computing. September.

MANOLIOS, P. AND SRINIVASAN, S. K. 2004. Automatic verification of safety and liveness for XScale-like processor models using WEB refinements. In *Proceedings of the Design, Automation, and Test in Europe (DATE)*. IEEE Computer Society, 168–175.

MANOLIOS, P. AND SRINIVASAN, S. K. 2005a. A complete compositional reasoning framework for the efficient verification of pipelined machines. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, San Jose, CA. IEEE Computer Society, 863–870.

MANOLIOS, P. AND SRINIVASAN, S. K. 2005b. A computationally efficient method based on commitment refinement maps for verifying pipelined machines. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, 188–197.

MANOLIOS, P. AND SRINIVASAN, S. K. 2005c. A parameterized benchmark suite of hard pipelined machine verification problems. `http://www.cc.gatech.edu/~manolios/benchmarks/charme.html`.

MANOLIOS, P. AND SRINIVASAN, S. K. 2005d. Refinement maps for efficient verification of processor models. In *Proceedings of the Design Automation and Test in Europe (DATE)*. IEEE Computer Society, 1304–1309.

MCMILLAN, K. L. 1998. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *Proceedings of the Computer Aided Verification (CAV)*, Van Couver, British Colombia, Canada. A. J. Hu and M. Y. Vardi, eds. Lecture Notes in Computer Science, vol. 1427. Springer, 110–121.

MISHRA, P. AND DUTT, N. D. 2002. Modeling and verification of pipelined embedded processors in the presence of hazards and exceptions. In *Proceedings of the IFIP WCC Stream 7 on Distributed and Parallel Embedded Systems (DIPES)*, Montreal, Qubec, Canada, vol. 219. B. Kleinjohann et al., eds. Kluwer, 81–90.

OWRE, S., SHANKAR, N., RUSHBY, J. M., AND STRINGER-CALVERT, D. W. J. 2001. PVS system guide. `http://pvs.csl.sri.com/doc/pvs-system-guide.pdf`.

PATANKAR, V. A., JAIN, A., AND BRYANT, R. E. 1999. Formal verification of an ARM processor. In *Proceedings of the 12th International Conference on VLSI Design*, Goa, India. IEEE, 282–287.

RYAN, L. 2008. Siege homepage. `http://www.cs.sfu.ca/~loryan/personal`.

SAWADA, J. 1999. Formal verification of an advanced pipelined machine. Ph.D. thesis, University of Texas at Austin. `http://www.cs.utexas.edu/users/sawada/dissertation/`.

SAWADA, J. AND HUNT, W. A. 2002. Verification of FM9801: An out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability. *Formal Meth. Syst. Des. 20,* 2, 187–222.

SRIVAS, M. AND BICKFORD, M. 1990. Formal verification of a pipelined microprocessor. *IEEE Softw. 7,* 5, 52–64.

VELEV, M. N. 2004. Using positive equality to prove liveness for pipelined microprocessors. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan. M. Imai, ed. IEEE, 316–321.

VELEV, M. N. AND BRYANT, R. E. 2000. Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. In *Proceedings of the ACM Design Automation Conference (DAC)*, Los Angeles, CA. ACM Press, 112–117.

Received December 2005; revised September 2006; accepted November 2007