Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB Refinements

Panagiotis Manolios
College of Computing
Georgia Institute of Technology
Atlanta, GA 30318
manolios@cc.gatech.edu

Sudarshan K. Srinivasan
School of Electrical & Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30318
darshan@ece.gatech.edu

Abstract

We show how to automatically verify that complex XScale-like pipelined machine models satisfy the same safety and liveness properties as their corresponding instruction set architecture models, by using the notion of Well-founded Equivalence Bisimulation (WEB) refinement. Automation is achieved by reducing the WEB-refinement proof obligation to a formula in the logic of Counter arithmetic with Lambda expressions and Uninterpreted functions (CLU). We use the tool UCLID to transform the resulting CLU formula into a Boolean formula, which is then checked with a SAT solver. The models we verify include features such as out of order completion, precise exceptions, branch prediction, and interrupts. We use two types of refinement maps. In one, flushing is used to map pipelined machine states to instruction set architecture states; in the other, we use the commitment approach, which is the dual of flushing, since partially completed instructions are invalidated. We present experimental results for all the machines modeled, including verification times. For our application, we found that the time spent proving liveness accounts for about 5% of the overall verification time.

1. Introduction

We present what we believe is the first method for automatically and efficiently verifying both safety and liveness properties of pipelined machine models. Verification entails constructing a WEB-refinement proof, which implies that, relative to a refinement map, a pipelined machine has exactly the same infinite executions as the machine defined by the instruction set architecture, up to stuttering. A consequence is that the pipelined machine satisfies exactly the same $\mathrm{CTL}^* \setminus \mathsf{X}$ properties satisfied by the instruction set architecture. For the types of machines we study, we can reduce the WEB-refinement proof to a statement expressible in the logic of Counter arithmetic with Lambda expressions and Uninterpreted functions (CLU), which is a decidable logic [2]. We use the tool UCLID [9] to transform the CLU formula into a CNF (Conjunctive Normal Form) formula, which we then check with a SAT solver. We provide experimental results for eight XScale-like [4] pipelined machine models of varying complexity and including 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 4.3% longer than the verification times for the standard Burch and Dill type proofs, which do not address liveness.

The use of WEB-refinement for proving the correctness of pipelined machines was introduced in [10], where a number of simple three stage pipelined machines were verified using the ACL2 theorem proving system [7, 8]. The paper also showed that the variant of the Burch and Dill notion of correctness [3] used by Sawada [18] can be satisfied by machines that deadlock, and an argument was given that such anomalies are not possible if WEB-refinement is used. Our main contribution is to show how one can prove WEB-refinement theorems automatically and efficiently, which we accomplish by defining "rank" functions and refinement maps automatically. The WEB-refinement theorem contains quantifiers and involves exhibiting the existence of certain rank functions. We achieve automation in two steps. First, we strengthen the theorem in a way that leads to a simplified statement that is expressible in the CLU logic and which holds for the examples we consider. Second, we show how to define the rank function in a general way that does not require any deep understanding of the pipelined machine and in fact is simpler to define than flushing.

The paper is organized as follows. In Section 2, we provide an overview of refinement based on WEBs, the theory upon which our correctness proofs depend. In Section 3, we give a quick overview of the XScale-like processor models we use, and in Section 4, we outline how we verify such models. In Section 5, we report verification times and statistics for eight processor 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 and we compare the running times of the SAT solvers Siege [17] and Chaff [15] on our problems. Everything required to reproduce our results, *e.g.*, machine models, correctness statements, CNF formulas, etc., is available upon request. Related work is described in Section 6, while conclusions and an outline of future work appear in Section 7.

2. Refinement

Why develop a theory of refinement? Why not just use the Burch and Dill notion of correctness augmented with some kind of liveness criterion? First, pipelined machines are complicated enough that it is easy to make mistakes, *e.g.*, after we prove a WEB-refinement, it follows directly from the metatheory that the pipelined machine cannot deadlock, whereas this does not necessarily follow from a Burch and Dill correctness proof, even when augmented with theorems that researchers thought would establish liveness [10]. 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 paper, the first method we know of for automatically proving both safety and liveness properties for pipelined machines.

The point of a correctness proof is to establish a meaningful relationship between ISA, a machine modeled at the instruction set architecture level and MA, a machine modeled at the microarchitecture level, a low level description that includes the pipeline. 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. A problem with this approach is that it requires reasoning about infinite paths, which is difficult to automate. In [11], we give an equivalent formulation, WEB-refinement, that requires only local reasoning. We now give the relevant definitions, which are given in terms of general transition systems (TS). 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 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 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 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 *$B \subseteq S \times S$ 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 $rankl : S \times S \to \mathbb{N}, rankt : S \to W$, such that $\langle W, \lessdot\rangle$ is well-founded, and

$$\langle \forall s, u, w \in S :: sBw \;\wedge\; s \dashrightarrow u \;\Rightarrow$$
$$\text{(a)}\ \langle \exists v :: w \dashrightarrow v \;\wedge\; uBv\rangle \;\vee$$
$$\text{(b)}\ (uBw \;\wedge\; rankt(u) \lessdot rankt(s)) \;\vee$$
$$\text{(c)}\ \langle \exists v :: w \dashrightarrow v \;\wedge\; sBv \;\wedge\; rankl(v,u) < rankl(w,u)\rangle\rangle$$

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

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 the MA states that map to the ISA state under r. Now, condition 2 of the WEB definition clearly holds. Our ISA and MA machines are deterministic (actually they are nondeterministic, but we use oracle variables to make them deterministic [12]), 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.

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

In the formula above 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 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 4, after we describe the processor models.

3. Processor Models

Figure 1 shows the high-level organization of the XScale-like processor model, a seven stage pipeline whose stages are IF1, IF2 (2-cycle fetch), ID (instruction decode), EX (execute), M1, M2 (2-cycle memory access), and WB (write back). Five abstract instruction types are modeled 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 branch prediction, ALU exceptions, and interrupts. The models are similar to those appearing in [20] (which use modeling techniques from [22]), except that our models are written in the CLU logic and we model interrupts. Modeling issues are not the point of this paper, nevertheless, a brief overview of CLU and the processor models we use is helpful for understanding the rest of the paper. The CLU logic [2] consists of Uninterpreted Functions (UFs) and Predicates (UPs), restricted lambda expressions, ordering, and successor and predecessor functions. Interrupts are detected in the M1 stage and squash all previous instructions including the instruction that caused the interrupt. We use temporal abstraction to model the behavior of interrupts: after an interrupt the PC is set to the program counter corresponding to the oldest instruction that was squashed by the interrupt, the data memory is modified by applying a UF to the current data memory, and the register file remains the same.

4. Verification of Processor Models

In Section 2, we showed that proving the "core theorem" allows us to establish a WEB-refinement, and in this section we present two methods for defining r and *rank*. One is based on flushing. The other, based on the commitment approach, can be loosely thought of as the dual of flushing, since partially completed instructions are invalidated instead of completed.

In the flushing approach, r maps an MA state, w, to the ISA state obtained by flushing w and projecting out the programmer visible components, where by flushing we mean feeding the pipeline with bubbles to complete partially executed instructions without fetching any new instructions. A pipelined machine model can be easily instrumented to enable such flushing. 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 [3].

The rank of an MA state, w, is the number of steps required to fetch a new instruction that eventually completes. We determine the rank by stepping w, to obtain v, flushing v to obtain v', and comparing v' with w', the flushed state of w, to check if $v' \neq w'$, *i.e.*, to check if the instruction fetched by stepping from w to v completed. (A branch mispredict may lead to a valid instruction being squashed.) The number of steps required to fetch an instruction that completes is the rank. The straightforward implementation of this idea requires 174 symbolic simulations, which UCLID was not able to handle. We implemented an optimized version based on the observation that stepping and flushing the MA states can be folded together so as to reduce the number of symbolic simulations. In more detail, we determine the number of steps required to flush the pipeline (by flushing it) and we set a counter to this value. The MA state is simulated for this number of steps and the rank of the MA state is the number of steps required for the latch between M2 and WB to become valid. Notice that an invalid optimization of the rank function will be caught during verification because the core theorem guarantees that the rank function provided is appropriate.

In the commitment approach, r maps an MA state, w, to the ISA state obtained by committing w and projecting out the programmer visible components, where by committing we mean invalidating all the partially executed instructions in the pipeline and rolling back the PC so that it points to the oldest invalidated instruction. A pipelined machine model can be easily instrumented to enable such committing by using history variables. The history variables record the components of MA states that may need to be modified in the process of invalidating instructions. In our case, we needed to remember 5 state components, including the PC and the data memory, some for multiple steps, leading to 18 history variables.

The rank of an MA state, w, is the number of steps required to commit a new instruction, which can be computed by starting at the end of the pipeline and counting how many consecutive latches are invalid. For example, if there are 3

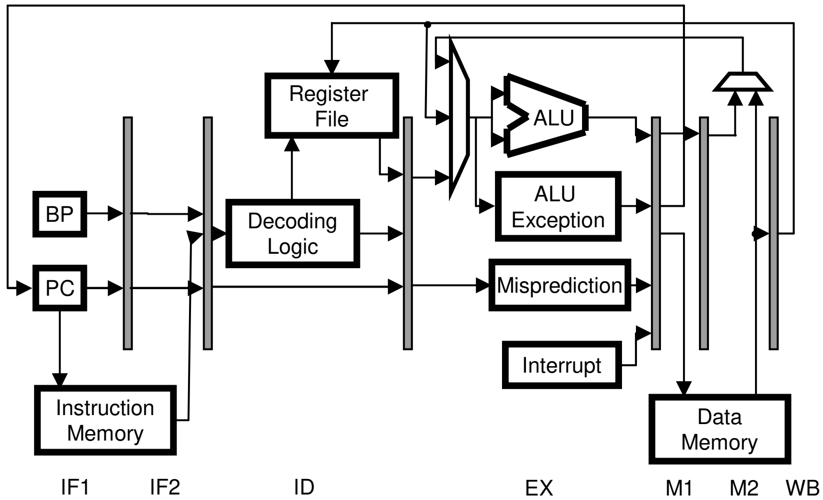

Figure 1. Pipeline organization of processor model

invalid latches at the end of the pipeline, then after three steps, an instruction will be ready to commit. Determining the rank of a state is therefore straightforward.

In order to use the commitment approach we have to use an invariant. To see why, consider a pipelined machine state w whose only valid latch is IF1, but where IF1 is inconsistent with MA, *e.g.*, IF1 does not match any instruction in memory. Suppose s is the state obtained by committing w and projecting. Clearly, s and w do not have the same infinite behaviors, as when the instruction in IF1 completes, it will not match the successor of s. We address this issue by characterizing the set of reachable MA states, the "good" MA states. An MA state is "good" iff it is reachable from a committed state. To check that an MA state, w, is "good," the committed state, c, corresponding to w is determined. State w is "good" if it can be reached from c in 0 to 6 steps. Proving that if a state is good, so is its successor amounts to proving that starting from an arbitrary committed state and taking 7 steps gives a state that is good. The CLU formula corresponding to this invariant is straightforward to write down.

The flushing approach does not usually require the use of an invariant. Why not? Because inconsistent states, such as the one mentioned above get flushed away, but that means that inconsistent states are related to ISA states. Deciding between using the commitment approach or the flushing approach depends on how comfortable one is with this aspect of flushing. Sometimes, even the flushing approach requires the use of invariants and since the invariant we describe above is the *strongest* invariant, it tends to do the trick.

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 latches 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 and performance is essentially the same. Finally, the UCLID tool generates a counterexample if it finds a bug.

5. Results

In this section, we review our experimental results. We start with two base processor models, CXS and FXS: the prefix C indicates the use of the commitment approach for defining the refinement map and prefix F indicates the use of flushing for defining the refinement map. The base models are extended to implement: branch prediction, designated by "-BP"; ALU exceptions, designated by "-EX"; and interrupts, designated by "-INP".

Table 1 presents the results. 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. For the safety and liveness proofs, we also report the size of the CNF files and the verification times taken by both Siege and Chaff. 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 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 really the average over 10 runs and we report the standard deviations for the runs. The experiments were conducted on an Intel XEON 2.20GHz processor with an L1 cache size of 512KB.

Processor	Safety				Safety and Liveness						
	CNF Vars	CNF Clauses	Verification Time [sec]		CNF Var	CNF Clauses	CNF Size [KB]	Verification Time [sec]			
			Siege	Total				Siege	Chaff	Stdev	Total
CXS	12,930	38,215	35	38	12,495	36,925	664	29	6,552	3.4	32
CXS-BP	24,640	72,859	284	289	23,913	70,693	1,336	300	7,861	48.7	305
CXS-BP-EX	24,651	72,841	244	249	24,149	71,350	1,344	233	4,099	50.2	238
CXS-BP-EX-INP	24,669	72,880	255	261	24,478	72,322	1,368	263	3,483	34.1	269
FXS	28,505	84,619	140	154	53,441	159,010	3,096	160	796	24.4	175
FXS-BP	33,964	100,624	170	185	71,184	211,723	4,136	187	586	50.4	203
FXS-BP-EX	35,827	106,114	179	195	74,591	221,812	4,344	163	759	17.6	180
FXS-BP-EX-INP	38,711	11,4742	128	147	81,121	241,345	4,736	170	1,427	32.3	189

Table 1. Statistics for boolean correctness formula and formal verification time.

As is clear from Table 1, Siege provides superior performance when compared to Chaff. If we divide the total running time of Chaff with Siege, we see that Siege provides a speedup of about 17 and in the case of CXS the speedup is 226. The overall cost of liveness, computed by subtracting the sum of the Safety Siege column from the sum of the Safety and Liveness Siege column and dividing by the latter is 4.6%; notice that for the commitment approach it is 0.75%, whereas it is 9.3% for the flushing approach. The verification time for liveness alone seems to be about the same as the verification time for safety, *e.g.*, when proving liveness for CXS, Siege takes 37 seconds (this is the average of ten runs). Since the liveness and safety theorems share considerable structure (*e.g.*, the machine models), the SAT solvers are able to prove the conjunction of the two theorems more quickly than proving each conjunct separately; in fact, in some cases the verification times for safety and liveness are slightly less than the verification times for safety alone, indicating that the heuristics of the SAT solver are able to effectively exploit the shared structure.

The differences in verification times between CXS and CXS-BP 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. In all the CXS models, the proof of the "good" invariant accounted for all but about .2 seconds of the verification times. The strongest invariant can be very useful for checking other types of properties, *e.g.*, performance properties.

6. Related Work

We now selectively review previous work on pipelined machine verification. A very early approach by Srivas and Bick was based on the use of skewed abstraction functions [21]. Burch and Dill showed how to automatically compute the abstraction function using flushing [3]. There are approaches based on model-checking, *e.g.*, in [13], McMillan uses compositional model-checking and symmetry reductions. Theorem proving approaches are also popular, *e.g.*, in [18, 19], Sawada uses an intermediate abstraction called MAETT to verify some safety and liveness properties of complex pipelined machines. Another approach by Hosabettu et al. uses the notion of completion functions [5]. Symbolic Trajectory Evaluation (STE) is used by Patankar et al. to verify a processor that is a hybrid between ARM7 and StrongARM [16]. SVC is used to check the correct flow of instructions in a pipelined DLX model [14]. Abstract State Machines are used to prove the correctness of refinement steps that transform a non-pipelined ARM processor into a pipelined implementation [6]. An XScale processor model is verified using a variation of the Burch and Dill approach in [20]. This paper directly depends on previous work on decision procedures for boolean logic with equality and uninterpreted function symbols [1]. The results in [1] were further extended in [2], where a decision procedure for the CLU logic is given. The decision procedure is implemented in UCLID, which has been used to verify out-of-order microprocessors [9].

7. Conclusions and Future Work

Our main contribution is to show how to automatically verify both safety and liveness properties of complex XScale-like pipelined machine models with a slight performance penalty over verifying safety properties alone. This improves on previous automatic methods, which can only check safety properties. Verifying pipelined machines involves establishing a WEB-refinement theorem, which implies that the pipelined machine satisfies exactly the same $\text{CTL}^* \setminus \mathsf{X}$ properties satisfied by the instruction set architecture. We show how to automate the verification of the WEB-refinement theorem, which contains quantifiers and involves exhibiting the existence of certain rank functions. The automation is achieved in two steps. First, we strengthen the theorem in a way that leads to a simplified statement that holds for the examples we consider. Second, we show how to define the rank function in a general way that does not require any deep understanding of the pipelined machine; in fact, it is much simpler to define the rank function than it is to define how the machine is flushed. As a result, we are left with a formula in the CLU logic and can use UCLID to obtain a CNF formula, which we then check with a SAT solver.

For future work, we plan to explore how to connect UCLID with the ACL2 theorem proving system [7, 8]. This will allow us to use ACL2 for efficient simulation and advanced debugging. In addition, we plan to explore methods for verifying larger instructions sets more efficiently than is currently possible with either approach alone.

References

[1] R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *CAV'99*, vol. 1633 of *LNCS*, pages 470–482. Springer, 1999.

[2] R. E. Bryant, S. K. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *CAV'02*, vol. 2404 of *LNCS*, pages 78–92. Springer, 2002.

[3] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV'94*, vol. 818 of *LNCS*, pages 68–80. Springer, 1994.

[4] L. Clark, E. Hoffman, J. Miller, M. Biyani, Y. Liao, S. Strazdus, M.Morrow, K. Velarde, and M. Yarch. An embedded 32-bit microprocessor core for low-power and high-performance applications. *IEEE Journal of Solid-State Circuits*, 36(11):1599–1608, 2001.

[5] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In *CAV'99*, vol. 1633 of *LNCS*. Springer, 1999.

[6] J. K. Huggins and D. V. Campenhout. Specification and verification of pipelining in the ARM2 RISC microprocessor. *ACM Transactions on Design Automation of Electronic Systems*, 3(4):563–580, 1998.

[7] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.

[8] M. Kaufmann and J. S. Moore. ACL2 homepage. See URL `http://www.cs.utexas.edu/users/-moore/acl2`.

[9] S. Lahiri, S. Seshia, and R. Bryant. Modeling and verification of out-of-order microprocessors using UCLID. In *Formal Methods in Computer-Aided Design*, vol. 2517 of *LNCS*, pages 142–159. Springer, 2002.

[10] P. Manolios. Correctness of pipelined machines. In *Formal Methods in Computer-Aided Design*, vol. 1954 of *LNCS*, pages 161–178. Springer, 2000.

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

[12] P. Manolios. A compositional theory of refinement for branching time. In *CHARME'03*, vol. 2860 of *LNCS*, pages 304–318. Springer, 2003.

[13] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV'98*, vol. 1427 of *LNCS*, pages 110–121. Springer, 1998.

[14] P. Mishra and N. Dutt. 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.

[15] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *DAC'01*, pages 530–535, 2001.

[16] V. A. Patankar, A. Jain, and R. E. Bryant. Formal verification of an ARM processor. In *Twelfth International Conference On VLSI Design*, pages 282–287, 1999.

[17] L. Ryan. Siege homepage. See URL `http://www.cs.-sfu.ca/~loryan/personal`.

[18] J. Sawada. *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/`.

[19] J. Sawada. Verification of a simple pipelined machine model. In M. Kaufmann, P. Manolios, and J. S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 137–150. Kluwer Academic Publishers, June 2000.

[20] S. K. Srinivasan and M. N. Velev. Formal verification of an Intel XScale processor model with scoreboarding, specialized execution pipelines, and imprecise data-memory exceptions. In *MEMOCODE'03*, pages 65–74, 2003.

[21] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, pages 52–64, Sept. 1990.

[22] M. N. Velev and R. E. Bryant. Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. In *DAC'00*, pages 112–117. ACM Press, 2000.