Refinement Maps for Efficient Verification of Processor Models

Panagiotis Manolios, Sudarshan K. Srinivasan

Refinement Maps for Efficient Verification of Processor Models*

Panagiotis Manolios
College of Computing
Georgia Tech, Atlanta, GA 30318

Sudarshan K. Srinivasan
School of Electrical & Computer Engineering
Georgia Tech, Atlanta, GA 30318

Abstract

While most of the effort in improving verification times for pipeline machine verification has focused on faster decision procedures, we show that the refinement maps used also have a drastic impact on verification times. We introduce a new class of refinement maps for pipelined machine verification, and using the state-of-the-art verification tools UCLID and Siege we show 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.

1. Introduction

We show that the refinement maps used for pipelined machine verification can have a drastic impact on verification time and therefore should be studied as first-class objects. In this paper, 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.

* This research was funded in part by NSF grants CCF-0429924 and IIS-0417413.

We automatically and efficiently verify the pipelined machines described in this paper by showing that they have exactly the same infinite executions as the machines defined by the corresponding instruction set architectures, up to stuttering. This is accomplished by constructing a WEB-refinement proof, which implies that the pipelined machine satisfies exactly the same $\mathrm{CTL}^* \setminus \mathsf{X}$ properties satisfied by the instruction set architecture [10]. Thus, we verify both safety and liveness properties of the pipelined machine models we study. Automation is attained by expressing the WEB-refinement proof obligation in the logic of Counter arithmetic with Lambda expressions and Uninterpreted functions (CLU), which is a decidable logic [3]. We use the tool UCLID [8] to transform the CLU formula into a CNF (Conjunctive Normal Form) formula, which we then check with the SAT solver Siege [12].

We now selectively review previous work on pipelined machine verification that is directly related to our work. Burch and Dill showed how to automatically compute the abstraction function using flushing [4] and gave a decision procedure for the logic of uninterpreted functions with equality and Boolean connectives. Another, more efficient decision procedure was given in [2]. The work was further extended in [3], where a decision procedure for the CLU logic that exploits optimized encoding schemes [15] is given. The decision procedure is implemented in UCLID, which has been used to verify out-of-order microprocessors [8] and which we use to verify the models presented in this paper. The notion of correctness for pipelined machines that we use was first proposed in [9], and is based on WEB-refinement [10]. The first proofs of correctness for pipelined machines based on WEB-refinement were carried out using the ACL2 theorem proving system [6, 7]. The advantage of using a theory of refinement over using the Burch and Dill notion of correctness, even if augmented by a "liveness" criterion, is that deadlock may avoid detection with the Burch and Dill approach [9], whereas it follows directly from the WEB-refinement approach that deadlock (or any other liveness problem) is ruled out. In [11], it is shown how to automatically verify safety *and liveness* properties of pipelined machines using WEB-refinement. Theorem proving approaches include [13, 14, 5].

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)
1530-1591/05 $ 20.00 IEEE

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 describe the pipelined machines we use in the paper and how we model the novel aspects of these machines using UCLID. In Section 4, we present experimental data measuring verification time and related statistics for the standard refinement maps. In Section 5, we propose our new refinement maps and analyze their performance experimentally. Everything required to reproduce our results, *e.g.*, machine models, correctness statements, CNF formulas, etc., is available upon request. Conclusions and an outline of future work appear in Section 6.

2. Refinement

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 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.

What exactly do we mean when we say MA refines ISA? We mean that the two systems are *stuttering bisimilar*: for every pair of states w, s such that w is an MA state and $s = r(w)$, we have that 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 is equivalent to σ up to finite stuttering (repetition of states). 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. Of course, reasoning about infinite paths is difficult to automate, and in [10], WEB-refinement, an equivalent formulation is given that requires only local reasoning, involving only MA states, the ISA states they map to under the refinement map, and their successor states.

In [11], it is shown how to automate the proof of WEB-refinement in the context of pipelined machine verification. The idea is to strengthen, thereby simplifying, the WEB-refinement proof obligation; the result is a CLU-expressible formula that guarantees that the two machines satisfy the same formulas expressible in the temporal logic $\mathrm{CTL}^* \setminus \mathsf{X}$, over the state components visible at the instruction set architecture level. $\mathrm{CTL}^* \setminus \mathsf{X}$ is a very expressive temporal logic, allowing one to express both safety and liveness properties. The CLU-expressible formula that implies WEB-refinement follows, where *rank* is a function that maps states of MA 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 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 proof obligation relating s and v is the safety component, and the proof obligation that $\mathit{rank}(v) < \mathit{rank}(w)$ is the liveness component.

3. Processor Modeling and Verification

We use as benchmarks for our experiments various complex pipelined machines whose modeling and verification is described in this section. 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 implement the following instruction types: ALU instructions with register-register and register-immediate addressing modes, loads, stores, and branches. We further extend this base processor model with features such as a pipelined fetch stage, an instruction queue (holding up to 3 instructions), direct mapped instruction and data caches, and a write buffer. The model has a simple branch prediction scheme that always predicts taken. A pipelined machine containing all of the above features is shown in Figure 1.

The modeling of pipelined machines using UCLID is well documented. Here we focus on the new features we have modeled, which include the write buffer and the direct mapped instruction and data caches. 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 Boolean value 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 return the index, tag, and 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 match the contents of the instruction memory and the instruction cache.

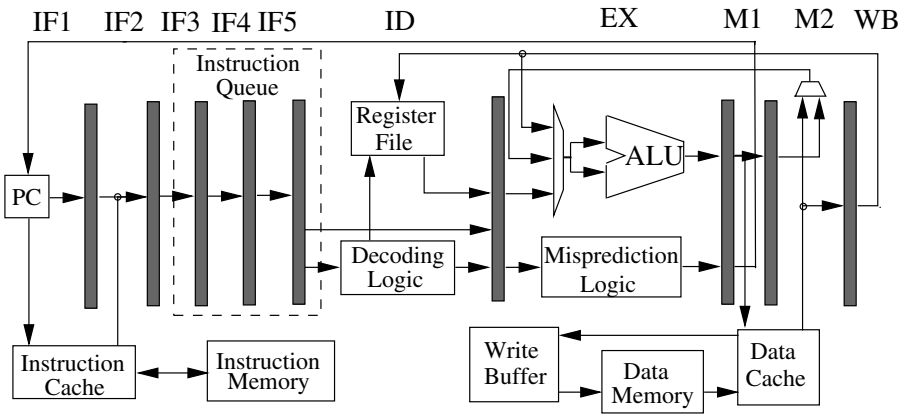

Figure 1. High-level organization of pipelined machine model.

We need an invariant stating that valid instruction cache entries should be consistent with those in 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, it also holds for any successor state.

The data cache is direct mapped and is similarly modeled. Writes to the data memory are write-through and update the data cache. An inductive invariant similar to the instruction cache is required for the data cache, stating that all the valid entries in the data cache are consistent with the data memory.

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 establishing that if we update the data memory with all the valid entries in the write buffer, then we obtain the memory we would have obtained if a write buffer were not used. That is, if D is the data memory and U is the memory state obtained after updating all the valid write buffer entries to D, then $U = R$, where R is a memory that is similar to D except that store instructions directly update R (instead of going through the write buffer).

4. Refinement Maps

Burch and Dill proposed the use of flushing to automatically define the refinement map used to establish the correctness of pipelined machines [4]. 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 [1] to simplify the definition of this refinement map. Also, we need an inductive invariant that we call the "Good MA" 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.

We used both the flushing and the commitment approach to verify the pipelined machines described in Section 3. For all experimental results presented in this paper, we used the UCLID decision procedure (version 1.0) coupled with the siege SAT solver [12] (variant 4), using a 3.06 GHz Intel Xeon, with an L2 cache size of 512 KB.

Table 1 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 instruc-

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 1. Verification statistics for various pipelined machine models.

tion 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 2 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 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 1 and Figure 2 is the general rule of thumb that verification times

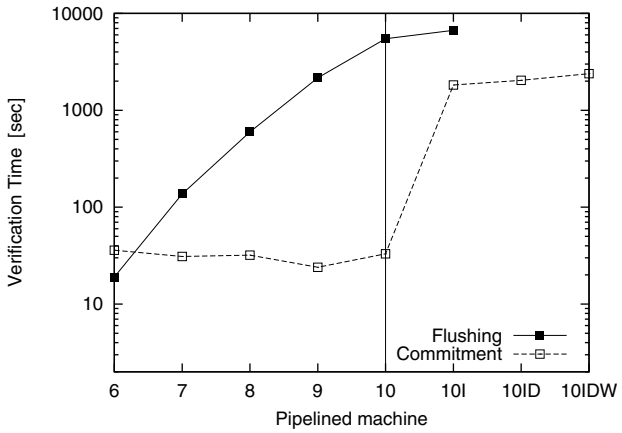

Figure 2. Verification times obtained by first increasing the length of the pipeline and then adding an instruction cache, a data cache, and a write buffer.

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.

5. 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 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.

Based on the above considerations, we define a set of intermediate refinement maps for NBP, 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 NBP is more complex than 10IDW (see Table 1), which only has a simple branch predictor that

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 2. 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.

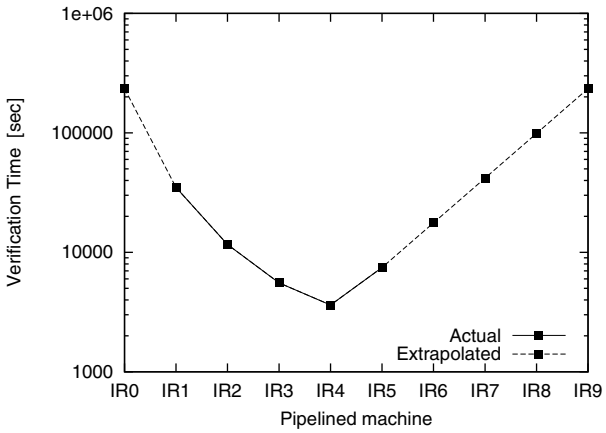

Figure 3. Verification times for a 10-stage processor model with an instruction cache, a data cache, and a write buffer using various refinement maps.

always predicts taken. The refinement maps are IR0, ..., IR9, where IR*i* commits the first *i* latches of the pipeline and flushes the remaining latches. For example, IR0, IR5, and IR9 correspond to pure flushing, commiting 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 IR*i* we define the rank function `rank`*i* to return a pair of natural numbers: the first is the commitment component, computed by `rank`$i_{\mathtt{c}}$, and the second is the flushing component, computed by `rank`$i_{\mathtt{f}}$. These two functions are essentially the standard rank functions used for flushing and commitment [11]: `rank`$i_{\mathtt{c}}$ returns the number of steps required for a new instruction to reach latch $i+1$, the first flushed latch, while `rank`$i_{\mathtt{f}}$ returns the number of steps required to fetch and complete a new instruction for a machine that consists of the latches after latch *i*, *i.e.*, the flushed latches. The less-than ordering on `rank`*i* is the component-wise order.

Table 2 and Figure 3 show the verification times we obtain as we apply the various intermediate refinement maps to NBP. Note that the Y-axis in Figure 3 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 3, the verification time for IR0 is extrapolated, using Table 1 as a guide, and is shown as a dotted line. As *i* increases from 0 to 4, the verification times for IR*i* 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 2 and 3. In addition, by using flushing for the wide, later part of the pipeline, the "Good MA" 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 "Good MA" 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 3, 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 "Good MA" 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 1 and Figure 2 show, verification times are exponential in the length of the pipeline; thus, such a decomposition leads to drastic improvements in ver-

ification times. The reason why we suggest that thought be given to the complexity of the formula for the "Good MA" 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. Conclusion

We have introduced a new class of refinement maps for pipelined machine verification, and using the state-of-the-art 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 — where partially completed instructions are invalidated and the programmer visible components are rolled back to correspond with the last committed instruction— on the latches at the front of the pipeline and the flushing approach — where partially completed instructions are made to complete without fetching any new instructions— 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. Given our results, it seems worthwhile to further investigate the role of refinement maps on verification times. For future work, we plan to apply these techniques to a wider class of pipelined machines.

Acknowledgments. We thank an anonymous referee for suggesting that we change the paper's title.

References

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

[2] R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*, pages 470–482. Springer-Verlag, 1999.

[3] 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 E. Brinksma and K. Larsen, editors, *Computer-Aided Verification–CAV 2002*, volume 2404 of *LNCS*, pages 78–92. Springer-Verlag, 2002.

[4] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

[5] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*. Springer-Verlag, 1999.

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

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

[8] S. Lahiri, S. Seshia, and R. Bryant. Modeling and verification of out-of-order microprocessors using UCLID. In *Formal Methods in Computer-Aided Design (FMCAD'02)*, volume 2517 of *LNCS*, pages 142–159. Springer-Verlag, 2002.

[9] P. Manolios. Correctness of pipelined machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design–FMCAD 2000*, volume 1954 of *LNCS*, pages 161–178. Springer-Verlag, 2000.

[10] 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`.

[11] P. Manolios and S. Srinivasan. Automatic verification of safety and liveness for xscale-like processor models using web refinements. In *Design, Automation, and Test in Europe*, 2004.

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

[13] 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/`.

[14] 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.

[15] S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Design Automation Conference (DAC 03)*, pages 425–430, 2003.