Monolithic Verification of Deep Pipelines with Collapsed Flushing[*]

Roma Kane and Panagiotis Manolios
College of Computing
Georgia Tech, Atlanta, GA 30318
{kroma, manolios}@cc.gatech.edu

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

Abstract

We introduce collapsed flushing, a new flushing-based refinement map for automatically verifying safety and liveness properties of term-level pipelined machine models. We also present a new method for handling liveness that is both simpler to define and easier to verify than previous approaches. To empirically validate collapsed flushing, we ran extensive experiments which show more than an order-of-magnitude improvement in verification times over standard flushing. Furthermore, by combining collapsed flushing with commitment refinement maps, we can monolithically verify complex pipelined machine models with deep pipelines—a salient feature of state-of-the-art microprocessor designs—that previous approaches cannot handle.

1. Introduction

We verify pipelined machine models by showing that they refine their instruction set architecture (ISA). The notion of refinement we use is based on stuttering bisimulation. It guarantees that the pipelined machine and its ISA satisfy the same safety and liveness properties and that they have the same infinite visible executions, up to stuttering. The refinement theorems we prove are parameterized by refinement maps: functions that map pipelined machine states to ISA states. One of the most commonly used refinement maps is the *flushing refinement map*: given a pipelined machine state, it returns an ISA state by completing the partially executed instructions in the pipeline, without fetching any new instructions, and then projecting out the ISA visible components. In general, refinement maps are complex functions, but necessarily so, as they relate machines with multiple instructions in various stages of completion to machines in which instructions complete atomically. A consequence is that refinement maps have a profound impact on verification times, which makes finding efficiently verifiable refinement maps of crucial importance.

In this paper, we introduce collapsed flushing, a variant of flushing that, as our extensive empirical evaluation shows, leads to drastically faster verification times than is possible with standard flushing. We also show how to combine collapsed flushing with commitment, another well-known refinement map that can be thought of as the dual of flushing, as partially executed instructions are invalidated instead of being completed. We show that the resulting refinement map can be used to efficiently reason about complex machine models with deep pipelines. This is an important problem, as recent state-of-the-art microprocessor designs have very deep pipelines, *e.g.*, Intel's$^{\circledR}$ hyper-pipelined technology appearing in the Pentium 4 processor has a pipeline with 31 stages [7].

The refinement proofs are verified automatically using the UCLID system [1], which implements a decision procedure for the logic of Counter arithmetic with Lambda expressions and Uninterpreted functions (CLU). In order to use UCLID, the pipelined machine models we use are defined at the *term-level*: the data path is abstracted away using integers, and combinational circuit blocks, such as the ALU, are abstracted away using uninterpreted functions. The advantage of term-level models is that they can be compiled to CLU expressions. Another issue is that the refinement-based correctness statements cannot be expressed in CLU. Fortunately, we can express the main "core" of the correctness statements in CLU. The core correctness statements are given to the UCLID system, which compiles them to propositional formulas in CNF format. These formulas are then checked using a SAT solver. We use the Siege SAT solver [16], but any SAT solver will do.

The rest of the paper is organized as follows. We start in Section 2 by providing a brief overview of the refinement-based notion of correctness we use. We then turn our attention to refinement maps and describe the commitment refinement map (in Section 3) and the standard and collapsed flushing refinement maps (in Section 4). We show how to combine the commitment and flushing refinement maps in

[*] This research was funded in part by NSF grants CCF-0429924, IIS-0417413, and CCF-0438871.

Section 5. In Section 6, empirical evaluations show that we obtain over an order-of-magnitude improvement in verification times when using collapsed flushing instead of standard flushing. Related work is briefly discussed in Section 7, and we conclude in Section 8.

2. Refinement

We prove that pipelined machines (MA) are correct by showing that they refine their instruction set architecture (ISA). A refinement proof is relative to a *refinement map*, r, a function from MA states to ISA states. Our notion of refinement is based on *stuttering bisimulation*: 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 proofs 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 the following CLU-expressible formula.

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

In the formula above `ISA-step` is the function that steps the `ISA` machine once, `MA-step` is the function that steps the `MA` machine once, and *rank* is a function that maps pipelined machine states to the natural numbers. The proof obligation relating s and v can be thought of as the safety component, and the proof obligation that $rank(v) < rank(w)$ can be thought of as the liveness component.

3. Commitment

In this section, we give an overview of the commitment refinement map. The idea is to invalidate the partially executed instructions in the pipeline and to undo any effects these instructions had on the programmer visible components. This is accomplished with the use of history variables, variables that record past values of state components. The rank function is defined as the number of steps required to commit an instruction, which is the length from the end of the pipeline to the first valid pipeline latch.

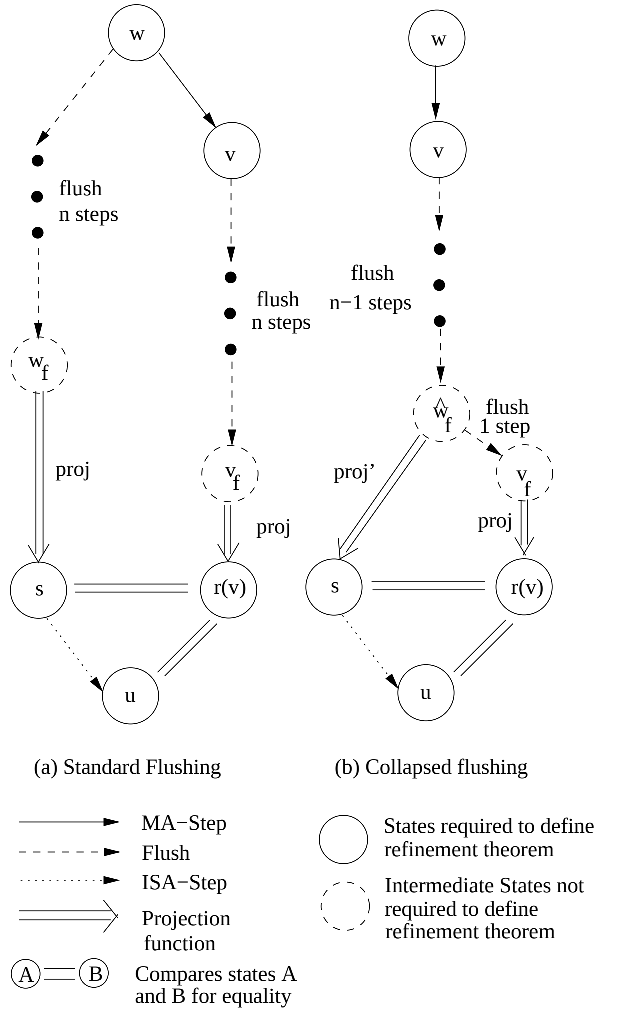
The commitment approach requires an invariant that characterizes the set of reachable states. Two methods of doing this have been investigated: the "Good MA" invariant, and, more recently, the Greatest Fixpoint (GFP) invariant. The GFP invariant is easier to use and gives rise to drastic reductions in verification times over the "Good MA" approach [13]. The GFP invariant characterizes the set of states that are n steps away from an arbitrary state, where n is the number of steps required to replace all partially executed instructions in the pipeline with new instructions from the instruction memory. For the pipelined machines we consider, n is the number of steps required to flush the pipeline. The GFP invariant is an invariant by definition and does not require an invariant proof. In contrast, the "Good MA" approach requires establishing an invariant, which accounts for about 98% of the total verification time.

4. Collapsed Flushing

In this section, we describe collapsed flushing, an implementation of the flushing refinement map that leads to faster verification times over previous methods. The use of flushing as a refinement map was proposed by Burch and Dill [3]. As mentioned previously, flushing can be thought of as the dual of commitment, as partially executed instructions in the pipeline are completed (without fetching any new instructions) instead of being invalidated.

In Figure 1(a) we represent the refinement theorem based on standard flushing as a graph we call the *refinement graph*. The nodes of the graph are variables whose names match the ones given in Section 2; the edges correspond to symbolic simulation steps, flushing steps, or projections. Pipelined machine state w is flushed for n steps, resulting in flushed state w_f, where n is the number of steps required to invalidate all of w's pipeline latches. The ISA state returned by the flushing refinement map is s, the state obtained by projecting out the ISA components of w_f. State u is obtained by stepping state s, and state v is obtained by stepping w. Flushing state v gives us state v_f, and projecting out the ISA components gives us state $r(v)$. The safety component of the refinement theorem compares $r(v)$ with s and u. The liveness component depends on the ranks of w and v. Thus, the refinement theorem depends only on states w, v, s, u, and $r(v)$, nodes depicted with a solid circle in Figure 1.

The verification times for the flushing method depend on two factors. The first factor is number of symbolic simulation steps required to reach u from w. We call this factor the *flushing length*. If n is the number of symbolic simulation steps required to flush the pipelined machine, then the flushing length is $n+1$, as can be seen from the refine-



(a) Standard Flushing

(b) Collapsed flushing

Figure 1. Implementation of standard and collapsed flushing refinement maps.

ment graph in Figure 1(a). The number of steps required to flush the pipelined machine depends on the pipelined machine under consideration, and it is an inherent parameter of the flushing refinement map. Therefore, there is no way to reduce the flushing length without abandoning the use of the flushing refinement map.

The second factor is the *state distance*, the number of symbolic simulation steps separating u from $r(v)$. This is approximately the length of the shortest path between u and $r(v)$ in the refinement graph, when viewed as an undirected graph. As can be seen from Figure 1(a), the state distance for standard flushing is $2n+2$. The intuition as to why this metric is related to verification times is that the statements $u = r(v)$ and $s = r(v)$ are quite complex, each requiring about $2n$ symbolic simulation steps to state.

We now describe collapsed flushing, depicted in Figure 1(b). The insight is that we can reduce the state distance from $2n+2$ to 2. Consider the state $\hat{w}_f$, obtained by stepping w once, to obtain v, and then flushing v for $n-1$ steps. If no new instruction is fetched during the initial step (as is the case during a stall or a branch mispredict) then $\hat{w}_f$ is exactly w_f. Otherwise, we can obtain w_f from $\hat{w}_f$ by using history variables to factor out any effect that the instruction fetched from the transition to v has on the programmer visible components. That is, w_f can be obtained by slightly modifying the process of computing v_f. This allows us to collapse the two flushing computations arising in the implementation of the standard flushing refinement map into one, which is why we name this method collapsed flushing. Figure 1(b) shows that the state distance is 2, improving upon the $2n+2$ value for standard flushing. We validate these intuitions in Section 6, where we show empirically that collapsed flushing leads to much faster verification times and scales better than standard flushing.

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

In the standard flushing method, the rank of a pipelined machine state is defined as the number of steps required to fetch an instruction that eventually completes. For the collapsed flushing method, we use an alternate rank function that can be easily implemented and that leads to faster verification times. The new rank function is defined as the number of steps required to flush a pipelined machine state. To compute this for v, we simply determine how many flushing steps are needed before all pipelined latches are invalid. Since we step w before flushing it (see Figure 1(b)), the rank of w is the number of steps required before all pipelined latches are either invalid or contain a tagged instruction (only the step from w to v can lead to a tagged instruction).

5. Intermediate Refinement Maps

In this section, we give a brief description of intermediate refinement maps (IRs) and explain how we use collapsed flushing to define IRs. Intermediate refinement maps (IRs) are a relatively new class of refinement maps, obtained by combining flushing and commitment [15]. An IR is defined by choosing a reference point (a stage in the pipelined machine), and committing all the pipeline latches before the

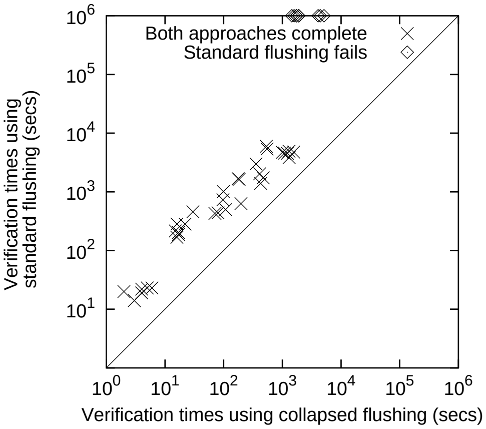


Figure 2. A comparison of standard and collapsed flushing based on verification times.

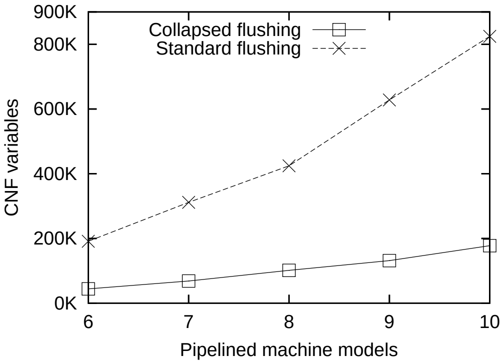


Figure 3. A comparison of standard and collapsed flushing based on the number of CNF variables generated.

reference point and flushing all the pipeline latches after the reference point. IRs result in drastic reductions in verification times over both flushing and commitment as they give rise to two simpler problems, each roughly half the complexity of the original. The first verification problem corresponds to the part of the pipeline being committed, and the second verification problem corresponds to the part of the pipeline being flushed. The best choice of reference point is one that leads to roughly the same complexity for the two resulting problems. Thus, the reference point is usually chosen to be close to the middle of the pipeline. The rank function of an IR (used for checking liveness) is defined as a pair of natural numbers computed by functions $\texttt{rank}i_\texttt{f}$ for the flushing component and $\texttt{rank}i_\texttt{c}$ for the commitment component. The functions $\texttt{rank}i_\texttt{f}$ and $\texttt{rank}i_\texttt{c}$ are essentially the ranks for flushing and commitment, respectively. The less-than ordering for the rank of the IR is defined as the lexicographic ordering with priority given to $\texttt{rank}i_\texttt{f}$.

We describe how to define the IR obtained by combining GFP-based commitment with collapsed flushing. In the following discussion, we refer to the pipeline latches that are committed and flushed as the commit latches and the flush latches, respectively. We require an invariant for the commit latches and it is based on the GFP invariant: starting from an arbitrary state, we step the machine for the number of steps required to flush the commit latches. The flush latches are also stepped, as the commit latches depend on the flush latches, but once this process is finished, we assign arbitrary values to the flush latches. This defines the IR invariant.

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

6. Experimental Results

In this section, we present our empirical evaluation of collapsed flushing, which is based on an extensive set of experiments. To summarize, we found that using collapsed flushing gives an order-of-magnitude improvement in verification time when compared with standard flushing. We also show that the CNF files generated when using collapsed flushing are much smaller than when using standard flushing. Both observations validate our analysis of collapsed flushing in Section 4. In the second set of experiments, we show that by using intermediate refinement maps based on the combination of collapsed flushing with GFP-based commitment, we can monolithically verify pipelines that are too deep to verify with the best previously known monolithic approach, which uses intermediate refinement maps based on standard flushing and GFP-based commitment.

For the experiments, we used and extended the pipelined machine models described in [14]. These models contain branch prediction mechanisms, instruction caches, data caches, write buffers, and instruction queues. They were formally verified using the UCLID decision procedure (Version 1.0) along with the Siege SAT Solver [16] (variant 4), using a 3.06 GHz Intel Xeon with an L2 cache size of 512 KB.

Figure 2 compares collapsed flushing with standard

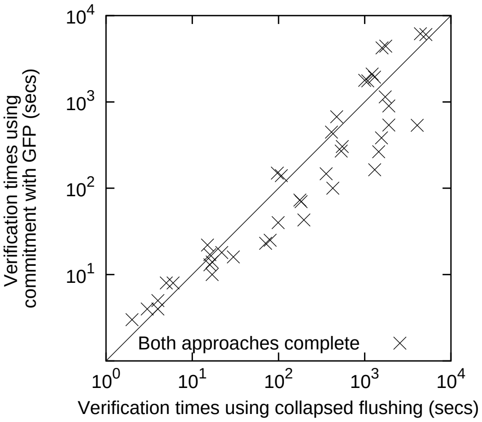


Figure 4. A comparison of verification times for collapsed flushing and GFP-based commitment.

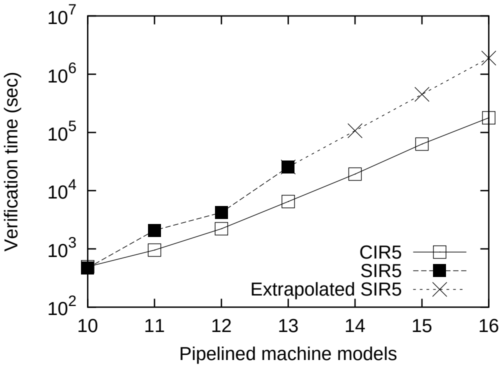


Figure 5. A comparison of verification times for CIR5 and SIR5, defined using collapsed and standard flushing, respectively.

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

Our analysis in Section 4 shows that the complexity of pipelined machine verification problems is greatly reduced when standard flushing is replaced by collapsed flushing, because of the differences in state distance. As a metric of the complexity of these problems, we use the number of CNF variables generated. In Figure 3, we plot the number of CNF variables generated when verifying pipelined machines of varying length for both standard and collapsed flushing. Recall, that for the machine models we consider, the number of flushing steps required to define either standard or collapsed flushing is the same and is directly proportional to the length of the pipeline. From the figure, it can be seen that as the length of the pipeline increases, the CNF variables generated for standard flushing rapidly increase, whereas the increase for collapsed flushing is more modest. The reason for this, as explained in Section 4, is that the state distance for standard flushing depends linearly on the number of steps required to flush the machine, but remains a constant for collapsed flushing. Therefore, collapsed flushing scales much better than standard flushing as the length of the pipeline increases, and it can even handle problems that standard flushing cannot.

In Figure 4, we compare collapsed flushing with GFP-based commitment on the same 42 pipelined machine models used in Figure 2. As can be seen from the scatter plot, the two approaches are comparable.

A major benefit of collapsed flushing can be seen when it is combined with commitment (GFP) to define intermediate refinement maps (IRs) as shown in Figure 5, where we compare IRs defined using commitment (GFP) and collapsed flushing (CIRs) with IRs defined using commitment (GFP) and standard flushing (SIRs). IRs are effective for handling problems that are beyond the scope of pure flushing or commitment refinement maps, such as deep pipelines. For the experiments, we use IR5, which is the IR obtained by committing the first 5 pipeline latches and flushing all other pipeline latches. The *x*-axis shows pipelined machine models obtained by increasing the number of stages from 10 to 16 for a machine containing features such as a branch prediction mechanism, instruction and data caches, and write buffers. Notice that the *y*-axis is a logarithmic scale. From the figure, it can be seen that SIR5 is not able to handle machine models with pipelines that have more than 13 stages. For these models, we have extrapolated the verification times using the average slope of the SIR5 models that could be verified. CIR5 scales better as we increase the number of pipeline stages and is able to handle pipelines with 16 stages (and beyond). We note that some modern microprocessors have very deep pipelines, *e.g.*, Intel's Pentium 4 processor, with hyper-pipelined technology, has 31 stages [7].

7. Related Work

We briefly review previous work on pipelined machine verification that is directly related to our work. Burch and Dill introduced flushing and gave a decision procedure for the logic consisting of boolean connectives, equality, and uninterpreted functions [3]. Several variants of flushing have been previously considered. One example is controlled flushing, an implementation of the flushing refinement map that uses a fixed stalling and flushing pattern, leading to simpler formulas and faster verification times [2]. A second example is incremental flushing, which uses an inductive argument, making it difficult to apply, *e.g.*, the authors conclude that the effort required to deductively justify the proof decompositions offsets the benefits obtained [8]. Also note that neither of these approaches deals with liveness. Recent work on compositional methods [12] can handle deep pipelines, but several verification steps are required. There are also theorem proving methods [17, 6] that can be used to verify deep pipelines, but they often require extensive expert user guidance.

A complimentary approach to extending the complexity of pipelined machines that can be handled automatically has focused on decision procedures. This includes the work on UCLID [9], but we expect recent advances in decision procedures to provide even more significant improvements (*e.g.*, see [5, 4]).

8. Conclusion

We have introduced collapsed flushing, a new refinement map based on flushing that results in about an order-of-magnitude improvement in verification times over standard flushing. We also presented a new, simpler, and easier-to-verify rank function, which is used for handling liveness. We showed how to obtain intermediate refinement maps by combining collapsed flushing with GFP-based commitment. These maps allowed us to extend the reach of monolithic pipelined machine verification, enabling the verification of deep pipelines. The utility of collapsed flushing was empirically validated with an extensive set of experiments on a benchmark suite containing a large number of pipelined machines.

References

[1] 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.

[2] J. R. Burch. Techniques for verifying superscalar microprocessors. In *Design Automation Conference (DAC '96)*, pages 552–557, Las Vegas, Nevada, June 1996. ACM Press.

[3] 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.

[4] L. de Moura. Yices homepage. See URL `http://fm.csl.sri.com/yices`.

[5] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

[6] 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.

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

[8] R. B. Jones, J. U. Skakkebæk, and D. L. Dill. Formal verification of out-of-order execution with incremental flushing. *Formal Methods in System Design, Special Issue on Microprocessor Verification*, 20(2):139–158, Mar. 2002.

[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 (FMCAD'02)*, volume 2517 of *LNCS*, pages 142–159. Springer-Verlag, 2002.

[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, DATE'04*, pages 168–175, 2004.

[12] P. Manolios and S. Srinivasan. A complete compositional reasoning framework for the efficient verification of pipelined machines. In *ICCAD-2005, International Conference on Computer-Aided Design*, pages 863–870, 2005.

[13] P. Manolios and S. Srinivasan. A computationally efficient method based on commitment refinement maps for verifying pipelined machines models. In *ACM-IEEE International Conference on Formal Methods and Models for Codesign*, pages 189–198, 2005.

[14] P. Manolios and S. Srinivasan. A parameterized benchmark suite of hard pipelined-machine-verification problems. In D. Borrione and W. Paul, editors, *Correct Hardware Design and Verification Methods (CHARME'05)*, LNCS, pages 363–366. Springer-Verlag, 2005.

[15] P. Manolios and S. Srinivasan. Refinement maps for efficient verification of processor models. In *Design Automation and Test in Europe, DATE'05*, pages 1304–1309, 2005.

[16] L. Ryan. Siege homepage. See URL `http://www.cs.sfu.ca/ ∼loryan/personal`.

[17] 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.