Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2010, Article ID 515021, 7 pages
doi:10.1155/2010/515021

Research Article

Optimization Techniques for Verification of Out-of-Order Execution Machines

Sudarshan K. Srinivasan

Department of Electrical & Computer Engineering, North Dakota State University, Fargo, ND 58105, USA

Correspondence should be addressed to Sudarshan K. Srinivasan, sudarshan.srinivasan@ndsu.edu

Received 22 March 2010; Accepted 17 August 2010

Academic Editor: Dhiraj K. Pradhan

We develop two optimization techniques, *flush-machine* and collapsed flushing, to improve the efficiency of automatic refinement-abased verification of out-of-order (ooo) processor models. Refinement is a notion of equivalence that can be used to check that an ooo processor correctly implements all behaviors of its instruction set architecture (ISA), including deadlock detection. The optimization techniques work by reducing the computational complexity of the refinement map, a function central to refinement proofs that maps ooo processor model states to ISA states. This has a direct impact on the efficiency of verification, which is studied using 23 ooo processor models. *Flush-machine*, is a novel optimization technique. Collapsed flushing has been employed previously in the context of in-order processors. We show how to apply collapsed flushing for ooo processor models. Using both the optimizations together, we can handle 9 ooo models that could not be verified using standard flushing. Also, the optimizations provided a speed up of 23.29 over standard flushing.

1. Introduction

Many main stream microprocessor architectures are based on the out-of-order (ooo) execution model, which is designed to maximize the number of instructions that can be processed in parallel given the hardware resource limitations. The control complexity of such designs is however quite involved and more easily prone to deadlock errors. Test simulations are primarily used in industry to guarantee correctness of such designs. The simulation-based approach is however far from exhaustive. Property-based verification approaches are also widely used, but suffer from the similar problems as a large number of hard to write properties are required to describe the correct behavior of such designs [1].

The formal verification of ooo processor models has therefore been an active area of research [2–10]. The focus of these approaches has been to verify safety, that is, to check that if the ooo processor model makes progress, then that progress is correct as specified by the instruction set architecture (ISA) of the processor. In these verification approaches for ooo processors however, deadlock detection (otherwise known as liveness checking) has either been ignored or not automated. In previous work, we have developed automatic formal methods to check for both safety and liveness for ooo processor models [11]. In this work, we present two optimization techniques that drastically improve the efficiency of verification.

Dynamic verification methods [12] have also been proposed, where the idea is to use dynamic hardware checkers to verify the computations of the more complex optimized processor core. Formal verification techniques have been found to catch deep bugs and improve coverage of functional verification [13]. Dynamic verification can be used in conjunction with formal verification to improve reliability of the system. However, note that the algorithms and implementations used for dynamic verification are prone to bugs as well and have to be verified.

The notion of correctness that we use for out-of-order processors is based on Well-Founded Equivalence Bisimulation (WEB) refinement. Refinement is a notion of equivalence that can be used to verify that an implementation system (ooo processor model) correctly implements all behaviors (including both safety and liveness) of its specification system (ISA). Refinement proofs are relative

to a refinement map r, which is a function that maps implementation states to specification states. A detailed description of the theory of refinement can be found in [14]. It is enough to prove the following correctness formula to show that an out-of-order processor model correctly implements its ISA. In the formula below, IMPL denotes the set of implementation states, Istep is a step of the implementation machine, and Sstep is a step of the specification machine. rank, used for deadlock detection, is a witness function from implementation states to natural numbers whose value decreases when there is stutter (phenomena where the implementation takes multiple steps to match a single step of the specification).

Definition 1 (Core WEB refinement correctness formula).

$$\begin{aligned}\langle \forall w \in \text{IMPL} :: \quad & s = r(w) \ \wedge \ u = \texttt{Sstep}(s) \wedge, \\ & v = \texttt{Istep}(w) \wedge u \neq r(v) \\ \longrightarrow \ & s = r(v) \wedge \text{rank}(v) < \text{rank}(w)\rangle.\end{aligned} \tag{1}$$

To verify that an ooo processor model works correctly, a refinement map and a rank function—that can be used to establish a refinement relation (as given by the correctness formula above) between the ooo model and its ISA—has to be computed. Once these functions are available, the correctness formula can be stated and checked automatically using a decision procedure. Thus the verification approach can be automated, if easy-to-use methods are available to compute refinement maps and rank functions that can relate ooo models with their ISA specifications. Two such methods were developed in previous work [11], one based on flushing [15] and the other based on commitment [16]. The idea with flushing is to push the partially executed instructions in the reorder buffer of the ooo modeling forward and force them to complete without fetching any new instructions. After all the instructions are completed, the programmer visible components of the ooo model (program counter, instruction memory, and register file) are projected out to give an ISA state. The commitment approach can be thought of as the dual of flushing, where the partially executed instructions are pushed back until the reorder buffer is empty.

The refinement map is a complex function, and it is one of the primary contributors toward the computational complexity of the refinement-based verification approach. In this paper, we present two techniques, *flush-machine* and collapsed flushing, for optimizing flushing-based refinement maps. The *flush-machine* is a novel approach described in Section 3. Collapsed flushing (described in Section 4) was developed in [17], where it was applied to verify in-order processors. Here, we show how to apply collapsed flushing for ooo models. The efficiency of these methods are studied using 23 ooo processor models (described in Section 2). The results are given in Section 5. Overall we find that when both the optimization techniques are employed, we get speed up in verification times of about 23.29 over flushing and 57.48 over commitment. We conclude in Section 6.

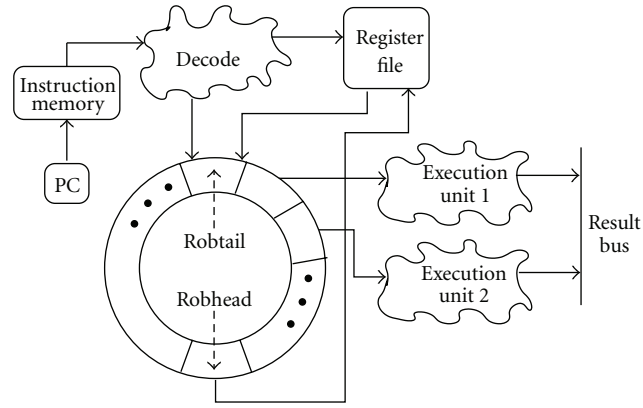


Figure 1: High-level organization of the out-of-order machine model.

2. Out-of-Order Machine Model

The high-level organization of the out-of-order pipelined machine models is shown in Figure 1. The models are defined using the ACL2 programming language at the term level. Note that we have not developed any novel modeling techniques. In fact, our models are based on the out-of-order UCLID model developed by Lahiri et al. [6].

The ooo models have three high-level stages that include fetch and decode, execute, and retire. A rob is employed to enable instructions to execute out-of-program order. The rob is implemented as a circular buffer with a head (robhead) and a tail (robtail) pointer. An instruction is fetched and decoded only if the rob is not full and is allocated to the entry in the rob corresponding to robtail. There are two execution units both of which can process ALU instructions (which is the only instruction type supported by the models). If any of the execution units become free, the oldest instruction whose source operands are available is selected to execute. The execution units are multicycle, fixed-latency units. Once an execution unit has processed an instruction, the result is written back to the rob entry that holds the instruction. The result is also propagated to other instructions in the rob that are dependent on this result. When the instruction reaches the head of the rob it is allowed to retire by writing the result back to the register file.

3. Flush Machine

The flushing refinement map is computed by creating a variation of the ooo model, such that the new model completes instructions in the reorder buffer, but does not fetch any new instructions. We call this new model a flushing step (as opposed to a regular step of the ooo machine). The number of flushing steps required to flush an ooo model state is given by the following equation [11]:

$$n_{ooof} = 1 + (\text{robsize} * \max(\text{eu1}_{\text{lat}}, \text{eu2}_{\text{lat}}, \ldots, \text{eu}m_{\text{lat}})). \tag{2}$$

In the above formula $\text{eu}i_{\text{lat}}$ indicates the latency of the ith execution unit and robsize is the number of entries in

the reorder buffer. The time and memory resources required for verification directly depends on $n_{\mathrm{ooo}f}$. The idea with the *flush-machine* is to construct a simple machine that executes and completes only one instruction every cycle from the front of the rob and therefore, does not use the complex control mechanism of the ooo implementation. This simpler machine can now be used to compute the flushing refinement map, instead of the original complex ooo implementation, which will result in a flushing refinement map with reduced computational complexity. The following two modifications can be used to construct the *flush-machine* from the ooo machine.

(i) Replace multicycle functional units with a single-cycle equivalent. For example, replace a multicycle ALU with a function that performs the ALU operation in one cycle. The idea of replacing multicycle functional units with a single-cycle equivalent in the flush function has been explored before in the context of in-order processors [18, 19]. We use this idea to deal with multicycle functional units in the *flush-machine* approach in the context of ooo execution based on the employment of a rob.

(ii) Remove complex control logic that determines the next available instruction to execute. Instead, execute only the instruction at the head of the reorder buffer. If the instruction in the rob head has completed execution, then, retire this instruction and execute the next instruction in the rob, if it has not completed.

The number of flushing steps required by the *flush-machine* that employs both the above modifications is given by the following equation:

$$n_{\mathrm{ooo}fm} = 1 + \mathrm{robsize}. \tag{3}$$

Note that $n_{\mathrm{ooo}fm}$ is independent of the latency of the execution units. Using the *flush-machine*, we get a drastic improvement in the number of flushing steps required to compute the flushing refinement map, the speed up in the number of flushing steps is in fact close to the latency of the execution unit with the maximum latency $(\max(\mathrm{eu1}_{\mathrm{lat}}, \mathrm{eu2}_{\mathrm{lat}}, \ldots, \mathrm{eu}m_{\mathrm{lat}}))$. Also, the flush-machine does not have the complex control logic of the out-of-order implementation, thereby reducing the complexity of the refinement map. As can be seen from Section 5, this has a drastic impact on the efficiency of the verification approach.

To prove refinement, a rank function is also required and is used for deadlock detection. As state earlier, rank is a function from ooo processor model states to natural number, whose value decreases when there is stutter, that is, multiple steps of the ooo machine is matched by a single step of the ISA. We now give a formula for computing rank for ooo machines, which can be used with the *flush-machine* approach that does not reduce its efficiency (rank_{fm}). The rank itself is defined using the following function (rob latency), which essentially gives the number of cycles required for an instruction in the rob to complete +1. Note that rob latency ignores the presence of other instructions.

$$\text{rob-latency}(\mathrm{rob}_i) = \begin{cases} \neg\text{active} \cdot \mathrm{rob}_i, & 0 \\ \text{valid} \cdot \mathrm{rob}_i, & 1 \\ \mathrm{ex} \cdot \mathrm{rob}_i = 1, & \mathrm{excyc}(1, \mathrm{rob}_i) + 1 \\ & \cdots \\ \mathrm{ex} \cdot \mathrm{rob}_i = m, & \mathrm{excyc}(m, \mathrm{rob}_i) + 1 \\ \text{default}, & \mathrm{exlat}(\mathrm{rob}_i \cdot \mathrm{inst}) + 2. \end{cases} \tag{4}$$

In the above definition of rob-latency, rob_i is the ith entry in the rob. The active function checks if the rob entry is not empty and the valid function checks if the instruction in the rob entry has completed execution and is waiting to be retired. Function exlat takes an instruction as input and gives the latency of the execution unit that can process the instruction. m is the number of execution units. The ex function gives the execution unit that is processing the instruction in the rob entry. Function excyc takes as input an execution unit number and a rob pointer. If the execution unit can process the instruction present in the input rob pointer, then it returns the number of cycles for that execution unit to become available.

The rank_{fm} itself is given by the following equation:

$$\mathrm{rank}_{fm} = \sum_{i=1}^{\mathrm{robsize}} \text{rob-latency}(\mathrm{rob}_i). \tag{5}$$

There are two possible scenarios where stutter can occur in an ooo machine: (1) if the rob is full, then no new instruction is fetched in the next step; (2) an instruction was speculatively executed and the speculation was incorrect (e.g., a mispredicted branch) leading to squashing of all younger instructions in the rob. No new instructions are fetched in the second scenario as well. Therefore, at the ISA level, it seems as if the ooo machine has made no progress leading to stutter. In both the above scenarios, the rank_{fm} value will decrease. Note that the machines we have verified do not incorporate any form of speculation.

4. Collapsed Flushing

Collapsed flushing was introduced in [17] and was used to verify in-order processor models. We show how to use collapsed flushing to verify out-of-order processor models. The idea with collapsed flushing is shown in Figure 2. The use of flushing as a refinement map to state the refinement correctness formula. (Definition 1 is depicted in Figure 2(a)). The processor state w is flushed using n flushing steps of the processor model (*Iflush*) to get state w^f, which is the flushed state corresponding to w (state in which all pipeline latches are empty or invalid). The ISA state s can be obtained from w^f by projecting the programmer visible components. State w is also stepped (*Istep*) to get state v. Flushing v gives the flushing state corresponding to v, v^f, which is used to compute $r(v)$. Stepping ISA state s gives u. The refinement correctness formula can be stated using states w, v, s, u, and $r(v)$.

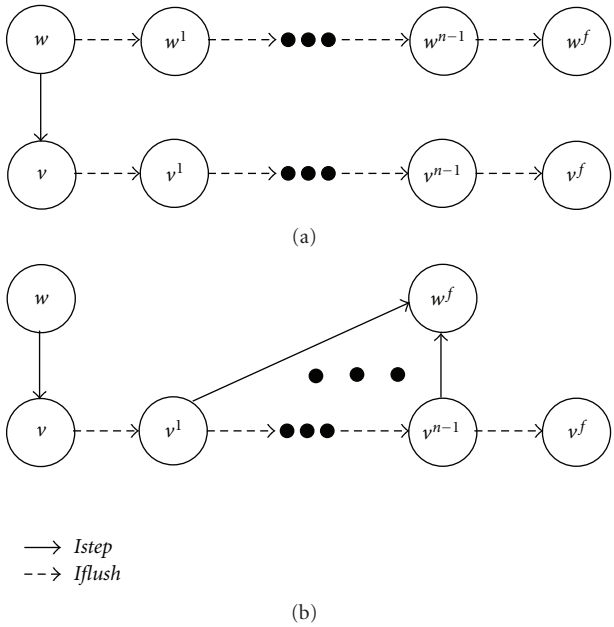


Figure 2: Figure depicts flushing (a) and collapsed flushing (b).

The idea with collapsed flushing is that since a flushing step of the processor model is very similar to a regular step, we could use states $v^1, \ldots, v^{n-1}, v^f$ to compute w^f as depicted in Figure 2(b). This would be possible if a mechanism was used to keep track of the newly fetched instruction when stepping from w to v. Note that in this step an instruction need not necessarily be fetched. Using this optimization halves the number of flushing steps required to state the correctness formula. Since the flushing step is a computationally expensive function, this optimization results in significant improvement in the efficiency of verification as can be seen from Section 5.

To implement collapsed flushing for out-of-order processor models, we include a tag filed with every rob entry. The tag fields of all rob entries in w are reset. In state v, the tag field of the rob entry that houses the new instruction when stepping from w to v is set. We now show how to compute state w^f from states $v^1, \ldots, v^{n-1}$, and v^f.

Since our models do not use any form of speculation, the program counter is never updated by instructions in the rob. Therefore, the program counter (pc) of state w^f is

$$w^f \cdot \mathrm{pc} = v^1 \cdot \mathrm{pc}. \tag{6}$$

The instruction memory (imem) is never updated. Therefore the instruction memory of state w^f is

$$w^f \cdot \mathrm{imem} = v^1 \cdot \mathrm{imem}. \tag{7}$$

Table 1: Verification Times.

OOO	Verification Times [sec]			
Model	Flushing	GFP	FM	FM & CF
2–2	3.87	2.24	1.52	1.26
2–3	6.06	3.71	1.71	1.4
2–4	9.4	6.13	2.12	1.56
2–5	13.15	10.41	2.6	2
3–2	14.51	11.21	4.02	2.91
3–3	26.34	28.93	4.93	3.55
3–4	43.01	64.76	6.02	4.38
3–5	65.8	273.13	7.02	5.09
4–2	73.48	103.67	15.92	9.67
4–3	182.38	680.55	19.56	11.87
4–4	428.5	3,409.96	29.06	15.32
4–5	973.92	TO	33.8	20.01
5–2	OM	1,687.55	96.41	50.28
5–3	OM	TO	127.33	54.61
5–4	OM	OM	141.83	63.45
5–5	OM	OM	159.09	91.85
6–2	OM	OM	592.51	282.56
6–3	OM	OM	951.69	233.62
6–4	OM	OM	973.49	336.54
6–5	OM	OM	1,710.76	540.03
7–2	OM	OM	3,375.68	1,450.23
8–2	OM	OM	3,351.34	1,294.73
9–2	OM	OM	OM	OM

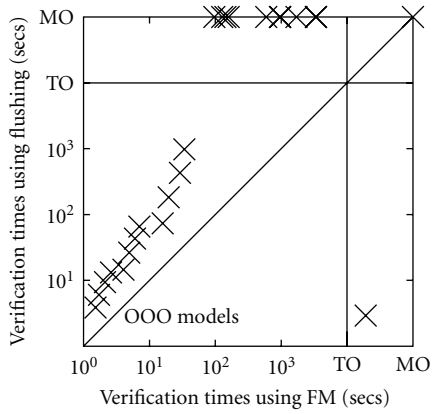
The register file (rf) of state w^f is

$$w^f \cdot \mathrm{rf} = \begin{cases} v^1 \cdot \mathrm{robhead} \cdot \mathrm{tag} \wedge v^1 \cdot \mathrm{robhead} \cdot \mathrm{valid} & v^1 \cdot \mathrm{rf}, \\ \cdots & \\ v^{n-1} \cdot \mathrm{robhead} \cdot \mathrm{tag} \wedge v^{n-1} \cdot \mathrm{robhead} \cdot \mathrm{valid} & v^{n-1} \cdot \mathrm{rf}, \\ \mathrm{default} & v^f \cdot \mathrm{rf}. \end{cases} \tag{8}$$

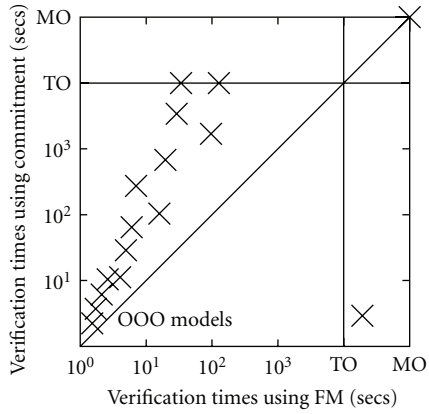
The above formula is based on the observation that when the rob entry holding the tagged instruction becomes the robhead, all other instructions will have been completed and the value of the register file in this state corresponds to the value of the register file after flushing state w (w^f).

5. Results

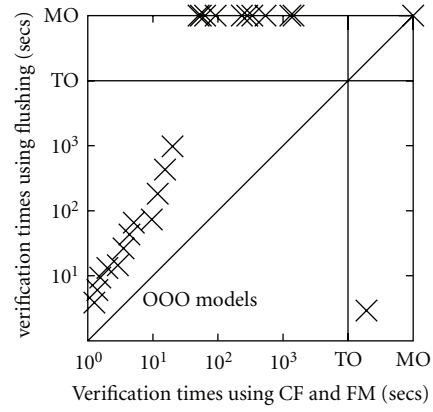
The *flush-machine* approach and collapsed flushing were evaluated using 23 ooo machine models. The models were generated by varying the the size of the reorder buffer and the latency of the execution units. Since the methods are highly automated and the degree of automation is similar, the verification time is the most important parameter for comparison and has been used to evaluate the relative



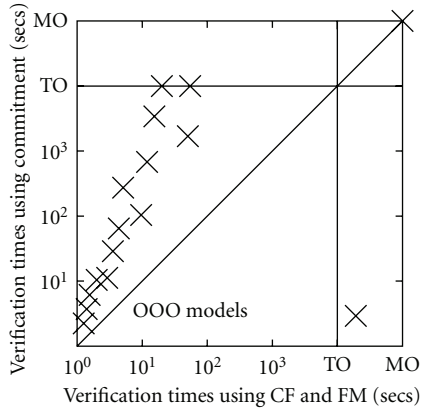
(a) Scatterplot compares the *flush-machine* (FM) approach with standard flushing using 23 ooo pipelined machine models



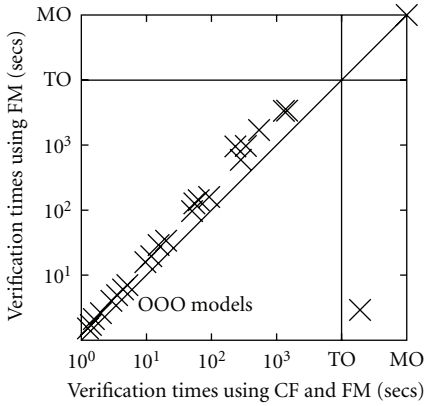
(b) Scatterplot compares the *flush-machine* (FM) approach with standard GFP-based commitment using 23 ooo pipelined machine models



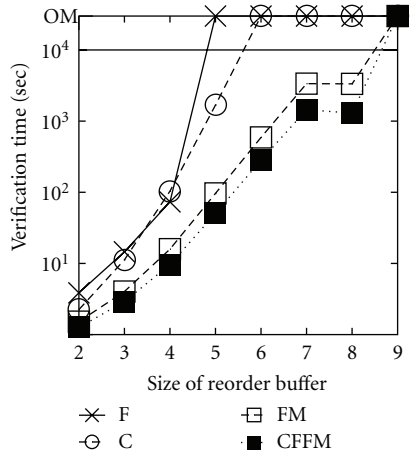
(c) Scatterplot compares the use of both collapsed flushing (CF) and *flush-machine* (FM) with standard flushing based on 23 ooo pipelined machine models



(d) Scatterplot compares the use of both collapsed flushing (CF) and *flush-machine* (FM) with GFP-based commitment on 23 ooo pipelined machine models



(e) Scatterplot compares the use of both collapsed flushing (CF) and *flush-machine* (FM) with FM alone on 23 ooo pipelined machine models



(f) Plot shows the increase in verification time when using flushing (F), commitment (C), *flush-machine* (FM), and collapsed flushing (CF) and FM, when the size of the reorder buffer is increased. All the models have two execution units with a 2-cycle latency each

Figure 3:

efficiency of the various refinement-based methods. The following naming convention was used for the models. A model name is of the form "$r-l$", where "r" indicates the size of the reorder buffer and "l" indicates the latency of the two execution units.

Table 1 shows the verification times for the 23 models using GFP-based commitment, flushing, *flush-machine* (FM), and *flush-machine* and collapsed flushing (CFFM). Verification was performed using the ACL2-SMT system, which is obtained by combining ACL2 (version 3.3) and the Yices decision procedure (version 1.0.10). The experiments were run on a 800 MHz Intel(R) Core(TM)2 Duo CPU with a cache size of 2048 KB. Note that in the table and the graphs in Figure 3, "OM" indicates that ACL2-SMT ran out of memory, and "TO" indicates time out. We set a time-out limit of 5000 seconds.

The optimization techniques are compared with standard flushing and GFP commitment using 4 scatter plots shown in Figures 3(a), 3(b), 3(c), and 3(d). The scatter plot in Figure 3(e) depicts the gains from using collapsed flushing in addition to *flush-machine*. Note that in all the scatter plots in Figure 3, both the X- and Y-axis use logarithmic scales. Overall flushing ran out of memory on 11 benchmarks, and commitment timed out on 2 benchmarks and ran out of memory on 9 benchmarks. FM and CFFM were able to verify all the benchmarks except for 9–2. FM provided a speedup of 14.35 over flushing and 32.91 over commitment. CFFM provided a speedup of 23.29 over flushing and 57.48 over commitment. When collapsed flushing was used in addition to *flush-machine* (*i.e.*, CFFM versus FM) a speedup of 2.59 was obtained over just using FM. Note that the speedups were calculated without taking into account verification times of benchmarks that timed out or ran out of memory on either of the approaches being compared.

Figure 3(f) shows the scalability of each of the approaches by plotting the increase in verification times as the size of the reorder buffer is increased. Note that the Y-axis uses a logarithmic scale. FM and CFFM scale much better than flushing and commitment as expected, but as with monolithic approaches, the verification times increase exponentially as the size of the reorder buffer is increased.

6. Conclusions

We have introduced a novel optimization, *flush-machine*, to improve the efficiency of refinement-based verification of ooo processor models. We have also shown how to use collapsed flushing to verify ooo processor models. The optimizations were studied using 23 models. When using both optimizations, speed ups of 23.29 and 57.48 were obtained over standard flushing and commitment, respectively. Also, using the optimizations, 9 ooo models could be verified that could not be handled using flushing and commitment. However, the scalability of the approach is still an issue, which we plan to address in future work. We also plan to extend these methods to verify ooo models that incorporate speculative execution.

Acknowledgment

The author would like to thank and acknowledge Panagiotis Manolios for discussions on verifying liveness based on refinement when dealing with multicycle functional units. Some initial ideas on this topic emerged from these discussions.

References

[1] G. Yanyan and L. Xi, "Formal verification of out-of-order processor," in *Proceedings of the International Conference on Computer Modeling and Simulation (ICCMS '09)*, pp. 129–135, IEEE, February 2009.

[2] W. A. Hunt Jr. and J. Sawada, "Verifying the FM9801 microarchitecture," *IEEE Micro*, vol. 19, no. 3, pp. 47–55, 1999.

[3] R. Hosabettu, M. Srivas, and G. Gopalakrishnan, "Proof of correctness of a processor with reorder buffer using the completion functions approach," in *Proceedings of the International Conference on Computer Aided Verification (CAV '99)*, N. Halbwachs and D. Peled, Eds., vol. 1633 of *Lecture Notes in Computer Science*, Springer, 1999.

[4] T. Arons and A. Pnueli, "A comparison of two verification methods for speculative instruction execution," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '00)*, vol. 1785 of *Lecture Notes in Computer Science*, pp. 487–502, Springer, March 2000.

[5] D. Kroning, *Formal verification of pipelined microprocessors*, Ph.D. thesis, Universität des Saarlandes, 2001.

[6] S. Lahiri, S. Seshia, and R. Bryant, "Modeling and verification of out-of-order microprocessors using UCLID," in *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD '02)*, vol. 2517 of *Lecture Notes in Computer Science*, pp. 142–159, Springer, 2002.

[7] 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*, vol. 20, no. 2, pp. 139–158, 2002.

[8] M. N. Velev, "Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer," in *Proceedings of the Design, Automation and Test in Europe (DATE '02)*, pp. 28–35, IEEE Computer Society, 2002.

[9] H. I. Shehata and M. Aagaard, "A general decomposition strategy for verifying register renaming," in *Proceedings of the Design Automation Conference (DAC '04)*, S. Malik, L. Fix, and A. B. Kahng, Eds., pp. 234–237, ACM, 2004.

[10] M. N. Velev, "Using automatic case splits and efficient cnf translation to guide a sat-solver when formally verifying out-of-order processors," in *Proceedings of the AMAI Symposium*, 2004.

[11] S. K. Srinivasan, "Automatic refinement checking of pipelines with out-of-order execution," *IEEE Transactions on Computers*, vol. 59, no. 8, pp. 1138–1144, 2010.

[12] T. M. Austin, "DIVA: a dynamic approach to microprocessor verification," *Journal of Instruction-Level Parallelism*, vol. 2, 2000.

[13] B. Bentley, "Validating the Intel Pentium 4 microprocessor," in *Proceedings of the 38th Design Automation Conference*, pp. 244–248, June 2001.

[14] P. Manolios, *Mechanical verification of reactive systems*, Ph.D. thesis, University of Texas, Austin, Tex, USA, August 2001,

http://www.ccs.neu.edu/home/pete/research/phd-dissertation.html.

[15] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Proceedings of the International Conference on Computer Aided Verification (CAV '94)*, vol. 818 of *Lecture Notes in Computer Science*, pp. 68–80, Springer, 1994.

[16] P. Manolios, "Correctness of pipelined machines," in *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD '00)*, W. A. Hunt Jr. and S. D. Johnson, Eds., vol. 1954 of *Lecture Notes in Computer Science*, pp. 161–178, Springer, 2000.

[17] R. Kane, P. Manolios, and S. K. Srinivasan, "Monolithic verification of deep pipelines with collapsed flushing," in *Proceedings of the Design, Automation and Test in Europe, (DATE '06)*, G. G. E. Gielen, Ed., pp. 1234–1239, European Design and Automation Association, Leuven, Belgium, 2006.

[18] M. N. Velev and R. E. Bryant, "Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 112–117, June 2000.

[19] M. N. Velev, "Automatic formal verification of liveness for pipelined processors with multicycle functional units," in *Proceedings of the Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '05)*, D. Borrione and W. J. Paul, Eds., vol. 3725 of *Lecture Notes in Computer Science*, pp. 97–113, Springer, 2005.