Resource Conflict Detection in Simulation of Function Unit Pipelines

Pekka Jääskeläinen, Vladimír Guzma, and Jarmo Takala

Department of Information Technology
Tampere University of Technology
P.O. Box 553
FIN-33101 Tampere
Finland
{pekka.jaaskelainen,vladimir.guzma,jarmo.takala}@tut.fi

Abstract. Processor simulators are important parts of processor design toolsets in which they are used to verify and evaluate the properties of the designed processors. While simulating architectures with independent function unit pipelines using simulation techniques that avoid the overhead of instruction bit-string interpretation, such as compiled simulation, the simulation of function unit pipelines can become one of the new bottlenecks for simulation speed.

This paper evaluates commonly used models for function unit pipeline resource conflict detection in processor simulation: a resource vector based-model, and an finite state automata (FSA) based model. In addition, an improvement to the simulation initialization time by means of lazy initialization of states in the FSA-based approach is proposed. The resulting model is faster to initialize and provides equal simulation speed when compared to the actively initialized FSA. Our benchmarks show at best 23 percent improvement to the initialization time.

1 Introduction

Processor simulators possess different level of accuracy depending on their purpose. Instruction set simulation is mainly used for program verification and development in cases which do not require detailed modeling of timing. More accurate cycle-based simulators can produce cycle counts and utilization statistics for directing processor design space exploration – a process of finding the most suitable processor architecture for the applications at hand. In automated design space exploration of application-specific processors, the number of examined candidate architectures can reach thousands, thus the time it takes to produce the utilization data and cycle counts for each explored architecture can affect the total exploration time dramatically.

Structural hazards are situations in which multiple operations or instructions try to use the same processor resource simultaneously. Commonly, structural hazards result in processor stall cycles in which the processor waits mostly idle for the hazard to resolve. Cycle-accurate simulators detect these stall cycles and model them accurately. At minimum, the stall cycles should be counted and added to the total cycle count. On the other hand, some architectures, such as the Transport Triggered Architectures (TTA) [1] do not provide hardware locking support in case of structural hazards. In this

S. Vassiliadis et al. (Eds.): SAMOS 2007, LNCS 4599, pp. 233–240, 2007.

case, the detection of structural hazards during simulation is a fundamental part of the program verification process.

Simulation of statically scheduled architectures with relatively simple control logic, such as VLIWs and TTAs, concentrates on simulating the data transports between function units and register files, the functionality of operations in function units, and the function unit latencies. In this type of simulators, especially if the simulation overhead of instruction decoding phase is avoided, simulating the function units and their pipelines can become the new bottleneck for simulation speed.

This paper evaluates models to detect function unit pipeline resource conflicts in cycle-accurate simulation: a resource vector based-model and an finite state automata (FSA) based model. Finally, an improvement to the simulation initialization time by means of lazy initialization of states in the FSA-based approach is proposed and evaluated. Using this model, our benchmarks show that up to 23 percent improvement to the simulation initialization time can be achieved.

The rest of paper is organised as follows. Section 2 analyses existing solutions for improving processor simulation speed. Section 3 gives brief overview of common book keeping methods for structural hazard detection during instruction scheduling and simulation. Section 4 describes our test setup, followed by Section 5 with results from the performed experiments. Section 6 concludes the work and outlines future research directions.

2 Related Work

Several research papers discuss the techniques to avoid the instruction bit string interpretation overhead during simulation. These techniques are commonly referred to as "compiled simulation". For example, Shade is a simulator which includes a technique for translating the simulated instructions dynamically to host instructions during simulation and caching the translated instructions for later execution [2]. However, the presented work is a simulator with functional accuracy, as detecting structural hazards and other microarchitectural details required for cycle-accuracy are not discussed.

JIT-CCS technique applies just-in-time (JIT) compilation, common in Java virtual machines, to instruction set simulation. This technique removes the limitation of translating simulator not capable of simulating self-modifying code [3]. Use of JIT techniques for simulation is explored also in DynamoSim, which improves the simulator flexibility by combining interpretive and compiled techniques by compiling only parts of the simulation that benefit the most [4]. The paper also extends the scope of the simulation compilation from basic blocks to traces to exploit better the instruction-level parallelism capabilities of the host processor.

FastSim uses the idea of compiled simulation in detailed out-of-order microarchitectural simulation [5]. The main contribution of the paper is a technique to "memoize" microarchitectural configurations and "fast-forward" the actions to the processor state when the simulation enters a previously executed microarchitectural configuration. The idea is extended in [6] with a language for easy implementation of this type of "fast-forwarding" simulators.

Pees, Hoffman, and Meyr present an architecture description language LISA, which allows generating compiled processor simulators for several architectures automatically [7,8]. The resulting simulators are cycle-accurate thanks to the capabilities of the language to allow detailed modeling of pipeline resources used by the instructions. Similar work is presented in [9] in which ANSI C is used to model the instructions to avoid a new modeling language.

An interesting simulation speedup technique worth noting is "token-level simulation" [10] and "evaluation reuse" [11]. The principle of these techniques is to simulate the program first in functional level for obtaining the basic block traces. Using the basic block traces, the accurate cycle count is produced by evaluating the effects of each basic block to the processor pipeline state but without simulating the actual functionality again since it has already been performed in the previous faster pass. This technique seems very promising for speeding up the collection of the total cycle counts but does not produce cycle-accurate simulation for exact timing or debugging features such as cycle-stepping, due to the separation of the functional and timing simulation.

Literature covering techniques for speeding up processor simulation, in general, is widely available. However, avoiding the bottlenecks in simulation of architectures with independent function unit pipelines is rarely discussed. This paper considers in particular the bottlenecks in simulating such architectures.

3 Structural Hazard Detection in Simulation

This section gives a brief overview of the most common methods for keeping book of structural hazards during simulation or instruction scheduling.

3.1 Resource Vectors

Reservation table is a two-dimensional (2D) table with one dimension representing the machine resources and the other one representing the latency cycles [12,13]. A resource usage is marked by placing 'X' in the table cell at the position of the cycle and the resource. The same information can be represented in a 1D structure called *resource vector*. A column in this vector lists all resources that are reserved at a cycle [14].

When using this table for resource modeling, the simulator keeps book of the occupied resources at each cycle of the simulation in a *composite resource vector*. Before an operation or instruction is to be executed in the simulator, conflicts are detected by comparing the composite resource vector to the resource usage of the candidate operation. In case there are overlapping resource usages between the candidate operation's resource vector and the composite vector, a structural hazard is detected. Otherwise, the composite vector is updated to reflect the resources occupied by the started operation.

3.2 Finite State Automata

The resource vector based structural hazard detection scheme can be refined to a more advanced version by exploiting a Finite State Automaton (FSA) [15] for representing all the legal state transitions in the processor.

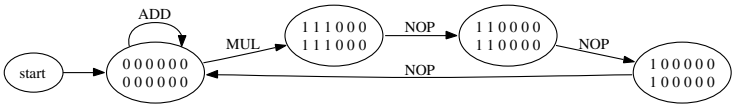

Fig. 1. Resources modeled with a finite state automaton

In the FSA-based approach, each state is represented by a collision matrix, a 2D table S, which contains rows for each operation and as many cycle columns as the longest latency operation or instruction requires. The element $S[o,t]$ is 0 only if operation o does not "collide" when issued t cycles later after entering the state. That is, if $S[o,1] = 0$, the operation o can be issued at the next cycle after entering state S, resulting in a transition to state S'. The collision matrix of the target state is computed by shifting the collision matrix of the starting state to the left (which simulates a cycle advance) and ORing it with the issued operation's collision matrix. [12]

Figure 1 illustrates an example automaton for a function unit with two operations: ADD and MUL. The FSA can be used to quickly detect the legal operation sequences that can be executed by the function unit. For example, in the automaton, it is easy to notice that after executing ADD, it is possible to execute both ADD and MUL but, after executing MUL, three cycles are needed (issue NOPs or stall the processor) before issuing new operations.

The FSA-based conflict detection models are known to be very fast, but their initialization time can be long due to large number of states in the automaton that need to be built based on the operation resource usage patterns. This leads to an optimization to the FSA-based approach which is also evaluated in this paper. One of the evaluated models is an FSA-based model in which the states are built "lazily" the first time they are entered, hoping to reduce the initialization time to a minimum. The optimization is derived from the observation that in many cases only the minority of the states are visited by the simulated program, thus, the construction time for the unused states is wasted.

4 Test Setup

We evaluated different models for function unit resource conflict detection during simulation by implementing them in our TTA simulator [16] and executing synthetic simulations of function units using these models.

The initialization times were evaluated by initializing each model 100 000 times in a row and the total time was measured.

The simulation speed of each model was measured by simulating sequences of operations and by measuring the total real time it took to simulate the operation sequence. Each operation in each function unit was executed in round-robin fashion in successive cycles with total of 10 000 000 operation executions. All resource conflicts reported by the models were caught and ignored.

The measurements were made in a Pentium 4 CPU with 3.4 GHz clock and 1 GB of RAM. The operating system was Ubuntu Linux 6.10, with GNU GCC compiler version

	0	1	2
MUL	R	A	A+W

Fig. 2. Resource vector for multiplication with three resources R, A and W

4.1.1-13ubuntu5. The compiler optimization switch used to compile the models was '-O3'. All the tests were executed under equal overall system load after a fresh boot. Each test was run three times in a row and the best result was picked. Picking the best result instead of, for example, the average, allowed us to evaluate the peak speed each model can reach. However, the differences between the results were negligible.

The following conflict detection models were evaluated:

none. A model without conflict detection. This model simulates only the operation latency, but does not detect if there are conflicting pipeline resource usages between started operations. This model could be used in quick design space exploration.

vectors. The traditional resource vector-based approach for conflict detection. It maintains the composite vector and checks resource conflicts against the composite vector each time an operation is started.

active FSA. Uses an FSA for conflict detection. FSA is fully constructed before starting the simulation. The used construction algorithm is similar to the one presented in [17].

In this model, the automaton is fully constructed before starting the simulation. Therefore, in case of function units with complicated pipeline resource usage patterns, an "state explosion" can happen, which lengthens the simulation initialization. The simulation itself should be very fast as conflicts are detected with a single table lookup.

lazy FSA. Like "active FSA", but the FSA is not fully constructed before starting the simulation. Instead, only the start state is created and other states are created when they are visited for the first time during simulation.

Our hypothesis is that this model should improve startup time when compared to the active FSA model, but the simulation itself might be slower due to the need of checking whether a required state exists and building one if the transition is valid.

Models were evaluated with the following function unit resource usage patterns trying to cover the wide range of function units used in processors.

ALU. Arithmetic-logic unit with 18 integer operations. Latency of each operation is one cycle.

MUL. A single-operation function unit that implements integer multiplication with latency of 3. The operation uses three pipeline resources (symbols R, A, and W) as illustrated in Fig. 2.

FPU. Function unit that models a floating-point unit. Its pipeline matches the one of MIPS R4000 floating-point unit, as described in [18]. The unit includes floating-point operations that share eight different pipeline stages. The double precision floating-point operations range from a simple "absolute value" operation (latency of two) to a long latency operation "square root" (latency of 112).

Table 1. Count of created states in the active FSA model

FU	states
MUL	3
ALU	2
FPU	258

The count of states in FSA affects the initialization time for the actively initialized FSA-based simulation model. State counts for each function unit pipeline model are listed in Table 1.

5 Results

Table 2 lists the startup times for each of the models and Table 3 shows the simulation times. The simulation times do not include the model initialization time, but they do include the time to simulate the actual functionality of the operation.

The startup and simulation times are compared to the model "none" to indicate the slowdown compared to no conflict detection at all. This "baseline" represents an ideal model without any conflict detection overhead.

The results show that the simplest conflict detection model using resource vectors is relatively fast to initialize (still measured a slowdown of 32 to 45 percent), but its simulation speed is at worst about 7.6 times slower than the FSA-based approaches. The FSA-based conflict detection slowed the simulation down about 32 to 59 percent, compared to the model with no conflict detection, while with resource vectors, the slowdown was more drastic, from 736 to 1065 percent. The simulation results for lazy FSA were identical to those of active FSA.

The lazy initialization of the FSA seemed to be a profitable optimization as it reduced the overhead of building the states during initialization from 9 to 23 percent when compared to the active FSA, while still providing equal simulation speed to the active FSA.

Table 2. Simulation startup times

	none	vectors	active FSA	lazy FSA
MUL	1.00 (5.7 s)	1.32 (7.5 s)	1.72 (9.8 s)	1.56 (8.9 s)
ALU	1.00 (38.3 s)	1.45 (55.5 s)	3.24 (124.4 s)	2.66 (101.8 s)
FPU	1.00 (116.0 s)	1.35 (157.0 s)	3.27 (379.4 s)	2.51 (290.6 s)

Table 3. Simulation times

	none	vectors	active FSA	lazy FSA
MUL	1.00 (2.0 s)	10.65 (21.3 s)	1.40 (2.8 s)	1.40 (2.8 s)
ALU	1.00 (2.2 s)	7.36 (16.2 s)	1.32 (2.9 s)	1.32 (2.9 s)
FPU	1.00 (5.8 s)	9.66 (56.2 s)	1.59 (9.2 s)	1.59 (9.2 s)

Low initialization time is important especially during a processor design space exploration with smaller test programs during which frequent short simulations of evaluated architecture variations is usual.

6 Conclusion

In this paper, simulation models for detecting function unit pipeline resource conflicts in simulation of architectures with independent function unit pipelines were evaluated. The evaluated models included the traditional resource vector based approach, and an approach that uses an finite state automaton (FSA) to detect resource conflicts quickly.

Additionally, an improvement to the FSA-based approach was proposed. In this "lazy FSA" model, the states are not constructed at simulation initialization time, but at the time they are used the first time, thus reducing the simulation initialization time in case of complex resource usage patterns in the simulated function unit.

The different models were implemented and benchmarked using three different test function units with resource usage patterns of varying complexity and with operations with both short and long latencies. The conclusion from the benchmarks is that the proposed "lazy FSA" approach, due to its reasonable initialization time combined with good simulation speed, is a suitable default model for function unit simulation in a processor simulator.

In the future, we plan to evaluate more techniques for speeding up the simulation of statically scheduled architectures with simplified control logic, like VLIWs and TTAs. Producing a very fast simulator especially for TTAs is quite challenging as it is not a traditional instruction set architecture, thus cannot be easily mapped to the host instruction set by means of compiled simulation. In addition, its architecture is very close to its microarchitecture, thus, even a functional simulation is forced to model quite low level details. However, techniques like combining speed of functional simulation with accuracy of cycle-level simulation or the use of techniques such as "memoization" could be interesting to adapt for our case [10,5].

Acknowledgement

This work has been supported in part by the Academy of Finland under project 205743 and the Finnish Funding Agency for Technology and Innovation under research funding decision 40441/05.

References

1. Corporaal, H.: Microprocessor Architectures: from VLIW to TTA. John Wiley & Sons, Chichester (1997)
2. Cmelik, B., Keppel, D.: Shade: a fast instruction-set simulator for execution profiling. In: Proc. SIGMETRICS '94, Nashville, Tennessee, May 1994, pp. 128–137. ACM Press, New York (1994)
3. Nohl, A., Braun, G., Schliebusch, O., Leupers, R., Meyr, H., Hoffmann, A.: A universal technique for fast and flexible instruction-set architecture simulation. In: Proc. DAC '02, New Orleans, Louisiana, June 2002, pp. 22–27. ACM Press, New York (2002)

4. Poncino, M., Zhu, J.: Dynamosim: a trace-based dynamically compiled instruction set simulator. In: Proc. ICCAD '04, San Jose, CA, November 2004, pp. 131–136. IEEE/ACM Press, New York (2004)
5. Schnarr, E., Larus, J.R.: Fast out-of-order processor simulation using memoization. In: Proc. ASPLOS-VIII, San Jose, California, October 1998, pp. 283–294. ACM Press, New York (1998), doi:10.1145/291069.291063
6. Schnarr, E.C., Hill, M.D., Larus, J.R.: Facile: a language and compiler for high-performance processor simulators. In: Proc. PLDI '01, Snowbird, Utah, June 2001, pp. 321–331. ACM Press, New York (2001)
7. Pees, S., Hoffmann, A., Meyr, H.: Retargeting of compiled simulators for digital signal processors using a machine description language. In: Proc. DATE '00, Paris, France, March 2000, pp. 669–673. ACM Press, New York (2000)
8. Pees, S., Hoffmann, A., Meyr, H.: Retargetable compiled simulation of embedded processors using a machine description language. ACM T. Des. Autom. Electron. Syst. 5(4), 815–834 (2000)
9. Engel, F., Nührenberg, J., Fettweis, G.P.: A generic tool set for application specific processor architectures. In: Proc. CODES '00, San Diego, CA, pp. 126–130. ACM Press, New York (2000)
10. Kim, J.K., Kim, T.G.: Trace-driven rapid pipeline architecture evaluation scheme for asip design. In: Proc. ASPDAC '03, Kitakyushu, Japan, pp. 129–134. ACM Press, New York (2003)
11. Kim, H.Y., Kim, T.G.: Performance simulation modeling for fast evaluation of pipelined scalar processor by evaluation reuse. In: Proc. DAC '05, San Diego, CA, June 2005, pp. 341–344. ACM Press, New York (2005)
12. Davidson, E.S., Shar, L.E., Thomas, A.T., Fatel, J.H.: Effective control for pipelined computers. In: COMPCON75 Digest of Papers, February 1975, pp. 181–184. IEEE Computer Society Press, Los Alamitos (1975)
13. Faraboschi, P., Fisher, J.A., Young, C.: Instruction scheduling for instruction level parallel processors. In: Proc. IEEE, Washington, DC, vol. 89, pp. 1638–1659. IEEE Computer Society Press, Los Alamitos (2001)
14. Bradlee, D.G., Henry, R.R., Eggers, S.J.: The marion system for retargetable instruction scheduling. In: Proc. PLDI '91, Toronto, Ontario, Canada, June 1991, pp. 229–240. ACM Press, New York (1991)
15. Cormen, T.H., Leiserson, C.E., R.L.R.: Introduction to Algorithms. The MIT Press, Cambridge, Massachusetts (1999)
16. Jääskeläinen, P.: Instruction Set Simulator for Transport Triggered Architectures. Master's thesis, Department of Information Technology, Tampere University of Technology, Tampere, Finland, P.O.Box 553, FIN-33101 Tampere, Finland (Sepember 2005), See `http://tce.cs.tut.fi/`
17. Bala, V., Rubin, N.: Efficient instruction scheduling using finite state automata. Int. Journal of Parallel Programming 25(2), 53–82 (1997)
18. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 3rd edn. Morgan Kaufmann Publishers, San Francisco (2003)