Resource Conflict Detection in Simulation of Function Unit Pipelines

Pekka Jääskeläinen, Vladimír Guzma, and Viljami Korhonen

Department of Computer Systems
Tampere University of Technology
P.O. Box 553
FIN-33101 Tampere
Finland
{pekka.jaaskelainen,vladimir.guzma,viljami.korhonen}@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 several resource conflict detection models, commonly used in compiler instruction scheduling, in the context of function unit pipeline simulation. The evaluated models include the conventional reservation table based-model, the dynamic collision matrix 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 comparable simulation speed to the actively initialized FSA.

1 Introduction

Processor simulators possess different level of accuracy depending on their purpose. Functional 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 have a dramatic effect on the total exploration time.

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 are unable to handle all types of structural hazards

gracefully in hardware. In such cases, the detection of structural hazards during simulation is considered part of the program verification process. One example of this type of architecture is the TMS320C64x, a popular clustered VLIW architecture from Texas Instruments. This architecture handles certain types of structural hazards with exceptions, thus often resulting in program termination [1].

Simulation of statically scheduled architectures with simplified control logic concentrates on simulating the data transports between function units and register files, the functionality of operations in function units, and the function unit latencies. Our results show that in this type of simulators, especially if the simulation overhead of instruction decoding phase is avoided, structural hazard detection can become the bottleneck for simulation speed. This effect is emphasized when simulating architectures that have multiple pipelined function units with shared pipeline resources.

This paper evaluates several common models to detect function unit pipeline resource conflicts during compiler instruction scheduling, and applies them to processor simulation. The explored models include the conventional reservation table (CRT), the dynamic collision matrix (DCM) and an finite state automata (FSA) based model [2]. 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 the lazily initialized model, our benchmarks show significant improvements to the actively initialized FSA model initialization time with very little overhead to the simulation time.

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 in compiler instruction scheduling which are applied in this paper to processor simulation. Section 4 describes our test setup, followed by Section 5 with results from the executed benchmarks. 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 [3]. 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. In addition to the dynamic translation, the paper describes a methodology to flexibly adjust the selection of traces to produce from simulation, as it is noted that the production of traces is very expensive when compared to the simulation itself.

JIT-CCS technique applies just-in-time (JIT) compilation, common in virtual machines, to instruction set simulation. This technique removes the limitation of translating simulator not capable of simulating self-modifying code [4]. 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 [5]. 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 in order to achieve higher simulation speed.

The growing trend of multiprocessors in desktop machines is exploited in processor simulation in [6]. Multiple processor cores are taken into use by compiling the simulation code in other available host processor cores while simulating the application in one. This method hides the compilation overhead of dynamic translation, which is very beneficial especially when simulating smaller applications.

FastSim uses the idea of compiled simulation in detailed out-of-order microarchitectural simulation [7]. 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 [8] which introduces a language for easier 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 [9, 10]. The resulting simulators are cycle-accurate thanks to the capabilities of the language to allow detailed modeling of pipeline resources used by the instructions.

An interesting simulation speedup technique worth noting is "token-level simulation" [11] and "evaluation reuse" [12]. 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.

Notable speedups to functional simulation are achieved in [13] by combining fast host native execution of selected functions of the simulated program with more detailed, but slower, instruction set simulation of platform-specific functions. The proposed method is targeted to accelerate functional simulation used in cross-platform software development while our work addresses cycle-accurate processor simulation.

Literature covering techniques for speeding up and retargeting processor simulation, in general, is widely available. However, avoiding the bottlenecks in cycle-accurate simulation of architectures with independent function unit pipelines seems to be rarely discussed. Overall, it was hard to find publications addressing the problem of detecting structural hazards during simulation effectively while taking in account the trade-offs in simulation setup time. This paper considers structural hazard detection models in the context of simulating architectures with independent function unit pipelines that share hardware resources. The work is potentially applicable to a wider range of processor simulators that require any kind of structural hazard detection.

3 Structural Hazard Detection in Processor Simulation

Processor resource usage models have been traditionally used in instruction schedulers of compilers. In instruction scheduling, the model is asked whether an instruction can

	cycle		
resource	0	1	2
r_1	1	0	0
r_2	0	1	0
r_3	0	1	1

Fig. 1. Conventional Reservation Table.

be placed to a given cycle without causing structural hazards. This question is often referred to as "a contention query" [2]. Contention queries are done similarly in cycle-accurate processor simulators, except that the cycle parameter of the query is limited to the currently simulated one.

An important issue to notice while adapting the resource usage models to processor simulation is that at the worst case, the contention query is done every time before an instruction is simulated. In addition, for each successfully executed instruction, the "compound resource usage model" that stores the function unit or processor resource usage state must be updated to reflect the resource usage of the newly executed instruction. Finally, for each simulated cycle the compound resource usage model must be updated to reflect the passing of simulation time. These three operations required from the resource usage models, later referred to as "conflict check", "operation issue", and "cycle advance", must be fast, as the instruction functionality itself can be often simulated with relatively low number of host cycles with compiled simulation techniques.

Initialization time of the resource conflict model is important in such automated processor design space exploration algorithms which also explore the function unit pipeline resources. In case the application used to evaluate the explored architecture is short, the initialization time can become the bottleneck for the total simulation time. On the other hand, in case the design space exploration does not vary the structure of function units, the resource conflict model for each function unit used in exploration can be computed once and reused later using model caching. In this case the model initialization time is less relevant.

Next sections present a brief overview of common methods for keeping book of occupied pipeline resources during processor simulation or instruction scheduling and their simulation runtime complexity.

3.1 Conventional Reservation Table

Conventional Reservation Table (CRT) is a matrix with one dimension representing the machine resources and the other one representing cycles [2, 14, 15]. A resource usage is marked by storing 1 in the matrix element at the position defined by the cycle and the resource. An example of a reservation table for an operation that uses three resources and has a latency of three cycles is shown in Fig. 1.

When using CRT for function unit (FU) pipeline resource modeling, the simulator keeps book of the occupied resources at each cycle of the simulation in a per FU *compound reservation table*. Resource conflicts are detected by comparing the compound

	cycle		
operation	0	1	2
O_1	1	1	0
O_2	0	0	0

Fig. 2. Collision Matrix for Operation O_1.

reservation table to the resource usage pattern of the candidate operation. In case there are overlapping resource usages between the candidate operation's reservation table and the global reservation table, a structural hazard is detected.

The CRT model can be implemented with a bit matrix using host machine words to store the bits. In this kind of implementation the conflict check can be implemented as an AND operation between the corresponding words of the compound reservation table and the reservation table of the issued operation. In case the result of any of the AND operations is nonzero, a conflict is detected. This check is potentially quite fast, especially if there are less cycles in the reservation table than bits in the host machine word, in which case only one AND operation for each reservation table row is required. However, the worst case complexity of the resource conflict check is $O(rc)$ where r is the number of resources and c denotes the number of cycles in the longest operation in the function unit.

Operation issue can be implemented similarly as the conflict check, by using the OR operation to merge the resource usages of the issued instruction to the global reservation table. Cycle advance is implemented by left shifting the reservation table once to reflect the freeing of resources as the simulation time passes.

3.2 Dynamic Collision Matrix

Another approach to the conflict detection problem is to store conflict information directly to a matrix instead of the resource usage patterns that are needed to compute it. In this approach, a *collision matrix* (CM) is computed for each operation in the FU. A collision matrix contains rows for each operation and as many columns as there are cycles in the longest latency operation in the FU. The element M_{ij} in a conflict matrix of operation O_k is 0 only if the operation O_i (the i:th row in the CM) does **not** cause a resource conflict when issued j cycles (the j:th column in the CM) after issuing operation O_k [14].

Figure 2 presents a collision matrix of an imaginary operation O_1 in a function unit with two operations. From the matrix it is easy to see that after executing O_1, one cannot execute O_1 again for the first two cycles, but O_2 can be freely executed, as all elements in its row contain 0, denoting "no conflict".

The *Dynamic Collision Matrix (DCM)* model, proposed in [2], uses a *global collision matrix* which can be used to keep book of the pipeline resource state of the simulated function unit, similarly to the compound reservation table in case of CRT. The improvement of DCM in comparison to CRT is that instead of needing to loop through the matrix elements, the conflict check is now a single table look-up with $O(1)$ complex-

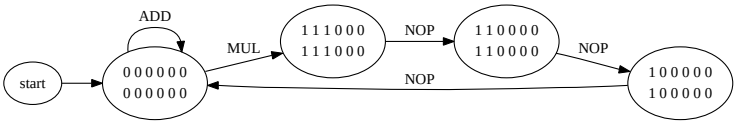

Fig. 3. Finite state automaton of valid operation sequences in an FU.

ity. Operation issue still requires a loop that ORs the corresponding bits in the issued operation with the DCM. The cycle advance implementation is also similar to CRT: a left shift of the matrix. The loops required for operation issue and cycle advance can be optimized similarly as CRT, using host machine words to store the bits. Complexity of these two operations is dependent on the count of operations in the function unit, the worst case being $O(oc)$ where o is the number of operations and c number of cycles in the longest operation in the function unit.

3.3 Finite State Automata

One popular structural hazard detection scheme is to construct a Finite State Automaton (FSA) [16] that contains information of all non-conflicting operation execution sequences in the function unit. In the FSA-based approach, each state includes a collision matrix (as described in the previous section) that stores information of all the operations that are possible to issue without conflicts while at that state. The collision matrix of the target state when issuing an operation is computed by left shifting the collision matrix of the starting state and ORing it with the collision matrix of the issued operation [14]. Left shifting the collision matrix simulates a cycle advance, which assumes that at most one operation can be started per cycle in the function unit. Cycles in which no new operations are issued are modeled by issuing a "no operation" (NOP), a pseudo operation with an all-zeros collision matrix. In some architectures, such as the traditional VLIW, NOP is not a pseudo instruction but issued like any other function unit operation.

Figure 3 illustrates an example automaton for a function unit with two operations: ADD and MUL. For example, in the automaton, it is apparent that after executing ADD, it is possible to execute both ADD and MUL, but after executing MUL three idle cycles are needed before issuing new operations.

Conflict check using FSA requires minimal computation; a single table look-up to the transition table of the FSA. In case the transition table provides a valid destination state id for the operation, the tested operation is possible to issue without conflicts, otherwise a hazard is detected. Similarly, updating the function unit state in operation issue is a constant operation: the target state obtained in conflict check is simply stored to a variable representing the current state of the function unit. The state update includes also the cycle advance simulation, and in case of an idle cycle, extra functionality of issuing a NOP is required. Thus, all model operations: conflict check, operation issue, and cycle advance happen in constant time using the FSA.

A disadvantage of the FSA-based model is that their initialization time can grow long due to the potentially large number of states in the automaton that need to be built

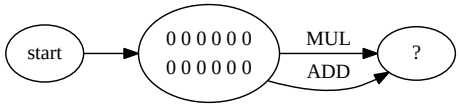

Fig. 4. The lazy FSA before issuing any operations.

based on the operation resource usage patterns. An algorithm for building a minimum automata from instruction resource vectors is presented in detail in [17]. Briefly, the algorithm starts by initializing the start state with the all-zeros conflict matrix. Then it tries to "issue" all possible operations at that state, creating possibly new target states, issues all possible operations at the newly created states and so on, until there are no unhandled states left. Therefore, the worst case complexity of initialization of the function unit FSA is the case when all possible states are produced. Each state in the FSA is identified by its conflict matrix. Therefore, the maximum number of states is the number of different permutations of the conflict matrix, which is $O(2^{co})$ where c is the number of cycle columns and o the number of operation rows in the conflict matrix.

The potential "state explosion" for complex function unit pipelines leads to an optimization to the FSA-based approach 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. The optimization is derived from the observation that in many cases only the minority of the FSA states are visited by the simulated program, thus, the construction time for the unused states is wasted.

The time to initialize a new state consist of first generating a new collision matrix for the state using the starting point state and the collision matrix of the issued operation. The size of the matrix depends on the complexity of the function unit pipeline, as described in the previous section. In addition to the initialization overhead, there is some overhead from finding whether a state with the created collision matrix is already found in the state machine, in which case the transition must be made to that state instead of the newly created one. Overhead of this look-up grows as a search algorithm dependent function of the count of already created states.

The lazy FSA is initialized to contain the all-zeros starting point state. All state transitions from the state are initialized to point to a pseudo *uninitialized* state. Every time an operation that causes a transition to the *uninitialized* state is issued, a concrete target state is created, if it does not exist already, of which transition table entries are set to point to the *uninitialized* state. An example of lazy FSA in action is shown in Fig. 4 which shows how the lazily initialized version of the FSA in Fig. 3 looks initially. Figure 5 shows the FSA after issuing the MUL operation.

4 Test Setup

We evaluated different models for function unit resource conflict detection during simulation by implementing them in a simulator for Transport Triggered Architectures (TTA) [18]. TTA is an architecture that resembles VLIW and includes function units

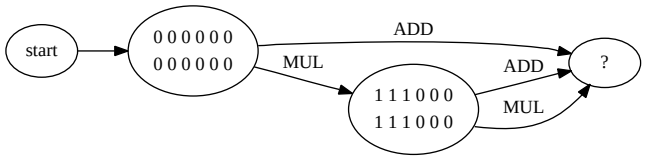

Fig. 5. The lazy FSA after issuing MUL.

with independent pipelines [19]. The main difference of TTA to VLIW in the simulation point of view is the programmer-controlled interconnection network which adds some extra complexity to the simulation. However, it is important to note that the results are not at all TTA-specific, but applicable to all processor architectures that require structural hazard modeling. TTA was used as an architecture template for this benchmark merely because this work was conducted as part of a project developing a codesign toolset for TTA-based application specific processors [20].

The experiments were divided to synthetic benchmarks and full program benchmarks. Synthetic benchmarks evaluated the models separately from the rest of the simulator to isolate their contribution to the simulation speed. The purpose of full program benchmarks was to give some perspective on how much the choice of the resource conflict model affects the actual total simulation speed. In this benchmark we evaluated the different models when used as part of a compiled simulator engine that simulated a whole TTA processor running nontrivial test programs.

In the synthetic benchmarks, the model initialization times were evaluated by initializing each model 1000 times in a row and measuring the total wall-clock time spent in the initialization routine. The model simulation speed was measured by simulating sequences of operations and by measuring the total wall-clock time it took to simulate the operation sequence. The synthetic simulation benchmark included a conflict check, an operation issue, and a cycle advance for each simulated operation, thus mimics a program with average of one operation execution per cycle. Each operation in each function unit was executed in round-robin fashion in successive cycles with total of $4 \cdot 10^9$ operation executions. All the resource conflicts reported by the models were caught and ignored without producing notable additional overhead to the simulation benchmark. Each test in the synthetic benchmark was executed three times in a row and the average wall clock time was measured.

The different models were evaluated with the following function unit resource usage patterns:

MUL A single-operation function unit that implements integer multiplication with latency of 3. The operation uses three pipeline resources as illustrated in Fig.1. This function unit was picked in order to evaluate how the different models perform in case the simulated function unit hasa very small number of opcodes.

ALU An arithmetic-logic unit with 18 integer operations. Latency of half of the operations is one cycle, two cycles for the other half.

FPU Function unit that models a floating-point unit. Its pipeline matches the one of MIPS R4000 floating-point unit, as described in [21]. The unit includes eight floating-

point operations that share eleven different pipeline resources. The double precision floating point operations range from a simple “absolute value” operation (latency of two cycles) to a long latency “square root” operation (latency of 112 cycles).

In the full program benchmarks, the impact to the actual experienced simulation speed was measured by testing each model in a compiled simulation engine. The benchmarked programs along with their total cycle counts are listed in Table 1. In order to create a realistic test machine for the benchmark, we used the architecture description language of our TTA-based codesign toolset [20] to define a TTA with the resources listed in Table 2. All benchmarked programs were using fixed point arithmetic so the floating point unit was not used in this benchmark.

Table 1. Benchmarked programs and their total cycle counts.

application	description	total cycles [Mcycles]
Video decoding	EEMBC DENBench [22] MPEG-4 Decode: ’graphic’	375
Audio decoding	Tremor [23] Ogg vorbis decoder: a 40 KB sample	304

Table 2. Relevant components in the simulated TTA-processor.

resource	quantity
ALU	2
MUL	2
32-bit registers	32
Register file read ports	4
Register file write ports	2
Transport buses	4

The measurements were made in a 3.4GHz Pentium 4 CPU with 2 GB of RAM. The operating system was Ubuntu Linux 7.10, with GNU GCC compiler version 4.1.2-16ubuntu2. The compiler optimization switch used to compile the models was ’-O3’.

In the full program simulation benchmark, a simulation code in C++ language was generated and compiled to host native code with the above mentioned GCC compiler and optimization switch.

The evaluated conflict detection models are the following:

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

Table 3. The number of created states in the active FSA model.

FU	# of states
MUL	3
ALU	4
FPU	2109

CRT The conventional reservation table model.
DCM Dynamic collision matrix model.
active FSA Uses an FSA for conflict detection. The FSA is fully constructed before starting the simulation. The used construction algorithm is similar to the one presented in [24].
lazy FSA Like "active FSA", but the FSA is not fully constructed before starting the simulation. Initialization routine includes initializing the collision matrices of function unit operations, and the creation of the initial state. Other states are created when they are visited for the first time during simulation.

The number of states in FSA affects the initialization time for the actively initialized FSA-based simulation model. The number of states for each function unit pipeline model are listed in Table 3. These numbers provide an indication of the complexity of the evaluated function unit pipelines. The FPU with 2109 states is an example of how the number of states grow dramatically when the complexity increases. On the other hand, simple function units can be simulated with a very low number of states.

5 Results

The results from the synthetic model benchmarks and full program simulation benchmarks are presented in the following sections.

5.1 Synthetic Model Benchmarks

The initialization times for each of the models are given in Table 4. Table 5 shows the simulation times which do not include the model initialization time.

The initialization numbers for CRT are not very interesting as our architecture description format (ADF) uses resource vectors, a structure very close to the reservation

Table 4. Average model initialization times [μs].

	CRT	DCM	active FSA	lazy FSA
MUL	10 (1X)	10 (1X)	30 (3X)	30 (3X)
ALU	70 (1X)	200 (2.9X)	330 (4.7X)	230 (3.3X)
FPU	3510 (1X)	143 000 (41X)	285 000 (81X)	143 000 (41X)

Table 5. Model simulation times [Moperations per second].

	CRT	DCM	active FSA	lazy FSA
MUL	9.7	30.6	40.0	32.9
ALU	14.7	4.7	40.3	34.8
FPU	1.8	2.7	33.2	32.2

table to describe the pipeline resource usages. The conversion from the ADF format to the optimized CRT structure produces only very negligible overhead.

The interesting part of this benchmark is how large overhead the rest of the models inflict on top of building the CRT. The ratio number is provided for easier comparison because DCM uses CRT for its initialization, and both of the FSA models include the overhead of the DCM model initialization because of the need to construct the operation collision matrices. This is apparent in the results. For example, the lazy FSA initialization time is roughly equal to the DCM initialization time because they both need to initialize the operation conflict matrices and an additional all-zeros conflict matrix: the global collision matrix in case of DCM, and the collision matrix of the initial state in case of the lazy FSA.

The effect of "state explosion" to the model initialization time is substantial in case of the FPU: creating all the states actively doubles the initialization time when compared to the lazy initialization.

The simulation time benchmark results, shown in Table 5 emphasize the speed advantage of the $O(1)$ complexity FSA models. CRT and DCM, the models that require table traversal in simulation show gradual slowdown as the complexity of the simulated function unit grows. The active FSA detects resource conflicts 2.7 to 18.4 times faster than the CRT and 1.3 to 12.3 times faster than the DCM. The slowdown of the lazy FSA in comparison to the active FSA is at worst 18% and at best only 3%. Our analysis of the code revealed that most of the slowdown is not caused by the initialization of the states lazily, but due to the additional comparison that checks whether the target state is the uninitialized state. These kind of small overheads are significant when the actual detection code is only a single array look-up.

5.2 Full Program Simulation Benchmarks

The main purpose of the full program simulation benchmarks was to show that the choice of the conflict detection model makes a difference in a highly-optimized simulation engine in real simulated programs. The results support this assumption, as shown in Table 6. For easier comparison, the table includes the relative speed normalized to the model with no conflict detection.

It can be seen that the overhead of conflict detection in FSA models is significantly smaller than in other models. No major difference was found between the lazily and actively initialized FSAs.

It was suprising to note that the DCM model with its constant time conflict detection was actually slower than the traditional CRT model in real simulation, although the

Table 6. Simulation speed with different models [Mcycles per second].

	none	DCM	CRT	active FSA	lazy FSA
Video decoding	4.88 (1X)	1.17 (0.24X)	1.79 (0.37X)	3.51 (0.72X)	3.48 (0.71X)
Audio decoding	3.96 (1X)	1.08 (0.27X)	1.53 (0.39X)	2.96 (0.75X)	2.96 (0.75X)

synthetic benchmarks showed different results. This was due to the fact that there was less than one operation per cycle in the simulated programs, thus the cycle advance simulation overhead become more dominant.

The DCM cycle advance speed was slower than in CRT since the matrix shifted in a cycle advance was larger for the function units in the simulated architecture. For example, the ALU has more operations (rows in the DCM) than shared pipeline resources (rows in the CRT).

5.3 About the Effect of Model Initialization Time to the Total Simulation Time

How much the FU resource conflict detection model initialization affects the total simulation time depends on the type of simulator and its implementation. Naturally, the relative contribution to the total simulation time is small in case other parts of the simulation initialization are complex. For example, in a straightforward implementation of compiled simulation, the simulation model along with the simulated program is compiled to host native code before running the simulation, which is often a lengthy process. In addition, in case the simulated program itself runs for a long time, the simulation initialization time loses relevance on every simulated cycle.

For our co-design toolset we have implemented a simulation engine that is optimized for fast initialization time at the expense of simulation speed [18, 20]. This engine is used in automated design space exploration in which relatively short kernels of crucial algorithms are simulated with a large number of candidate architectures. In this setting, the initialization time is important, as can be seen in Figure 6. The figure shows the relation between the function unit model initialization time and the total simulation time as a function of the length of the simulation in cycles for the FSA-based models. The data for the graph was produced by simulating a machine with two function units as complex as the FPU model used in the benchmarks, which is a realistic scenario in design space exploration of instruction set extensions that share pipeline resources. For example, it can be read from the figure that in simulations as long as 200 000 cycles, the model initialization time still consists almost 40% of the total time for the active FSA, while for the lazy FSA this number is much lower, at about 23%. The difference diminishes radically as the simulation length increases.

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.

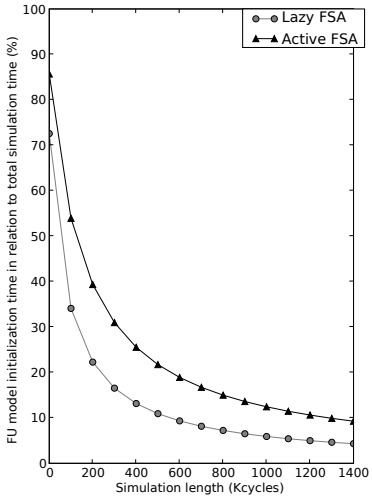

Fig. 6. Relation between model initialization time to the total simulation time.

The evaluated models included the traditional resource vector based approach, the dynamic collision matrix model, and a model that uses an finite state automaton (FSA) to quickly detect the resource conflicts.

In addition, 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 performed experiments show 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 resource conflict detection 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. 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 simulator is forced to model quite low level details. However, techniques aiming to combine the speed of functional simulation with the accuracy of cycle-level simulation or the use of processor state "memoization" could be interesting to adapt for our case [7, 11].

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. Texas Instruments: TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide. (December 2007) http://www-s.ti.com/sc/techlit/spru732.
2. Ramanan, V.J., Govindarajan, R.: Resource usage models for instruction scheduling: two new models and a classification. In: ICS '99: Proceedings of the 13th international conference on Supercomputing, New York, NY, USA, ACM Press (1999) 417–424
3. Cmelik, B., Keppel, D.: Shade: a fast instruction-set simulator for execution profiling. In: Proc. SIGMETRICS '94, Nashville, Tennessee, United States, ACM (May 1994) 128–137
4. 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, USA, ACM Press (Jun 2002) 22–27
5. Poncino, M., Zhu, J.: Dynamosim: a trace-based dynamically compiled instruction set simulator. In: Proc. ICCAD '04, San Jose, CA, USA, IEEE/ACM (Nov 2004) 131–136
6. Qin, W., D'Errico, J., Zhu, X.: A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. In: CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis, New York, NY, USA, ACM Press (2006) 193–198
7. Schnarr, E., Larus, J.R.: Fast out-of-order processor simulation using memoization. In: Proc. ASPLOS-VIII, San Jose, California, United States, ACM (Oct 1998) 283–294
8. 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, United States, ACM (Jun 2001) 321–331
9. 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, ACM (Mar 2000) 669–673
10. 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) (2000) 815–834
11. Kim, J.K., Kim, T.G.: Trace-driven rapid pipeline architecture evaluation scheme for asip design. In: Proc. ASPDAC '03, Kitakyushu, Japan, ACM (Jan 2003) 129–134
12. 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, USA, ACM (Jun 2005) 341–344
13. Gao, L., Kraemer, S., Leupers, R., Ascheid, G., Meyr, H.: A fast and generic hybrid simulation approach using C virtual machine. In: CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems, New York, NY, USA, ACM Press (2007) 3–12
14. Davidson, E.S., Shar, L.E., Thomas, A.T., Fatel, J.H.: Effective control for pipelined computers. In: COMPCON75 Digest of Papers, IEEE (Feb 1975) 181–184
15. Faraboschi, P., Fisher, J.A., Young, C.: Instruction scheduling for instruction level parallel processors. In: Proc. IEEE. Volume 89., Washington, DC, USA, IEEE (2001) 1638–1659

16. Thomas H. Cormen, Charles E. Leiserson, R.L.R.: Introduction to Algorithms. The MIT Press, Cambridge, Massachusetts, London, England (1999)
17. Proebsting, T.A., Fraser, C.W.: Detecting pipeline structural hazards quickly. In: Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon (1994) 280–286
18. 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 (Sep 2005) *See* http://tce.cs.tut.fi/.
19. Corporaal, H.: Microprocessor Architectures: from VLIW to TTA. John Wiley & Sons, Chichester, UK (1997)
20. Jääskeläinen, P., Guzma, V., Cilio, A., Takala, J.: Codesign toolset for application-specific instruction-set processors. In: Proc. Multimedia on Mobile Devices 2007. (2007) 65070X–1 – 65070X–11
21. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 3rd edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003)
22. EEMBC: Denbench 1.0 software benchmark databook. PDF *See* http://www.eembc.org/TechLit/Datasheets/denbench_db.pdf.
23. The Xiph Open Source Community: Tremor - the reference ogg vorbis decoder. WWW *See* http://xiph.org/vorbis/.
24. Bala, V., Rubin, N.: Efficient instruction scheduling using finite state automata. Int. Journal of Parallel Programming **25**(2) (1997) 53–82