Fast Cycle-accurate Behavioral Simulation for Pipelined Processors Using Early Pipeline Evaluation

In-Cheol Park
KAIST - Daejeon, Korea
icpark@ee.kaist.ac.kr

Sehyeon Kang
KAIST - Daejeon, Korea
shkang@ics.kaist.ac.kr

Yongseok Yi
KAIST - Daejeon, Korea
yslee@ics.kaist.ac.kr

ABSTRACT

Modeling and simulating pipelined processors in procedural languages such as C/C++ requires lots of cost in handling concurrent events, which hinders fast simulation. A number of researches on simulation have devised speed-up techniques to reduce the number of events. This paper presents a new simulation approach developed to enhance the simulation of pipelined processors. The proposed approach is based on early pipeline evaluation that all the intermediate values of an instruction are computed in advance, creating a future state for the next instructions. The future state allows the next instructions to be computed without considering data dependencies between nearby instructions. We apply this concept to building a cycle-accurate simulator for a pipelined RISC processor and achieve almost the same speed as the instruction-level simulator.

1. INTRODUCTION

Modern embedded systems usually include at least one programmable instruction set processor, as the software approach helps the designer catch up the shrinking time-to-market with less effort. The portion to be implemented in software or hardware is determined in the early design stage, and simulation, among several approaches, is a prevalent method to do this.

In the performance evaluation of an embedded system, timing accuracy of the instruction set simulator becomes more and more important. As the programmable core embedded in the system has interactions with memory, memory-mapped I/O and other system components, its timing accuracy has a significant impact on the confidence of the performance analysis [2]. While an instruction level processor simulator just connects the hardware which cooperates with the processor and the software running on the processor at functional level, a cycle-accurate processor simulator helps software interact with hardware at the exact time. Additionally a cycle-accurate instruction set simulator can be used to validate the implementation of the processor by performing comparisons between the two states.

In the simulation of pipelined processors, there are several levels of abstraction from gate level to functional level. Gate-level models described in HDL are very accurate in dealing with detailed timing, but simulation speed is slow. And it is difficult to try a lot of design alternatives in the design space exploration stage. In contrast, functional-level models written in procedural programming languages like C/C++ are easier to modify and run faster. Unfortunately, however, building a cycle-accurate instruction set simulator requires a detailed simulation model to resolve the pipeline hazards [1] and often needs to rewrite it in another language. What is worse, it is required in handling pipeline hazards to analyze the data dependencies between the instructions within the pipeline, which prevents fast simulation. Therefore many simulation techniques are proposed to improve the simulation speed while maintaining the timing accuracy [3].

One approach is to provide an optimized simulation library that makes it easy to model concurrent behaviors. SystemC [16] is an example of this approach. It exploits an object-oriented programming technique to provide methods to describe concurrent behaviors of hardware components and corresponding simulation engines. A built-in message-passing mechanism is used in Incas [4] for a cycle-accurate model of UltraSPARC. This approach, however, does not improve the simulation speed in spite of the efficient modeling capabilities.

A notable technique is proposed in [5], which introduces the delay operator to solve pipeline hazards automatically and implicitly. The delay operator is used when the source operand of a fetched instruction is in the destination register of a previous instruction that is alive in the pipeline. In this case a delay operator is assigned to point the value that should be taken from the preceding context. However, since the delay operators can be locally connected and spread out over the pipeline stages, handling them makes another burden to fast simulation.

The compiled simulation is regarded faster than the interpretive simulation because the former benefits from a priori information, moving the frequent operations such as instruction decoding from the simulation run-time to the compile-time [6] [7]. Furthermore the compiled simulation exploits host machine instructions to speed up the simulation. This concept was introduced by HSS [8], which converts the cycle-free portions of a circuit into machine codes optimized for a host machine that execute the simulation. Similarly, LISA [9] translates executables from one machine to another one like binary translation [10]. Low-level code generation is presented in [11] to aggressively utilize host machine resources. However, even in the compiled simulation, run-time scheduling is still inevitable [12] since not all the operations can be scheduled in the compile-time. Moreover, since a compiled simulator attempts to resolve pipeline hazards statically at the compile time, the simulator must consider every possible case that can be encountered in dynamic scheduling.

These previous works focusing on the implementation of a cycle accurate instruction set simulator are commonly devoted to mimicking the pipelined processors, not considering the software characteristics of the simulator. Consequently, they just translate the concurrent behavior of the pipelined processor in high-level programming languages, which invokes a considerable amount of scheduling events. This paper proposes a new simulation technique to reduce these scheduling events. The main point is to provide correct values of the processor state at the right time, not how to calculate the values as in hardware.

This work was supported by the Korea Science and Engineering Foundation through the MICROS center and by IC Design Education Center(IDEC).

ICCAD'03, November 11-13, 2003, San Jose, California, USA.
Copyright 2003 ACM 1-58113-762-1/03/0011 ...$5.00.

The rest of this paper is organized as follows. Section 2 presents the concept of the proposed simulation technique based on early pipeline evaluation and how to apply the proposed technique to an instruction set simulator by taking an example. Subsequently the overall structure of the simulator is described in Section 3. The implementation and the performance of the proposed simulator are presented in Section 4.

2. EARLY PIPELINE EVALUATION

Pipelining is a key implementation technique to increase processor performance by overlapping the execution of multiple instructions. As the pipelining is to split the execution of an instruction into smaller pieces, the instructions alive in the pipeline can have a number of interactions caused by the pipeline hazards. In software modeling of a pipelined processor, however, a clock period is not a physical value but a virtual concept. This implies the execution of an instruction does not need to be split to fit into a short clock period as in hardware.

From this perception, we propose a new simulation technique that all the pipelined operations of an instruction are computed at one time to produce all the intermediate values of every pipeline stage. The values proceed along the pipelined datapath without further computation as depicted in Fig. 1. This technique is named *early pipeline evaluation*.

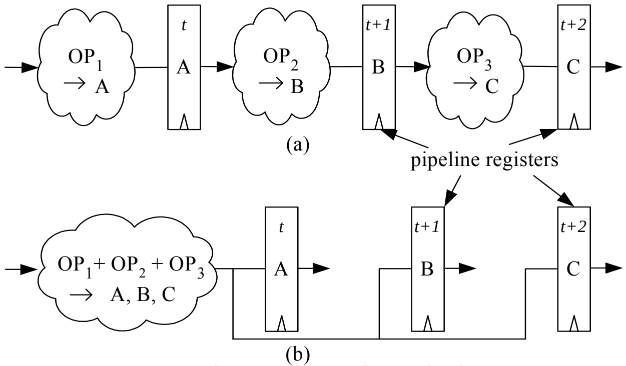


Fig. 1. Pipelining: (a) Traditional scheme. (b) Early pipeline evaluation.

The early pipeline evaluation removes the necessity of considering pipeline hazards by executing the program effectively at instruction level, eventually improving the simulation speed. This is possible since the results from the previous instructions are already available and all the subsequent pipeline operations are determined. The proposed simulation technique is a switchover from the conventional cycle-accurate simulator which attempts to mimic the hardware operations.

2.1 Future State

The resulting values from the early pipeline evaluation create a *future state* that is a reference state to be used in the computation of the following instructions. The future state is obtained by executing the program at instruction level, while the processor state resulted from executing a cycle-accurate pipelined model is called the *architectural state*. The architectural state is updated as the same way as in the original pipelined processor, providing an *architectural view* to the programmers. This separation of states, future and architectural states, is also applied to implementing the precise interrupt handling in out-of-order processors [13].

As the processor state consists of a register file and a data memory, the future state consists of a *future register file* and a *future memory* and the architectural state consists of an *architectural register file* and an *architectural memory*. Unlike the register file, a memory buffer that has entries as many as the number of pipeline stages is used for the future memory state. As the memory is considerably large compared to the register file and the difference between the future memory and the architectural memory is small, having a duplicated memory to store the latest values is wasteful. Therefore a small memory buffer is used to keep the values of 'store' instructions alive in the pipeline. Since these values are the more recent values than those in the data memory, they are searched before directly accessing the data memory and bypassed to 'load' operations if the reference address is found in the buffer.

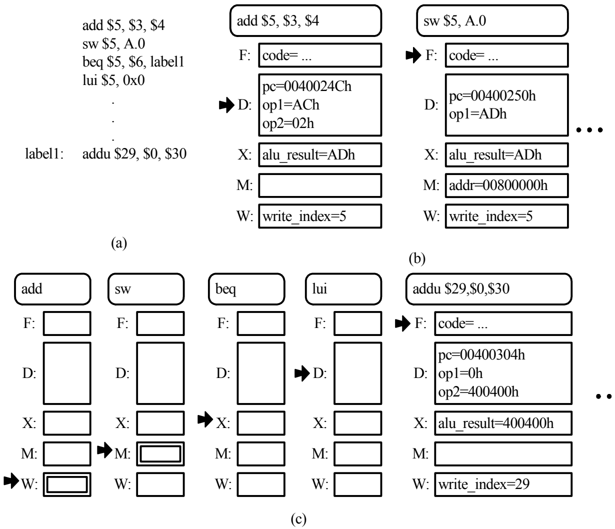
The associative search is used to carry out the bypass between nearby memory instructions referencing the same address. For efficient associative search, a hash table is used. In the memory buffer, a number of LSBs of a memory address can be used as a key. When a 'store' instruction is fetched, the early pipeline evaluation inserts a value to the memory buffer, i.e. future state. When the instruction reaches the memory stage, this value is deleted and written to the data memory, i.e. architectural state. Between these operations a 'load' instruction may access the same address as that of the 'store' instruction. The early pipeline evaluation searches the memory buffer using the LSBs of the address and the value is fetched from the buffer. If a 'load' instruction accesses a different address, the memory buffer returns nothing and the value is fetched directly from the data memory.

Early pipeline evaluation calculates all the intermediate values by exploiting the future state. This brings considerable speed-up over conventional cycle-accurate simulation. If the operation of an instruction is split into the corresponding pipeline stages, the instruction must be identified at every pipeline stage to do a proper processing for intermediate values. Though it is not complex as much as in the decode stage, the decoding time at each pipeline stage takes a large portion of the simulation time. Moreover, not only the current instruction but also previous instructions must be considered to resolve the data dependencies. The proposed early pipeline evaluation removes the works and saves the simulation time significantly.

2.2 Simulation Example

Fig. 2(a) illustrates an example code, written with compliance to the convention of MIPS assembly language. The processor under consideration is assumed to have a 5-stage pipeline consisting of {F, D, E, M, W}, where F, D, X, M and W stand for Fetch, Decode, eXecute, Memory and Write-back, respectively. Assume that the general-purpose registers \$3, \$4 and \$6 contain the values of 0xAC, 0x02 and 0xAD, respectively. As the first instruction, `add`, is fetched at cycle *t*, the values of all the intermediate pipeline registers, the program counter (pc), and the results of the operation are computed at cycle *t*. Proceeding ahead, the next instruction `sw`, store word, is fetched at cycle *t*+1. The source operand is in the destination register of the previous instruction, thus requires a data forwarding in a cycle-based simulator. In the proposed simulation, however, since all the results of the `add` instruction has been already computed, the value of the source operand is readily available in the future file, and the results of the `sw` instruction can be computed from the future file without a forwarding mechanism. As the pipeline proceeds by a cycle, the update pointer of the `add` instruction is moved to the next stage, i.e. the decode stage and the intermediate values of stage D are updated to the architectural state. At the same time, the pointer of the `sw` instruction points to the fetch stage as depicted in Fig. 2(b).

The third instruction `beq`, branch if equal, needs to evaluate the branch condition: true if its two operands \$5 and \$6 have the identical values. Although \$5 has a data dependency and \$6 does not, both are fetched from the future register file. In this example the two registers have the same value, 0xAD, thus the branch is taken. Therefore, the calculated pc points to the branch target address, labeled `label1`, whose actual address is 0x00400300. As MIPS has one branch slot, `lui`, load upper immediate, which is located next to `beq` is fetched at cycle *t*+3. The source operand of this instruction is supplied by its code word, so the results are computed without invoking the future register file.



```
        add $5, $3, $4
        sw $5, A.0
        beq $5, $6, label1
        lui $5, 0x0
        .
        .
        .
label1: addu $29, $0, $30
```

Fig. 2. Simulation Example: (a) An example code, (b) Circular buffers at cycle (*t*+1), and (c) Circular buffers at cycle (*t*+4), where ➡ denotes the part of the future state to be updated to the architectural state.

At cycle *t*+4, the first instruction reaches its write-back stage, letting the $5 in the architectural register file be updated to the value 0xAD as in Fig. 2(c). Likewise, since the second instruction reaches the memory stage, the memory is updated and the corresponding element in the memory buffer is removed. Here we assume that the memory operation, either 'write' or 'read', is completed in one cycle, so that the pipeline proceeds without stall.

3. SIMULATOR ORGANIZATION

Fig. 3 illustrates the organization of the simulator. As explained in the previous section, the simulator has the architectural state, i.e. the architectural register file and the architectural memory, which represents the real state of the processor and the future state, i.e. the future register file and the future memory buffer, which is a reference state for the computation of the following instructions. The simulation body consists of an early pipeline evaluation part and circular buffers that store the intermediate values of pipelined registers. A column is named a slot and it is occupied by one instruction. Each entry of the slot is used to store the values to be updated at a corresponding pipeline stage.

For a fetched instruction, I_i, the early pipeline evaluation part computes the result and the intermediate values to be stored in the pipeline registers, $\{F_i, D_i, E_i, M_i, W_i\}$, and writes them to a slot, S_i. As the pipeline advances, a new instruction is fetched and its intermediate values are placed in the next circular buffer as numbered in Fig. 3 and the values from the previous instructions are updated to the architectural register file and memory after the pipeline latencies. Since the maximum number of instructions alive in the pipeline is equal to the pipeline depth, the number of slots in the circular buffers is the same as the pipeline depth (*n*). Therefore the intermediate values computed for the (*i*+*n*)-th successively fetched instruction can be stored into S_i as all the intermediate values of I_i are already updated into the architectural state at that time.

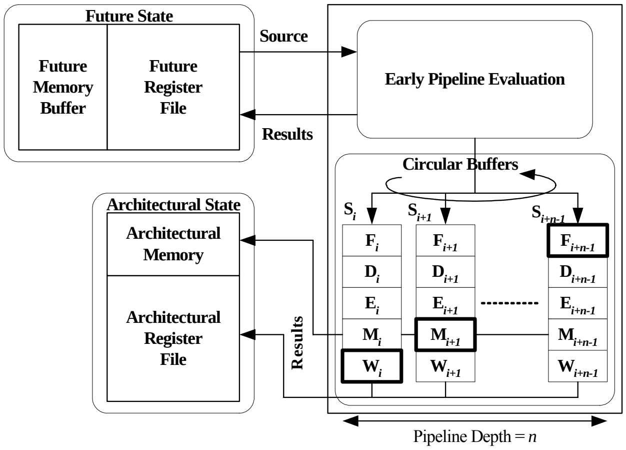


Fig. 3. Overall organization of the simulator.

While the future state is updated instantly, the architectural state is updated according to the pipeline latency. This latency, however, may be variable with the previous instructions. For example, the pipeline may be stalled due to lazy memory operation, multi-cycle instruction and so on. To achieve accurate performance analysis these pipeline stalls should be considered. In the real time environment, the operation speed of memory is usually slower than that of the processor core, as a result a 'load' or 'store' operation requires more than one cycle. Even though a cache is placed between the processor and the memory, a cache miss requires a multi-cycle memory operation. And practical processors often contain multi-cycle instructions that make the following instructions stalled.

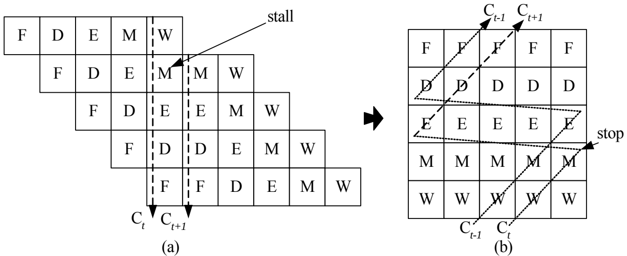


Fig. 4. (a) Pipeline stops at C_t because of a stall. (b) Active stage pointer movement.

To resolve these situations, an *active stage pointer* is used to indicate an active stage set of instructions whose entries should be written to the architectural state at a cycle. Fig. 4 shows how the active stage pointer indicates the update sequence of the architectural state. Since only one entry of each slot in the circular buffers is active at every cycle, the active stage set consists of the entries of the slots located in the diagonal line starting from the active stage pointer which normally points the write-back stage of the instruction fetched before (*n-1*) cycles. The update of the architectural state occurs along the line starting from the active stage pointer. If a stall occurs at cycle *t* due to lazy memory operation as in Fig. 4(a), the update stops at the stall stage, i.e. memory stage, and the active stage pointer remains there. The update starts again from the stall stage after the stall condition is removed. Therefore only the write-back stage is updated at C_t and the following stages are updated at C_{t+1} in Fig. 4(b). To reflect accurate memory access cycles, stall information can be added to the memory operation instead of connecting a memory or cache simulator which may cause severe speed degradation.

3.1 Debugging Mode

A debugging mode is added to the simulator in order to make the user observe the pipeline registers by stepping through the program cycle by cycle. Since the user might be doubtful about the value of a certain pipeline register, a simulator must provide a way to trace back the origin of the value. This is trivial if the value comes solely from the preceding pipeline stage. However, if it is forwarded from the other part of the pipeline, tracing back becomes rather complicated because the proposed simulation technique does not accommodate any forwarding mechanism in nature.

In the debugging mode the simulator keeps the number of preceding instructions, as many as the pipeline depth, which still reside in the pipeline. To trace the origin of a value, it looks up the preceding instructions to find out the data dependency using a conventional dependency-checking algorithm and indicates a proper forwarding path based on the result of the evaluation. Although this is quite a cumbersome and time-consuming task, it is not a serious problem because the debugging mode does not require fast simulation speed.

4. EXPERIMENTAL RESULTS

Three versions of instruction set simulator for MIPS R2000/3000 are implemented for experiments: instruction-level, cycle-accurate and early pipeline evalutaion versions. A Simple machine description language is used to generate the three versions, as manipulating the instruction format is error-prone and common to all. By using the machine description language, we can also generate various decoding styles such as nested '*if, else if*', grouped '*if, else if*' and function pointer tables. According to our experiments, the function pointer and grouped '*if, else if*' statements are more effective in saving the average simulation time. On the average, 45% and 42% are saved, respectively, compared to the nested '*if, else if*' statements.

The size of the machine description for the MIPS processor is 1078 lines. The generated simulators support the interactive mode to monitor the processor status step by step. Its functionalities are fully verified using a reference simulator, SPIM [15]. DSPStone benchmark programs compiled with the GNU cross-compiler are used as simulation inputs. The simulation is executed on a SUN Blade2000 workstation and the result is presented in Fig. 5.

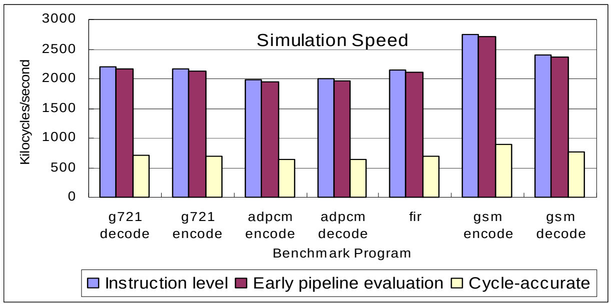


Fig. 5. Comparison of simulation speed.

The simulation results show that the simulation speed is 2.2 million cycles/sec on the average, almost as fast as the instruction level simulator. And it is quite fast compared to other retargetable interpretive instruction set simulators. The authors of paper [5] report that the average simulation speed of their simulator is 240 kilocycles/sec, and UPFAST [14], is 200 kilocycles/sec. Even compared to the simulation speed of retargetable compiled simulators, the simulation speed achieved by the proposed simulation method is comparable. For example, LISA framework achieves 2.5 million instructions/sec for '`fir`' and 0.8 million instructions/sec for '`adpcm`' [9]. Note that the proposed method is incorporated into interpretive simulation to provide flexibility and debugging convenience. However, the proposed approach is not limited to interpretive simulation, but can be easily incorporated to compiled simulation.

5. CONCULSION

This paper has presented a new simulation technique based on early pipeline evaluation and demonstrated the potential of the proposed approach by implementing an instruction set simulator. As opposed to real pipelined processors, the proposed approach computes all the pipelined operations of an instruction at a time including all the intermediate values. The early pipeline evaluation creates a future state to maintain the latest values of the processor state, which makes it possible to compute the subsequent values without considering the data dependency between nearby instructions. This technique can be applied not only to instruction set simulators but also to pipelined IP blocks even if they have different levels of abstraction.

6. REFERENCES

[1] J.L. Hennessey and D.A. Patterson, "Computer Architecture: a quantitative approach," Morgan Kaufmann Publisher, 1990.

[2] Lisa Guerra et al. "Cycle and Phase Accurate DSP Modeling and Integration for HW/SW Co-verification," in proceedings of 36th Design Automation Conference, 1999.

[3] Mayan Moudgill, "Techniques for Implementing Fast Processor Simulators," in proceedings of 31st Simulation Symposium, 1998.

[4] Guillermo Maturana et al. "Incas: A Cycle Accurate Model of UltraSPARC^{TM}," in proceedings of International Conference on Computer Design, 1995

[5] Felix Sheng-Ho Chang and Alan J. Hu, "Fast Specification of Cycle-Accurate Processor Models," in proceedings of International Conference on Computer Design, 2001.

[6] George Hadjiyiannis et al. "A Methodology for Accurate Performance Evaluation in Architecture Exploration," in proceedings of 36th Design Automation Conference, 1999.

[7] Mark R. Hartoog et al. "Generation of Software Tools from Processor Descriptions," in proceedings of 34th Design Automation Conference, 1997.

[8] Zeev Barzilai et al. "HSS – A High Speed Simulator," IEEE Transactions on Computer-Aided Design, July 1987.

[9] Vojin Živojnović et al. "Compiled Simulation of Programmable DSP Architectures," in proceedings of IEEE Workshop on VLSI Signal Processing, 1995.

[10] Richard L. Sites et al. "Binary Translation," Communications of the ACM, February, 1993.

[11] Jianwen Zhu et al. "An Ultra-Fast Instruction Set Simulator," Transaction on VLSI Systems, June 2002.

[12] Gunnar Braun et al. "Using Static Scheduling Techniques for the Retargeting of High Speed, Compiled Simulators for Embedded Processors from an Abstract Machine Description," in proceedings of International Symposium on System Synthesis, 2001.

[13] James E. Smith and Andrew R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Transaction on Computers, May 1998.

[14] Soner Önder and Rajiv Gupta, "Automatic Generation of Microarchitecture Simulaors," in proceedings of IEEE International Conference on Computer Language, 1998.

[15] J.R. Larus. SPIM S20: A MIPS R2000 simulator, http://www.cs.wisc.edu/~larus/spim.html

[16] SystemC, available in http://www.systemc.org