A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors[1]

Venkata Krishnan and **Josep Torrellas**

Department of Computer Science
University of Illinois at Urbana-Champaign, IL 61801
venkat,torrella@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

Abstract

Multiprocessor system evaluation has traditionally been based on direct-execution based Execution-Driven Simulations (EDS). In such environments, the processor component of the system is not fully modeled. With wide-issue superscalar processors being the norm in today's multiprocessor nodes, there is an urgent need for modeling the processor accurately. However, using direct-execution to model a superscalar processor has been considered an open problem. Hence, current approaches model the processor by interpreting the application executable. Unfortunately, this approach can be slow.

In this paper, we propose a novel direct-execution framework that allows accurate simulation of wide-issue superscalar processors without the need for code interpretation. This is achieved with the aid of an *Interface Window* between the front-end and the architectural simulator, that buffers the necessary information. This eliminates the need for full-fledged instruction emulation. Overall, this approach enables detailed yet fast EDS of superscalar processors. Finally, we evaluate the framework and show good performance for uni- and multiprocessor configurations.

1 Introduction

Software simulation plays an integral role in the design and validation of high-performance uni- and multiprocessor systems. Two major simulation methodologies used are Trace-Driven Simulation (TDS) and Execution-Driven Simulation (EDS). In TDS, hardware probes [27, 30] or software instrumentation of the application [22, 24, 25] allow information like basic block or data addresses to be collected in a trace buffer during the application's execution. The generated information is later used to drive a simulator of the system under study. Given that an application may execute billions of instructions, to reduce the storage requirements of the traces, sampling [5, 21, 26] is often used.

[1] This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, ASC-9612099 and MIP-9619351, DARPA Contract DABT63-95-C-0097, NASA Contract NAG-1-613 and gifts from IBM and Intel.

While TDS is an effective methodology in the study of high-performance uniprocessor systems, it is not applicable to multiprocessor systems. The reason is that, in the latter systems, it is very important to faithfully model the interleaving of the memory accesses of the different processors. The actual interleaving depends, among other things, on the memory system used. Since the trace is generated using an existing memory system and the TDS uses a different, simulated memory system, TDS may result in the incorrect interleaving of the traces [1].

It is in this area that EDS plays a crucial role. Here, the execution of the application and the simulation is completely interleaved. The application is instrumented at appropriate points so as to generate events for the back-end simulator. When the simulator is called, it processes the event. Then, it returns control to the application. This method facilitates feedback from the simulator to guide the execution of the parallel application.

There have been many EDS-based multiprocessor system simulators for both RISC and CISC architectures [2, 4, 6, 15, 16, 20, 23, 31]. Most of these EDS-based systems are based on *direct execution*. Here, the instrumented application is executed directly on the host machine where the simulation runs. This results in fast simulation. Unfortunately, using simulation based on direct execution comes at a price. Since there is no simulated processor model, it can be used only in those studies where the processor need not be modeled accurately.

Unfortunately, though direct-execution based EDS is used commonly for multiprocessor environments where the main focus is to study memory subsystems, not modeling the processor in detail may give rise to inaccurate results [18]. This is because current multiprocessor systems are typically based on complicated processors like wide-issue dynamic superscalars. Accurately simulating current dynamic superscalar processors using direct execution has been considered an open problem [17]. Consequently, when it is necessary to model such a processor, the simulators have to resort to interpreting the application executable [18, 23]. This often results in slowdowns of orders of magnitude.

Addressing this problem is the motivation behind this paper. The contribution of this paper is the design and evaluation of a novel direct-execution framework that allows accurate simulation of advanced processor architectures in both uni- and multi-processor environments without incurring the slowdowns of an interpretational approach. This is achieved with the aid of an *Interface Window* between the front-end and the architectural simulator, that buffers limited instruction information. This eliminates the need for full-fledged instruction emulation.

Overall, our direct-execution framework addresses the need for having a single, efficient simulation environment that allows the detailed modeling of the processor architecture in both uni- and multi-processor environments.

This paper is organized as follows: Section 2 details our simulation framework, Section 3 evaluates its performance and, finally, Section 4 concludes the paper.

2 A Direct-Execution Framework for Processor-Level Simulation

This section presents our framework. We first describe the methodology for modeling the execution of an instruction, then present a more complete view of the framework, and finally examine the way we collect statistics.

2.1 Modeling Instruction Execution

Our primary aim is to model advanced processor architectures for both uni- and multi-processor environments. One way of achieving this would be to interpret each instruction in a simulated processor. However, this is likely to lead to significant slowdowns in many cases [17, 23]. Our approach, instead, is to perform a limited functional emulation. We elaborate on this approach in the rest of this section.

We instrument the application to generate events at basic-block boundaries. This tracks the control flow of the application. Furthermore, all load and store instructions are also instrumented to generate events for the simulator. Finally, for each instruction in a basic block, we need some limited information to change the state of the simulated processor. This information is register usage and partial opcode information. There is no need for any full-fledged instruction emulation.

More specifically, we map the host ISA to an internal instruction set, which we term the *RISCier* ISA. The RISCier ISA captures the full functionality of the native instruction set without being concerned by the different flavors of each instruction type. For instance, we combine into a single class all ALU instructions that take the same number of cycles to execute in the simulated processor and that can use the same functional units in the simulated processor. In the RISCier ISA, one such instruction is represented as a generic `alu` operation along with the registers that it uses. Also, branch delay slot scheduling is undone when the mapping to the RISCier code is performed. The complete set of RISCier instructions are shown in Table 1.

Overall, our simulation framework consists of three modules. First, we have a front-end that instruments and then directly executes an executable file. For this, we use MINT [31], a well-known system that operates on MIPS executables. We have modified MINT to handle MIPS-II binaries. Then, unlike a conventional simple direct-execution simulator, events are no longer passed directly to the memory simulator back-end. Instead, they are intercepted by the processor simulator. The processor simulator changes its internal state based on these events and later releases memory references to the memory simulator.

RISCier Instruction	**Operations Covered**
alu	add,sub,set,and,or,xor,move
shift	all logical and arithmetic shifts
md	integer multiply and divide
uncond	unconditional branch
cond	conditional branch
call	function call
ret	function return
ld	load
st	store
fpadd	floating point add, sub, mov, abs, trunc, floor, comparisions etc.
fpmult	floating point multiply
fpdiv	floating point square root and div.

Table 1: Description of the RISCier instruction set.

Incorporating the processor simulator into existing MINT-based memory simulators is extremely simple. The only requirement for the memory simulator is that it should have the ability to support hit-under-miss and also handle multiple accesses in the same cycle. The information passed between modules is detailed next.

2.2 The Interface Window

When the application is executed using the modified MINT front-end, it generates a trace of basic-block and memory addresses. However, the processor simulator cannot be invoked for each and every event. There are two reasons for it. First, the simulation would be too slow. Secondly and more importantly, the simulator cannot consume just one instruction at a time. Indeed, since the size of a basic block is only a few instructions in integer applications [32], wide-issue superscalars use advanced branch-prediction schemes [19, 34] to fetch instructions across multiple basic blocks. Thus our model should permit sufficient information from the front-end to be gathered as dictated by the branch-prediction scheme of the simulated processor, before invoking the simulator.

This effect is achieved with an interface module between the front-end and the processor simulator called the *Interface Window*. The interface window buffers the events issued by the front-end (basic block and memory addresses) before calling the processor simulator (Figure 1). When the outcome of a branch in the program is known, we use the branch prediction mechanism of the simulated processor to see if the latter would have predicted the branch in the correct way. If so, we continue filling the interface window with events from the front-end. Otherwise, we stop filling the window and call the processor simulator. The simulator will then consume all the events in the window. Note that the processor simulator has access to an encoded representation of the application that contains, for each basic block and on an individual instruction basis, the limited opcode information and the register usage information explained above.

When the simulator reaches the mispredicted branch instruction, it continues executing instructions from the wrong path. Note again that the basic blocks in the wrong path are known to the simulator because it has access to the application file. When the simulator reaches a point where the mispredicted branch is finally resolved in

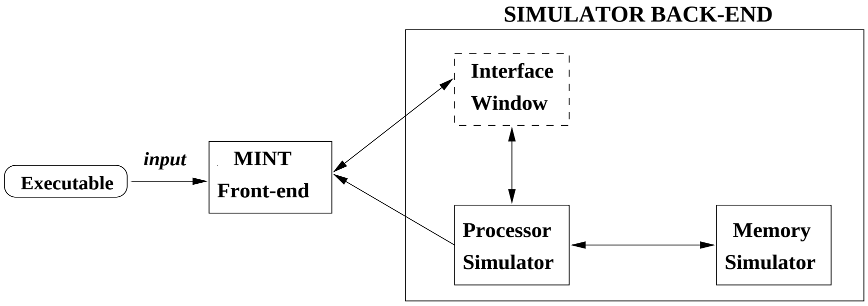

Figure 1: A high-level view of the simulation framework.

the simulated processor, the simulated pipeline is flushed and the front-end invoked again. With this approach, we correctly model the speculative execution of the simulated processor.

It is possible that the interface window is filled completely before reaching a mispredicted branch. This typically occurs only when running floating-point applications, where the branch prediction is close to ideal. In that case, the front-end stops and the processor simulator is invoked. When the window is low in instructions, the front-end is invoked again. In our studies, we set the interface window to 512 entries.

We see that the interface window allows the correct modeling of different branch-prediction schemes, independently of the processor model. In addition, it enables instruction scheduling and correct modeling of speculative execution. We consider each issue in turn.

2.2.1 Instruction Scheduling

Typically, the application is compiled to run optimally on the host processor. The latter is often less aggressive and has a narrower issue width than the simulated processor. Consequently, we need to generate the appropriate instruction schedule for the simulated processor. Otherwise, we may degrade the performance of the simulated processor, especially when simulating in-order issue superscalars, which completely depend on good compiler scheduling.

To solve this problem, we exploit the fact that we have many basic blocks in the interface window, and reschedule the instructions. A typical compiler for a superscalar or VLIW machine performs inter basic-block code scheduling in a window of instructions that represents the most-frequently executed path. The most-frequently executed path may be either estimated statically or acquired through profiles. The Superblock in the IMPACT compiler [3] or the Trace Window in the Multiflow compiler [12] are typical examples of this approach. In our work, we take the instructions in the interface window and perform a resource-constrained list-scheduling of instructions [14] as well as limited register renaming to remove false dependences, before invoking the processor simulator. Furthermore, loops with small bodies are automatically unrolled inside the window, with the amount of unrolling constrained by the size of the window.

Overall, therefore, we perform compile-time optimizations at run-time without using any run-time information such as load or store addresses. The only information that we use is the branch-prediction of the simulated processor that is used for growing the window. Unfortunately, unlike a compiler, which can look at huge instruction windows spanning thousands of instructions and can perform advanced optimizations such as software pipelining [9], we are restricted by the size of the run-time interface window in our scheme. Thus, our approach falls short of a pure compiler-based approach to scheduling. Nevertheless, as we show in our evaluations, our approach considerably improves the ILP when modeling in-order issue superscalars. It must be noted that a similar strategy was adopted by the UltraSparc team to generate code for the in-order issue UltraSparc-I processor [28].

2.2.2 Speculative Execution

We model speculative execution by using the branch prediction mechanism of the simulated processor as indicated before. This allows multiple branch predictions to be performed when there are pending unresolved branches. In addition, it realistically allows the pipeline to be polluted with instructions that will later be squashed once the outcome of the mispredicted branch is known. This allows the simulation to proceed somewhat independently of the native execution of the application.

Without further analysis, however, we do not know the addresses of the data accessed by the wrongly-speculated instructions. This is because the application, running under MINT, will never execute those instructions. To address this problem, we can perform a careful analysis of the value of the registers of the wrongly-speculated instructions and, in most cases, determine the data addresses. However, to simplify the problem, in our current setup, we neglect this issue and assume that the memory accesses of the wrongly-speculated instructions always hit in the L1 cache and do not pollute it.

2.3 The Simulator Framework

Our framework is quite flexible and allows different processor types and configurations to be modeled. The base processor models that are supported include an in-order issue superscalar that can speculatively fetch across branches and an out-of-order issue superscalar that is modeled on the lines of the MIPS R10000 [13] and is shown in Figure 2.

The model is quite detailed and incorporates all superscalar features such as a register renaming mechanism,

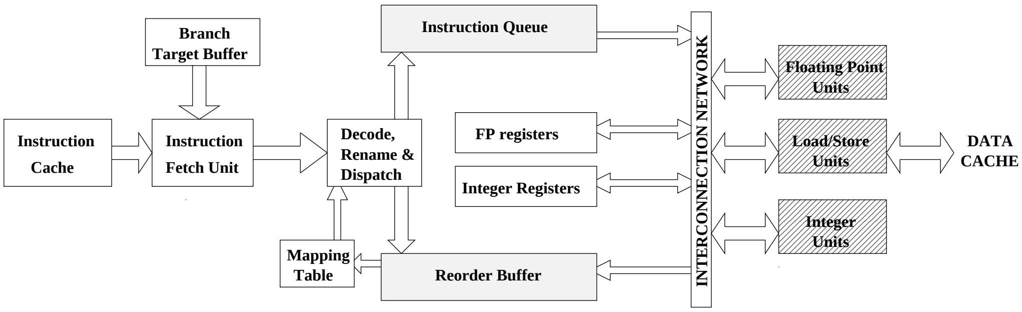

Figure 2: Architecture of the out-of-order issue superscalar.

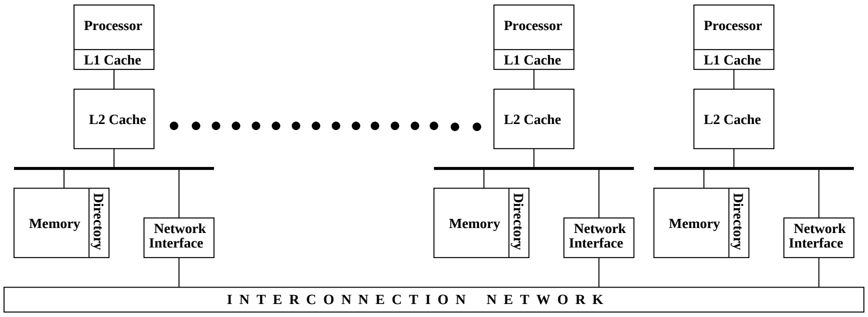

Figure 3: Multiprocessor architecture.

a large associative instruction queue for out-of-order issue, speculation across multiple branches and in-order instruction retirement.

We model the pipeline in great detail. A scoreboard keeps track of operand availability. An instruction is dispatched when its operands are ready and the corresponding functional unit is available. We use event queues to speed-up the simulation. Consider, for instance, the instruction queue. Rather than scanning it every cycle, we do so only when there is a possibility that an instruction changes status. For example, if a load instruction that misses in the cache causes dependent instructions to clog the instruction queue, we can skip checking the instruction queue until the load data becomes available.

Our framework also uses the superscalar core as a building block to model advanced processor architectures such as a chip multiprocessor (CMP), where multiple superscalar cores share a single chip [7]; a simultaneous multithreaded (SMT) processor, which adds multiple threads to a superscalar and allows instructions from different threads to be issued in the same cycle [29]; and finally a hybrid of the two architectures that supports several SMT processors on a single chip [8]. By varying parameters such as the number of on-chip processors and the number of threads in a SMT processor, we can model different architectures.

We also model a multiprocessor system, where each node in the system can be any of the above processor types. The memory subsystem for the uniprocessor is based on a conventional L1-L2 cache hierarchy, while the multiprocessor system (Figure 3) supports a DASH-like [11] directory-based cache-coherence protocol. Some of the processor and memory subsystem parameters that can be specified at startup time for modeling the machine are given in Table 2.

2.4 Statistics Collection

The simulator allows detailed statistics collection each cycle for each thread and processor. For the processor, we gather statistics on an issue slot basis. The total number of slots is the product of the total number of cycles taken by the application and the maximum issue rate. Wasted or non-*useful* slots are classified according to the type of hazard that prevented the slot from being used. The different categories of hazards are: lack of functional units (*structural*), memory access (*memory*), data dependences (*data*) and branch mispredictions (*control*). In addition, when executing parallel applications, the threads can also spin on barriers and locks (*sync*). Finally, we have an *other* category for less frequent events such as running out of renaming registers.

In an in-order issue processor, the hazard type that is responsible for unfilled slots in a given cycle can be uniquely identified. An extension needs to be made when running multiple threads as in in-order issue SMT processors. Here, since several threads can compete for a given issue slot, we assign the non-*useful* slots proportionally. For example, consider 8 threads in an 8-issue SMT processor. Further assume that, in a given cycle, four slots are filled with useful instructions. Thus we have a total of four wasted slots in this cycle. Assume that the threads cannot issue more instructions because, say, four are stalled on memory accesses, three are stalled on a data hazard and the remaining one cannot issue because of a structural hazard. In our statistics, we assign the four non-*useful* slots to the different reasons on a proportional basis. We use the formula:

$$\begin{array}{c} type_i \\ wasted \\ slots \end{array} = \begin{array}{c} total \\ wasted \\ slots \end{array} \times \begin{array}{c} threads \\ stalled \\ on\ reason_i \end{array} / \ total\ threads$$

In the above example, the four wasted slots are assigned as follows: $4 \times 4/8 = 2$ *memory* slots, $4 \times 3/8 = 1.5$

Parameter	Subparameter
Processor type	Number of processors/chip (CMP-level) Threads/Processor (SMT-level)
Issue mechanism	Issue policy (in-order/out-of-order) Issue width
Dynamic issue mechanism	Renaming registers Instruction queue entries
Branch prediction (correlating) [19]	Prediction table entries Misprediction penalty (cycles)
Functional units (int, load/store, fp)	Number of units Type of instructions handled Instruction latency and repeat rate (for non-pipelined units)
Multiprocessor support	Number of processing nodes
Memory-related	Number of outstanding loads and stores L1 and L2 cache size, associativity, block size and banks Local and remote memory latency

Table 2: Sample simulation parameters that can be specified at startup time.

data slots and $4 \times 1/8 = 0.5$ *structural* slots.

In an out-of-order issue processor, it is usually impossible to attribute only one hazard type in a given cycle. Therefore, we scan the instruction queue every cycle and record the type of hazard faced by each instruction that is unable to issue. If, in addition, the processor has several threads, we do this for all the threads. At the end, the wasted slots are divided proportionally among the different types of hazards.

2.5 Other Features

Finally, our framework has the ability to dynamically turn on and off the simulation. This allows sampling at regular intervals. We can also selectively turn off the processor simulator while allowing memory references to the memory back-end. This is particularly helpful when running parallel applications in which the initialization phase need not be processor-simulated, while the caches have to be warmed-up before the program enters the parallel section [33]. Additionally, it allows the development and testing of the memory back-end independently of the processor simulator.

3 Evaluation

The complete framework is implemented in C/C++ and is around 20K lines of code. Both accuracy and speed were kept in mind during the implementation. Unlike in interpretation-based simulations, where the simulator can be validated based on the results produced by the application, simulations based on direct execution are difficult to validate as they model only the timing aspect of the application. To mitigate this problem, we have implemented our simulator so that it can be executed in two different modes: a safe mode with assertions added at every stage and a fast mode that disables the checking. The safe mode validates the implementation of the processor model during the course of execution. We use this mode for a set of sample inputs before using the fast mode for our experiments.

We evaluate our framework for both a uni- and a multiprocessor environment. The evaluation for the uniprocessor environment is performed with two SPECint95 (*compress* and *ijpeg*), one multimedia [10] (*mpeg*) and four SPECfp95 (*swim*, *tomcatv*, *hydro2d* and *mgrid*) applications. Two SPLASH-2 [33] (*radix* and *cholesky*) applications are used to evaluate the multiprocessor environment.

The applications are compiled for a MIPS R4400 processor, using the highest level of optimization and the *-mips2* option. The MIPS R4400 is a simple single-issue processor. Since we want to model a more aggressive architecture, we will have to reschedule the instructions in the interface window.

The architecture modeled is a 4-way dynamically- or statically-scheduled superscalar processor. The value of some of its architectural parameters is specified in Table 3. The processor has a 2K-entry direct-mapped correlating branch prediction table [19] that allows multiple branch predictions in a cycle. All functional units are fully pipelined, with most instructions taking 1 cycle to complete. The only exceptions are as follows: integer multiply and divide take 2 and 8 cycles respectively; floating-point multiply takes 2 cycles, while divide takes 4 (single precision) and 7 (double precision) cycles. The processor can have up to 16 outstanding memory accesses, of which half can be loads. The characteristics of the memory hierarchy are shown in Table 4.

Issue Width	Number of Funct. Units (int/ld-st/fp)	Entries in Instruction Queue	Number of Renaming Regs. (int/fp)
4	4/2/2	32	32/32

Table 3: Characteristics of the superscalar processor.

Parameter	Value
[L1 / L2] size (Kbytes)	[64 / 1024]
[L1 / L2] line size (Bytes)	[32 / 64]
[L1 / L2] associativity	[2-way / 4-way]
L1 banks	7
L1 latency (Cycles)	1
L2 latency (Cycles)	6
Memory latency (Cycles)	40

Table 4: Characteristics of the memory hierarchy. Latencies refer to round trips.

In the following, we perform two sets of experiments. First, we measure the IPC increases that result from performing instruction scheduling in the interface window as described in Section 2.2. Then, we evaluate the slowdowns that result when the simulator is incorporated to

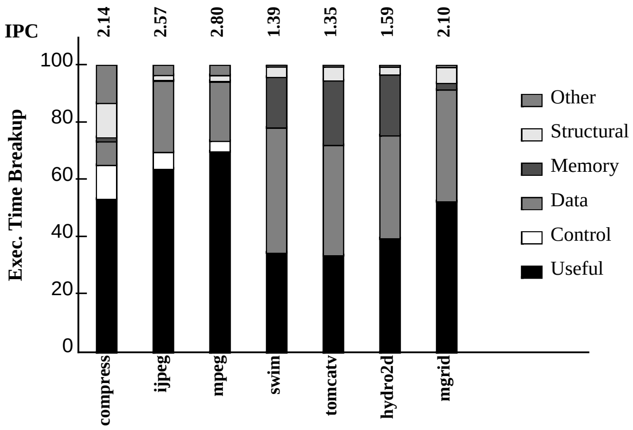

Figure 4: IPC and execution time breakup in a 4-way out-of-order issue superscalar for a real memory system.

the MINT front-end. For the latter experiments, we only include the processor simulator, not the memory back-end simulator. This is done to isolate the slowdowns caused by the processor simulator alone.

3.1 Interface Window Scheduling

To see the impact of rescheduling the instructions in the interface window as described in Section 2.2, we simulate the statically-scheduled processor with and without instruction rescheduling. Table 5 shows the average IPC for the applications with and without instruction rescheduling. The table has two sets of data, namely one assuming the memory system characteristics of Table 4 (real memory system) and one where all data accesses hit in the first-level caches (ideal memory system). From the table, we see that, thanks to the rescheduling, the IPC increases for all applications and memory systems. The increases range from 7% to 37%, and are on average 22%.

Instruction rescheduling in the interface window does not have much effect for out-of-order issue processors. This is because the processor hardware itself already aggressively reorders the instructions. The result is that the IPC changes are negligible. Figure 4 shows the IPC for the out-of-order issue superscalar with the real memory system of Table 4. The average IPC is now 1.99. The figure also shows a detailed breakup of the total execution time in terms of the different categories mentioned in Section 2.4. This provides valuable information in identifying performance bottlenecks. For instance, a large *data* contribution in the loop-intensive floating-point applications highlights the need for software pipelining [9].

3.2 Slowdowns for Uniprocessor and Multiprocessor Simulation

In this section, we evaluate the slowdown of the simulator when modeling a uni- and multi-processor system. To isolate the slowdowns caused by the processor simulator, we do not include the memory back-end in these experiments. The original MINT and the version enhanced with the processor simulator were linked with empty memory back-end functions. For each application, we report the average time taken by 5 runs. All measurements are made

Ideal Memory System

Application	IPC No Resched.	IPC Resched.
compress	1.89	2.35
ijpeg	1.65	2.09
mpeg	1.63	2.24
swim	1.35	1.52
tomcatv	1.33	1.62
hydro2d	1.17	1.56
mgrid	1.24	1.38
Average	1.46	1.82

Real Memory System

Application	IPC No Resched.	IPC Resched.
compress	1.72	1.99
ijpeg	1.65	2.06
mpeg	1.63	2.19
swim	0.89	0.97
tomcatv	0.83	0.97
hydro2d	0.93	1.18
mgrid	1.21	1.29
Average	1.26	1.52

Table 5: Impact of the ILP-enhancing instruction rescheduler in a 4-way in-order issue superscalar

on one node of a SGI PowerChallenge based on MIPS R10000 processors.

Columns 2 and 3 of Table 6 give the slowdown over MINT when modeling the 4-issue static and dynamic superscalars. From the table, we can see that the slowdown varies widely across the applications. Overall, the average slowdown for these experiments is around 10.

Application	Ours / MINT		MINT / Native
	Static	Dynamic	
compress	4.5	5.9	115
ijpeg	18.5	23.8	139
mpeg	14.6	18.7	152
swim	5.4	7.4	70
tomcatv	1.5	1.8	131
hydro2d	3.3	4.3	123
mgrid	15.4	18.1	199
Average	9.0	11.4	133

Table 6: Slowdowns with an empty memory back-end when simulating a 4-way static and dynamic superscalar.

Column 4 shows the MINT slowdown over the native execution of the applications. MINT has been reported to have a slowdown of around 40-70 [31]. However, the

Application	Number of Nodes	MINT / MINT.1	Ours / MINT.1	Ours / MINT
radix	1	1.0	28.2	28
	8	1.1	31.1	28
	16	1.2	34.3	29
	32	1.5	41.9	28
cholesky	1	1.0	27.0	27
	8	1.4	31.1	22
	16	1.5	39.5	26
	32	2.8	67.2	24

Table 7: Slowdown when simulating a multiprocessor configuration based on a 4-way dynamic superscalar. *MINT.1* represents the time taken by MINT for a single-node configuration.

applications used in [31] are quite different from those that we use. In our case, we see slowdowns in the range of 70-200. Factoring this, the full simulator is around 1300 times slower than native execution. This is in contrast to the several thousand-time slowdowns of interpretation-based systems, such as the MXS processor simulator in SimOS [23].

In addition, since our processor simulator is independent of the front-end, we have the flexibility of using a faster front-end such as Shade [22], which utilizes dynamic compilation and caching for improved performance.

Finally, we examine the slowdowns of the simulator when simulating a multiprocessor configuration in which each node is the 4-way dynamic superscalar. Two parallel applications, namely *radix* and *cholesky* are used for this purpose. Table 7 gives the results.

Columns 3 and 4 of Table 7 show the slowdown of MINT and our simulator relative to the simulation of a single node on the original MINT simulator. Recall that, in all experiments, we use an empty back-end memory simulator. As expected, the slowdown increases with more processors.

More importantly, the last column is the ratio between the previous two columns. It shows the slowdown of our system relative to MINT for the same number of processors. We can see that the slowdowns have increased relative to the uniprocessor slowdowns of Table 6. They are now in the high twenties. However, the numbers vary little with different numbers of processors. This indicates that our simulator scales quite efficiently with the number of processors.

4 Conclusion

TDS has typically been used to evaluate high-performance uniprocessor systems. However, it is not applicable to multiprocessor systems, which require EDS to model the exact interleaving of memory accesses. A common approach for modeling such systems has been to use a direct-execution based EDS. However, using such as a system to model superscalar processors has long been considered an open problem. Consequently, existing schemes resort to application code interpretation, which results in extremely slow simulations. This paper proposed and evaluated a novel approach to using a direct-execution based EDS for modeling advanced processor architectures, both in a uniprocessor and multiprocessor setup. Our evaluations showed that we can model the processor in great detail without resorting to application code interpretation while, at the same time, achieve fast simulations.

Acknowledgments

We thank the referees and the members of the I-ACOMA group for their valuable feedback. We greatly appreciate Matt Reilly's comments that substantially improved the earlier version of this paper. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] P. Bitar. *A Critique of Trace-Driven Simulation for Shared-Memory Multiprocessors*, pages 37–52. Kluwer Academic Publishers, Editors: M. Dubois and S. Thakkar, May 1990.

[2] E. Brewer, C. Dellarocas, A. Colbrook, and W. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS-TR-516, MIT Laboratory for Computer Science, September 1991.

[3] P. P. Chang, S. Mahlke, W. Y. Chen, N. J. Water, and W. Hwu. IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. In *18th International Symposium on Computer Architecture*, pages 266–275, May 1991.

[4] H. Davis, S. Goldschmidt, and J. Hennessy. Multiprocessor Simulation and Tracing using Tango. In *International Conference on Parallel Processing*, pages II99–107, August 1991.

[5] D.A.Wood, M.D.Hill, and R.E.Kessler. A Model for Estimating Trace-Sample Miss Ratios. In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 79–89, May 1991.

[6] S. Dwarkadas, J.R.Jump, and J.B.Sinclair. Execution-Driven Simulation of Multiprocessors: Address and Timing Analysis. *ACM Transactions on Modeling and Computer Simulation*, 4(4):314–338, October 1994.

[7] L. Hammond, B. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, September 1997.

[8] V. Krishnan and J. Torrellas. A Clustered Approach to Multithreaded Processors. In *12th International Parallel Processing Symposium (IPPS)*, April 1998.

[9] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Processors. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[10] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems . In *30th International Symposium on Microarchitecture (MICRO-30)*, pages 330–335, December 1997.

[11] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In *17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[12] G. Lowney, S. Freudenberger, T. Karzes, W. D. Lichtenstein, R. Nix, J. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[13] MIPS Technologies, Inc. *R10000 Microprocessor Chipset, Product Overview*, 1994.

[14] S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann, 1997.

[15] S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator. In *Workshop on Performance Analysis and its Impact on Design (PAID) (held in conjunction with ISCA'97)*, June 1997.

[16] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures. In *International Conference of Computer Design*, pages 486–490, October 1996.

[17] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997.

[18] V. S. Pai, P. Ranganathan, and S. V. Adve. The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodolgy. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, pages 72–83, February 1997.

[19] S.-T. Pan, K. So, and J. Rameh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 76–84, September 1992.

[20] D. Poulsen and P.-C. Yew. Execution-Driven Tools for Parallel Simulation of Parallel Architecture and Applications. In *Proceedings of Supercomputing*, pages 860–869, November 1993.

[21] R.E.Kessler, M.D.Hill, and D.A.Wood. A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches. *IEEE Transactions on Computers*, C-43:664–675, June 1994.

[22] R.F.Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[23] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, Winter 1995.

[24] M. Smith. Tracing with Pixie. Technical Report CSL-TR-91-497, Center for Integrated Systems, Stanford University, November 1991.

[25] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[26] T.M.Conte and W.W.Hwu. Systematic Prototyping of Superscalar Computer Architectures. In *3rd IEEE International Workshop on Rapid System Prototyping*, June 1992.

[27] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 162–174, October 1992.

[28] M. Tremblay, D. Greenley, and K. Normoyle. The Design of the Microarchitecture of UltraSPARC-I. *Proceedings of the IEEE*, 83(12):1995, December 1995.

[29] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *23rd International Symposium on Computer Architecture*, pages 191–202, May 1996.

[30] R. Uhlig, D. Nagle, T. Stanley, T. Mudge, S. Sechrest, and R. Brown. Design Tradeoff for Software-Managed TLBs. *ACM Transactions on Computer Systems*, 12(3):206–235, August 1995.

[31] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *MASCOTS'94*, pages 201–207, January 1994.

[32] D. Wall. Limits of Instruction Level Parallelism. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–189, April 1991.

[33] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[34] T. Yeh and Y. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *19th International Symposium on Computer Architecture*, pages 124–134, May 1992.