Fast Out-Of-Order Processor Simulation Using Memoization

Eric Schnarr and James R. Larus[1]
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706 USA
[schnarr,larus]@cs.wisc.edu

Abstract

Our new out-of-order processor simulator, FastSim, uses two innovations to speed up simulation 8–15 times (vs. Wisconsin SimpleScalar) with no loss in simulation accuracy. First, FastSim uses speculative direct-execution to accelerate the functional emulation of speculatively executed program code. Second, it uses a variation on memoization—a well-known technique in programming language implementation—to cache microarchitecture states and the resulting simulator actions, and then "fast forwards" the simulation the next time a cached state is reached. Fast-forwarding accelerates simulation by an order of magnitude, while producing exactly the same, cycle-accurate result as conventional simulation.

Keywords: Out-of-order processor simulation, direct-execution, memoization.

1. Introduction

Microarchitectural simulation is an essential tool in the research and design of processors, compilers, and other system software. Unfortunately, existing simulators of out-of-order processors run programs thousands of times slower than actual hardware. By applying techniques used to implement functional programming languages, we reduced the cost of simulation by up to an order of magnitude, with no effect on its accuracy.

FastSim is a new direct-execution simulator of a speculative, out-of-order uniprocessor with non-blocking caches. Its two primary contributions are *speculative direct-execution*, which efficiently performs the functional simulation of a program, and *fast-forwarding*, which dramatically accelerates the time-consuming simulation of an out-of-order microarchitecture.

Direct-execution simulators run machine code from a target program directly on a host processor, and use a variety of methods to interleave simulation code. This widely used technique allows functional simulation to run at near-hardware speed. Direct-execution, however, has not been previously used to simulate out-of-order processors, because of the difficulty of reconciling the fixed behavior of an executing program with the fluid behavior of a speculative out-of-order microarchitecture. FastSim solves this problem by decoupling the simulation of out-of-order execution from the functional execution of instructions. With a new technique called *speculative direct-execution*, FastSim allows mispredicted branch paths to be executed directly, then rolled back. Without further optimization (e.g., fast-forwarding), FastSim runs 1.1–2.1 times faster then the well-known SimpleScalar out-of-order simulator, which does not use direct-execution.[2]

FastSim's primary contribution is the application of *memoization*—result caching—to the expensive process of simulating an out-of-order microarchitecture. Traditionally, memoization was used to implement functional programming languages by caching function return values. Expensive computation can be avoided by returning a previously cached value, when available.

FastSim's *fast-forwarding* technique is similar. Fast-forwarding records microarchitecture configurations and the simulator actions that result from them. When a previously recorded configuration is encountered, the associated actions can be replayed at high speed until a previously unseen configuration is encountered. Fast-forwarding makes the simulator run 5–12 times faster, with no change in simulation results (e.g., cycle count.) Combining direct-execution and memoization, FastSim simulates a MIPS R10000-like architecture with a 190–360 times slowdown (i.e., simulation time over native benchmark execution time on the host), which is an order of magnitude faster than SimpleScalar.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 describes the implementation of direct-execution in conjunction with an out-of-order microarchitecture simulator, including our new technique for simulating speculative execution. Section 4 describes memoization of FastSim's out-of-order pipeline and discusses strategies for limiting the size of the memoization cache. Section 5 presents performance results for these optimizations including a comparison of FastSim

ASPLOS VIII 10/98 · CA, USA
© 1998 ACM 1-58113-107-0/98/0010...$5.00

[1] Current address: James Larus, Microsoft Research, One Microsoft Way, Redmond, WA 98052. Email: larus@microsoft.com.

[2] We do not have a version of FastSim without direct-execution. Instead, we use SimpleScalar as a surrogate, as it simulates comparable processors at an equivalent level of detail.

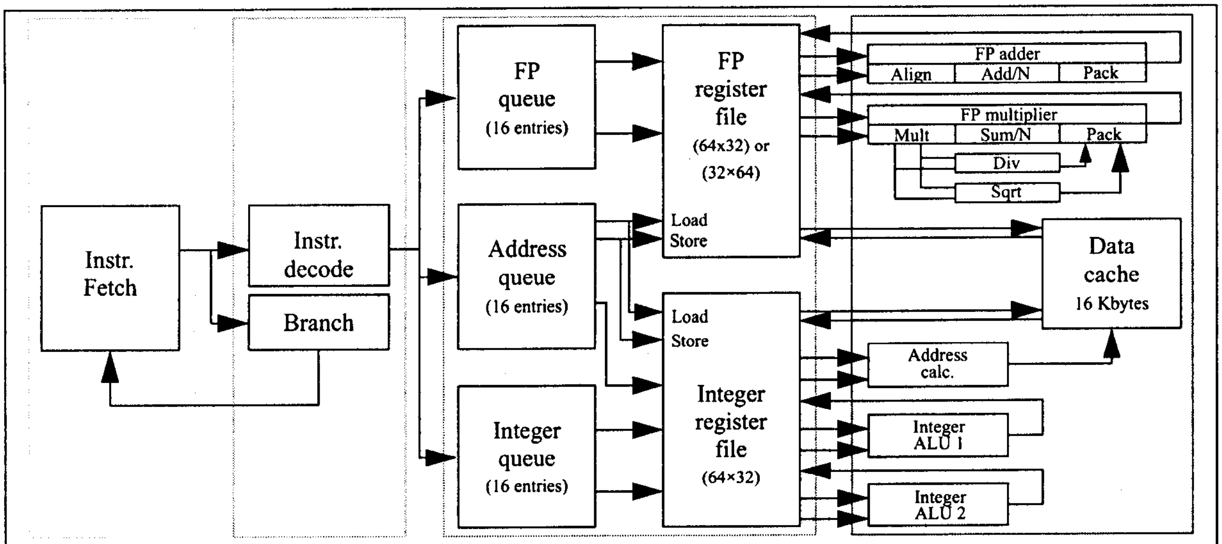

Figure 1. Block diagram of the dynamic execution microarchitecture modeled by FastSim.

and the SimpleScalar out-of-order simulator. Section 6 concludes.

2. Related Work

SimpleScalar [1], RSIM [8], and MXS [4] are contemporary simulators for out-of-order processors. None use direct-execution or memoization. They all execute thousands of host cycles per simulated cycle. SimpleScalar, one of the fastest out-or-order simulators using traditional technology, simulates a MIPS-like architecture and runs target programs with a 4,000 times slowdown [1]. RSIM emulates a multiprocessor with a SPARC-like architecture and typically simulates 10,000–15,000 instructions per second on a SUN Ultra 1/140 workstation [8]. MXS is the detailed, dynamic execution processor simulator from SimOS. It executes approximately 20,000 instructions per second, with a "several thousand times slowdown" [4].

Other simulators use direct-execution. Shade performs functional simulation and instrumentation by dynamically translating target instructions into host instructions. Collecting traces and similar kinds of information incurs a 2.8–6.1 slowdown [2]. Mipsy and Embra are functional CPU models in SimOS. Mipsy does not use direct-execution and runs 100–200 times slower than native hardware, while Embra runs only 10–30 times slower by translating target instructions into native instructions that execute directly on the host [4]. None of these direct-execution simulators perform detailed out-of-order microarchitectural simulation, as does FastSim.

Some simulation strategies trade-off accuracy for speed. Trace sampling has been used successfully in cache and processor simulation. Tom Conte et al. applied trace sampling to the simulation of an out-of-order processor and describe techniques for reducing state loss between sample clusters [3]. Another strategy is to approximate complex hardware using a simplified processor model. For example, WWT2 statically determines pipeline performance within basic blocks and updates the simulated cycle count each time a basic block is executed [6]. Pai et al. have shown that out-of-order processors cannot be approximately accurately by in-order pipeline models due to the unpredictable effects of memory instruction reordering [7]. In comparison, FastSim has no loss of accuracy, preferring to trade space for speed.

3. The Structure of FastSim

FastSim is a cycle-accurate, direct-execution simulator of an out-of-order uniprocessor. Like RSIM, it models a SPARC v.8 [9] instruction set running on a MIPS R10000-like [10] microarchitecture—Figure 1—although, unlike RSIM, FastSim only simulates a single processor. FastSim's processor model supports out-of-order instruction execution, speculative execution, and an aggressive non-blocking cache. Table 1 lists the processor parameters used in this paper.

Direct-execution is not easily applicable to speculative, out-of-order processor simulation. The first problem is simulating out-of-order execution using direct-execution, which is inherently in-order. As discussed in section Section 3.1, FastSim directly executes groups of instructions in program order, then subsequently simulates their behavior with respect to the out-of-order pipeline model. This is possible in FastSim, because loads, stores and other instructions do not require precise timing information to execute correctly on a uniprocessor machine.

Section 3.2 discusses FastSim's speculative direct-execution. Briefly, FastSim saves register and memory state at branches, then allows mispredicted branches and consequent execution paths to directly execute. Feedback from the μ-architecture simulator tells direct-execution when to restore register and memory state and restart execution at the corrected branch target. Hence mispredicted execution paths are directly executed, and data is collected for use in FastSim's processor simulator.

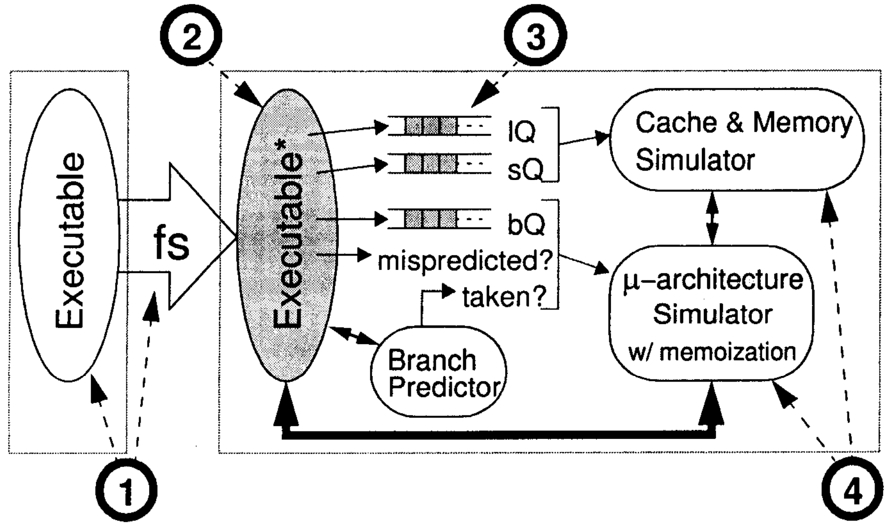

Figure 2. Overview of the FastSim simulator: 1) The tool fs rewrites a target executable. 2) The edited executable directly executes target instructions. 3) Instrumentation records information about loads, stores, and branches that is later used for μ-architecture and cache simulation. 4) The μ-architecture and cache simulators are called periodically to simulate FastSim's processor model.

Decode 4 instructions per cycle.
2 integer ALUs, 2 FPUs, and 1 load/store address adder.
64 physical 32-bit integer registers, and 64 32-bit (or 32 64-bit) floating point registers.
2-bit/512-entry branch history table for branch prediction.
Speculatively execute instructions through up to 4 conditional branches.
Non-blocking L1 and L2 data caches, 8 MSHRs each.
16 KByte 2-way set associative write through L1 data cache.
1 MByte 2-way set associative write back L2 data cache.
8 byte wide, split transaction bus

Table 1: FastSim's processor model parameters.

3.1 Direct-execution & OOO Simulation

Figure 2 shows the major components of the FastSim simulator. FastSim uses a binary rewriting tool (**fs**) based on the Executable Editing Library (EEL) [5] to instrument a statically linked SPARC program executable and link it with FastSim's μ-architecture and cache simulators.

The key to using direct-execution in out-or-order processor simulation is to separate functional—in order—execution of target instructions from simulation of the out-of-order pipeline. This is possible for two reasons. First, FastSim simulates a uniprocessor, hence loads and stores can be executed before their precise timing is known without affecting their result. Second, out-of-order pipelines preserve the appearance of executing instructions in program order. FastSim exploits these properties by directly executing groups of instructions in program order, then simulating their behavior with respect to FastSim's out-of-order pipeline model.

A target executable is instrumented to record the addresses accessed by every load and store, and the target of every conditional branch and indirect jump. Load and store addresses are put in queues, called lQ and sQ respectively, for FastSim's cache simulator. Instrumentation also calls FastSim's μ-architecture simulator at every conditional branch and indirect jump (including return instructions). Since FastSim's μ-architecture simulator is invoked at every control transfer instruction with more than one possible target, a single variable records whether a branch is taken or not-taken or the target of an indirect jump.

FastSim's μ-architecture simulator decides when the processor being modeled would have fetched, decoded, executed, and retired instructions previously executed via direct-execution. This simulator does not manipulate program data values or compute any functional results of the target program. These tasks are handled by direct-execution. When invoked, the μ-architecture simulator advances the out-of-order pipeline simulation up to the fetch

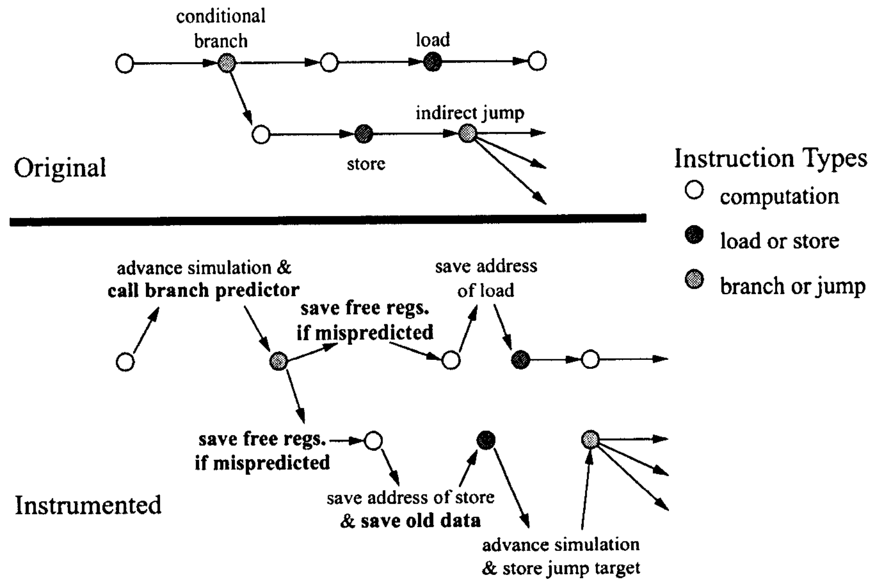
conditional branch
load
indirect jump
Original
store
Instruction Types
computation
load or store
branch or jump
advance simulation & call branch predictor
save free regs. if mispredicted
save address of load
save free regs. if mispredicted
Instrumented
save address of store & save old data
advance simulation & store jump target

Figure 3. Instrumentation inserted among original program instruction by fs, to maintain FastSim's data structures and call the μ-architecture simulator. Instrumentation to implement speculative execution is shown in bold.

of the current branch or indirect jump. Control flow information previously recorded for the last conditional branch or indirect jump is used to fetch instructions along the same execution path as direct-execution. When μ-architecture simulation catches up with direct-execution, the simulation is suspended and direct-execution continues to the next branch or indirect jump.

The μ-architecture simulator in turn calls FastSim's cache simulator. The queued load and store addresses along with timing information provided by the μ-architecture simulator permit accurate simulation of an aggressive non-blocking cache. The μ-architecture simulator computes the cycle at which load and store instructions are issued to the cache simulator. The simulator then models the cache's behavior for loads and stores, and informs the μ-architecture simulator how long each load will take to produce the requested data. Note that no program data is returned by the simulator, only the time taken to obtain the data.

3.2 Simulating Speculative Execution

In the behavior described so far, direct-execution drives FastSim's μ-architecture and cache simulators and no information flows in the other direction. Speculative execution, however, requires feedback from the μ-architecture simulator. The decision when to roll-back, following a mispredicted branch, is made by the μ-architecture simulator and must control direct-execution. On the other hand, the μ-architecture and cache simulators require data collected by direct-execution before they can run. *Speculative direct-execution* is our new technique that solves this problem. The idea is to directly execute mispredicted execution paths, while recording enough information to restore processor and memory state after a misprediction is detected by the μ-architecture. Figure 3 shows where instrumentation is inserted into a target executable to perform speculative direct-execution.

All conditional branches in a target executable are replaced with instrumentation that first calls the μ-architecture simulator then consults FastSim's branch predictor and branches in the predicted direction. Mispredictions are detected immediately by comparing the original branch condition to the predicted branch direction. Instrumentation along the two arcs out of each branch detect mispredictions—the original branch instruction is used as part of this instrumentation. If mispredicted, all register values—integer, floating point and control registers—are saved in FastSim's bQ data structure. The bQ can hold register data for up to four mispredicted branches, which is all that is required by FastSim's current processor model. In the common case, in which a branch is predicted correctly, no state is saved, as the simulation never rolls-back a correctly predicted branch.

The bQ allows FastSim to restore register values when the μ-architecture simulator detects a misprediction. Other techniques are used to restore memory. Instrumentation before every store instruction records the value in memory before the store is executed (its pre-store value) and puts this data in the same sQ entry that records the store's effective address. When a misprediction is detected, all pre-store memory values following the mispredicted branch are restored, in reverse order

Using these techniques, mispredicted execution paths directly execute on a host processor, thereby collecting information needed by FastSim's μ-architecture and cache simulators. Conditional branches are executed based on the

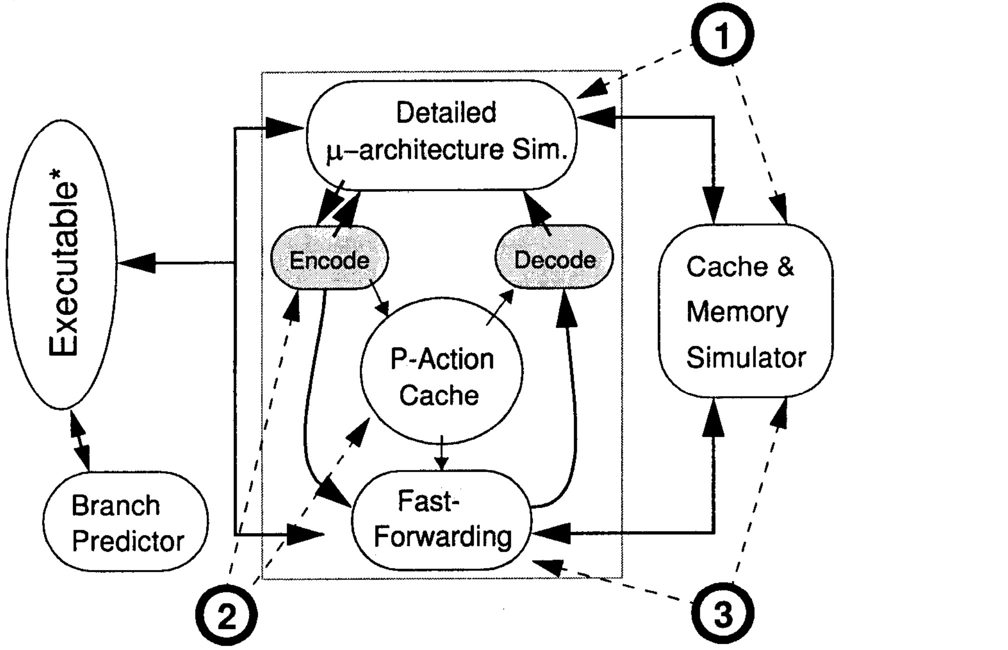

Figure 4. Memoization of FastSim's μ-architecture simulator. 1) A detailed simulator models the out-of-order microarchitecture. 2) Actions of the detailed simulator are recorded in the p-action cache, indexed by μ-architectural configurations. 3) Cached simulator actions are replayed for previously simulated configurations, thereby avoiding expensive detailed simulation.

results of prediction rather than using target program values, and only state changes subsequent to mispredictions are recorded for roll-back. When the μ-architecture simulator discovers a misprediction, FastSim rolls-back execution of the target program by restoring host memory and registers, then continues direct-execution of the target program from the corrected target of the mispredicted branch.

4. Fast-Forwarding

FastSim's primary contribution is the application of memoization to microarchitecture simulation. FastSim uses a technique called *fast-forwarding* that caches μ-architecture configurations and the resulting simulator actions for use in subsequent simulation. Figure 4 shows the structure of FastSim's fast-forwarding μ-architecture simulator.

The next section (Section 4.1) describes the construction of FastSim's μ-architecture simulator, focusing on the techniques used to centralize simulator state and reduce the space requirements for encoding this state—a necessary first step for implementing fast-forwarding. Section 4.2 describes how simulator configurations (i.e., μ-architecture state) and the resulting simulator actions are further compressed and cached into FastSim's p-action cache. This p-action cache is subsequently used to fast-forward simulation. Fast-forwarding produces the same result as detailed simulation, since μ-architecture simulator state stored in the p-action cache completely determines consequent actions of the detailed simulator. Finally, Section 4.3 discusses strategies for further reducing the size of the p-action cache.

4.1 μ-architecture Simulator

FastSim's μ-architecture simulator has been carefully designed to minimize the space needed to represent the state of its out-of-order pipeline—approximately 16 bytes plus 2 bytes per instruction in the pipeline—without reducing the complexity of its processor model. At the same time, we have minimized the amount of interaction between the μ-architecture simulator and other FastSim components. These are necessary first steps to perform fast-forwarding simulation. Larger state encodings consume more space in the cache. Interactions between the μ-architecture simulator and other components result in explicit states, which must be stored in the cache.

FastSim's μ-architecture simulator is simplified by only simulating the timing of instructions, not their functional behavior. For example, values in registers and memory are not considered by the μ-architecture simulator, although the cache simulator does use addresses recorded in the lQ and sQ.

Another simplification is that FastSim's cache simulator is not memoized. The cache simulator is called by the μ-architecture simulator as infrequently as possible through a simple interface. The cache simulator is invoked each time a load or store is chosen from FastSim's R10000-like address queue and begins its simulated execution. For loads, the cache simulator immediately returns the shortest interval (in cycles) before the requested data could become available, considering all other loads and stores already executing. The μ-architecture waits for this interval before again invoking the cache simulator for this load, although the simulator may be called in the meanwhile to handle other loads and stores. This call to the cache simulator either returns that data is now available or returns a new interval for the µ-architecture to wait. A common example is a load that first misses in the L1 cache (usually a 6 cycle delay), then misses in the L2 cache resulting in an additional delay depending on the current state of the cache. With this interface, the µ-architecture simulator is oblivious to the internal workings of its associated non-blocking cache simulator.

FastSim's µ-architecture simulator is built around one central data structure, the iQ, which contains one entry for every instruction currently in the out-of-order pipeline. Between simulated cycles, the iQ contains the entire configuration of the µ-architecture simulator, which can be used to index into FastSim's cache of memoized actions. The iQ is only an abstraction in FastSim's µ-architecture simulator that centralizes simulator state. It can be easily adapted to model a variety of pipeline designs.

Entries remains in the iQ from the time an instruction is fetched until it is retired. The iQ records an instruction's address (from which the instruction itself can be looked up) and a small amount of state information. This per-instruction state information identifies in which pipeline stage an instruction resides and the minimum number of cycles before this stage might change. For example, an integer divide instruction may be executing—in the **execute** stage—with up to 34 cycles before it finishes executing and can be retired.

At every simulated cycle, FastSim's µ-architecture simulator makes a complete pass over instructions in the iQ, in program order, from oldest to newest. Retired instructions are removed, state information for each instruction is updated for one cycle of execution, and new instructions are fetched into the queue. Most implementation constraints in FastSim's µ-architecture model can be implemented with simple counters. One constraint is that R10000's integer instruction queue (see Figure 1) holds at most 16 instructions. FastSim counts the number of integer instructions already in the **queue** stage before allowing later integer instructions to move into this stage. Since this type of constraint is recomputed every cycle, it is not part of the µ-architecture state carried between cycles.

Other constraints are more complex, but can still be implemented without explicit state information. Consider the R10000 register renaming scheme. FastSim recomputes register renaming information every cycle. This is possible, since the actual map of logical to physical registers does not affect the simulated time. The only consideration is the number of physical registers required to hold all output values of enqueued and executing instructions. FastSim builds up a new logical to physical register map every cycle, which models the physical register limitation of an R10000 and finds all true data dependencies between instructions. Similarly, a simple counter limits the pipeline to execute at most four speculative branches.

4.2 P-Action Cache and Fast-Forwarding

FastSim's processor-action cache (the *p-action cache*) stores a map from µ-architecture configurations to simulator actions that result from those configurations. A µ-architecture configuration is simply a snapshot of the iQ taken between cycles. Simulator actions are events, such as calling the cache simulator for a load or store, returning to the direct-execution, or updating the simulation cycle counter. In general, actions stored in the p-action cache represent the ways in which FastSim's µ-architecture simulator interacts with direct-execution or cache simulation, or update counters, such as the simulation cycle counter. Figure 5 shows one possible µ-architecture configuration and some of the actions resulting from this configuration.

At the start of simulation, FastSim's p-action cache is empty. µ-architecture simulation starts by running FastSim's detailed µ-architecture simulator. Whenever the detailed simulator interacts with either direct-execution or FastSim's cache simulator, it allocates a new action, describing the interaction, in the p-action cache. These actions are linked to the most recent µ-architecture configuration, which captures the simulator state before these actions executed.

When FastSim encounters a configuration already in the p-action cache, it looks up and replays the associated actions rather than using the detailed (slow) µ-architecture simulator to recompute them. We call this process *fast-forwarding*, and it produces **exactly the same results** as the detailed µ-architecture simulation. Actions are replayed in the same order—calling the cache simulator, returning to direct-execution, and updating simulation statistics—as when they were first generated.

The only variation in µ-architecture behavior arises from different cache behavior (caused by the unpredictable internal state of the cache simulator or different values in the lQ or sQ) or from different control flow in the direct-execution. These variations are checked when the actions are replayed, and previously unseen behaviors terminate fast-forwarding, so that the detailed simulator can simulate the new scenario.

Configurations stored in the p-action cache are a compressed representation of data in the iQ. This compression takes advantage of having instructions listed in program order. To encode the sequence of instruction in the iQ, we only save the starting addresses (PC and nPC) of the oldest instructions in the iQ, plus one bit per conditional branch (taken/not-taken), plus the target address of any indirect jumps. The iQ's per instruction state information can be compressed into 1.5 bytes per instruction, which subsumes the 1 bit of taken/not-taken information needed for conditional branches. Including some additional header information, this compresses a configuration to 16 bytes plus 4 bytes per indirect jump plus 1.5 bytes per instruction. New configurations are allocated at the end of a cycle in which an action was allocated. Hence at most one

Addr.	Instruction	Tag1	Tag2
0x10074	**clr** %fp	done	
0x10078	**ld** [%sp + 0x40], %l0	cache	6
0x1007c	**add** %sp, 0x44, %l1	exec	1
0x10080	**sub** %sp, 0x20, %sp	queue	
0x10084	**tst** %g1	queue	
0x10088	**be** 0x10098	queue	
0x1008c	**mov** %g1, %o0	queue	
0x10098	**sethi** %hi(0x5b000), %o0	fetch	
0x1009c	**or** %o0, 0x148, %o0	fetch	
0x100a0	**call** 0x3f378	fetch	
0x100a4	nop	fetch	
0x3f378	**save** %sp, -96, %sp		
0x3f380	**sethi** %hi(0x75c00), %o0		
0x3f384	**call** 0x56ce8		
0x3f388	**or** %o0, 0x2e0, %o0		
0x56ce8	**save** %sp, -96, %sp		
0x56cec	**sethi** %hi(0x77000), %g1		
0x56cf0	**ld** [%g1 + 0x17c], %g1		
0x56cf4	**call** %g1		
0x56cf8	restore		

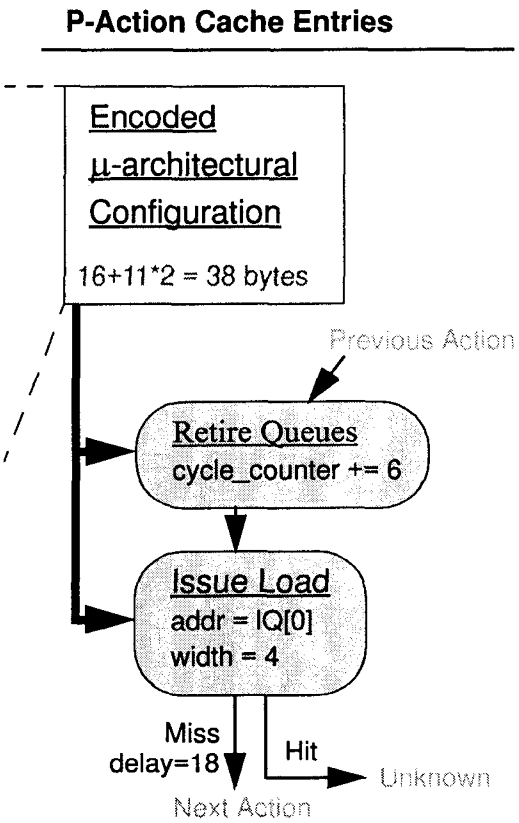

Figure 5. A µ-architectural configuration in the detailed simulator and associated entries recorded in the p-action cache. The instructions on the left are taken from the dynamic instruction stream. The top 11 instructions are currently in the pipeline and considered part of the configuration. As a result of this configuration, the µ-architecture will execute for 6 cycles, then call the cache simulator for the load at 0x10078. These actions are encoded in the p-action cache, indexed by a compressed representation of the µ-architecture configuration.

configuration is stored per simulated cycle, but several simulated cycles are often associated with a single configuration. Note that all interactions between the µ-architecture and other FastSim components take place in the last cycle associated with a configuration because of the way configurations are allocated.

Multiple actions can be associated with a single configuration. FastSim allocated 2.9–5.7 actions per configuration while simulating the SPEC95 benchmarks. The first action following a configuration identifies the number of simulated cycles associated with the configuration. Other actions, such as calling the cache simulator or returning to direct-execution, are linked in the order in which they were produced by the detailed simulator. The last action in a chain of actions associated with a configuration is linked to the first action of the following configuration, forming an unbroken chain of actions.

Variations in behavior, caused by different values from the cache simulator or changes in control flow following a branch or indirect jump, cause the fast-forwarding simulator to choose one of several possible successor actions in the action chain. For example, there are four possible outcomes following a conditional branch in the direct-execution (i.e., taken/predicted, taken/mispredicted, not-taken/predicted, and not-taken/mispredicted) and arbitrarily many return values for a load event sent to the cache simulator (i.e., possible intervals before data becomes available). If the action for a particular outcome is not in the p-action cache (e.g., the outcome has not yet occurred for the current configuration), fast-forwarding stops and detailed simulation resumes. Subsequent detailed simulation computes the µ-architecture behavior for this new outcome, and generates actions along a new branch of the action chain to handle this outcome in the future. Figure 6 illustrates the graph structure of the p-action cache in terms of configurations and action chains, and shows how new configurations and actions are linked into the existing graph structure to handle new outcomes.

4.3 Limiting P-Action Cache Size

Fast-forwarding accelerates µ-architecture simulation at the cost of increased memory consumption. Without limitation, the p-action cache can grow to hundreds of megabytes for the more complex SPEC95 benchmarks (e.g., 889MB for

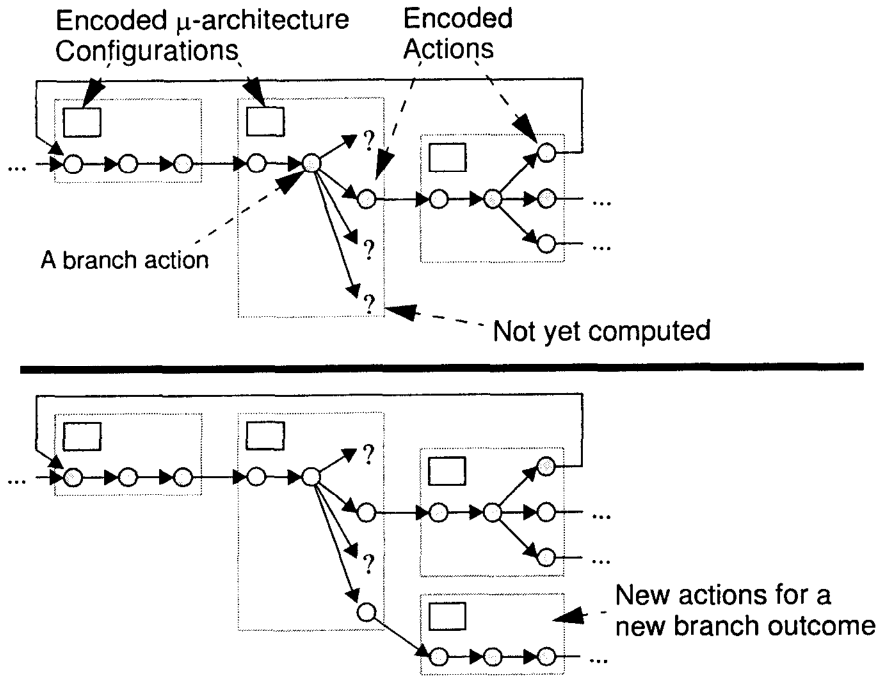
Encoded μ-architecture Configurations
Encoded Actions
A branch action
?
?
?
Not yet computed
?
?
New actions for a new branch outcome
...

Figure 6. The upper diagram is a sample graph of configurations and actions in the p-action cache. The lower diagram shows how new configurations and actions might get linked into the existing graph when the detailed simulator is invoked to handle an alternate branch outcome.

go, 296MB for gcc). Test results presented in the next section were collected on a host machine with 2GB of physical memory, but few people will have machines this large in the next few years. Consequently we investigated several techniques for handling FastSim's memory consumption.

Our first trivial p-action cache replacement policy allows unbounded growth of the p-action cache. This policy produces fast simulation times, providing the p-action cache fits in physical memory. If it does not fit in physical memory, then the OS will page (and likely thrash). A better replacement policy is to flush the p-action cache when full. This cache flush policy is easy to implement and can limit the p-action cache to any size, but there is a performance trade-off. Whenever the cache is flushed, FastSim must use detailed (slow) μ-architecture simulation to recompute actions and configurations.

A drawback of the cache flush policy is that useful actions are flushed along with never to be uses ones. An alternative policy, which also maintains pointers and avoids fragmentation, is to use a copying garbage collector. Only actions that were accessed since the last garbage collection are copied. This policy incurs extra overhead—the cost of copying—which would be offset by increased reuse of cached actions. A further refinement is to use a generational garbage collector, so frequently replayed actions will not be copied by the garbage collector as often—hopefully reducing garbage collection overhead.

5. FastSim Performance

This section describes some performance measurements of FastSim running the SPEC95 benchmarks. Experiments were run on a Sun Microsystems Ultra Enterprise E5000 with 167MHz UltraSPARC processors and 2 GBytes of physical memory. All programs, except compress, were run using their "test" input sets to reduce simulation time. Compress, which requires less time, used its "train" data set.

Table 2 shows the performance of FastSim, as compared against the original benchmarks (before they were instrumented) and against a direct-execution simulator without memoization. SlowSim is FastSim with memoization disabled—the fast-forwarding simulator was turned off and no configurations were encoded or put in the p-action cache. The table shows that memoization improves overall simulation performance by a factor of 4.9–11.9 times. Despite this dramatic speedup, the cycle counts—and all other processor statistics—generated by FastSim are identical.

Table 3 compares FastSim against the SimpleScalar out-of-order simulator [1] using similar processor and cache parameters. Despite their differences—e.g., SimpleScalar models a different instruction set—SimpleScalar provides a good baseline for measuring FastSim's performance and demonstrating the benefit of its techniques. With only direct-execution, FastSim runs 1.1–2.1 (mgrid–gcc) times faster than SimpleScalar. With fast-forwarding, FastSim runs 8.5–14.7 (fpppp–ijpeg) times faster than SimpleScalar.

One reason for memoization's large benefit is that FastSim

Benchmark	Program	SlowSim /	FastSim /	Slow / Fast
099.go	138.2	1,554.2	248.4	6.3
124.m88ksim	2.9	1,363.3	249.5	5.5
126.gcc	12.3	1,122.7	215.1	5.2
129.compress	0.3	1,304.4	218.2	6.0
130.li	8.6	1,435.6	293.5	4.9
132.ijpeg	3.3	1,837.5	199.4	9.2
134.perl	18.0	1,115.9	177.8	6.3
147.vortex	82.2	1,310.7	221.8	5.9
101.tomcatv	12.6	1,322.3	199.8	6.6
102.swim	4.5	1,460.4	191.3	7.6
103.su2cor	6.9	1,934.6	251.4	7.7
104.hydro2d	9.1	2,174.1	232.8	9.3
107.mgrid	33.3	2,569.6	215.9	11.9
110.applu	122.7	1,982.8	292.5	6.8
125.turb3d	114.1	1,992.9	254.5	7.8
141.apsi	66.8	2,758.1	357.7	7.7
145.fpppp	14.9	2,423.7	322.9	7.5
146.wave5	36.6	2,169.4	303.8	7.1

Table 2: Performance of the FastSim simulator running SPEC95 benchmarks. "Program" is time (in seconds) to execute the original, uninstrumented executables. The two simulator slowdowns show how many times slower the benchmarks ran in FastSim without memoization (SlowSim) and with memoization (FastSim). The final column is the factor by which memoization improved the simulation.

	Program				FastSim	FastSim /
Benchmark	cycles	insts.	SimpleScalar	SlowSim	Kinsts/sec.	SimpleScalar
099.go	1.14E+10	1.64E+10		76.2	477.0	
124.m88ksim	2.78E+08	4.81E+08	58.4	121.8	665.5	11.4
126.gcc	9.27E+08	1.41E+09	47.2	102.8	536.6	11.4
129.compress	2.74E+07	4.43E+07	51.9	104.5	624.6	12.0
130.li	8.87E+08	1.24E+09		100.4	491.0	
132.ijpeg	2.61E+08	4.81E+08	50.2	80.4	740.6	14.7
134.perl	1.34E+09	1.93E+09	48.1	96.5	605.6	12.6
147.vortex	5.76E+09	1.09E+10		101.1	597.3	
101.tomcatv	9.83E+08	1.55E+09	57.9	93.0	615.7	10.6
102.swim	2.35E+08	4.23E+08	55.3	64.5	492.3	8.9
103.su2cor	5.48E+08	9.14E+08	56.1	68.7	528.8	9.4
104.hydro2d	6.28E+08	8.46E+08		42.7	399.0	
107.mgrid	2.96E+09	5.26E+09	56.5	61.4	731.1	12.9
110.applu	8.53E+09	1.51E+10		61.9	419.8	
125.turb3d	8.87E+09	1.59E+10		70.0	547.8	
141.apsi	6.28E+09	8.57E+09		46.5	358.7	
145.fpppp	1.20E+09	1.99E+09	48.9	55.1	413.6	8.5
146.wave5	2.59E+09	4.64E+09		58.4	417.3	

Table 3: Program cycles and instructions are the total number of cycles and retired instructions resulting from out-of-order simulation in FastSim. Next are the average instructions retired per second by the SimpleScalar simulator, FastSim without memoization (SlowSim), and FastSim with memoization (FastSim). The last column shows FastSim's performance improvement relative to SimpleScalar.

was able to replay simulator actions for almost all instructions. Table 4 shows the fraction of instructions simulated in detail compared against the much larger proportion of instructions for which actions were replayed. For all benchmarks except gcc and ijpeg, FastSim used its detailed µ-architecture simulator for fewer than 0.1% of target instructions. However, the performance improvement does not appear to be directly attributed to this fraction (compare ijpeg).

Table 5 reports measurements of the memoization process. The first column reports size of the p-action cache. In many

Benchmark	Detailed (insts.)	Replay (insts.)	Detailed / Total
099.go	1.61E+07	1.64E+10	0.099%
124.m88ksim	6.49E+04	4.81E+08	0.013%
126.gcc	4.40E+06	1.41E+09	0.311%
129.compress	3.41E+04	4.42E+07	0.077%
130.li	4.17E+04	1.24E+09	0.003%
132.ijpeg	9.78E+05	4.80E+08	0.203%
134.perl	4.34E+05	1.93E+09	0.022%
147.vortex	8.37E+05	1.09E+10	0.008%
101.tomcatv	4.02E+04	1.55E+09	0.003%
102.swim	9.93E+04	4.23E+08	0.023%
103.su2cor	2.35E+05	9.14E+08	0.026%
104.hydro2d	2.41E+05	8.46E+08	0.028%
107.mgrid	6.72E+04	5.26E+09	0.001%
110.applu	1.40E+05	1.51E+10	0.001%
125.turb3d	8.75E+04	1.59E+10	0.001%
141.apsi	1.52E+05	8.57E+09	0.002%
145.fpppp	2.53E+05	1.99E+09	0.013%
146.wave5	2.39E+05	4.64E+09	0.005%

Table 4: Instructions that FastSim simulated by fast-forwarding (Replay) and by detailed simulation (Detailed). The last column is the fraction of instructions that FastSim simulated in detail.

	P-Action	# Static	# Static	Actions /	Cycles /	Dyn. Chain Length	
Benchmark	Cache (MB)	Configs.	Actions	Config.	Config.	Avg.	Max.
099.go	889.4	5,096,560	14,764,742	3.5	1.5	17,300	1,882,101
124.m88ksim	4.6	26,660	89,180	3.6	1.5	190,974	592,750,035
126.gcc	296.0	1,774,016	5,353,318	3.5	1.5	5,354	1,618,693
129.compress	2.8	13,475	57,429	3.5	1.4	35,711	5,231,549
130.li	3.2	18,944	60,581	3.4	1.4	645,873	49,204,501
132.ijpeg	199.5	816,075	3,343,805	3.7	1.5	19,142	2,679,671
134.perl	142.9	559,449	3,205,519	3.6	1.6	51,189	13,495,080
147.vortex	108.6	557,362	2,037,172	3.7	1.3	259,160	32,527,035
101.tomcatv	5.6	27,191	114,445	3.9	1.4	1,934,565	619,213,774
102.swim	16.8	79,002	262,422	4.5	1.2	426,471	491,018,150
103.su2cor	32.8	156,603	642,213	4.1	1.1	178,467	182,556,421
104.hydro2d	35.5	174,422	679,767	4.5	1.2	244,809	194,389,159
107.mgrid	9.5	47,035	192,098	3.4	1.0	3,788,172	322,900,913
110.applu	19.5	94,893	375,606	4.7	1.0	7,414,106	38,010,020,845
125.turb3d	10.4	50,275	205,181	4.1	1.2	10,490,459	2,555,810,836
141.apsi	20.3	98,550	409,502	4.7	1.0	5,122,367	784,023,417
145.fpppp	25.4	127,051	460,440	3.8	1.0	272,104	27,784,740
146.wave5	38.3	180,398	752,237	4.9	1.0	1,049,836	458,444,554

Table 5: Measurements of memoization. "P-Action Cache" is the total memory used to record configurations and actions. The next two columns report the static number of configurations and actions allocated. "Actions/Config." is the average dynamic number of actions associated with each configuration, and "Cycles/Config." is the dynamic number of simulation cycles per configuration. The final two columns report average and maximum lengths of action chains played back without stopping to perform detailed simulation.

programs, it was manageably small. However, in five applications it grew to over one hundred megabytes. The go benchmark generated nearly 900MB of p-action data, by far the most. Fortunately, a simple cache replacement policy, discussed later, can greatly reduce the memory requirements for simulating most benchmarks.

Table 5 also reports the number of actions and configurations statically generated for each program. Although the number of actions and configurations varied greatly between programs, the dynamic number of actions per configuration remains relatively consistent—between 3.4 and 4.9—for all benchmarks. This number is a measure

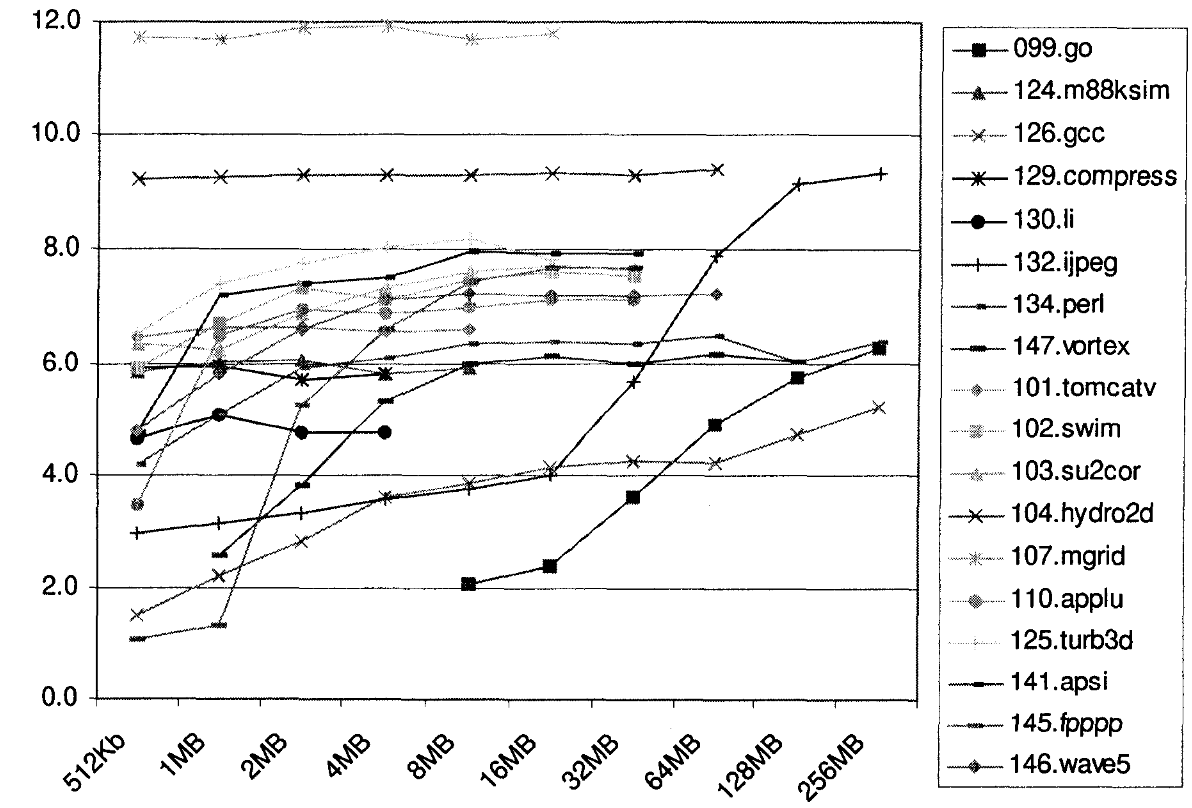

Figure 7. Graph of FastSim's speedup due to memoization with limited p-action cache space using the cache flush replacement policy.

of how much work can be directly replayed at a memoized configuration. By dividing the actions per configuration by the dynamic cycles per configuration, we get an indication of how much simulated work is performed by the μ-architecture each cycle. The average actions per cycle over all the integer benchmarks is 2.4, compared to 3.9 for floating-point benchmarks, which corresponds to our pipeline's ability to execute more instructions in parallel if there is a mix of integer and floating point operations. The final two columns report the average and maximum number of chained actions that fast-forwarding was able to replay without calling the detailed simulator. The large values in both columns reflect the extremely long intervals during which only previously cached configurations were encountered.

Figure 7 shows the result of limiting p-action cache size using the cache flush on full policy. The graph shows simulator speed-up (non-memoized/memoized time) for p-action cache sizes ranging from 512Kb to 256MB. Most benchmarks could tolerate an order-of-magnitude reduction in p-action cache size with little or no impact on simulator performance. This includes the go benchmark, which naturally uses 889MB but shows no slowdown when limited to 256MB and only moderate slowdown at 64MB. A few benchmarks did not perform well with reduced cache sizes—notably ijpeg, which slowed dramatically with only moderate cache reductions—although even these benchmarks ran several times faster than simulation without memoization for all but the most restrictive cache sizes.

We also tried garbage collecting the p-action cache, keeping only those configurations and actions that had been used since the last garbage collection. Despite the potential savings from keeping useful actions in the cache, FastSim's performance with garbage collection was nearly identical to its performance using the simple flush on full policy. Furthermore, since we used a copying garbage collector, the total memory in use during a collection could be up to twice the maximum allowed p-action cache size. Taking this into account, garbage collecting the p-action cache is almost always worse than simply flushing it. Experiments with a generational garbage collector were no better. The additional complexity—e.g., handling pointers from older generations back to younger generations—offset any savings from copying smaller portions of the cache.

The garbage collector's poor performance can be attributed to two factors: Garbage collections (or cache flushes) are infrequent and few actions survive each collection. 1-4 garbage collections or cache flushes occur when the p-action cache is sized just smaller than the maximum space used by a benchmark. For each factor of two decrease in cache size there is only a 3.8 times increase in the number of collections on average. Infrequent collections means that few configurations are discarded over a program's execution and that the amortized cost of regenerating them is small. Another factor is that only 18% of the p-action cache survives each garbage collection on average. Little is gained by finding and copying these actions over flushing the cache and regenerating them, when compared to the total simulation time.

6. Conclusion

FastSim uses two, well-known, but previously unapplied techniques to greatly speed the detailed, accurate microarchitectural simulation of an out-of-order uniprocessor. FastSim demonstrates that direct-execution is compatible with out-of-order simulation, although the benefits are small because of the cost of simulating a complex microarchitectural model.

FastSim also directly attacks this cost, by using memoization to dramatically reduce the cost of detailed simulation. A key observation is that out-of-order microarchitecture configurations are often repeated and result in identical simulator behavior. By caching these configurations and their corresponding simulator actions, subsequent visits to a configuration can be replayed many times faster. This fast-forwarding speeds processor simulation by a factor of 5–12 times, at the cost of increased memory consumption.

Experiments with cache replacement policies show that most benchmarks only need a fraction of the p-action data they generate over the course of simulation. A simple flush on full policy is sufficient to limit the p-action cache size without a large impact on performance. More complex cache replacement policies, such as copying garbage collection, are not worth the effort, since they are difficult to implement and perform no better than the simple flush on full policy.

7. Acknowledgments

We would like to thank Mark Hill and David Wood for their suggestions and guidance. Our thanks also goes to Sarita Adve for discussions on incorporating direct-execution into out-of-order processor simulation.

8. References

[1] Doug Burger, and Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0," *University of Wisconsin-Madison Computer Sciences Tech Report #1342*, June, 1997.

[2] Robert F. Cmelik, and David Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," in the *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994.

[3] T. M. Conte, M. A. Hirsch, and K. N. Menezes, "Reducing state loss for effective trace sampling of superscalar processors," in the *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, Austin, TX, October 1996.

[4] Steve Herrod, Mendel Rosenblum, Edouard Bugnion, Scott Devine, Robert Bosch, John Chapin, Kinshuk Govil, Dan Teodosiu, Emmett Witchel, and Ben Verghese, "The SimOS Simulation Environment," Computer Systems Laboratory, Stanford University, 1996.

[5] James R. Larus and Eric Schnarr, "EEL: Machine-Independent Executable Editing," in the *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, June 1995.

[6] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Steve Huss-Lederman, Mark D. Hill, James R. Larus, and David A. Wood, "Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator," in the *Workshop on Performance Analysis and Its Impact on Design (PAID)*, June 1997.

[7] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve, "The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodolgy," in the *Proceedings of the 3rd International Symposium on High Performance Computer Architecture (HPCA)*, February 1997.

[8] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve, "RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors," in the *Proceedings of the 3rd Workshop on computer Architecture Education (held in conjunction with the 3rd International Symposium on High Performance Computer Architecture)*, February 1997.

[9] Sun Microsystems, The SPARC Architecture Manual (Version 8), December 1990.

[10] Yeager, "The Mips R10000 Superscalar Microprocessor," in *IEEE Micro*, April 1996.