Microprocessor designers use multiple simulation tools with varying degrees of modeling detail ranging from the instruction set of the microprocessor to the circuit implementation. Here, we focus on tool design for the development of microarchitectures, which implement the instruction set. Microarchitecture design involves both functional and performance simulators. A functional simulator models a machine's architecture, or instruction set, with functional correctness. A performance simulator models the machine organization, or microarchitecture, and is concerned with machine performance. Sometimes these performance simulators are also referred to as cycle-accurate simulators to reflect their concern with timing issues.

Background
To reduce simulation time, designers have traditionally implemented performance simulators as trace-driven tools; that is, their inputs are traces of dynamic instructions, without full-function simulation capability. This type of simulator processes an execution trace of a benchmark to produce measurements of the dynamic use of machine resources, the throughput at various pipeline stages, and ultimately the performance of the machine, measured in IPC (average instructions per cycle). Figure 1 illustrates one such tool called MW (microarchitecture workbench), which was developed at Carnegie Mellon and which we have used extensively in our microarchitecture research. An earlier work validated the MW PowerPC 604 performance simulator used in this study against an actual PowerPC 604 system.

Since the 1980s, trace-driven performance simulators have become popular for assessing microprocessor performance. Avoiding full-function simulation, these performance simulators can process extremely long traces in a reasonable amount of time. However, in recent years four weaknesses of trace-driven performance simulators have emerged.

First, the complexity of microarchitecture has increased dramatically, causing trace-driven performance simulation to become quite time consuming, thus reducing the simulation-time benefit of the trace-driven approach. Since trace-driven simulation has become much more time consuming than functional simulation, it is possible to include functional capabilities in the traditional trace-driven simulator without significantly impacting simulation time.

Second, there are inherent limitations to the capabilities of a trace-driven simulator. Typically, such a simulator processes only the trace of instructions executed (I trace) and the trace of memory addresses referenced (M trace).
Most contemporary microprocessors employ some form of dynamic branch prediction. During program execution, it is possible for the branch predictor to mispredict and send the machine temporarily on a mispredicted path. If a misprediction is detected, during branch resolution, the machine recovers by flushing the mispredicted path instructions.

Since both the I and M traces contain only non-speculative instructions, it is impossible to simulate the processing and the dynamic effects of mispredicted path instructions using only these traces. To correct this problem, some trace-driven simulators insert a fixed number of branch stall cycles or inject artificial instructions to mimic the mispredicted path instructions. Both of these approaches only approximate the actual machine behavior. The better solution is to simulate the branch predictor, the associated speculative processing of instructions, and the recovery mechanism when misprediction is detected. This involves the direct simulation of the mispredicted path instructions, including their fetching, decoding, dispatching, execution, and flushing, which requires a full-function performance simulator.

The third weakness of a trace-driven performance simulator is the lack of instruction execution results. Both data-dependent instruction execution and more recent value prediction techniques require instruction results to be accurately simulated. Unfortunately, trace-driven simulators cannot accurately provide instruction results using data traces.

Finally, researchers have proposed systematic methods for generating instruction sequences that thoroughly test the microarchitecture. These methods rely on an accurate performance model of the microarchitecture to confirm the effectiveness of the test sequences. Since trace-driven simulators do not model the execution of mispredicted instructions, it is impossible to validate any speculative device or recovery mechanism.

Motivations

There is a serious need for a new performance simulation tool that can address these shortcomings. The key requirements for the new tool include full-function simulation and cycle-accurate microarchitecture modeling. The addition of full-function capability should not significantly increase the total simulation time. Direct simulation of mispredicted path instructions and value prediction mechanisms could then be supported. Such a simulation tool can also be used to validate speculation and recovery mechanisms.

The functional MW (fMW)

This article presents the design and implementation of a new performance simulator with full-function capability called fMW (functional microarchitecture workbench). This tool replaces our original MW tool. Both MW and fMW are based on the PowerPC architecture and faithfully model all the PowerPC instructions. The fMW builds on MW by incorporating a customized version of the PSIM functional simulator and by extending the capabilities of the original MW. (See Figure 2.) This coupling of PSIM and MW guarantees accurate execution of instructions (including runtime data values) and cycle-accurate timing simulation. Further-
more, the development of this tool enables future research involving multiple instruction streams, such as simultaneous multithreading and dual-path execution, as well as research on value prediction\(^8,9\) that requires runtime register and memory values.

To demonstrate the effectiveness of the fMW tool, we present two recent research studies. The first study investigates the effects of mispredicted path instructions on the cache hierarchy,\(^13\) while the second concerns the validation of speculation and recovery mechanisms.\(^10,11\) The capabilities of fMW are quite similar to that of SimpleScalar;\(^4\) however, fMW is based on the PowerPC architecture and executes binaries compiled for netBSD. PSIM’s ability to translate PowerPC system calls to the local machine\(^12\) makes fMW highly portable and does not require native execution on PowerPC platforms.

Implementation details

MW models the pipeline resources and cache hierarchy, and simulates at the level of machine cycles. PSIM, on the other hand, operates at the instruction level and interprets or “executes” one instruction at a time. PSIM maintains the architectural state by tracking all register and memory updates. It has no knowledge of the cache hierarchy or any other features of the microarchitecture.

As PSIM executes each instruction, it bundles the instruction with its execution results and passes it on to MW. When the branch predictor mispredicts a branch instruction, MW instructs PSIM to traverse the mispredicted path. PSIM checkpoints its current state, for later recovery and begins execution on the mispredicted path.

In Figure 3, the branch at the end of basic block A is mispredicted. MW instructs PSIM to checkpoint the architectural state and begin execution on the mispredicted path (basic block B). Later, the MW executes the branch instruction, detects the misprediction, and corrects it by flushing the mispredicted path instructions. As the MW simulation recovers, PSIM reverts to the saved state and begins execution on the correct path (basic block C). In this manner, all mispredicted path instructions are accurately accounted for and directly simulated in MW, while the machine state and data values stored in PSIM are correctly maintained.

Implementation difficulties

Simulating instructions on a mispredicted path creates several interesting problems. System calls, interrupts, exceptions, and unnatural data values encountered on a mispredicted path can cause irreparable state changes. For example, if an exit() call is encountered, PSIM will execute the call and terminate execution, ending the simulation. Other problems include accessing unmapped memory due to incorrect address values, and exceptions caused by unnatural incorrect data values. To alleviate these problems, PSIM suspends all system calls, interrupts, and exceptions when executing mispredicted paths.

fMW performance

The interaction overhead between PSIM and MW, and the execution of mispredicted paths, slightly reduce the simulation speed. The original trace-driven MW tool simulated approximately 20,000 to 25,000 instructions per second (KIPS). The fMW can simulate approximately 15 to 20 KIPS. Both measurements are performed on 200-MHz Pentium Pro machines running Linux. The fMW’s speed is encumbered by years of software evolution. Extensive code optimization is currently underway, which will significantly improve the speed of the fMW tool.

fMW applications

Two studies demonstrate the usefulness of fMW. The first examines the effects that mispredicted path instructions have on the cache hierarchy. Figure 3 shows the checkpointing of PSIM as implemented in the fMW framework.
hierarchy.13 The second study quantifies the coverage achieved by microarchitecture validation test sequences.10,11 The fMW tool enabled both of these studies, which were not possible with the earlier MW tool.

Cache effects of mispredicted paths

To achieve high instruction fetch bandwidth, modern microprocessors employ branch predictors to speculatively fetch instructions beyond conditional branches. If these speculative instructions are determined to be on a mispredicted path, they must be invalidated and removed from the machine.

Mispredicted path instructions can affect many parts of the machine, particularly the functional units, branch predictors, and caches. This study examines the effect of mispredicted path execution on performance and the cache hierarchy.

Previous work. A handful of studies14-16 have examined the effects of mispredicted paths; however, each of these efforts is hampered by inadequate modeling techniques. One simulator15,16 is trace driven, leading to several inaccuracies. Since trace-driven simulators cannot execute mispredicted path instructions, Pierce and Mudge injected a fixed number of instructions to emulate the mispredicted path.15 However, the number of cycles a given machine spends on each mispredicted path depends on the aggressiveness of the branch predictor and the branch resolution latency. Fixing the branch resolution latency at a constant number of instructions introduces significant error. Nevertheless, using this method, Pierce and Mudge found that the mispredicted path instructions tend to prefetch the data cache.14 However, the methods that measured these effects had serious limitations and are inherently inaccurate. The fMW removes such inaccuracies by directly simulating the mispredicted path instructions in the machine model. The following section summarizes our experimental results; an earlier work provides more detailed results.13

Experimental results. We used the SPECint95 benchmark suite for our experiments. Table 1 summarizes the input sets and run lengths of each benchmark. To focus the current study on the effects of speculative execution and to emphasize the effect of mispredicted instructions in the pipeline, we extended the PowerPC 604 microarchitecture to remove resource constraints and widened it to allow a greater number of in-flight instructions. The instruction window is limited to 512 instructions with an unlimited number of functional units and an unlimited number of rename registers. Instruction fetch and dispatch widths are increased to 16 instructions per cycle. A 64-entry, fully associative branch target address cache (BTAC) and a 512-entry branch history table (BHT) handle branch prediction.

The memory hierarchy includes a perfect (100% hit rate) main memory; a 32-Kbyte, four-way set-associative level-1 instruction cache (IL1); a 32-Kbyte, eight-way set-associative level-1 data cache (DL1); and a 512-Kbyte, eight-way set-associative, unified level-2 cache (UL2). All caches use a write-back, write-allocate scheme. Access latencies are 1, 3, and 100 cycles for the L1, L2, and main memory respectively.

Due to space constraints, we discuss only instruction cache results here. For more
To determine when the instruction cache is polluted or prefetched, we used two copies of the memory system during simulation. One maintains the memory state for both correct path and mispredicted path instructions, while the other maintains the memory state for only correct path instructions. Any latency difference between the two memory systems is due to mispredicted path instructions. If the access latency of the correct path-only memory is greater than the latency of the memory updated by the mispredicted path, the mispredicted path has prefetched into the instruction cache. If the opposite occurs, the mispredicted path has polluted the instruction cache. Prefetching causes a performance improvement and is considered a gain to be measured in cycles with pollution being a loss.

Table 2 shows the instruction cache access discrepancies observed when mispredicted path instructions are simulated in fMW. The table lists the number of prefetching and polluting accesses along with the average number of cycles gained (for each prefetch access) or lost (for each polluting access). The “Net Change” column records the overall cache latency cycle changes caused by the mispredicted path accesses [(prefetching accesses × cycles gained per access) – (polluting accesses × cycles lost per access)]. A positive net change indicates a reduction in cycle count (performance gain), whereas a negative change indicates an increase in cycle count (performance loss).

The results in Table 2 show that most benchmarks have a positive net change due to mispredicted path cache accesses, however the extent varies greatly. Only perl and gcc show negative net changes.

Examination of the average number of cycles lost/gained shows that most of the prefetching accesses are prefetched from main memory (100-cycle latency), since the average number of cycles gained per prefetching access is in the range of 25 to 50 cycles. On the other hand

detailed and extensive results, please refer to Combs, Bechem, and Shen.13

The number of cycles shown in the “Net Change” column of Table 2 does not translate directly into IPC change. The dynamic execution of the benchmark determines the effect of each instruction fetch. When the machine is stalled, performance is not affected by cache pollution because the next instruction is not currently needed. Cache prefetching also has a diminished impact when the next instruction is not currently needed.

Figure 4 shows the actual impact of mispredicted path execution on the IPC. The percent of change ranges from the greatest increase in go of 12.0% to the perl decrease of −7.93%. The average across all benchmarks is approximately 1.0%. This IPC increase is due to the positive effects of prefetching, while the reduction of IPC is due to cache pollution effects induced by the mispredicted path instructions.

Although the gains are positive for most benchmarks, the IPC changes vary significantly from benchmark to benchmark. Pierce and Mudge observed that all benchmarks had more prefetching than pollution and thus

---

**Table 2. Instruction cache access discrepancies caused by mispredicted path instructions.**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Pollution accesses</th>
<th>Avg. cycle loss/access</th>
<th>Prefetch accesses</th>
<th>Avg. cycle gain/access</th>
<th>Net change (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>207</td>
<td>1.00</td>
<td>110</td>
<td>24.44</td>
<td>2,481</td>
</tr>
<tr>
<td>gcc</td>
<td>2,356,036</td>
<td>14.92</td>
<td>543,954</td>
<td>34.35</td>
<td>−16,479,381</td>
</tr>
<tr>
<td>go</td>
<td>349,958</td>
<td>3.95</td>
<td>188,388</td>
<td>50.51</td>
<td>8,133,593</td>
</tr>
<tr>
<td>ljpeg</td>
<td>108,252</td>
<td>2.88</td>
<td>42,341</td>
<td>38.56</td>
<td>1,320,485</td>
</tr>
<tr>
<td>li</td>
<td>659</td>
<td>7.96</td>
<td>203</td>
<td>34.36</td>
<td>1,730</td>
</tr>
<tr>
<td>m88ksim</td>
<td>857</td>
<td>6.28</td>
<td>361</td>
<td>34.25</td>
<td>6,983</td>
</tr>
<tr>
<td>perl</td>
<td>96,656</td>
<td>29.60</td>
<td>18,885</td>
<td>45.63</td>
<td>−1,999,238</td>
</tr>
<tr>
<td>vortex</td>
<td>278,049</td>
<td>10.16</td>
<td>99,737</td>
<td>34.84</td>
<td>650,993</td>
</tr>
</tbody>
</table>
concluded that the cache effects of mispredicted path instructions would always be beneficial. This observation conflicts with the data of Figure 4.

To accurately assess the IPC performance impact, direct simulation of the mispredicted path instructions is essential. Using fMW, we found that the magnitude of the effect on IPC varies widely from benchmark to benchmark, ranging from $-8\%$ to $+12\%$. These results clearly differ from those in previous studies that did not perform direct cycle-accurate simulation of the mispredicted path instructions. These results demonstrate the usefulness and effectiveness of the new fMW tool at yielding more accurate and complete simulation data.

Validation of speculation and recovery

Currently, the microprocessor industry relies heavily on simulation for validating microarchitecture mechanisms. Validation involves exercising the simulation models and examining the outcome. To exercise these models for validation, the industry uses instruction sequences or test sequences as input stimuli to the simulation models. Generally, researchers used three types of test sequences. First, real application programs can be used as test sequences. While these programs may represent the actual user workload, they may not fully exercise the machine. Second, designers generate test programs to probe specific areas of the machine and to test the “corner conditions” of machine behavior. Third, randomly generated programs supplement the previous two types of test sequences.

Using real application programs and randomly generated programs as test sequences can be very inefficient. Explicitly generated test sequences are generated in a very ad hoc fashion based on the intuitive knowledge of the designer. Regardless of the test sequences used, there is no rigorous way to quantitatively assess their coverage at the microarchitecture level. There is a real need for a systematic method to generate highly efficient test sequences for microarchitecture validation that is based on rigorous models of microarchitecture mechanisms and can yield quantitative coverage figures.

Validation method. Recently, we presented a systematic method for generating efficient test sequences that would rigorously validate contemporary superscalar microarchitectures. These microarchitectures employ deep pipelines, aggressive speculation, and out-of-order execution. This method operates at the microarchitecture level and is intended to validate the behaviors of the key microarchitecture mechanisms:

- dynamic branch prediction,
- register renaming, and
- out-of-order instruction issuing from reservation stations and maintaining precise exception via the reorder buffer.

To handle the complexity of a modern microarchitecture, we partitioned the machine into a set of critical buffers, including the branch target address cache, the branch history table, register rename buffers, reservation stations, and the reorder buffer. Figure 5 (next page) illustrates a typical superscalar pipeline with these critical buffers. We view these buffers as critical because the bulk of the machine control logic manages the reading and writing of these buffers.

Each of these critical buffers has multiple symmetrical entries. In our validation method the behavior (reading and writing) of each buffer entry is modeled with a simple finite-state machine (FSM). This FSM model is used to automatically generate an efficient test
sequence that fully exercises the buffer behavior. This approach resembles automatic test pattern generation (ATPG) for logic testing and borrows some ideas from functional testing of iterative structures.

Traditional logic testing tests an iterative array-structured circuit by partitioning the array into its symmetrical modules. Then each module is separately and identically tested. We borrowed this concept for our validation method. Since the buffer entries are symmetrical, a buffer is validated by separately and identically validating each of its entries. Each buffer is validated by exercising all the FSM state transitions for each buffer entry. A test sequence of instructions is generated that will force a buffer entry to traverse all of its state transitions. The state transitions are verified by monitoring the simulation process and examining the simulation outcome. This process is repeated for each of the buffer entries, then for each of the buffers. The coverage of a test sequence is the percentage of all possible FSM state transitions exercised by that test sequence of instructions and verified by the simulation tool.

In summary, our ATPG-based validation method involves

1) partitioning a microarchitecture into its critical buffers,
2) generating the FSM models for each entry of all the key buffers,
3) constructing a transition tour for each FSM model, and
4) synthesizing a test sequence of instructions to carry out each transition tour.

All the test sequences are then used to exercise the simulation model of the microarchitecture to verify the coverage achieved.

FSM models. This study applies the FSM method to the register rename buffer and the reorder buffer of the PowerPC 604 microarchitecture. Figure 6a illustrates the FSM diagram that models the behavior of each entry of the register rename buffer. An entry is free until the dispatch unit allocates it for an instruction in the dispatch stage. The entry remains allocated until the instruction finishes.

There are two states for an allocated entry. At the time of renaming, each newly allocated rename entry will always hold the most recent (MR) value for the renamed register denoted by the MR Allocation state. If a rename entry is allocated to a register that is later renamed by another instruction, the previously allocated entry will no longer hold the most recent value and will therefore transition from the MR Allocation state to the NonMR Allocation state. Once the instruction finishes, the content of the rename entry becomes valid, which causes a transition from MR Allocation (NonMR Allocation) to MR Valid (NonMR Valid). The FSM stays in the valid state until the result is written to the register file (WB transition) or a prior instruction causes an exception that requires all subsequent instructions to be discarded (discard transition).

Figure 6b shows the FSM diagram that models each entry of the reorder buffer. A reorder buffer entry is available for allo-
cation if its FSM is in the Free state. When an instruction is dispatched to a reservation station, a reorder buffer entry is allocated, and the entry transitions from Free to Allocation. The FSM will transition from the Allocation state to the Execute state and finally to the Finish state as the instruction executes and finishes. Mispredicted path instructions are removed from the reorder buffer after branch execution. The discard transition is traversed when an instruction is flushed from the reorder buffer.

Experimental results. In earlier works, we used the original version of MW to simulate the microarchitecture. Given the limitation of the original MW, the simulation of certain FSM state transitions was not possible. The “discard” arcs in the FSM diagrams of Figure 6a,b indicate these transitions. Consequently, the coverage of these transitions by the test sequences cannot be verified, resulting in relatively low coverage of the total number of state transitions. With the availability of the fMW tool, we can now simulate and verify all of these state transitions. Here, we highlight the results and benefits of using the fMW tool for determining the coverage of the test sequences in performing validation of the PowerPC 604 microarchitecture.

Using the ATPG-based validation method, we generated a test sequence totaling 97,000 instructions so we could validate the rename buffer and the reorder buffer. For comparison we also used the SPECint benchmarks as a second test sequence. Figure 7 shows the coverage results for the ATPG sequence and the SPECint benchmarks for both the register rename buffer and the reorder buffer. The figure also shows verifiable coverages using both the original MW tool and the new fMW tool.

We can make two key observations. First, the SPECint benchmarks, although almost four orders of magnitude longer (totaling 685,000,000 instructions), achieve much lower coverages than the ATPG test sequence. Second, using fMW, we can verify much higher percentages of the state transitions in the FSM models. Verifiable coverage increases for both the ATPG and SPEC int sequences using fMW.

Using our ATPG sequence, the original MW can only achieve a verifiable coverage of 64% of the transitions for the rename buffer, while the fMW tool achieves 100% coverage. The register rename buffer includes 12 entries for renaming general-purpose registers and eight entries for renaming condition code registers. Each entry is validated for renaming each possible architectural register. With the original version of MW, only seven out of 11 transitions of the rename buffer FSM are not verified.
trackable during simulation. Therefore, the maximum coverage that can be verified for any sequence using the original MW is at best 64%. The four unverified transitions require the rename entry to be first allocated and updated but later discarded due to the mis-prediction on a preceding branch instruction. The fMW tool can easily simulate the four discard transitions, and reports 100% verifiable coverage for the ATPG sequence. For the SPECint benchmarks, the verifiable coverage of the rename buffer using the original version of MW is 38%. With the fMW tool the verifiable coverage increases to 55%.

The reorder buffer has 16 entries. The ATPG sequence using the original version of MW achieves 57% verifiable coverage, while using fMW achieves 100% verifiable coverage. For the SPECint benchmarks, the verifiable coverage of the reorder buffer using MW is 54%. With the fMW tool, the verifiable coverage increases to 88%. The verifiable coverage of the rename buffer and the reorder buffer increases for both the ATPG sequence and the SPECint benchmarks, when using fMW instead of the original MW. Using fMW, many more FSM transitions can be verified via simulation, which was not possible with the original MW tool. These results clearly demonstrate the usefulness and effectiveness of the new fMW tool for supporting simulation-based microarchitecture validation.

Our new fMW tool will be the workhorse for our future research on advanced microarchitecture techniques. We plan to use it to accurately evaluate the effectiveness and validate the correctness of new microarchitecture mechanisms. The fMW will be an effective tool for supporting value prediction techniques, trace prediction techniques, multiple-path instruction execution, simultaneous multithreading, and simulation-based microarchitecture validation. We believe that a tool like fMW is absolutely essential for future microarchitecture research.

Acknowledgments
Our research was supported in part by ONR (N 00014-95-1-1112 and N 00014-96-1-0347) and by Intel. This work has benefited from the generous donation of a large number of Pentium II systems from Intel.

References
Candice Bechem is currently a member of Motorola's Engineering Rotation Program and earlier worked on our project while she was at Carnegie Mellon University. She received her MS from the ECE Department at Carnegie Mellon University. Bechem was IEEE student branch president at the University of Illinois, Urbana-Champaign, where she received her BS in computer engineering.

Jonathan Combs is currently a component design engineer at Intel's Texas Development Center in Austin, Texas, working on a future-generation IA-32 processor implementation. He also worked on this project while at Carnegie Mellon. Combs received his MS from the ECE Department of Carnegie Mellon and his BS in computer engineering from the University of Illinois Urbana-Champaign.

Noppanunt Utamaphethai is a PhD candidate in the ECE Department at Carnegie Mellon University. Currently, he is working on the ATPG-based validation of microprocessors. He received his BS from Brown University and his MS from Carnegie Mellon.

Bryan Black is a PhD candidate in the ECE Department of Carnegie Mellon. His research interests span computer systems and tropical fruits. He spent four years at Motorola as a design engineer and was a member of the PowerPC 604 design team before returning to Carnegie Mellon for his PhD degree. Black received his BS and MS from Carnegie Mellon.

R.D. Shawn Blanton is an assistant professor in the ECE Department of Carnegie Mellon and a member of the Center for Electronic Design Automation. He has worked on the design and test of complex digital systems with General Motors Research Labs, AT&T Bell Labs, and Intel. Blanton received a BS from Calvin College, an MS from the University of Arizona, and a PhD in computer science and engineering from the University of Michigan, Ann Arbor. He is the recipient of a NSF Career Award.

John Paul Shen, a professor in Carnegie Mellon's ECE Department, heads the university's Microarchitecture Research Team (CMU ART). He spent several years at Hughes and TRW. His current research interest is high-performance microarchitectures. Shen received a BS degree from the University of Michigan and MS and PhD degrees from the University of Southern California, all in electrical engineering. He is an IEEE fellow and a member of the IEEE Computer Society.

Direct questions about this article to John Paul Shen, Electrical and Computer Engineering Department, Carnegie Mellon University, Schenley Park, Pittsburgh, PA 15213; shen@ece.cmu.edu.