# MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling

Eric Larson
larsone@eecs.umich.edu

Saugata Chatterjee
saugatac@eecs.umich.edu

Todd Austin
austin@umich.edu

Advanced Computer Architecture Laboratory
Electrical Engineering and Computer Science
University of Michigan

## Abstract

MASE (Micro Architectural Simulation Environment) is a novel infrastructure that provides a flexible and capable environment to model modern microarchitectures. Many popular simulators, such as SimpleScalar, are predominately trace-based where the performance simulator is driven by a trace of instructions read from a file or generated on-the-fly by a functional simulator. Trace-driven simulators are well-suited for oracle studies and provide a clean division between performance modeling and functional emulation. A major problem with this approach, however, is that it does not accurately model timing dependent computations, an increasing trend in microarchitecture designs such as those found in multiprocessor systems. MASE implements a micro-functional performance model that combines timing and functional components into a single core. In addition, MASE incorporates a trace-driven functional component used to implement oracle studies and check the results of instructions as they commit. The check feature reduces the burden of correctness on the micro-functional core and also serves as a powerful debugging aid. MASE also implements a callback scheduling interface to support resources with non-deterministic latencies such as those found in highly concurrent memory systems. MASE was built on top of the current version of SimpleScalar. Analyses show that the performance statistics are comparable without a significant increase in simulation time.

## 1. Introduction

Computer system simulation is a vital technology in the modern computer system design cycle. The flexibility to quickly update software simulation models speeds the evaluation of design changes, permitting architects to explore large portions of the design space. Software modeling infrastructure also decouples hardware and software design efforts so that software development may proceed in parallel with actual hardware design, thereby reducing time to market for products with hardware and software components.

A microprocessor performance model is a software representation of a hardware design. It tracks the timing of instructions and data through the processor pipeline and memory system. Very detailed performance models may also include I/O device models such as disks and network interfaces. It is important for the performance model to be closely matched to the hardware that is being emulated. An inaccurate model can lead to incorrect or misleading research results [6].

Figure 1 illustrates a trace-driven modeling infrastructure[1], the most prevalent simulator organization. The performance model is driven by an instruction trace that represents the dynamic stream of instructions executed for a specific processor architecture and workload. Traces are either read from a file [22], created through the use of instrumented hardware [1], or generated on-the-fly by emulating a program [14]. Sim-

1. We use a broad definition of trace-based simulation. We consider any simulation environment that decouples creation of the dynamic instruction stream from the model that computes timing to be trace-based.

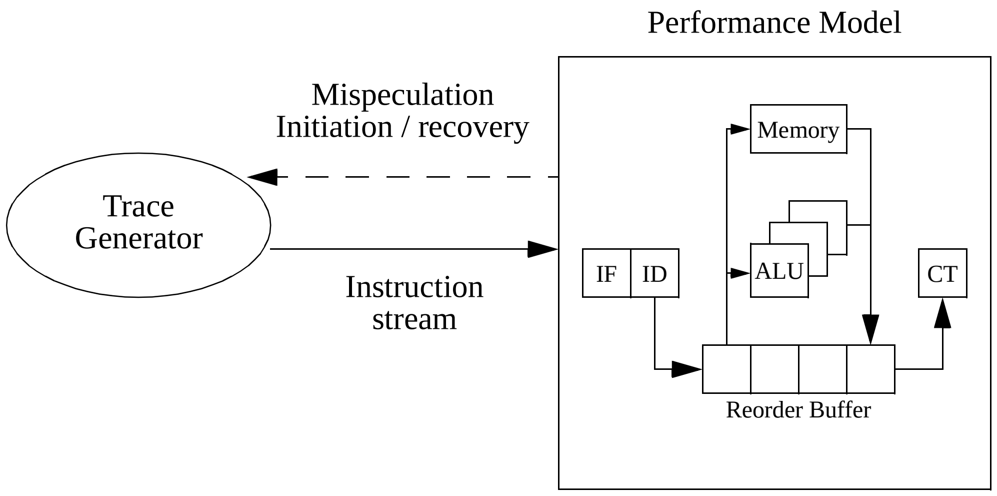


**Figure 1: Organization of a trace-driven simulation environment.** Instructions are supplied by a trace generator to a performance model that represents a detailed microarchitecture.

pleScalar [5], SMTSIM [23], and VMW [7] are some examples of trace-driven simulation infrastructures.

Techniques that use external trace files lack a functional simulation component. Since most trace files only contain the non-speculative instruction stream, performance models based on trace readers typically do not model mispeculation, that is, the instructions that are executed in the shadow of mispredicted branches, addresses, and instruction values. This can create large inaccuracies in the performance models; recent studies have shown that mispeculation streams provide instruction and data prefetching that is beneficial [8] [20]. Moreover, mispeculation reclamation techniques such as instruction reuse [18] and the misprediction recovery cache [4] cannot be modeled using external trace files. One advantage of external traces created from real hardware is that they can include operating system code and interrupt routines. Embedding this information into a trace allows these segments to be included without building a full-system simulator.

Dynamic trace generation provides the additional flexibility to model mispeculation by providing an interface for the performance model to direct the instruction emulator to compute mispeculation (as shown by the dashed line in Figure 1). The primary advantage of this simulation approach is that it provides a clean division of infrastructure for performance modeling and instruction set emulation (or trace processing). However, there are many drawbacks to this infrastructure that make it increasingly difficult to accurately model sophisticated, new microarchitectures.

Instruction streams can become inaccurate when they contain timing dependent computation, that is computation whose inputs (and thus the result) are dependent on *when* the instruction executes. An example of a timing dependent computation is an instruction that is subject to value prediction in the execute stage.[1] We assume that the scheduler of the processor will schedule instructions that have all of their operands first. If there are available functional units after all of the ready instructions are executed, the scheduler will attempt to predict the outcome of an instruction (at the execute stage) based on previous instances of the instruction. This approach would seem to be a good one. Accessing the value prediction table as late as possible would result in higher prediction accuracy and only accessing the table when needed would result

1. This example is based on work done by Calder et. al. [3]. The main difference is that the value prediction table is accessed in the fetch stage in their implementation while it is accessed in the execute stage in our example.We tried to find a real example of timing dependent value prediction from the literature but could not. The lack of any example is likely the result of deficiencies in existing simulation infrastructures.

in less power. The result of the instruction is dependent on when the instruction executes because it may get its value from the value prediction table or from its proper inputs. It is impossible to tell what the precise value is until the instruction is actually executed. Furthermore, if the value prediction is incorrect, the incorrect value is important because it may be an address that subsequently misses in the cache which could further affect the running time of the program. Accurate modeling of the mispeculated path is important since it competes for resources with the non-speculative instruction stream, which may cause additional cache misses or prefetch requests and affect the state of various predictors. It is our belief that the trend in microarchitectures is toward more aggressive speculation and dependence optimizations such as value prediction [11], instruction reuse [18], and speculative value coherence [10].

Another important example of timing-dependent operation are shared memory loads that are subject to race conditions, such as test-and-set instructions. The result of these operations are a function of the interleaving of operations in the shared memory system which is in turn a function of the timing of when the request is made by the processor executing the test-and-set operation.

The alternative to trace-based techniques is to employ a *micro-functional* performance model that not only times the activities of the program but also executes the program at that time, thereby reproducing the execution and timing of the program in a fashion identical to the simulated hardware. This approach can lead to more accurate models, an observation that is well recognized by architects and simulator developers [6]. However, most high-level simulation infrastructures remain trace-based due to two important drawbacks in micro-functional performance models. The first drawback comes from the coupling of timing and correctness. If the micro-functional performance model is incorrect in any way, the simulation can fail. While this exposes errors more readily in the functional model, this is not always the ideal design scenario. Inaccuracies may be the result of ongoing simulation development where details are left out because they were determined to be secondary. For instance, the forwarding of partial store values is a case that is complicated to get completely correct but happens so infrequently the overall impact on performance is negligible. Trace-driven models are more tolerant of infrequent inaccuracies because they don’t fail, and if the inaccuracies are infrequent, the overall simulation remains accurate. The second drawback with micro-functional performance models is that the simulation approach does not lend itself to oracle studies. Instruction inputs, results, and next PC values are not known until the instruction executes. For oracle studies, such as perfect branch or address prediction, not having this information earlier in the pipeline prevents performance bounds studies.

In this paper, we present MASE (Micro Architectural Simulation Environment), a novel performance modeling infrastructure that is built on top of the popular SimpleScalar toolset [5]. We address many of the drawbacks present in SimpleScalar’s trace-based modeling that make it difficult for researchers to accurately model features present in complex high-performance microarchitectures. The goal of MASE is to provide a flexible infrastructure to researchers that they can use to create accurate models of the hardware they are studying. MASE is not intended to model any particular microarchitecture as that would limit the flexibility and potential of the infrastructure. We provide four enhancements to the current implementation of SimpleScalar: (i) an oracle and checker that allows performance bound studies and removes the burden of correctness from the core, (ii) a micro-functional core that increases modeling accuracy, (iii) fine-grain state management facilities that simplify the implementation of control and data speculative optimizations, and (iv) an interface to support resources with non-deterministic latencies.

MASE possesses all the benefits of trace-based modeling, by decoupling model accuracy from simulator correctness and providing extensive support for oracle studies. We implement our new simulation strategy by combining a micro-functional performance simulation infrastructure with a trace-driven oracle execution unit. The oracle execution unit executes instructions at the front end of the simulated processor pipeline, producing instruction information suitable for directing perfect speculation and other oracle studies.

The burden of correctness is lifted from the performance model through the use of a checker execution component at the retirement stage of the performance model. Instruction results are checked as they retire into the architected state of the machine, and if they do not match those computed by the oracle, the performance model is flushed and restarted with correct simulation state. The checker provides a powerful model validation mechanism as well as a backup source of reference semantics that incomplete or inaccurate performance models may rely on to correctly complete any instruction. At the core of MASE is a micro-functional performance model; instructions are not only timed but executed in the core, accurately modeling timing-dependent computation.

We added to SimpleScalar a flexible speculative state management facility that permits restarting from any instruction. The current version of SimpleScalar only allows branch instructions to mispeculate and force a restart. The ability to restart from any instruction allows optimizations such as load address speculation and value prediction to be implemented. In these optimizations, instructions other than branches could be mispeculated, making it necessary to restart at the offending instruction. This allows external interrupts to be implemented since any instruction could be a candidate for a rollback. The checker also uses this mechanism to recover from any errors that are detected since any instruction could potentially cause an error.

In addition, MASE addresses another deficiency in SimpleScalar by providing an interface to model components with non-deterministic latency. Modern DRAM systems can reorder requests to reduce the overall access time, allowing later requests to affect the access time of the current request. For example, DRAM systems typically have a page cache that saves the last page accessed in the memory eliminating the row access time in subsequent requests to the same page. Assume the requests accessed pages in the following order: A, B, A. Since an access to page A was first, page A will reside in the page cache. If the second request to page A is seen before the access to page B is initiated, the memory system can reorder the memory requests so the second access to page A comes before the access to page B. This significantly speeds up the access time for page A since page A was already in the page cache, but it slows down the access to page B since it has to wait for the additional memory access. The memory interface in SimpleScalar requires a latency to be returned immediately after the memory request is sent. This is not sufficient in the scenario described above because it is not known if there will be another memory access to page A before page B gets to access memory, thus the true latency is not known immediately. In MASE, a callback interface is used that allows the memory system (or any resource) to invoke a callback function once the memory system has determined an operation's true latency. The callback interface provides for a more flexible and accurate method for determining the latency of non-deterministic resources. Several simulation infrastructures, such as RSIM [15] and SMTSIM [23], already provide this capability.

The remainder of the paper is organized as follows. The design of MASE and the implementation of the oracle, checker, micro-functional performance model, and our interface for resources with non-deterministic latency are described in Section 2. Section 3 gives results from detailed analyses of our implementation. This was done by comparing the new performance model against the SimpleScalar baseline performance model (sim-outorder), comparing its accuracy and simulation speed. We also present a case study that explores the implementation of a blind dependence speculation technique, a speculation technique that cannot easily be implemented in SimpleScalar. Anecdotal evidence is provided that describes how the checker proved to be very helpful when debugging. Section 5 summarizes and suggests additional work.

## 2. MASE modeling architecture

This section describes the architecture of MASE's performance model. A high-level view of the new architecture is shown in Figure 2. The following sections describe each of our key additions in more detail: the oracle and dynamic checker, the micro-functional performance model, the ability to restart from an arbitrary point, and a callback interface that supports non-deterministic resource latencies.

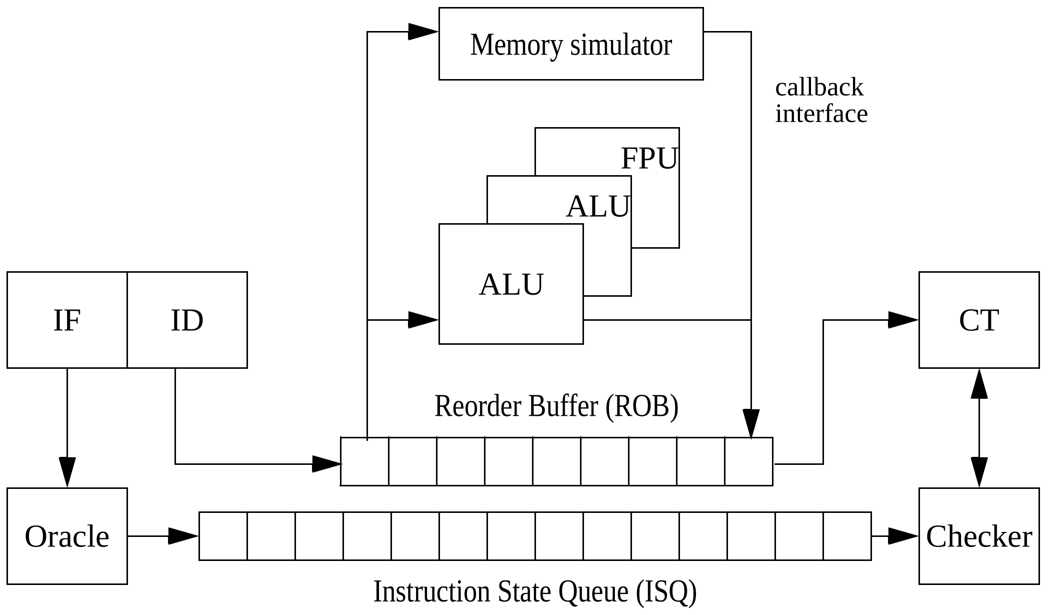
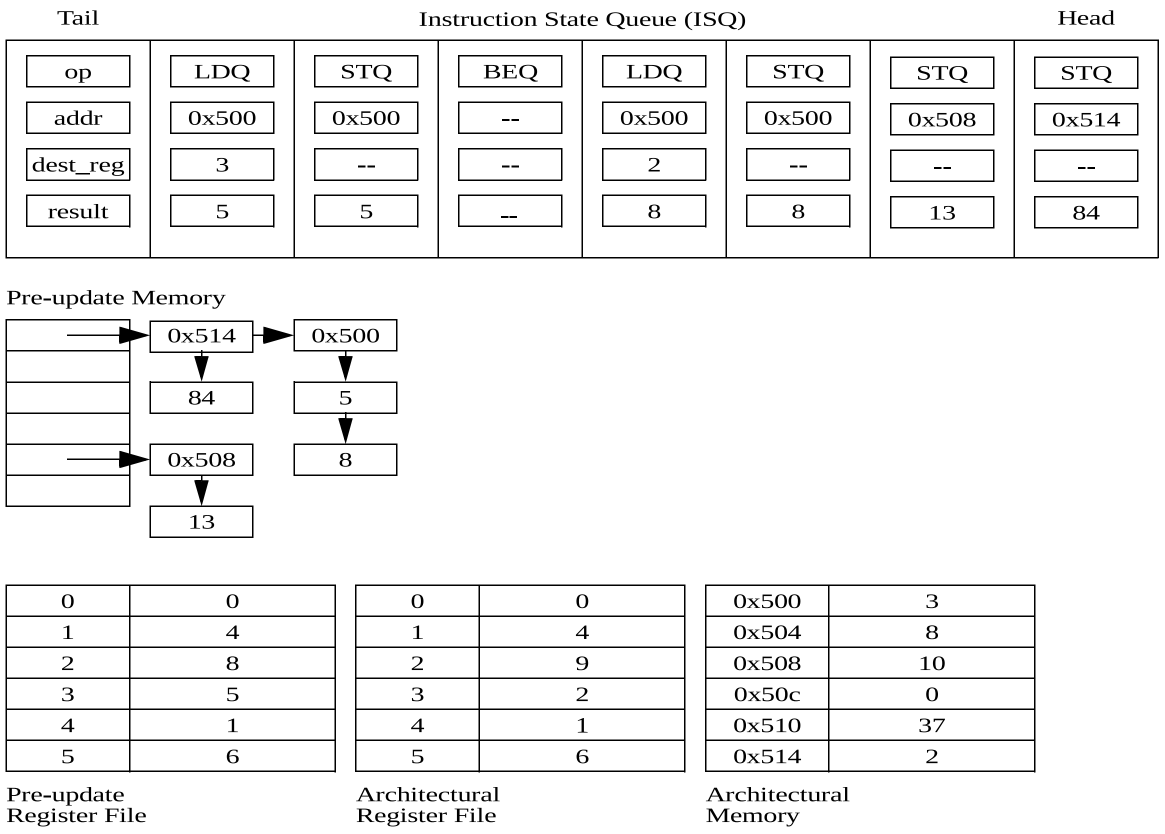


**Figure 2: Block diagram of the new performance architecture model.** The oracle executes instructions in advance and the results are stored in the ISQ so they can be checked by the checker when the instructions commit. The memory interface supports a callback interface to support non-deterministic memory latencies. The functional units execute the instruction instead of just returning a latency.

## 2.1 Oracle and checker execution component

The oracle sits in the fetch stage of the pipeline and executes instructions in program order. The oracle is a functional emulator as it does not model any timing. The oracle contains its own register file (pre-update register file) and memory. To minimize the overhead, the oracle does not have a separate copy of memory but uses a table (pre-update memory table) that contains all new values that have not been committed to architectural memory (all stores currently in the pipeline). Loads from memory are done by initially looking at the pre-update memory table. If there is a hit in the table, the value from the table is used. If there is a miss, architectural memory is accessed to obtain the value. Stores are executed by simply adding an entry to the pre-update memory table. When an instruction is executed, the oracle state is updated and the result of the instruction is stored in the instruction state queue (ISQ) as shown in Figure 2. This queue serves several purposes. It holds data computed by the oracle that is used by the checker to verify that the instruction executed correctly. The data can also be used to facilitate performance bounds studies where correct data is needed. The queue serves as a record of executed instructions in case a recovery is necessary due to a branch misprediction or some other event. The oracle must always be synchronized with the fetch stage of the microarchitectural model, as a result, if instructions are flushed from the microarchitectural model, the oracle must also do this. Since any instructions could potentially cause an error within the checker, the oracle and microarchitecture must be able to rollback from any instruction as described in Section 2.3.

Figure 3 shows an example of how the instruction state queue and pre-update state is updated in the oracle. The head of the queue contains the oldest instruction in the machine (the next instruction to commit) and the tail of the queue points to the next available entry. Pre-update memory is stored as a hash table. For each address in the hash table, there is a linked list of values with the most recent store at the head of the list. In the example, there are two stores to address 0x500. The most recent store to this address wrote the value of 5 so it appears first. It is not sufficient to overwrite the previous value of 8 because the branch instruction between the two stores may mispredict causing a removal of any entries that occur after the



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| op | LDQ | STQ | BEQ | LDQ | STQ | STQ | STQ |
| addr | 0x500 | 0x500 | -- | 0x500 | 0x500 | 0x508 | 0x514 |
| dest_reg | 3 | -- | -- | 2 | -- | -- | -- |
| result | 5 | 5 | -- | 8 | 8 | 13 | 84 |

Pre-update Register File

| | |
|---|---|
| 0 | 0 |
| 1 | 4 |
| 2 | 8 |
| 3 | 5 |
| 4 | 1 |
| 5 | 6 |

Architectural Register File

| | |
|---|---|
| 0 | 0 |
| 1 | 4 |
| 2 | 9 |
| 3 | 2 |
| 4 | 1 |
| 5 | 6 |

Architectural Memory

| | |
|---|---|
| 0x500 | 3 |
| 0x504 | 8 |
| 0x508 | 10 |
| 0x50c | 0 |
| 0x510 | 37 |
| 0x514 | 2 |

**Figure 3: Example of pre-update state and instruction state queue.** The instruction state queue holds the instructions that currently reside in the pipeline and is used to check the results when the instruction is committed. The pre-update memory is stored as a hash table and only contains values for stores that are in the pipeline.

branch (the store of 5 to address 0x500 in this case). Entries in the hash table only exist for stores in the instruction state queue. When the store to address 0x514 is committed, architectural memory is updated and the corresponding entry is removed from the hash table. If a load attempts to access an address not in the hash table (address 0x504 for example), it will initially look in the hash table. After it discovers it is not there, architectural memory is accessed.

The pre-update register file is stored in an array. It contains the latest write to the register. In the example in Figure 3, register 3 has a value of 5. Unlike memory, it contains values for all of the registers so it is only necessary to access the pre-update register file. Mispredictions are also handled differently. If the branch in the example mispredicted, the architectural register file is copied into the pre-update register file and the instruction state queue is scanned, from the head to the mispredicted branch, looking for instructions that write registers. If an instruction writes a register, it will update the register in the pre-update register file with the appropriate value.

Oracles are commonly used to provide “perfect” behavior to do studies that measure the maximum benefit of an optimization. A common case of this is perfect branch prediction where all branch mispredictions are eliminated. In order to provide this capability, the oracle resides in the fetch stage so it knows the correct next PC to fetch. When executing an instruction, the oracle also serves as a decoder in the sense of extracting various information about the instruction such as the type of instruction or branch target. To minimize the running time of the simulation, the oracle saves the decoding information and passes the information along to the dispatch/decode stage.

The checker monitors all instructions that are committed. The results of an instruction are passed along with the instruction until it is committed. The checker will compare these results with the results obtained by the oracle in the front-end of the machine. If the results match, the result will be committed to architectural state and the simulation will progress as normal. If the results do not match, the oracle result will be committed to architectural state and a recovery will be initiated. The remaining instructions in the pipeline are flushed and the front-end is redirected to the next instruction. The instruction with the bad result is allowed to commit in order to ensure forward progress. This policy is used because depending on the nature of the error, the bad instruction may repeatedly get the same error if it is re-executed, causing live-lock in the simulation. Some instructions are non-speculative and have non-deterministic values, i.e. shared memory loads that are subject to race conditions. The results of these instructions can differ with the checker. These instructions can be marked and it their result differs from the checker, the value from the micro-functional core will be regarded as correct and used to synchronize the checker.

The checker may be used in several different ways. The first way is to verify that any changes or enhancements to the simulator code are indeed correct. Since not all errors directly cause an error in the output, it provides extra security that the additions did not violate any microarchitectural dependencies or program semantics. Another advantage is to allow the checker to handle tricky infrequent corner cases to save programming time. For example, it is difficult to program all of the cases involving partial store forwards where the base addresses are different. Instead of adding code to handle this situation, the checker can detect the error and recover from that point. If the event is rare, the effects on the overall performance will be negligible. A third use is to have the checker be part of the microarchitecture itself. Dynamic on-chip verification is being investigated to reduce the burden of correctness in modern microarchitectures [2]. For these cases, the number of errors detected by the checker provides a measure of quality for the microarchitecture and its implementation in software. Fewer errors indicates a more complete microarchitectural core or a more well-behaved simulator. This can be useful when comparing aggressive microarchitectures.

## 2.2 Micro-functional performance model

The current version of SimpleScalar has no infrastructure for micro-functional simulation. Instructions are executed using an oracle in the dispatch stage - input values are read directly from architected state and results are written back right away. Values do not propagate through the pipeline and there is no notion of architectural storage, unlike true microarchitectures. The models of the various microarchitectural features in the current version of SimpleScalar concentrate exclusively on timing and performance aspects, and ‘true’ execution in microarchitecture is not modeled at all.

MASE models execution as it would be in a real microarchitecture. At dispatch, new instructions get reorder buffer entries allocated. Register renaming takes place and input operand values are either obtained from architectural storage or from a ‘completed’ creator’s reorder buffer entry and written into the instruction’s reservation station. If the creator of an operand has not completed execution, the instruction is attached to the dependence chain of the creator, and it gets its operand value when the creator completes execution. When all input operands of the instruction are ready, it is put into the ready queue. The issue stage attempts to issue instructions from the ready queue depending on the availability of functional resources. It is at this point that the actual execution of the instruction takes place. Input values for the instructions come from reservation station entries and the result is written to the reorder buffer entry. The instruction is entered into an event queue from which it will emerge as a completed instruction when the required amount of latency for executing the instruction has passed. During the commit stage, completed instructions are committed in-order and the results of instructions become visible at architectural storage.

The micro-functional performance model allows for forwarding of values from completed stores to waiting loads. The load-store queue keeps track of outstanding loads and stores. When stores complete, the data value is written into the load-store queue entry. For loads, the load-store queue is first searched to see if there is any possibility of a store-forwarded result. If there are store entries in the load-store queue before the load with unknown addresses or unknown data for an address match, the load is blocked. In case of an address match with a previous store whose data is ready, the load value is read off the load-store queue and hence such loads will not access the memory hierarchy. In case the data sizes match, or if the load data size is smaller than the store data size, either the data as is or the appropriately truncated data is transferred to the load. If there is no address match, the load probes the memory hierarchy to retrieve the data.

We have not implemented partial store forwarding, the case where a store produces only part of the value used by a later load. There is indeed a possibility that a load ends up with a stale value because we do not have complicated overlapping address checks in the LSQ. Since such situations occur rarely in practice, we decided that should such an event arise, we will address the deficiency by letting the checker capture and correct the error instead of implementing the complicated but rarely used circuitry needed for partial store forwards. We confirmed this result for the SPEC benchmarks in Section 3.3.

## 2.3 Recovery from any instruction

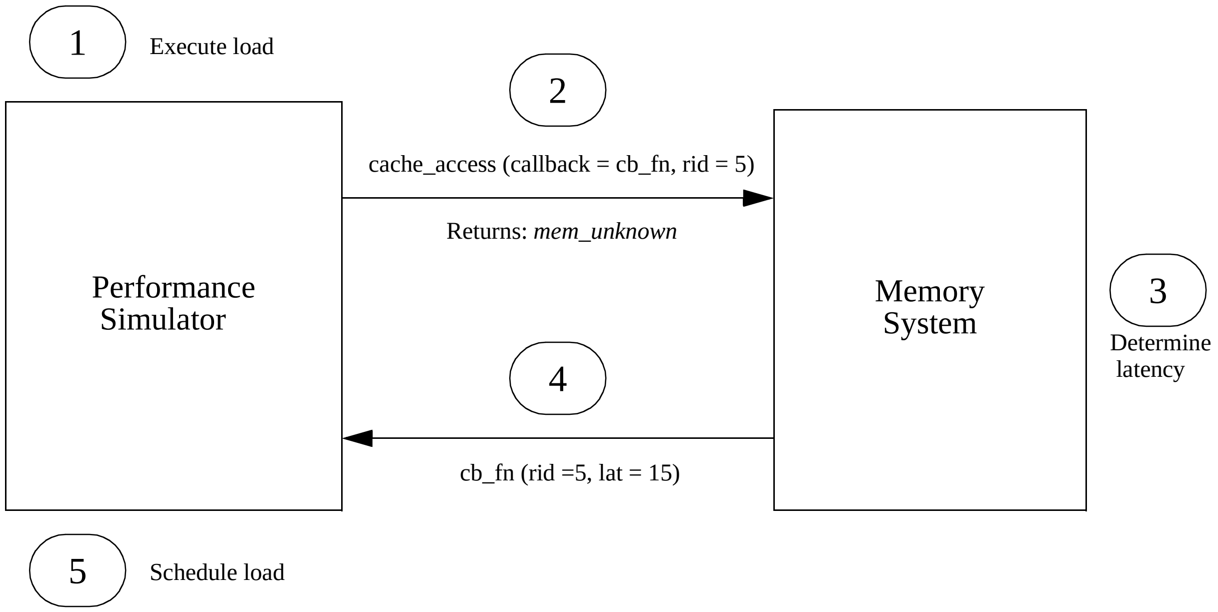
In the current version of SimpleScalar, branches are the only instructions that can cause mispeculation. A branch misprediction causes a speculation mode bit to be set. When this bit is set, certain data structures are not updated, eliminating the need to provide a rollback mechanism. This implementation has two major deficiencies: (i) it prevents speculative optimizations, such as blind load speculation, where it is impossible to determine if a recovery will occur until the later stages of the pipeline and (ii) it increases the inaccuracies in the mispeculation modeling since data structures are not updated during a misprediction.

MASE addresses these deficiencies by adding rollback mechanisms for several data structures throughout the machine. This includes updates to the reorder buffer, rename table, branch predictor. The oracle and checker also need to be synchronized so the instruction state queue, pre-update memory, and pre-update register file also have rollback mechanisms. All of these structures will continue to be updated even if the instructions are executed speculatively, increasing the accuracy of mispeculated instructions. When a misprediction occurs, a recovery occurs by restarting the fetch engine with the proper PC and rolling back the data structures to the appropriate point.

The rollback mechanism is illustrated by an example describing our implementation of the rename table. The rename table is used to break false register dependencies by keeping track of the latest creator (instruction that last created/wrote a value) for each logical register. In our implementation, each entry includes a linked list of all instructions that currently reside in the pipeline and create a value for the logical register. The head of the list points to the most recent creator in the pipeline. When an instruction that created a register value is committed, the corresponding entry is removed from the rename table. During a recovery event, the linked lists are scanned. Entries that refer to instructions that were squashed during the recovery are removed. Since the linked lists are ordered by position in the pipeline, once an entry is found that corresponds to an instruction that survived the recovery, the search can be terminated for that logical register.

## 2.4 Callback interface for non-deterministic latencies

MASE provides a callback interface for resources that have non-deterministic latency. The callback mechanism was used to update the memory interface in SimpleScalar. Currently, any call to the memory interface returns a latency value immediately for that particular operation. This is not strictly an accurate reflection of modern day DRAM systems, where the actual latency for a particular access might not be



Step 1: Load is executed.
Step 2: Call to memory system with unique request id (rid) 5. Unknown latency, so mem_unknown status is returned.
Step 3: Memory system determines latency.
Step 4: Callback function is invoked, latency value written into corresponding table entry.

**Figure 4: Processing a load with non-deterministic latency.** This diagram shows the steps of the execution of a load that requires the callback mechanism be used.

immediately known. The memory operation may have to be queued and subsequently reordered with later references. The actual latency is only known when the instruction accesses to the memory component.

In MASE, calls to the memory interface return two values - a status and the latency. The status can be of three types - *mem_known*, *mem_unknown* or *mem_invalid*. If the status returned is *mem_known*, then the memory system was accurately able to determine the latency of the operation (most likely due to a L1/L2 hit) and returns that latency immediately. If the status returned is *mem_unknown*, the memory system is indicating that it will handle the request but the latency cannot be determined at this point of time. One of the parameters passed to the memory call is a callback function, which will be invoked by the memory system when the latency has been determined. There are different types of callback functions for different types of memory accesses. For instruction cache misses, the fetch stage is blocked, and the callback function unblocks the fetch stage and sets the latency for the operation. For store data cache misses, the latency is ignored because the store data is either in the cache or in the writeback buffers; either can be read if required. For load data cache misses, there is a table that keeps track of callback memory requests. Each entry in the table contains a unique index and a reservation station entry. When the callback function is invoked, it scans the table for a matching index. If it finds the entry and it is still valid (the entry could be invalidated if it was squashed), it schedules a writeback event for the load with the given latency. Finally, if the status returned is *mem_invalid*, it means that the memory system cannot handle this request right now for some reason (for example, a full buffer). In instruction cache accesses, the fetch stage is blocked and we keep retrying until we get a different status. We do the same for store data cache accesses - block the commit stage and keep retrying. Loads that access the data cache are not scheduled to execute. The simulator will attempt to schedule the load each cycle until it gets a status of *mem_known* or *mem_unknown*.

## 3. Early analyses

This section describes some analyses we have done with MASE. We compare our infrastructure to the current version of SimpleScalar. The goals of the experiments is to compare the performance and running time of the two simulators. In addition, we analyze the benefits of the dynamic checker and look at blind load speculation, a study that would be difficult to implement with the current release of SimpleScalar.

### 3.1 Simulation methodology

We used the SPEC95 integer benchmarks, compiled using the Compaq C (version 5.9) and Fortran (version 5.3) compilers under using full compiler optimization (-O4). The train input set was used for all experiments. The benchmarks were simulated to completion or for a maximum of 250 million instructions.

Our infrastructure was built on top of the SimpleScalar/Alpha 3.0 tool set, a suite of functional and timing simulation tools for the Alpha ISA. This version of SimpleScalar was also used as our baseline. It contains a timing simulator that executes only user-level instructions, performing a detailed timing simulation of an aggressive 4-way dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is trace-driven using a functional component that includes execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction.

The simulation configuration models a modern out-of-order processor microarchitecture. The processor has a large window of execution; it can fetch and issue up to 4 instructions per cycle. It has a 128 entry reorder buffer with a 64 entry load/store buffer. Loads can only execute when all prior store addresses are known (except for the blind load speculation experiment). In addition, all stores are issued in program order with respect to prior stores. A 4k entry gshare branch predictor was used and there is a six cycle minimum branch misprediction penalty. The processor has 4 integer ALU units, 2 load/store units, 2 FP adders, 1 integer MULT/DIV unit, and 1 FP MULT/DIV unit. The latencies are: integer ALU 1 cycle, integer MULT 3 cycles, integer DIV 20 cycles, FP adder 2 cycles, FP MULT 4 cycles, and FP DIV 12 cycles. All functional units, except the divide units, are fully pipelined allowing a new instruction to initiate execution each cycle.

The processor we simulated has 64k 2-way set-associative instruction and data caches. Both caches have block sizes of 32 bytes. The data cache is write-back, write-allocate, and is non-blocking with two ports. The data cache access latency is two cycles (for a total load latency of three cycles). There is a unified second-level 512k 4-way set-associative cache with 32 byte blocks, with a 12 cycle cache hit latency. If there is a second-level cache miss it takes a total of 80 cycles to make the round trip access to main memory. There is a 16 entry 4-way associative instruction TLB and a 32 entry 4-way associative data TLB, each with a 30 cycle miss penalty.

### 3.2 Comparison to original model

This experiment compares the performance of each benchmark to the baseline infrastructure. The purpose of this experiment was to make sure the performance of the two simulators are comparable. This strategy tests the new recovery mechanism, micro-functional core, and the dynamic checker. The results of the experiment, shown in Table 1, confirm that the performance is comparable with negligible differences.

To test the performance of the memory interface requires the use of a memory system that takes advantage of the new callback mechanism. We adopted the existing SimpleScalar memory system, randomly selecting accesses to use the callback mechanism.

**Table 1: Performance validation statistics.** This table compares the resulting performance of our new performance model to the baseline model (current version of SimpleScalar). The performance results are given in instructions per cycle (IPC). As expected, the differences between the two are minimal.

| Benchmark | MASE | Baseline | Difference | Benchmark | MASE | Baseline | Difference |
|---|---|---|---|---|---|---|---|
| cc1 | 1.6537 | 1.6538 | 0.01% | li | 2.0481 | 2.0485 | 0.02% |
| compress | 2.3527 | 2.3510 | 0.07% | m88ksim | 2.1634 | 2.1644 | 0.05% |
| go | 1.3418 | 1.3418 | 0.00% | perl | 1.9862 | 1.9869 | 0.04% |
| ijpeg | 2.6659 | 2.6660 | 0.00% | vortex | 2.1099 | 2.0928 | 0.82% |

Our next experiment was to see the effect of the added functionality on the running time of our simulator. We compared the running time of our simulator to the baseline and the results are shown in Table 2. The time is listed in seconds and our simulator is twice as slow on average. This is not surprising given that out simulator is more detailed and accurate. The code has not been profiled yet to see where the bottlenecks are, so we hope to improve upon this before the code is released. The size of the source code increased by about 50%.

**Table 2: Running time of the simulator.** Compares the time it takes to run the simulator for each benchmark. Our simulator runs twice as long on average. Time is listed in seconds.

| Benchmark | MASE | Baseline | Benchmark | MASE | Baseline |
|---|---|---|---|---|---|
| cc1 | 5294 | 2964 | li | 10207 | 6796 |
| compress | 1093 | 606 | m88ksim | 6601 | 3724 |
| go | 6595 | 3665 | perl | 2238 | 1237 |
| ijpeg | 8427 | 2385 | vortex | 5326 | 2887 |

## 3.3 Use of the dynamic checker

One benefit of the dynamic checker is to reduce the burden of correctness on the performance model. When implementing an optimization, most of the time is spent on corner cases that happen very rarely. The programmer can simply not implement infrequent corner cases and rely on the dynamic checker to correct any instructions that have the incorrect value. In our implementation, we decided not to implement partial store forwarding from the load-store queue when the base addresses do not match. For instance, if there is a two-byte store to address 0x102h that is followed by a four-byte load from address 0x100h, it would not be detected by the load forwarding mechanism. Instead, we rely on the checker to fix this problem. Table 3 shows the number of checker errors that were obtained. In four of the eight benchmarks, this scenario was not encountered so there were no checker errors. In the other four benchmarks, the number of errors was very small. The benchmark *vortex* was highest with 129 errors, a very small number considering that millions of instructions were executed.

**Table 3: Checker errors.** Number of checker errors due to not handling partial store forwarding when base addresses do not match.

| Benchmark | cc1 | compress | go | ijpeg | li | m88ksim | perl | vortex |
|---|---|---|---|---|---|---|---|---|
| Errors: | 128 | 0 | 0 | 33 | 0 | 15 | 0 | 129 |

An additional benefit of the checker is that it serves as a debugging aid. This is hard to quantify in terms of numbers so we describe our debugging experience instead. The oracle and checker were among the first additions to our infrastructure so it could be used when implementing some of our other ideas. In most simulators, it is difficult to determine precisely where an error occurred when there is a difference in the output. Furthermore, some modeling errors don't surface as errors in the output. When the checker is used, each instruction that produces a result (writes to a register or memory) is checked. If an error is encountered, the number of errors is incremented and optionally the error can be printed out. When the errors are printed out, it will indicate what instruction has failed, allowing a programmer to precisely go to the point where the error occurred. It prints out the PC along with the instruction itself. The checker came in handy when implementing the micro-functional component. The first thing we realized during debugging is that most of the failing instructions referred to Alpha register $31 (the zero register). Almost immediately, we were able to determine that the processing of this special register was incorrect and it was fixed without running the simulator a second time. Once that problem was flushed out, we noticed that most of the problems dealt with conditional move instructions and how the output was incorrectly zero most of the time. We concentrated our debugging efforts at the conditional move and quickly identified that the case where the move was not executed was not handled properly. The checker was also handy when implementing our case study, blind speculation (described in the next section). As one might expect, loads were the only instruction that failed so the error message provided by the checker did not provide very much insight like in the previous cases. Instead, we focused on the first error that was signalled. We used gdb to debug the simulator and set a breakpoint on the failing instruction. Once we got to the failing instruction, we analyzed the state of the machine at the time and were able to isolate the problem relatively quickly.

## 3.4 Case study: Blind speculation

This section describes a case study we performed with our new performance simulator. We implemented blind speculation, a technique that allows loads to execute before all previous stores have executed [13]. Without blind speculation, loads are required to wait in the load-store queue until all addresses for previous stores have been resolved. In blind speculation, the scheduler assumes that any unknown store addresses will not match the address of the load, as such a load can be executed as soon as its address is known and there isn't an earlier matching store address in the load-store queue. The benefit of blind speculation is that it allows loads to execute earlier, but with a potential mispeculation penalty that is incurred when an earlier unknown store matches a speculated load. In this situation, instructions after the load are flushed and the fetch engine is directed to restart at the instruction after the mispeculated load.

Blind speculation is difficult to implement in the current version of SimpleScalar because SimpleScalar requires that mispeculations be detected at fetch. The contents of the load-store queue when a load is ready will determine if a blind speculation recovery is necessary. Branch mispredictions, the only recovery event currently in SimpleScalar, are easier to handle since it is possible to tell whether or not a branch will mispredict by using an oracle in the front-end. Consequently, updates to many data structures are conditionally based on whether or not the instruction is going to be squashed. In blind speculation, this is not possible since many data structures need to be updated before the simulator knows if the instruction will cause a recovery or not. This problem is addressed by adding a rollback mechanism for all of the data structures that need to keep track of the state of the instructions that reside in the pipeline.

Adding blind speculation to our infrastructure was relatively straightforward. Each cycle, the load-store queue is scanned to see if any loads can be scheduled to execute. With blind speculation, a load can be scheduled if both of the following conditions are true:

- The address is known.
- If there are store instructions that are known to write to the same address as the load, the store data must be known for the most recent of these store instructions.

Once a load is scheduled, all unknown store instructions in the load-store queue that are more recent than the last known store to the load address are scanned to see if they match the load address. The store address is obtained from the instruction state queue which holds oracle state. If an unknown store matches the load address, a recovery event is placed on the store which is invoked on writeback. The recovery is unusual in the sense that the instruction initiating the recovery is not the same as the recovery point. To get around this problem, the store's reorder buffer entry keeps track of the mispeculated load so the recovery routines know where to start squashing instructions. If a store causes multiple loads to be mispeculated, only the oldest load is saved since the other loads will get squashed when recovering from the first blind speculation miss. Another problem that can (and did) occur was that by the time a store signalled a blind speculation recovery, the load may have already been squashed due to a branch misprediction or a different blind mispeculation. The load instruction must be checked to make sure it is valid before a recovery is initiated.

After implementing blind speculation, we performed some small performance studies. The configuration was the same as the other experiments in this section. The baseline is our new simulator with blind speculation turned off. The simulation includes different load-store queue sizes and the reorder buffer size is twice the size of the load-store queue for each run. The results are shown in Figure 5 for the benchmark

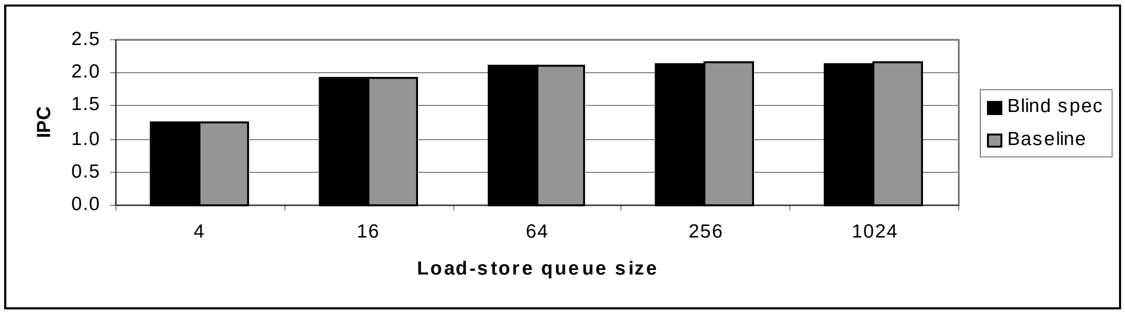


**Figure 5: Blind speculation results for the benchmark *vortex*.** As the load-store queue size increases, blind speculation becomes more of a penalty than a benefit. This is expected since the probability of a miss increases with the larger queue.

*vortex*. As you can see from the chart, the overall effectiveness of blind speculation is not that significant. It it most helpful when the load-store queue is moderately small and it actually hurts performance when the size of the load-store queue is increased. This result is intuitive since the probability of a miss in a smaller queue is smaller than a miss in a larger queue. This is confirmed by looking at the number of misses due to blind speculations, shown in Table 4. The number of misses increases greatly as the load-store queue size

**Table 4: Number of blind speculation misses for the benchmark *vortex*.** The number of misses increases as the load-store queue size increases.

| Load-store queue size: | 4 | 16 | 64 | 256 | 1024 |
|---|---|---|---|---|---|
| Misses: | 0 | 5159 | 18,058 | 36,842 | 39,398 |

is increased. The differences in misses between queue sizes of 256 and 1024 is not as significant as the other intervals. This is due to a saturating effect where the additional queue size does not provide any more benefit because the extra instructions in the queue are dependent on instructions that have not had a chance to execute. This can also be explained by looking at the lack of difference between the two data points in the baseline performance in Figure 5. The other Spec95 integer benchmarks were also simulated and obtained similar results that are not shown in order to conserve space. The only difference is that some benchmarks reached their queue limit with smaller queue sizes.

## 4. Related Work

There are a number of performance modeling infrastructures available to researchers today that implement various forms of these technologies. The Pentium Pro simulator [16], Dinero [9], and Cheetah [21] are examples of simulators that read external traces of instructions. Turandot [14], SMTSIM [23] and VMW [7], are simulators, like SimpleScalar, that generate instructions traces through the use of emulation. RSIM [15] is an example of a micro-functional simulator; instructions are emulated in the execution stage of the performance model. Unlike MASE, it does not have a trace-driven component in the front-end. This prevents oracle studies such as perfect branch prediction. SimOS [17] and SimICS [12] focus on system-level instruction-set simulation rather than detailed microarchitecture modeling. Similarly, MINT [25] and ATOM [19] are also simulation infrastructures, concentrating on fast instruction execution.

## 5. Summary and future work

At the heart of any detailed simulation infrastructure lies the issue of building an accurate performance model. There are two primary components in current modeling infrastructures: a functional simulator and a performance model which does timing analyses of instructions and data as they flow through the processor pipeline. While this division has its advantages, there are drawbacks to this approach. The solution to most of these problems is to integrate functional execution with performance modeling, such that programs are executed in the performance model. This, however, has two disadvantages. Firstly, inaccuracies in the performance model, even if they are infrequent, will cause programs to fail. Secondly, a micro-functional performance model does not lend itself to oracle-type studies.

We proposed MASE, a new micro-functional performance infrastructure that addresses problem with the trace-driven approach found in SimpleScalar while eliminating the disadvantages of micro-functional simulation. Our simulation infrastructure has an oracle at the front end that feeds off the fetch stage and executes instructions using its own state. This component provides a setting for all types of oracle studies like perfect branch and value prediction. MASE contains a detailed micro-functional performance model that not only does timing analyses on instructions and data, but also executes instructions similarly to real microprocessors. The performance model however can afford to be inaccurate for infrequent cases (for e.g., partial store forwards). This is because we have a checker that provides a validation infrastructure for our simulator. The checker will detect and correct infrequent corner cases by comparing the results of instructions from the performance model and the oracle. The checker also serves as a valuable debugging aid. MASE also supports arbitrary rollback to any point in the instruction window, which enhances our ability to handle any kind of misprediction as opposed to just branch mispredictions (as implemented in the current version of SimpleScalar). Lastly, our memory interface is more realistic than SimpleScalar's as it includes non-deterministic memory latency modeling for memory accesses that cannot immediately determine their latency.

We compared our new performance model to the current version of SimpleScalar. The performance of programs were comparable on both versions. Running time of our simulator was approximately twice as slow as compared to SimpleScalar because of the increased accuracy and micro-functional execution in our model. As a case study, we modeled and studied blind load speculation using MASE, something which would have been very difficult to do with the baseline SimpleScalar performance model.

In the future, we want to incorporate microcode enhancements in our simulator. Instructions can be divided into a sequence of micro-ops and the functionality of the micro-ops can be defined in the simulator code. Instructions would be executed by executing the required sequence of micro-ops. We also want to optimize the code so that our simulator is able to run faster and include a more accurate baseline models such as a better memory system.

## 6. References

[1] A. Agarwal, R. Sites, and M. Horowitz. ATUM: A new technique for capturing address traces using microcode. *Proceedings of the 13th Annual International Symposium on Computer Architecture*, June 1986, pp. 119-127.

[2] T. Austin. DIVA: A Dynamic Approach to Microprocessor Verification. *The Journal of Instruction-Level Parallelism Volume 2*, 2000.

[3] B. Calder, G. Reinman, D. Tullsen. Selective Value Prediction. *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[4] J. Bondi, A. Nanda, and S. Dutta. Integrating a misprediction recovery cache (MRC) into a superscalar pipeline. *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996.

[5] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. *University of Wisconsin Computer Sciences Technical Report #1342*, June 1997.

[6] R. Desikan, D. Burger, and S. Keckler. Measuring Experimental Error in Microprocessor Simulation. *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[7] T. Diep. VMW: A Visualization-based Microarchitecture Workbench. *Ph.D. thesis, Carnegie Mellon University,* June 1995.

[8] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss. *Proceedings of the 1997 International Conference on Supercomputing*, July 1997, pp. 68-75.

[9] J. Edler and M. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. *http://www.neci.nj.nec.com/homepages/edler/d4*.

[10] A. Lai and B. Falsafi. Memory sharing predictor: The key to a speculative coherent DSM. *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999, pp. 172-183.

[11] M. Lipasti, C. Wilkerson, J. Shen. Value locality and load value prediction. *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[12] P. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. SimICS/sun4m: A Virtual Workstation. *Usenix Annual Technical Conference*, June 1998.

[13] A. Moshovos and G. Sohi. Memory Dependence Speculation Tradeoffs in Centralized, Continuous-Window Superscalar Processors. *The 6th Annual International Symposium on High Performance Computer Architecture*, January 2000.

[14] M. Moudgill, J. Wellman, J. Moreno. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, May/June 1999, pp. 15-25.

[15] V. Pai, P. Ranganathan, and S. Adve. RSIM Reference Manual. Version 1.0. *Technical Report 9705, Department of Electrical and Computer Engineering, Rice University,* July 1997.

[16] D. Papworth. Tuning the Pentium Pro Microarchitecture. *IEEE Micro*, April 1996, pp. 8-16.

[17] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the SIMOS approach. *IEEE Parallel & Distributed Technology: Systems & Applications*, Winter 1995, pp. 34-43.

[18] A. Sodani and G. Sohi. Dynamic Instruction Reuse. *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, June 1997, pp. 194-205.

[19] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *Proceedings of the 1994 ACM Symposium on Programming Language Design and Implementation*, June 1994, pp. 196-205.

[20] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 2000.

[21] R. Sugumar and S. Abraham. cheetah - Single-pass simulator for direct-mapped, set-associative and fully asso-

ciative caches. *Unix Manual Page*, 1993.

[22] TraceBase. *Parallel Architecture Research Laboratory, New Mexico State University. http://tracebase.nmsu.edu/tracebase.html.*

[23] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[24] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. SoftSDV: A Pre-silicon Software Development for the IA-64 Architecture. *Intel Technology Journal*, 4th quarter 1999.

[25] J. Veenstra and R. Fowler. MINT: a front end for efficient simulation of shared-memory multiprocessors. *Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)*, January 1994, pp. 201-207.