Accurate Modeling of Aggressive Speculation in Modern Microprocessor Architectures

Harit Modi, Lawrence Spracklen, Yuan Chou and Santosh G. Abraham
Advanced Processor Architecture, Scalable Systems Group
Sun Microsystems, Sunnyvale, CA
{*harit.modi, lawrence.spracklen, yuan.chou, santosh.abraham*}*@sun.com*

Abstract

Computer architects utilize cycle simulators to evaluate microprocessor chip design tradeoffs and estimate performance metrics. Traditionally, cycle simulators are either trace-driven or execution-driven. In this paper, we describe ValueSim, a software layer that is interposed between a cycle simulator and either a functional simulator or a value-enhanced trace. By writing to the ValueSim API, the cycle simulator can run in either trace-driven mode or execution-driven mode, allowing it to exploit the advantages of both approaches.

The ValueSim API allows a cycle simulator to accurately model a complete range of aggressive speculative mechanisms developed by computer architects, even in the trace-driven mode. Using ValueSim, we illustrate, for three key commercial applications, the significant underestimation of off-chip bandwidth, queuing delays and cache pollution when modern speculative mechanisms are not accurately modeled, highlighting the importance of accurately modeling these mechanisms in chip multiprocessor designs.

1 Introduction

High-performance microprocessor designs represent a large investment, involving several hundred engineers over a 3-5 year period. Cycle-accurate (or cycle) simulators are extensively employed in the early phases of the design to fully evaluate design tradeoffs and enable accurate performance projections. Inaccurate cycle simulators in the early phases may necessitate expensive and time-consuming rework of the design in the later phases of development. A cycle simulator models all the major components of the microprocessor chip and reports extensive performance metrics for a range of design parameters on the relevant workloads. Cycle simulators are refined and validated over multiple design cycles spanning a number of years.

Computer architects rely on a variety of speculation mechanisms in order to effectively utilize the deep superscalar pipelines prevalent in modern microprocessors and to hide the increasing relative memory latencies, caused by the widening gap between processor and memory speeds. There are two broad classes of speculation mechanisms: control speculation and data speculation. Control speculation refers to the execution of an instruction before it is known that it is present in the dynamic instruction stream. For instance, branch predictors predict the direction and target of branches. This enables execution of the instructions from the predicted target before the outcome of the branch is known. Data speculation refers to the execution of an instruction before its inputs are known to be correct. For instance, a load that follows a store in program order may be speculatively executed prior to the store, under the usually correct assumption that the load and store target distinct memory locations (i.e. they do not alias).

In order to be accurate, cycle simulators need to model both classes of speculative mechanisms because misspeculation causes side effects that can impact various performance metrics. In the case of control speculation, when a mispredicted branch is resolved, instructions from this "wrong path" are flushed and execution is resumed at the correct target. However, the act of fetching and executing these wrong path instructions may affect the state of the instruction and data caches and, as a result, will impact various performance metrics, such as miss rates and CPI (cycles per instruction).

An important area of current microarchitecture research focuses on the development of speculation techniques, such as hardware scouting [10], run-ahead execution [12, 23], and value prediction [19], that target the large and increasing cache miss component of overall CPI in common commercial applications (typified by database-, application-, and web-servers). For the different pipeline stages (fetch (FE), rename (RN), execute (EX) and memory access (ME)), Figure 1 illustrates the increase in instructions processed when wrong path (WP) and hardware scouting (HWS) are modeled. For example, if the processor supports control speculation and hardware scouting, up to twice as many instructions may be fetched and up to 1.8X additional memory accesses performed. Clearly, as processors move towards ever more aggressive speculation and as Chip Multithreaded Processors (CMT) introduce interaction of these effects between hardware strands [26], these effects need to be modeled in greater detail.

1.1 Trace-Driven vs. Execution-Driven

Today's cycle simulators are either trace-driven or execution-driven, and both approaches have associated benefits and disadvantages.

In trace driven simulation, simulation is divided into two stages: trace generation and trace simulation. Trace generation

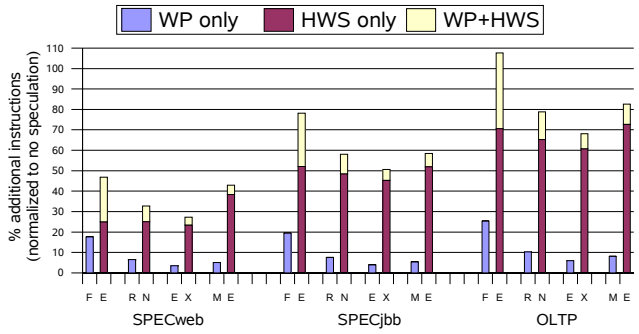

Figure 1. Increase in activity resulting from speculative techniques

is the process of generating (and validating) a trace that captures the behavior of the target workload. Traces can be collected via a variety of different mechanisms [27], but generally involve the use of functional simulators [20] to produce a trace consisting of the dynamic instruction stream and associated memory references of the workload. The trace is validated by comparing various metrics with those observed on an existing system. Once a trace is validated, it can be used as an input to the trace-driven cycle simulator and used to evaluate a range of design choices.

The major advantage of the trace-driven approach is that once a trace is collected, it can be used for many simulations with different parameters. This is important when the workload is very complex and expensive to set-up. For example, setting up a TPC-C benchmark run requires hundreds of gigabytes of memory, terabytes of disk space and hundreds of disks. Another advantage of the trace-driven approach is that trace-driven simulations are deterministic and easily repeatable [18]. A major drawback of the trace-driven approach is that a trace typically contains only non-speculative instructions, making it difficult to model control speculation. Furthermore, most traces do not contain values, making it difficult to model techniques such as value prediction.

Unlike trace-driven cycle simulators, execution-driven cycle simulators employ a close coupling of the functional and cycle simulators. This enables the simulation of many speculation mechanisms. Control speculation due to branch prediction can be accurately modeled, because the functional simulator maintains the architectural state. At each predicted branch, the cycle simulator requests that the functional simulator checkpoint the architectural state. The functional simulator provides the cycle simulator with the addresses generated along a potential wrong path. If the prediction is wrong, the cycle simulator directs the functional simulator to resume execution at the checkpoint. Otherwise, the checkpoint is discarded. Another advantage of execution-driven simulation is that it can accurately model multiprocessor effects such as the ordering of lock acquisitions and releases. A major disadvantage of the execution-driven approach is that it is often not practical for full-size commercial workloads, where the memory and secondary storage requirements are significant and increasing rapidly. Secondary storage requirements are already in terms of terabytes and petabyte data-sets will not be uncommon in the near future [14]. As a result, this frequently leads to many workloads being scaled down until they represent a manageable simulation

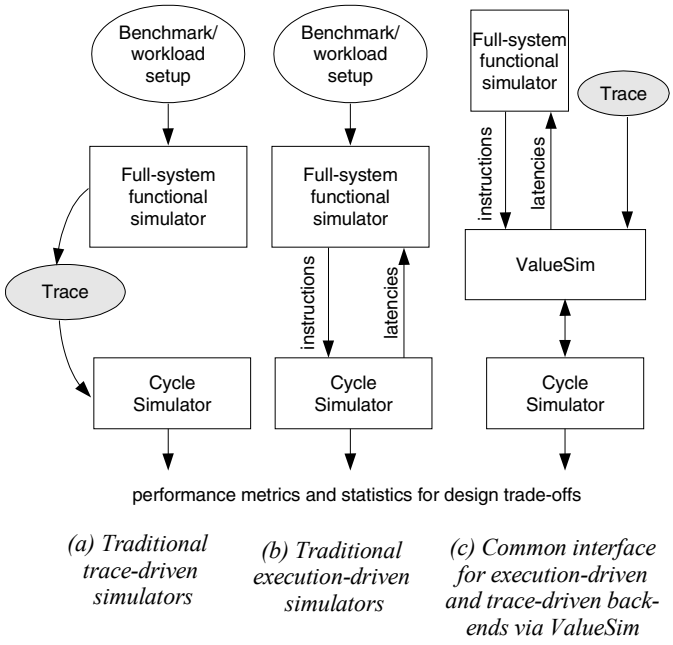

Figure 2. ValueSim: Unifying trace-driven and execution-driven simulators

environment [1]. This scaling is known to introduce significant inaccuracies [2].

As a result, there are situations where trace-driven simulation is preferred and other situations where execution-driven simulation is preferable. The ValueSim approach allows users to select from either depending on the situation.

1.2 The ValueSim Approach

ValueSim is a software layer that is interposed between a cycle simulator and either a functional simulator or a value-enhanced trace. Since the cycle simulator only makes calls to ValueSim's API (Application Programming Interface), the source of the simulation input is abstracted from the cycle simulator, allowing it to be run in either trace-driven mode or execution-driven mode without requiring any changes. Most importantly, ValueSim allows the cycle simulator to both easily and accurately model all forms of control and data speculation, even when the input is from the value-enhanced trace. In essence, ValueSim allows a cycle simulator to obtain the advantages of both trace-driven and execution-driven simulation, making it an ideal tool for writing cycle simulators that need to accurately model aggressive speculation.

In this paper, we describe ValueSim in detail, explaining its overall framework, API, and internal components. We also describe how ValueSim can be used to model a wide variety of speculation mechanisms. Finally, we present experimental data to illustrate that ValueSim can be successfully used to model both control and data speculation, such as hardware scouting and missing load value prediction, even when the simulation inputs are from a value-enhanced trace.

2 ValueSim

The primary design objectives behind ValueSim were as follows. First, the performance analysis infrastructure should enable the modeling of a range of existing speculation mechanisms, such as branch prediction and load-store disambiguation. Furthermore, the infrastructure should be based on a sufficiently general model of speculation to elegantly enable the modeling of novel speculative optimizations. Second, the existing performance analysis infrastructure should be reused and require minimal changes. Cycle/functional simulators, as well as the tracing infrastructure, represent a large software investment that acquire credibility over time as they are extensively validated over multiple design cycles. Accordingly, ValueSim was designed to work with existing cycle and functional simulators, while incorporating a general speculation model enabling the modeling of a variety of speculation mechanisms.

2.1 Common Interface

As shown in Figure 2(a), traditional trace-driven simulation decouples the functional or architectural simulation of the computer system from the cycle or timing simulation using a trace interface. In execution-driven simulation, as illustrated in Figure 2(b), the timing simulator feeds back the latencies of instructions to the functional simulator. This feedback may in turn affect the ordering of multiprocessor or I/O events leading to a potentially higher fidelity modeling of the system. Both approaches have their merits depending on the aspect of the system being optimized. ValueSim supports both options using a common interface: as illustrated in Figure 2(c), ValueSim is a software layer that is placed between the cycle simulator and the functional simulator or the trace and mediates the cycle simulator's access to the functional simulator or the trace.

2.2 Breaking Instructions Into Components

The API between ValueSim and the cycle simulator breaks up instruction execution into its natural components: fetch, rename, execute, retire and commit. This permits the flexible reordering of these components and the modeling of techniques that affect speculative values. Before describing the API in detail in Section 2.4, we first motivate the breakup of instruction execution into these components.

First, instruction fetch and instruction execution need to be separated because not all of the instructions that are fetched are executed. For example, some wrong-path instructions that are fetched as a result of a branch misprediction may not have reached the execute stage when the mispredicted branch is resolved.

Second, instruction fetch and instruction rename need to be separated because a single, complex fetched instruction may be broken into multiple, simpler "helper" micro-instructions, each of which must be renamed separately.

Third, instruction execute and instruction retire need to be separated because once renamed, while instructions may be executed in any order and multiple times, they have to be retired in order. For store instructions, instruction retire should be separated

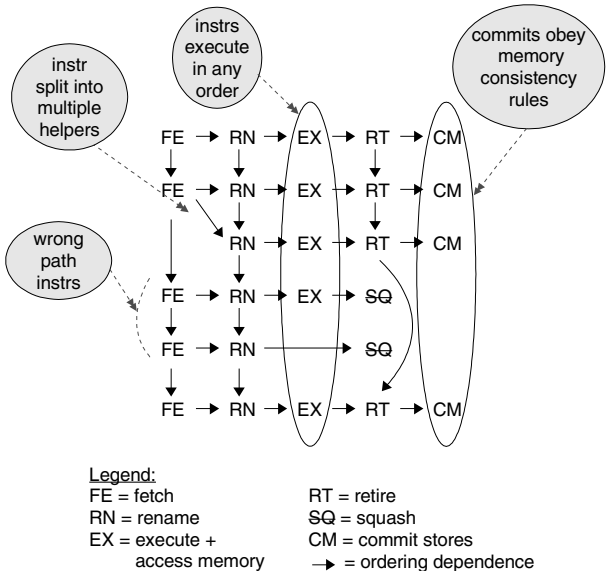

Figure 3. ValueSim splits instruction execution into its subcomponents

into two components: retire and commit. The commit component writes speculative memory values into architectural memory and occurs after retire. While retire must be in order, commit may occur out-of-order, depending on the memory consistency model.

Finally, some speculated instructions are not in the architectural dynamic instruction stream and these instructions must be squashed. Squash can occur anytime after fetch.

Figure 3 illustrates these concepts. Instructions proceed through Fetch (FE), Rename (RN), Execute (EX), Retire (RT), and, for stores, through Commit (CM). Each row indicates the progress of a particular dynamic instruction through these components. The arrows indicate the dependencies that govern instruction execution regardless of the nature of speculation. The horizontal arrows indicate that a particular instruction must go through these components in sequence. The vertical arrows indicate that instructions are fetched, renamed and retired in order. Note that no constraints are placed on execute and commit. The execute components may be re-ordered by a particular microarchitecture provided the results match that of an in-order execution. Similarly, commits may be re-ordered to the extent permitted by the memory consistency model, such as SPARC's Total Store Order (TSO) or x64's Processor Consistency. In Figure 3, the second instruction requires a helper instruction and the helper instruction, as well as the original instruction, go through the remaining phases starting with rename. The third and fourth instructions are along a wrong path and consequently, the third instruction is squashed following execute and the fourth following rename.

2.3 Speculative Register and Memory state

To enable the cycle simulator to model any form of control or data speculation, ValueSim maintains and updates speculative values on behalf of the cycle simulator and provides a general querying mechanism to enable retrieval of any speculative register or

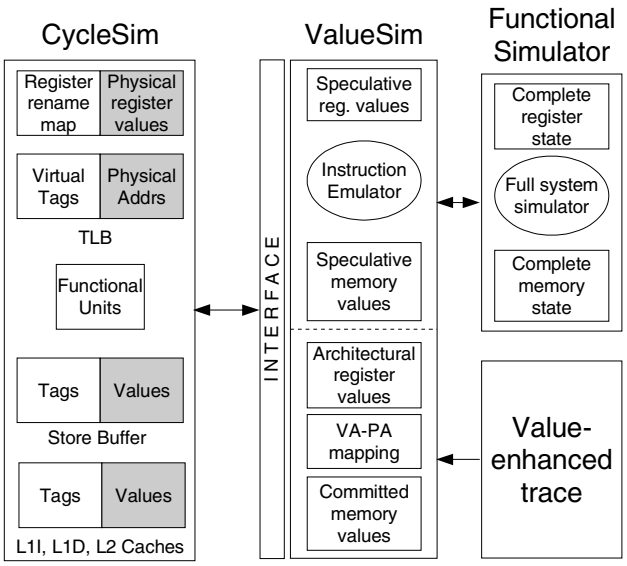

Figure 4. The components of the simulator

memory value. In current simulation frameworks, cycle simulators maintain microarchitectural state, such as the tags in the cache and the state of register rename maps, but do not maintain register and memory values. Functional simulators usually provide committed (or architectural) register and memory values, but not speculative values. Traces do not usually contain register and memory values. Until recently, cycle simulators did not require values to model contemporary microarchitectures, but with the exploitation of deeply speculative techniques in future microarchitectures, it is critical to provide these speculative values.

Figure 4 illustrates the distribution of responsibilities for state maintenance between the cycle simulator, ValueSim and the functional simulator. The cycle simulator is responsible for maintaining microarchitectural state. The cycle simulator box represents all the machine state that the cycle simulator "pretends" to maintain; however, traditional cycle simulators maintain only the *unshaded* parts of that machine state and ignore maintaining the *shaded* portions. The functional simulator maintains some of these shaded parts, corresponding to the last retired instruction (most importantly, the complete register and memory state of the system), as shown on the right side of the figure. What is missing are the speculative versions of the shaded parts corresponding to the in-flight instructions. In a processor that employs aggressive speculation, hundreds of instructions may be in-flight in a partially executed state. Though these instructions may have modified registers and memory, these modifications are not reflected in either the functional simulator (which only maintains the state of retired instructions) or the cycle simulator (which does not maintain any register or memory state). Though one can attempt to retrofit a cycle simulator with speculative state, such attempts are likely to be error-prone. Instead, we advocate that ValueSim maintain this speculative state, which consists of the register and memory values generated by instructions that have not yet been retired.

In order to utilize these speculative values, the cycle simulator must have the capability to access (and modify) these values using a convenient interface. A static register identifier is not sufficient to retrieve a particular speculative register as multiple in-flight instructions may be writing to the same static register number. Thus, we need a way to name the dynamic instance of a particular register in order to retrieve the correct speculative value. The combination of dynamic instruction number and a static register number provides such an identifier.

Instruction Progress Calls		
ValueSim call	**Input parameters**	**Return values**
`fetch()`	strandId PC Virt.Addr.(VA)	instr. word PC Phys.Addr.(PA) fetchId
`rename()`	strandId, fetchId renameId, [instr. word]	
`execute()`	strandId, renameId	*For load/stores:* access VA, PA *For branches:* direction, target
`accessMemory()`	strandId, renameId physical Address	
`retire()`	strandId, renameId clock cycle	async. events
`commitStore()`	strandId, renameId	
Accessor Calls		
ValueSim call	**Input parameters**	**Return values**
`getRegister()` `getMemoryWord()`	strandId, [renameId] reg. number or PA	register value or memory word
`convertToPa()`	strandId, [renameId] virtual address [contextId]	memory word
`setRegister()` `setMemoryWord()`	strandId, [renameId] reg. number or PA reg. value or mem. word	

Table 1. A subset of the ValueSim API

In our framework, each instruction that is renamed is assigned a monotonically increasing identifier called the renameId, thus providing us with a unique dynamic instruction identifier. Subsequently, when the cycle simulator retrieves a register value from ValueSim, the cycle simulator provides the instruction's renameId as well as the register number. ValueSim efficiently determines the youngest dynamic instruction that has the same register identifier and returns the value produced by the latest execution of that instruction. Finally, if there are no speculative values for the register, ValueSim returns the committed value obtained from the functional simulator.

The interface for accessing speculative memory values is similar: the cycle simulator provides the physical address and the renameId. ValueSim, from amongst the set of speculative stores with a lower renameId that have written to the identified physical address, returns to cycle simulator the last value written. If no speculative stores target the physical address provided by the cycle simulator, ValueSim returns the committed value in the functional simulator.

2.4 ValueSim API

The ValueSim API, shown in Table 1, provides for an implementation of the functionality described in the last two subsections: instruction progress calls and accessor calls that retrieve and modify specific speculative values.

The cycle simulator makes instruction progress calls to inform ValueSim about the progress of an instruction through the subcomponents identified earlier. ValueSim, in turn, returns values that are required by the cycle simulator, such as the instruction word on fetch and the load/store address on execute. At the fetch stage, the cycle simulator calls ValueSim `fetch()` with the strandId (hardware strand ID) and the virtual address of the PC (program counter). ValueSim queries the functional simulator to translate this virtual address to a physical address, retrieves the instruction word at the translated physical address from the functional simulator, and delivers both the physical address and the instruction word to the cycle simulator. In addition, ValueSim tags each `fetch()` with a fetchId, which is then used to relate it to a subsequent `rename()` of that instruction or its helpers. The `rename()` calls order the dynamic instructions sequentially and ValueSim tracks dependencies in this sequential stream of instructions.

Following the `execute()` call, ValueSim emulates the instruction and updates the speculative value of the register destinations. As discussed earlier, `execute()` may be called out-of-order and multiple times. For control transfer instructions, ValueSim returns the branch direction and target. For load/store instructions, the `execute()` call only performs virtual address generation and translation to a physical address, while the `accessMemory()` call performs the actual memory access and updates the speculative register (for loads) or speculative memory (for stores). This separation of address calculation from memory access enables the cycle simulator to modify the calculated address and elegantly model speculation mechanisms such as cache way prediction.

At `retire()`, ValueSim requests that the functional simulator advance instruction execution and update the architectural registers and memory at the specified `clock cycle`. If this instruction receives an asynchronous event (either due to I/O or other strands) in the full system functional simulator, ValueSim notifies the cycle simulator, which squashes the in-flight instructions and handles the asynchronous event. Using `squash()`, the cycle simulator informs ValueSim to squash instructions that are incorrectly speculated. Using `commitStore()`, store values are committed to architectural memory and made globally visible to all other processors. All instructions that are renamed should eventually be retired or squashed.

The accessor calls in Table 1 enable the cycle simulator to retrieve any speculative or architectural register or memory value and modify any speculative value. Furthermore, ValueSim checks the correctness of these speculation mechanisms by validating the values produced using the speculative optimization against the in-order functional simulator. This validation also helps find violations of dependency constraints and latent cycle simulator bugs.

Table 1 shows only the most common calls to ValueSim. For instance, there are also calls to `save` and `restore` simulator state for checkpoint support. The `copyState()` call supports multi-path speculation techniques (e.g., helper scout threads and dual-path execution) by copying relevant state to a new thread.

2.5 Supporting Trace-Driven Simulation

In order to generate speculative values in trace-driven mode, ValueSim requires an enhanced value trace that contains selective architectural values. Regular traces typically just contain the instruction stream, branch target/directions, and virtual and physical addresses of loads/stores. The enhanced value traces that we have devised additionally contain all register values at the beginning of the trace, and selective memory values. ValueSim internally emulates all instructions so it is redundant to include output destination register values in traces.

Given that the systems modeled may have hundreds of gigabytes of memory, it is not realistic to embed the entire memory image in the trace. A key observation is that an incorrectly speculated access to main memory in such systems is likely to take over 125 ns (or over 500 CPU cycles) to resolve and such accesses are extremely likely to be squashed before they deliver values to their dependent instructions and influence any subsequent speculative effects. Hence, in most cases, it is adequate to maintain a working set that is roughly the size of the largest cache, say the L2 or L3 cache. Accordingly, the functional simulator contains a simple, direct-mapped cache simulator, simulating say a 64 MB, 128-byte line cache. Only when a load/store misses in this cache does the functional simulator output a 128-byte cache line to the trace. The functional simulator handles I/O and DMA writes by invalidating relevant lines and not caching uncacheable device locations. Given such a trace, ValueSim also maintains a cache simulation of a larger cache (inclusive with respect to the one maintained by the functional simulator), say a 1GB, 16-way, 128-byte line cache. Since this cache is inclusive with respect to feasible on-chip caches, ValueSim is extremely likely to deliver memory values for accesses that hit in the target system's caches. We have developed sophisticated compression techniques for regular (non-value) traces, as well as value traces. Currently, value traces require less than twice the storage as regular traces, with room for further improvement.

At `retire()`, ValueSim discards the timing information from the cycle simulator and delivers asynchronous events to the cycle simulator in the order specified in the trace. ValueSim maintains additional structures for architectural register and memory values as shown in the bottom half of the ValueSim box in Figure 4, and also keeps track of virtual-physical translations.

In trace-driven mode, ValueSim is occasionally unable to provide address mappings (for ITLB, DTLB), or deliver a desired memory value (the instruction opcode for Icaches, the data value for DCaches) to the cycle simulator. Table 2 summarizes the frequency of these events when the cycle simulator is modeling wrong path effects and hardware scouting (HWS). We conclude that the occurrences of these events are very rare and have minimal impact on the accuracy of our speculation modeling of HWS. Thus, trace-driven simulation using ValueSim can simulate a similar range of speculation mechanisms as execution-driven simulation using ValueSim.

2.6 Application of ValueSim

In this subsection, we illustrate how ValueSim may be used to model some existing speculation mechanisms such as data speculation, control speculation, hardware scouting and value prediction. Figure 5 presents an example of data speculation. The store I1 and load I2 are to the same address A, and load I2 must execute after store I1 to ensure that %r3 is correctly set to the value

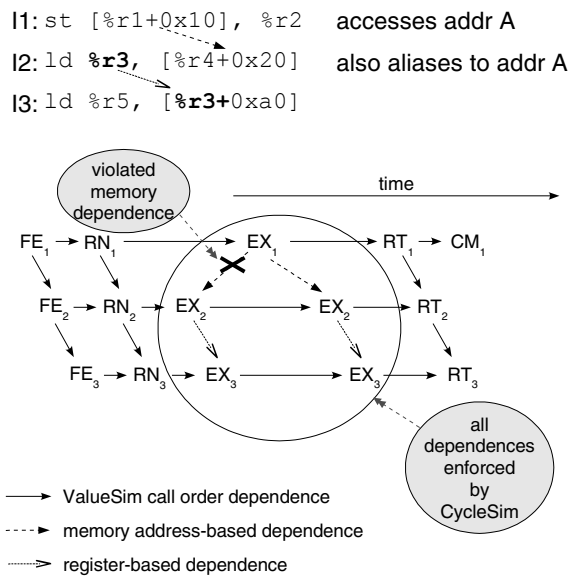

Figure 5. An example of load-store misspeculation

in %r2. However, this memory dependence is violated and load I2 is speculatively executed prior to load I3. As as result, load I3 executes with the wrong value of %r3 and potentially pollutes the data cache. The processor eventually determines the memory dependence violation and re-executes I2 and I3. Conventional trace-driven simulation does not maintain register values and so cannot deliver the value of %r3 prior to the execution of I2 which is required to generate the wrong speculative address of I3. Execution-driven simulators that use checkpoint, squash, and re-execute are unwieldy at modeling data speculation. Using ValueSim, the first speculative execution of I2 retrieves the architected value for address A, since I2 has not yet written the speculative value. This value is then delivered to the speculative (potentially polluting) execution of I3. Thus, ValueSim is able to model this accurately.

To model control speculation using ValueSim, the cycle simulator needs to call ValueSim in essentially the same way that it models the underlying microarchitecture. On reaching a branch, it fetches and executes instructions down the predicted path and eventually executes the branch to resolve it. If the initial prediction matches the branch resolution, the instructions are eventually retired. Otherwise, the instructions down the mispredicted path are squashed. Unlike most other simulators, ValueSim does not need to be explicitly checkpointed after each branch or even each mispredicted branch as it can recover to the state of any unretired instruction whenever required.

In hardware scouting, the processor fetches the instructions that follow a blocking load speculatively, executes the instructions that are independent of the load, and converts subsequent missing loads to prefetches [23]. When the original missing data returns, the processor resumes execution from a checkpoint at the original missing load. The cycle simulator uses ValueSim to model the speculative execution of instructions in HWS mode. The cycle simulator requests ValueSim to set the destination register of the original

Benchmark	ITLB ValueSim misses per 1M spec. instrs	ICache ValueSim misses per 100 spec. instrs	DTLB ValueSim misses per 100 spec. loads	DCache ValueSim misses per 100 spec. loads
SPECweb	28.56	0.05	0.03	0.44
SPECjbb	0.00	0.18	0.00	0.16
OLTP	0.00	0.00	0.01	0.13

Table 2. Effectiveness of ValueSim in trace-driven mode

missing load, as well as any other missing loads, to the NT (Not There) value, which ValueSim then automatically propagates to dependent instructions. If missing load value prediction is used during HWS, the register is set to the predicted value instead of a NT value. The cycle simulator issues prefetches for the loads that miss the cache and whose addresses are calculated as non-NT from ValueSim. When HWS terminates, the cycle simulator informs ValueSim that all instructions in HWS mode are squashed. The register and memory state prior to the missing load is automatically recovered, without needing an explicit checkpoint.

These examples illustrate how the general speculation model supported by ValueSim enables the user to prototype a range of speculation mechanisms quickly and accurately.

3 Methodology

In the remainder of this paper we highlight the ability of ValueSim to enable cycle simulators to accurately model a variety of aggressive speculative techniques, while obtaining their input from a value-enhanced trace. Evaluation is focused on three important commercial applications: OLTP, SPECjbb and SPECweb99. OLTP represents an on-line transaction processing workload. SPECjbb is a Java benchmark focusing on typical Java business applications, while SPECweb is a benchmark for measuring web server performance.

The value-enhanced traces are generated from optimized binaries using a full-system simulator and capture both application and operating system activity. The traces are of sufficient length to allow the initial portion to be used to warm the caches and other processor structures, and the remainder to collect statistics. All three applications are transaction oriented and do not exhibit phase changes. As a result, 50M instructions were found to be sufficient to warm the processor's level 2 (L2) cache, while 100M instructions yield a representative transaction mix that is sufficient for collecting accurate statistics.

The processor simulator used is an in-house cycle-accurate timing simulator, which models both bandwidth constraints and queuing effects in the memory hierarchy. Performance is evaluated with respect to a 4-way CMP (Chip Multiprocessor) processor, where each out-of-order (OOO) core is 8-wide fetch, 3-wide issue, has a 64 entry issue window, a 64 entry reorder buffer, and a 16-stage pipeline.

Each core has a private L1 instruction and data cache (each 32KB, 4-way set associative, 64-byte line size and 4-cycle latency) and the cores share a unified L2 cache (2MB, 4-way set associative, 64-byte line size and 25-cycle latency). The cores are

Expt.	Wrong Path	HWS	Value Prediction
A	No	No	No
B	Yes	No	No
C	No	Yes	No
D	Yes	Yes	No
E	No	Yes	Yes
F	Yes	Yes	Yes

Table 3. Experimental descriptions

assumed to run at 4GHz and the modeled off-chip bandwidth is 20GB/s. The conditional branch predictor is a 64K entry gshare predictor, the branch target buffer has 1K entries and is direct-mapped, while the return address stack has 16 entries.

For this CMP processor, the implications of not simulating wrong path instructions are investigated by analyzing the impact on several important system metrics, including, off-chip bandwidth, cache state, and queuing delays in the memory hierarchy. The performance ramifications of not accurately modeling this misspeculation are also illustrated, with performance measured in terms of instructions per cycle (IPC) metrics.

We also investigate how the importance of wrong-path effects is influenced by the use of aggressive speculative techniques, such as hardware scouting (HWS). For standard wrong path effects, the length of the wrong path is constrained by the size of the issue window. In contrast, HWS may potentially progress for hundreds of instructions down a wrong path before scouting is terminated by the return of the missing trigger load. Thus, modeling wrong path effects for HWS can be very important.

We also augment HWS with a value predictor (64-entry, direct mapped, PC indexed) [19], allowing HWS to target loads which are dependent on prior L2 cache misses [11]: if the predictor can supply the predicted value to the dependent load, it can be speculatively issued, rather than skipped, as required with standard HWS.

For each of these speculative techniques, the effects of not modeling wrong path effects are investigated. This is achieved by performing the simulation with wrong path modeling disabled and then re-simulating with wrong path modeling enabled, as illustrated in Table 3, allowing the impact of the wrong path effects to be accurately determined.

4 Results

In this section, we evaluate three aspects of ValueSim: simulation speed, validation and effectiveness. With regards to simulation speed, an important concern is the slow-down introduced by the ValueSim approach. An analysis of simulation runs with all the speculation features enabled on our flexible high-level trace-driven cycle simulator indicates that 52% of the total simulation time is spent in ValueSim. Thus, ValueSim introduces a slow-down of around 2x.

With respect to validation, we have found that conventional trace-driven cycle simulation does not match low-level Register Transfer Level (RTL) simulation for many diags (diagnostic tests) used for performance validation of our CMP designs; in one case the performance estimates differed by a factor of 2.6x. In these cases, when the detailed cycle simulator was coupled with ValueSim, the difference in performance estimates narrowed to less than 5% and the process of narrowing the discrepancies in the

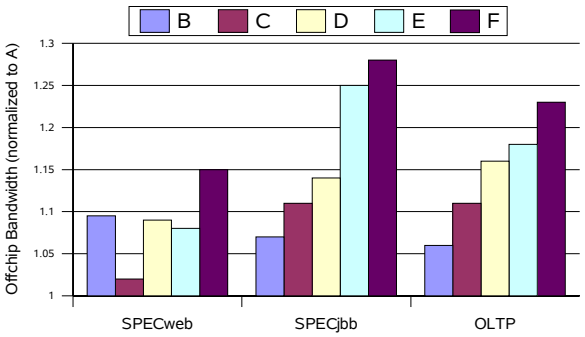

Figure 6. Increased off-chip bandwidth due to speculation

models led to a better understanding of both the cycle simulator and RTL. Thus, we have validated the ValueSim approach on many performance diags where speculative effects play a large role.

Finally, with regard to the effectiveness of ValueSim, a key ramification of not modeling wrong path effects is the effect of the memory references generated during these periods. With the support provided by ValueSim, our timing simulator is able to accurately model the off-chip requests generated by the application (load misses, instruction fetches, store fills and writebacks) on the correct path and on the wrong path.

Figure 6 illustrates the variation of in-bound off-chip requests when modeling of wrong path instructions is enabled for a variety of speculation techniques. The number of off-chip misses per correct path instruction can increase by up to 1.27X when aggressive speculation is utilized. Figure 6 also shows that not modeling wrong path effects results in a significant underestimation of the bandwidth consumed by the various speculation techniques. For instance, not modeling wrong path effects in SPECweb underestimates the bandwidth consumed by hardware scouting and value prediction by around 8% (C to D). Thus, accurate modeling of speculation techniques is critical for estimating the off-chip bandwidth requirements of future CMP designs.

The additional off-chip requests generated while on the wrong path can have other ramifications: they can result in increased pressure on the memory hierarchy, resulting in increased queuing delays for correct path memory requests, which, in turn, can have performance implications. Figure 7 illustrates the variation in

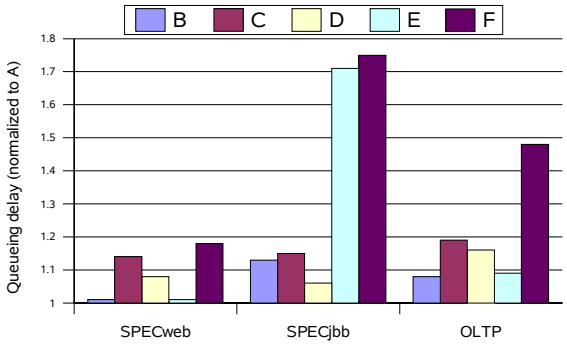

Figure 7. Increased queuing delays due to speculation

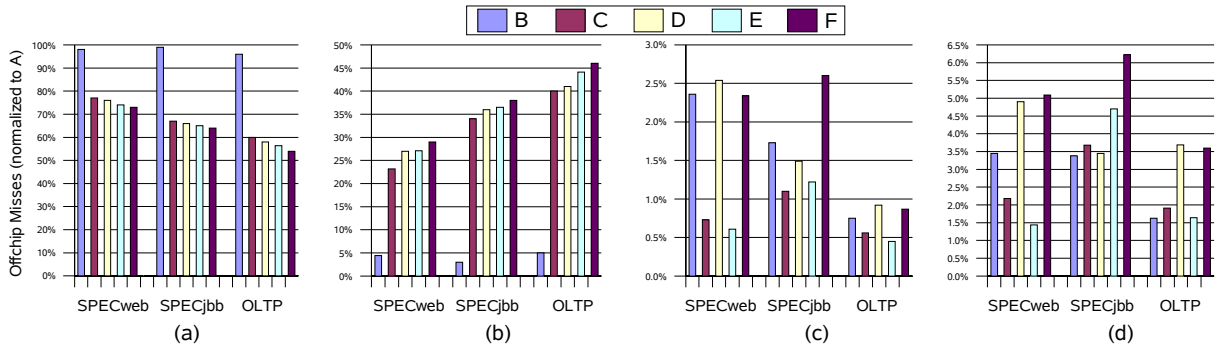

Figure 8. Variation in (a) demand fetches, (b) prefetching events, (c) pollution events and (d) useless prefetches due to speculation

the average queuing delay experienced by off-chip requests when speculative effects are modeled with/without wrong path effects. Aggressive speculation techniques increase queuing delays by up to 1.75X. Again, accurate modeling of wrong path effects associated with aggressive speculation techniques is especially relevant for CMP systems, where one core's demand fetches may have to contend for resources with speculative requests from other cores [26].

The impact of speculative off-chip references on the processor's cache state is characterized as prefetching (the cache line brought in is subsequently accessed by a demand fetch), polluting (the cache line displaces another line causing a demand fetch miss to that line), or useless (the cache line is never accessed by a demand fetch and displaces a dead line). Figure 8 illustrates that demand fetches decline and prefetches increase as increasingly speculative techniques are employed. All of the speculation techniques have a predominantly beneficial effect, with prefetching effects outweighing the pollution effects. However, the pollution effects of these speculation techniques can be significantly underestimated if wrong path effects are not modeled. For instance, hardware scouting increases the polluting events for SPECweb by a factor of 4 when wrong path effects are modeled (C to D). In CMP designs, where multiple cores share an L2 cache, one core's speculation can negatively effect the performance of other cores and a detailed, accurate analysis is crucial for tuning these speculation mechanisms [26].

Given the predominantly prefetching effects illustrated in Figure 8, performance gains are to be expected from these speculation techniques. Figure 9 illustrates the performance benefits associated with the speculation techniques can be significant; OLTP performance improves by over 1.15X when HWS is used in conjunction with value prediction. However, SPECweb performance decreases with HWS illustrating that aggressive speculation techniques, such as HWS, do not always result in performance improvements. In SPECweb, the increase in queuing delays and cache pollution outweighs the prefetching benefits. The performance benefit of hardware scouting in conjunction with value prediction is understated by several percent (E to F), when the wrong-path effects are not modeled for OLTP. This performance difference can easily swing the balance between whether or not a speculation technique is deemed sufficiently beneficial to be incorporated into a next-generation processor.

The aggressiveness of speculation mechanisms, such as the value predictor utilized during hardware scouting, can be altered by changing its size and the confidence required before a prediction is made. Figure 10 presents the changes in performance, prediction accuracy, and bandwidth (relative to a baseline of HWS without value prediction) as the aggressiveness of the value predictor is varied. While moving from a conservative to a moderately aggressive predictor improves performance, value misprediction rate and, consequently, off-chip bandwidth increase significantly, especially for SPECweb and SPECjbb. Furthermore, adopting an aggressive predictor actually reduces performance relative to a moderate predictor for all benchmarks due to the substantial increase in misprediction rate, off-chip bandwidth and other factors not shown, such as cache pollution and queuing delays.

A key benefit of ValueSim is the ability to accurately model these speculation techniques, allowing for a precise analysis of their impact on the system and the iterative refinement of their design. For example, simulation using ValueSim can provide quantitative data to assist in deciding whether, depending on the processor design, an aggressive value predictor is or is not desirable.

5 Related Work

Microprocessor performance analysis is normally undertaken using either trace- or execution-driven simulation, each with its own benefits and shortcomings.

5.1 Trace-Driven Simulation

In recent years, there has been increasing concern about the ability of trace-driven simulators to accurately predict the performance of next-generation microprocessors [8, 17]. A trace of correct path instructions does not contain the path that the processor temporarily follows when a misprediction occurs, preventing traditional trace-driven timing simulators from modeling wrong path effects. A variety of solutions have been proposed to this problem: modeling a fixed number of stall cycles [22], injecting artificial instructions to mimic the wrong path instructions [24], or scanning ahead in the trace to find the required sequence of instructions appearing as correct path instructions elsewhere [7]. Also, the lack of data values in many traces prevents investigation of enhancements such as value-prediction. While data values can be incorporated

into the trace or supplied via a separate value trace [30], these traces only contain correct-path values, hampering investigation of aggressive speculative data-dependent techniques [15].

With the use of the ValueSim framework and value-enhanced traces, many of these problems are eliminated. In its trace-driven mode, ValueSim provides the ability to simulate wrong path instructions and, via the ValueSim value caches, the majority of both correct path and wrong path values are made available to the cycle simulator.

5.2 Execution-Driven Simulation

While execution-driven simulators [5, 16, 20, 6, 25, 21, 13] address many of the accuracy concerns associated with trace-driven simulations, execution-driven simulation has associated problems. Current execution-driven techniques contain inaccuracies introduced by the use of checkpoints to reduce startup times, the frequent requirement for workload scaling [1] and the requirement for multiple simulations to better bound the range of potential performance variability [2].

5.3 Modeling of Speculative Effects

In the recent past, other simulator writers have realized the need to model speculative effects at a fine granularity in parallel with our efforts on ValueSim. ValueSim was developed four years ago and has now crossed the threshold of maturity and validation where its results can be viewed with confidence. Simics has implemented a Micro-Architecture Interface (MAI) [28, 29]. MAI forces the users to create filters that ensure the desired flow of speculative values between data speculative instructions. Thus, it is inconvenient to model a range of speculation techniques using MAI. The timing-first simulation approach uses the Simics full system functional simulator as well as a simpler emulator in the timing simulator, similar to the ValueSim approach [21]. SimpleScalar has proposed a Microarchitectural Simulation Environment (MASE) [4] (to be released with version 4) to enable simulation of some of the speculative effects that ValueSim was designed to address. MASE will allow users to violate data dependencies and model data speculation.

ValueSim provides a general speculation framework and enables the modeling of speculation techniques using value-enhanced traces as well as execution-driven simulators.

5.4 Effects of Speculation

A number of papers have investigated the potential errors introduced by not simulating wrong path effects. Initial analysis indicated that these effects were largely unimportant [9, 22], while more recent investigations have stressed the importance of modeling these effects [6, 24, 3]. These apparently contradictory conclusions can be reconciled by considering factors such as the modeled memory latency (ranging from 70-cycles in the initial studies to several hundred cycles in the more recent) and the aggressiveness of the speculation employed by the processors [24]. Our data further confirms the trend seen in the recent studies.

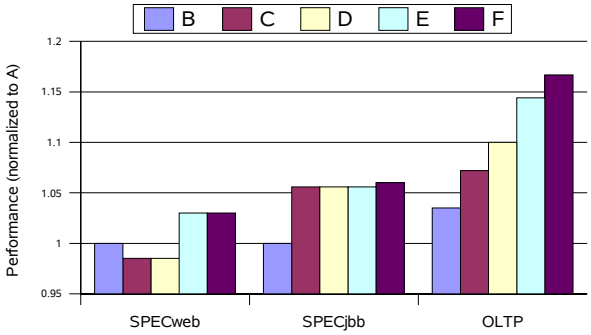

Figure 9. Performance benefits due to speculation

6 Conclusion

In an effort to mitigate the performance impact of the relative increases in the cost of off-chip misses, a range of speculation techniques have been proposed. As this speculation becomes more aggressive, so too do the ramifications of any mis-speculation. In this paper, we propose the ValueSim framework, which enables microprocessor architects to refine these speculation mechanisms and accurately model (and reduce) the effects of mis-speculation, such as cache pollution, bandwidth and increased power consumption.

As cycle simulators often represent considerable investment in validation, ValueSim can be utilized with only minimal modifications to existing simulators. Furthermore, ValueSim supports either trace-driven simulation or execution-driven simulation, enabling architects to leverage the best attributes of both approaches. The ValueSim interface enables a cycle simulator user to prototype a range of speculation mechanisms quickly and accurately.

To highlight the importance of ValueSim, we modeled the effects of Hardware Scouting and missing-load value prediction in future multi-core systems [26]. We clearly demonstrate that these techniques deliver significant performance gains at the expense of a substantial increase in off-chip bandwidth. We further show that value predictors have to be tuned and that unrestrained value prediction can, in fact, lead to noticeable increase in pollution and useless prefetching. As a result, without accurate modeling of speculation mechanisms provided by ValueSim, the effects of mis-speculation can go unnoticed (or significantly underestimated), likely leading to inaccurate performance estimates that are only discovered late in the design cycle with the completion of the RTL model.

References

[1] A. R. Alameldeen et al. Simulating a $2M commercial server on a $2K PC. *IEEE Computer*, 36(2):50–57, 2003.

[2] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Intl. Symp. High-Perf. Comp. Arch.*, pages 7–18, 2003.

[3] D. N. Armstrong, H. Kim, O. Mutlu, and Y. N. Patt. Wrong path events: Exploiting unusual and illegal program behavior

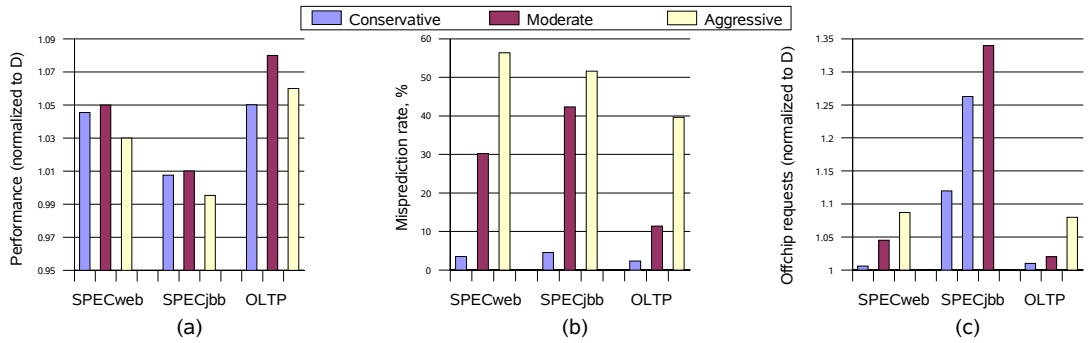

Figure 10. Variation in (a) performance, (b) value misprediction rate, and (c) off-chip bandwidth with different value prediction confidence thresholds (wrong path modeling enabled)

for early misprediction detection and recovery. In *Intl. Symp. Microarchitecture*, pages 119–128, 2004.

[4] T. Austin et al. SimpleScalar version 4 tutorial. *Tutorial in MICRO-34,* `http://www.simplescalar.com/docs/simple_tutorial_v4.pdf`, 2001.

[5] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[6] C. Bechem et al. An integrated functional performance simulator. *IEEE Micro*, 19(2):26–35, 2003.

[7] R. Bhargava, L. K. John, and F. Matus. Accurately modeling speculative instruction fetching in trace-driven simulation. In *Intl. Perf. Computing, and Communications Conf.*, pages 65–71, 1999.

[8] B. Black, A. S. Huang, M. H. Lipasti, and J. P. Shen. Can trace-driven simulators accurately predict superscalar performance? In *Intl. Conf. on Comp. Design, VLSI in Computers and Processors*, pages 478–485, 1996.

[9] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti. Precise and accurate processor simulation. In *Workshop on Comp. Arch. Evaluation using Commercial Workloads at HPCA*, pages 13–22, 2002.

[10] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High performance throughput computing. *IEEE Micro*, 25(3):32–45, 2005.

[11] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Intl. Symp. Comp. Arch.*, pages 76–87, 2004.

[12] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Intl. Conf. on Supercomputing*, pages 68–75, 1997.

[13] J. Emer et al. Asim: A performance model framework. *IEEE Computer*, 35(2):68–76, 2002.

[14] H. Eom and J. K. Hollingsworth. Achieving efficiency and accuracy in simulation for I/O-intensive applications. *Journal of Parallel and Distributed Computing*, 61(12):1732–1750, 2001.

[15] J. González and A. González. Speculative execution via address prediction and data prefetching. In *Intl. Conf. on Supercomputing*, pages 196–203, 1997.

[16] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: Simulating shared-memory multiprocessors with ILP processors. *IEEE Computer*, 35(2):40–49, 2002.

[17] E. J. Koldinger, S. J. Eggers, and H. M. Levy. On the validity of trace-driven simulation for multiprocessors. In *Intl. Symp. Comp. Arch.*, pages 244–253, 1991.

[18] K. M. Lepak, H. W. Cain, and M. H. Lipasti. Redeeming IPC as a performance metric for multithreaded programs. In *Intl. Conf. on Parallel Arch. and Compilation Techniques*, pages 232–244, 2003.

[19] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Arch. Support for Prog. Lang. & OS*, pages 138–147, 1996.

[20] P. S. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[21] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In *SIGMETRICS*, pages 108–116, 2002.

[22] M. Moudgill, J.-D. Wellman, and J. H. Moreno. An approach for quantifying the impact of not simulating mispredicted paths. In *Workshop on Perf. analysis and its Impact to Design at ISCA*, 1998.

[23] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Intl. Symp. High-Perf. Comp. Arch.*, pages 129–141, 2003.

[24] J. Pierce and T. Mudge. The effect of speculative execution of cache performance. In *Proc. Intl. Symp. on Parallel Processing*, pages 172–179, 1994.

[25] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. on Modeling and Comp. Simulation*, 7(1):78–103, 1997.

[26] L. Spracklen and S. G. Abraham. Chip multithreading: Opportunities and challenges. In *Intl. Symp. High-Performance Comp. Arch.*, pages 248–252, 2005.

[27] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: a survey. *ACM Computing Surveys*, 29(2):128–170, 1997.

[28] Virtutech. Introduction to Simics Full-System Simulator without Equal. *White Paper,* `http://www.virtutech.com/pdf/VirtutechSimicsIntroduction.pdf`, 2002.

[29] Virtutech. Simics Micro-Architectural Interface. *User Manual*, 2005.

[30] X. Zhang and R. Gupta. Whole execution traces. In *Intl. Symp. Microarchitecture*, pages 105–116, 2004.