VASA: A SIMULATOR INFRASTRUCTURE WITH ADJUSTABLE FIDELITY

Dan Wallin, Håkan Zeffer, Martin Karlsson and Erik Hagersten
Department of Information Technology
Uppsala University
P.O. Box 337, SE-751 05 Uppsala, Sweden
{dan.wallin, hakan.zeffer, martin.karlsson, erik.hagersten}@it.uu.se

ABSTRACT
This article presents Vasa, a configurable high-performance multiprocessor simulation package for the Virtutech Simics full-system simulator. Vasa includes models of multi-level caches, store buffers, interconnects and memory controllers and can model complex out-of-order SMT/CMP-processors in great detail. However, it can also be run in two less detailed simulation modes being up to 287 times faster on average.

We compare the simulation results from a 16-way cache coherent multiprocessor system with four 4-way SMT/CMP processors in the three simulation modes. Our results indicate that for many architectural studies, it is justifiable to run the simulations in a faster less detailed mode as long as it is not the behavior of the processor itself or the first level caches that is being studied.

KEY WORDS
Vasa, Simics, Full-System Simulation, Multiprocessor

1 Introduction

Computer architecture studies rely on accurate simulation models. Due to simulation slowdown, architectural studies are often restricted to either downscaled datasets or to study only parts of an application. Approximations that jeopardizes result reliability. It has for example been shown that reduced benchmark data sets can lead to very different results compared to reference datasets [1, 2]. Since faster simulation enables increased application coverage, a trade-off arises between experiment reliability and model fidelity.

As chip-multiprocessors become commonplace and an increasingly popular simulation target, the importance of simulator speed is exacerbated. Since multiprocessor system simulation is difficult to parallelize and has largely remained sequential, simulation slowdown grows linearly with the number of modeled processors.

In this paper we present the Vasa [1] simulator framework, which is a modular simulator infrastructure that is tightly integrated with the Virtutech Simics full-system simulator. It was designed to enable rapid prototyping of new architectural ideas. The framework includes processor and memory system models enabling simulation of large-scale multiprocessor computer system. To address the simulation slowdown problem, Vasa provides three different fidelity modes representing different levels of detail and simulation speed.

It is important to select the appropriate level of simulator fidelity because of the significant slowdown in detailed simulation. In this paper we show that a very detailed simulation mode with a pipelined out-of-order processor often can be avoided unless we study the processor itself or the behavior of the first level caches.

The contribution of this paper is threefold:

- We describe the approximations and trade-offs made in the design of the Vasa simulator framework, including the concepts of downstream timing and instant coherence.
- We present a simulator speed evaluation of the three different fidelity levels provided by Vasa.
- We quantify the effect on the experimental results when using the different fidelity levels.

2 Simics Overview

Virtutech Simics is a commercial full-system simulator [3] that can deterministically simulate multiprocessor systems with enough accuracy to boot unmodified operating systems. Simics guarantees that each instruction of the simulated machine is executed correctly from an ISA perspective and provides a timing interface to user modules such as Vasa components. Instruction fetches and data accesses are forwarded to the timing interface, allowing user models to model the corresponding memory system effect and timing of an access. For example, a module modeling a cache can stall the execution of an instruction for an arbitrary number of cycles on a cache miss. Hence Simics provides a simplified approach to execution driven simulation, where model writers can ignore ISA correctness issues.

When simulating a multiprocessor system in Simics, each processor is simulated in a round-robin fashion. The number of cycles executed in each time slot, *cpu-switch-time*, is variable allowing the coarseness in the thread interleaving to be scaled. This can have a significant effect when simulating multithreaded applications with contended locks. Throughout this study we have employed a

[1] Vasa is named after the 90 km cross-country ski race Vasaloppet, http://www.vasaloppet.se, which all authors successfully have completed.

cpu-switch-time of one, however all Vasa memory system components can be run with a higher *cpu-switch-time*.

Simics has the ability to checkpoint a simulation, which allows users to run an application to an interesting point and then save all states of the entire simulated machine to disk. Checkpointing can save simulation time since for example an application initialization phase only need to be run once. All later simulations can then be started from the checkpoint. For commercial benchmarks such initialization or warmup phase can require weeks of simulation [4].

Starting with release 2.0 Simics provides a Micro-Architectural Interface (MAI), which was designed to enable microarchitectural processor modeling. Through the MAI, users can overcome the in-order execution limitation of baseline Simics and write out-of-order execution models. In MAI mode, Simics still ensures that each instruction is executed correctly, but leaves it to a user model to determine when each instruction should pass phases like fetch, decode, execute and commit. This allows very detailed processor timing modeling. The Micro-Architectural Interface allows the processor model writer to focus on timing and leaves the ISA correctness to Simics.

3 Vasa Overview

Vasa is a new execution-driven simulation framework that enables full-system simulation of an entire multiprocessor computer system at various fidelity levels. The Vasa framework contains both processor and memory system components. Three different processor modes are supported, two in-order modes and one MAI enabled out-of-order mode. Each of the processor modes provides a different level of detail and speed. Vasa contains a single memory system model designed to handle all three processor modes.

The simulation infrastructure includes the following components:

- **Processor:** SMT-support, out-of-order execution, speculative execution, branch prediction, configurable number of pipeline stages and functional units, store buffer support etc.
- **Caches:** Write-through/write-back, variable cache size, cache line size, associativity and latency, LRU, random and NRU replacement policies, MOSI-coherence protocol between caches or shared caches, multiple banks and miss status holding registers.
- **Interconnect:** Support for a generic directory based protocol with variable latency and bandwidth for variable network topologies. The interconnect is built up from modules called coherence agents and links.
- **Memory controller:** Two different memory controllers, one simple with static access time to memory, the other advanced modeling bursts, channels, devices, banks and DRAM timing in detail.

Each component in the Vasa framework is built from Simics modules and connected using Simics interfaces. Setting up a model system, thus involves creating modules for each object to model, e.g., a cpu module, a store buffer module and a cache module and then connecting them together with Simics interfaces.

Vasa models are set up using the Simics configuration system which can be controlled through Simics built-in Python runtime environment. A system configuration is therefore simply created by a Python script, which simplifies varying and scaling the architectural features of a system.

Vasa also supports checkpointing. That is, at a certain time most of the memory system states can be stored to disk. However, the Vasa checkpointing facility does not capture outstanding instructions or transactions, hence some information is lost. Through checkpointing, memory system warming can be performed once and is thereafter avoided. The checkpointed states can be loaded in different simulation modes which makes it possible to fast-forward the simulation using a faster simulation mode and then use a more detailed mode at certain times. This is useful since it has been shown that it is common that not all parts of an application have to be modeled in great detail and still yield accurate results [1].

4 Time Modeling in Vasa

One of the largest challenges in simulator design is to strike a balance between the level of detail and flexibility in the model. One of the design goals of Vasa has been to create a framework that is as simple and easily modifiable as possible. To reduce model complexity we have employed two simulator strategies which we call *downstream timing* and *instant coherence*. We have found that these strategies lead to large simplifications and model clarity.

4.1 Downstream Timing

The main idea behind *downstream timing* is to model bandwidth and latencies for a transaction on its way down the memory system. Once the piece of data is found and latency has been modeled the memory system state is momentarily updated.

In Vasa there are two kinds of memory system messages, *requests* and *actions*. *Requests* are memory access messages passed on from the processor down the memory hierarchy until the requested piece of data is found. On the way through the memory hierarchy the memory request can be stalled multiple times. For example in a cache with N cycles access time, the lookup is performed N cycles after the request was forwarded from the previous level. If the access is a miss, the request is then forwarded to the next level in the hierarchy, where it again is stalled until the access time of the next cache has passed. In addition to modeling access time, the requests allocate bandwidth on

the way through the hierarchy. This is performed by using a configurable number of Miss Status Handling Registers (MSHRs). Each cache can only handle a fixed number of outstanding requests. If all MSHRs are occupied the request cannot enter the cache and forces it to stall. Once the piece of data is found, a fill action is carried out by an *action* message that is sent from the memory side of the hierarchy in the direction towards the processors. Actions take place momentarily in the entire cache and memory hierarchy and free the MSHRs that was allocated by the request.

The principle of downstream timing is to model the timing of messages that are passed down the memory hierarchy while upstream messages are performed instantly.

4.2 Instant Coherence

Instant coherence is a multiprocessor extension of downstream timing. When simulating coherence among multiple nodes, a module called the pseudo directory is used. The pseudo directory is Vasas internal directory, without counterpart in the modeled hardware, that keeps track of which nodes and states a certain piece of data is cached. The pseudo directory determines and sets up all coherence messages that would be necessary for a real directory coherence mechanism to satisfy a request. Depending on who is the requester, the owner or the home node, this can require a large number of messages. All messages have a certain size in bytes and are sent via *link* modules with adjustable latency and bandwidth. Time is modeled for all the messages that are sent across the simulated network. As soon as the requested piece of data has been been received by the requester, we can finish model timing. Now, actions are sent in the direction towards the processors, which momentarily update the entire cache/memory hierarchy. At the same time, we also update the pseudo directory to guarantee correct coherence throughout the modeled system. We call this method of using a centralized directory and momentary updates, *instant coherence*, since the messages required by a request is determined instantly.

Instant coherence in combination with downstream timing makes it possible to model realistic network latencies and bandwidths without introducing intermediate transition states in the coherence protocol. This simplifies the model significantly and makes it easier to verify and use. Instant coherence also has the benefit of separating the modeling of time from the coherence modeling, which makes it easier to modify either part.

5 Vasa Simulation Modes

Vasa can be run in three different modes, depending on what level of detail or speed that is required. The three modes provided are functional simulation (FUNC), timing simulation without a detailed processor model (STALL) and timing simulation with a detailed processor model (MAI).

5.1 Functional Simulation - FUNC Mode

In the functional simulation mode (FUNC), no timing is included. That is, all instructions take exactly one cycle and no timing feedback is provided from the memory system. Functional simulation is useful for studying application behavior in general but not for studies requiring the inclusion of timing sensitive behavior such as contention and thread interleaving. The main advantage of functional simulation is its superior speed. Since functional mode does not include timing feedback, exactly the same trace of instructions is observed between simulations of different systems. Hence scheduling effects cannot interfere with simulation results.

5.2 Timing Simulation without a Processor Model - STALL Mode

Many architectural studies require the inclusion of timing. To get an accurate thread interleaving in a multiprocessor simulation, the timing of each memory system access must be modeled. Time modeling in STALL mode is handled in an event driven manner. Simics provides a processor clock cycle-indexed event queue for each processor that allows models to post callbacks arbitrarily many cycles in the future. The Simics event queue is used in Vasa to post events such as cache lookups after a certain access time delay. This ensures that a request reaches the right level in the memory hierarchy at the right time.

5.3 Timing Simulation with a Processor Model - MAI Mode

The drawback with the previous modes is that they do not simulate complex deeply pipelined processors and therefore do not capture the effects of out-of-order and wrong-path execution. To be able to study modern systems with these features, Vasa can be run in a more accurate mode with a complex processor module. This mode is based on the Micro-Architecture Interface (MAI) mode described in Section 2. The cache hierarchy is simulated in the same way as in the STALL mode, except that instead of using the Simics event queue, we connect a module called the cycle handler to each object. The cycle handler steps all modeled objects forward each cycle. The reason for having the cycle handler module connected instead of using the Simics event queue mechanisms is that it enables randomization of the order time is advanced among the objects. By this randomization, we can avoid arbitration modeling and still prevent specific hardware threads from being favored.

6 Taking Advantage of the Simics Simulator Translation Cache

Simics uses a built-in software cache to speed up the simulations, the Simulator Translation Cache (STC). The STC

stores addresses for each processor that can be accessed without side-effects. A STC hit guarantees that an instruction fetch or a data access will not cause, e.g., an alignment exception or a TLB miss. The STC is divided into a data STC (dSTC) and an instruction STC (iSTC) which can be enabled and disabled independently. The simulator performance speedup provided by the STCs can be significant. However instruction fetches and data accesses that hit in the STCs are not forwarded to the timing interface and will therefore be missed by Simics user models such as Vasa. Hence, STCs are normally turned off to get a complete memory access trace.

The Simics API provides methods to force addresses to be evicted from the STCs. Vasa exploits this feature in order to provide a method of running Simics with the STCs turned on. Addresses that will not affect the memory system state can be allowed in the STC. For example in a system with first level caches that use a random or FIFO replacement, a fetch or data access that hits in the first level cache will not affect the state and can be allowed in the STCs. Similarly in a LRU cache, addresses residing in the MRU set of the L1 caches can be safely kept in the STCs. Once an address is evicted or invalidated, the corresponding entry in the STC must be flushed to make sure that an access to the same address is not filtered out the next time. Because of the principle of locality exposed in most programs many instructions can be safely kept in the STCs and thereby provide a simulator speedup. Note that using Vasa with the STCs turned on will yield incorrect cache access statistics. However, the number of cache misses will still be correct since all cache accesses that lead to a cache miss is modeled.

A similar simulation technique based on the STC have previously been explored by Ekman [6] in a uniprocessor context. STC simulation gains more speed in uniprocessor than multiprocessor simulations since uniprocessor simulations does not involve cache coherence invalidations.

7 Comparing Simulation Modes

In this section, we compare the experimental results in terms of memory system statistics and execution time between the different simulation modes in order to identify and quantify the differences between various fidelity levels.

7.1 Case Study

In this case study, we simulate a SPARC V9 system configured to resemble a scaled down IBM Regatta system with a total of 16 threads [7]. The system has two SMT threads per processor core, two cores per chip (2-way CMP) and a total of four chips. Each core has separate instruction and data L1 caches and shared L2 and L3 caches. Table 1 contains the simulated system parameters for the out-of-order processor model. The cache line size is the same in all caches in the system, but is varied between 64, 128 and 256 byte in different experiments. The model system is set up in Vasa according to Figure 1.

Processor	2- way SMT, dual core
Frequency	3 GHz
Pipeline stages	12
Fetch/Issue/Retire Width	16/6/8
Instruction window	256 entries
Branch predictor	36864 bits YAGS
Store buffer	32 entries/thread
L1 data cache	32 KB, 2-way, 2 cycles, 32 MSHRs
L1 instr cache	64 KB, 2-way, 2 cycles 32 MSHRs
L2 Shared cache	1 MB, 16-way, 11 cycles 128 MSHRs
L3 Shared cache	8 MB, 16-way, 81 cycles 128 MSHRs
Cache line size	64 B/128 B/256 B (varied in the experiments)
Interconnect	Fully connected
Bandwidth	3 GB/s per link
Local Mem Latency	200 cycles
Remote Mem Latency	600 cycles

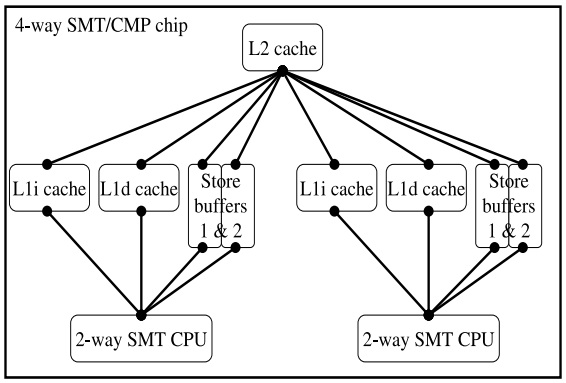
Table 1. Simulated Target System Parameters

The core model in the MAI mode works as follow: Each thread in the processor core fetches 8 instructions from the instruction cache. The instructions are selected using the ICOUNT fetch policy [8]. The decode and rename stages are 6-way superscalar before the insertion into the issue queue. Instructions can be issued and executed out-of-order but are committed in-order. The threads share the same branch predictor and branch-target buffer but have individual return address stacks.

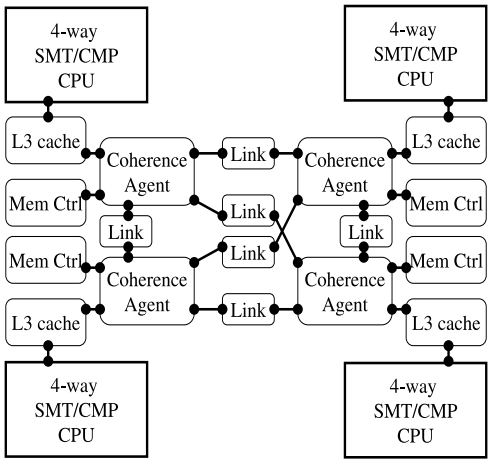
The STALL mode has exactly the same cache and memory system parameters. This mode cannot model a SMT-processor since it would require the modeling of the entire processor pipeline, which is not supported in baseline Simics. Therefore, we simply let two processors share the same L1 caches. Each core in the STALL configuration is from an architectural point of view a 2-way CMP instead of a 2-way SMT processor. The FUNC mode is identical to the STALL mode except that all memory accesses complete in exactly one cycle.

7.2 Benchmarks and Warming

The simulations were run with a total of ten different benchmarks. Eight of the benchmarks were taken from the SPLASH2 parallel benchmark suite [9]. The eight SPLASH2 benchmarks were BARNES, FFT, LU continuous (LU_C), LU non-continuous (LU_NC), OCEAN continuous (OCEAN_C), RADIX, WATER spa-



(a) The chip-configuration in the MAI mode of the modeled case study system. In the STALL and FUNC modes, each 2-way SMT core is replaced by a 2-way CMP core.



(b) The 16-way case-study system is built from four 4-way SMT/CMP chips.

Figure 1. Case study system configuration

tial (WATER_S) and WATER nsquared (WATER_N). These SPLASH2 benchmarks were run with the default input sets specified in the SPLASH2 code release except that OCEAN_C was scaled up to 258 times 258 data points. The caches were warmed according to Woo et al [9].

In addition to these benchmarks, we also modeled two commercial workloads, SPECJBB2000 and APACHE. SPECJBB2000 (JBB2000) is a commercial JAVA-based middleware benchmark which evaluates the performance of server-side JAVA [10]. APACHE is a benchmark modeling the Apache open source Web server to which URL-requests are sent by a client [11]. SPECJBB2000 was run for 4000 transactions after a warmup period of 100000 transactions. APACHE was run 400 transactions with 1000 transactions warmup period.

7.3 Model Verification

The Vasa simulation framework has been verified in functional mode against the simulator used in the SPLASH2 characterization article [9] and the SUMO-simulator [12]. The verification was made in terms of cache miss ratios, coherence and data traffic at variable cache line sizes. Unfortunately we have not been able to verify the more detailed modes of Vasa against other simulators or real hardware. The task is complicated by simulator model differences making apple to apple comparisons difficult. Hence we are unable to quantify the effects of the downstream timing and instant coherence simplifications. However for comparative simulation studies, i.e., an optimized model is compared against a baseline model, we believe that these simplifications is likely to affect both models equally and still provide the correct relative performance trends.

7.4 Simulation Performance

In this section we present the relative difference in simulation speed between the different modes. All runs are carried out on the same Opteron-based system. Compared with the most accurate MAI mode, the STALL mode is about 42 times faster and the FUNC mode about 164 times faster on average according to Table 2.

	MAI	STALL		FUNC	
		no-STC	iSTC	no-STC	iSTC
BARNES	1.00	27.9	35.2	67.4	134
FFT	1.00	47.1	71.6	54.8	142
LU_C	1.00	60.4	98.6	114	242
LU_NC	1.00	54.8	103	84.6	198
OCEAN_C	1.00	44.6	64.1	314	624
RADIX	1.00	42.6	60.8	392	666
WATER_S	1.00	30.8	50.2	62.6	141
WATER_N	1.00	41.1	67.4	70.0	169
APACHE	1.00	32.1	30.6	370	447
JBB2000	1.00	41.2	44.9	109	109
AVERAGE	1.00	42.3	62.6	164	287

Table 2. Normalized simulation speed for the MAI, STALL and FUNC modes.

The STC can be used to further improve the simulation speed according to the description in Section 6. Turning on the STC is most efficient when the cache miss ratio is very low. In this case the STC will remove a large amount of unnecessary cache accesses. The drawback of turning the STC on is that we have to remove certain physical addresses from the STC. This takes time. For the simulated

configuration it turns out that it is most efficient to only enable the STC on instruction cache accesses. Turning on the STC for data accesses actually decreased the performance for several of the studied applications.

Table 2 shows the relative performance for the FUNC and STALL modes with the instruction STC (iSTC) turned on. The STALL mode with the iSTC turned on, is on average 63 times faster than the MAI mode and the FUNC mode with the iSTC is on average 287 times faster than the MAI mode. The iSTC simulations are therefore about 48 percent faster in the STALL mode and 75 percent faster in the FUNC mode than the normal simulation on average for all applications. However, for APACHE which has a high instruction cache miss ratio [13], we actually get a slowdown in STALL mode. No results are presented for the MAI mode. In this mode the differences are very small, since almost all time is spent in simulating the processors rather than simulating the cache hierarchy.

7.5 Results in Different Modes

The simulations were performed with the case study system configuration in the three simulation modes. We set up Vasa to vary the access time to memory in order to get more statistically correct results. The simulated execution time can differ significally even if very small memory latency differences are used [14].

Table 3 shows the median cache miss ratios, data traffic, coherence traffic and execution time for the 64 Byte coherence unit configuration for each application. The L1d cache miss ratio is for many applications notably higher for the FUNC mode than it is for the MAI and the STALL modes. The reason for this is that each instruction, including memory operations, only takes a single cycle to execute in FUNC mode. The entire machine is updated instantly. This leads to an overestimate in the cache miss ratio for applications with lock contention. Critical sections typically takes shorter time to execute in FUNC mode than in other Vasa modes, which leads to fewer cache hits from threads spinning on contended locks.

Simics makes it possible to chose the number of instructions that is executed on a processor before switching to the next processor. This parameter is called the cpu-switch-time and is set to one in all configurations used in this paper. We tried to increase this parameter and found the cache miss ratio to decrease, especially in applications with many locks and barriers such as LU_C and OCEAN_C. This can be explained by more lock spinning caused by longer periods from the time a lock becomes available to when it becomes observable by other processors.

The relative differences in cache miss ratios between the simulation modes decrease the further away we come from the processor. The cache miss ratios in the third level cache only differ with a few percent between the modes.

The data traffic is somewhat underestimated in the STALL and FUNC modes compared to the MAI mode. The STALL and FUNC modes estimate the data traffic to be 8 respectively 16 percent less than the MAI mode data traffic on average. The STALL (FUNC) mode coherence traffic is 5 (13) percent less than that of the MAI mode on average. The reason for this is probably that the MAI mode's processor model lets speculative memory references from wrong path execution pass out to the memory system.

It is not really relevant to compare the absolute cycle count in the MAI and STALL modes since the STALL mode assumes no pipelining and that the processor issues a single instruction each cycle except when the memory system causes it to stall. The executed number of cycles is higher for the STALL mode than the FUNC mode since the FUNC mode assumes no latency at cache accesses.

The comparison presented in Table 3 is only carried out using the 64 byte coherence unit configuration. It does not show how the performance changes when we make an architectural change to the simulated system. The ability to study performance effects of architectural modifications is very important for computer architects since it it necessary for making correct design decisions. We therefore continue the experiments by comparing the studied applications when the coherence size in the entire system is varied between 64, 128 and 256 byte. A large coherence size normally leads to better performance as long as false sharing is limited [9].

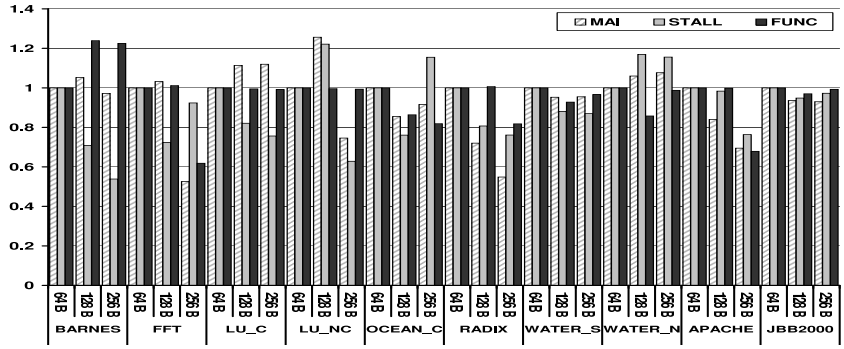
In Figures 2 and 3, the cache miss ratios for all caches and the execution time for all applications are shown. All results are normalized to 1.0 relative to the 64 byte configuration of each application in the corresponding execution mode. The FUNC mode is omitted since an instruction takes one cycle to execute regardless of cache line size, and hence, the number of cycles is exactly the same for all configurations.

The figures show that the trends in cache miss ratios and execution time are similar for almost all applications in the three simulation modes. The only time a less detailed simulation mode yields a significantly incorrect trend is for L1d cache miss ratio in some of the applications, e.g, BARNES, FFT, LU_C and OCEAN_C. In all other cases, it is possible to pick out the best candidate also in faster simulation modes.

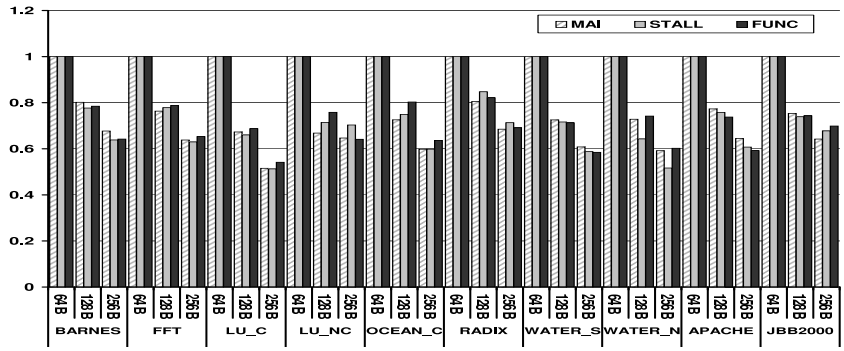
The results also show that the differences in the results between the simulation modes are smaller closer to the memory. These results indicate that the most detailed simulation mode only has to be used when the processor itself or the first level cache behavior is studied. Note that more aggressive processors than the out-of-order model described here, may have an increased effect in the higher levels of the memory hierarchy.

8 Related Work

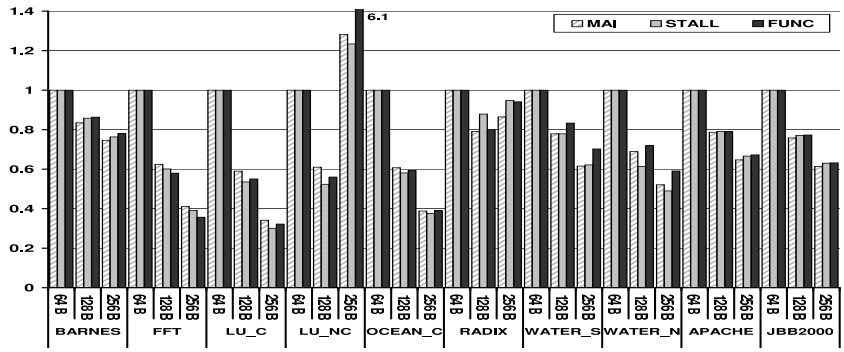
Many different simulation strategies have been presented both in terms of simulation techniques and finding relevant benchmark applications and working sets. The trend towards multithreaded computers, with SMT/CMP-



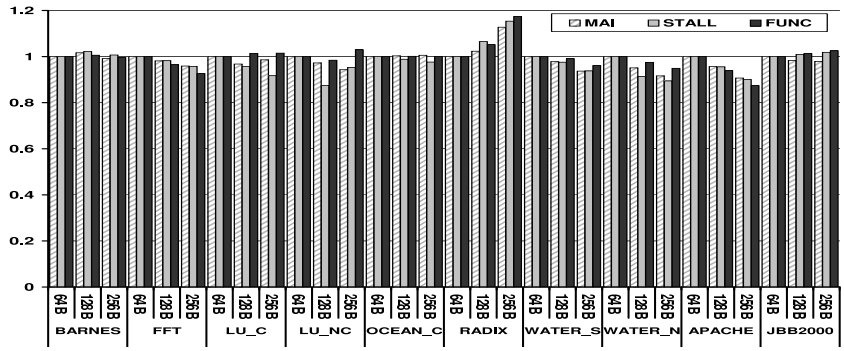
(a) Normalized L1d cache miss ratio



(b) Normalized L1i cache miss ratio



(c) Normalized L2 cache miss ratio



(d) Normalized L3 cache miss ratio

Figure 2. Normalized cache miss ratios for all applications when scaling coherence unit.

		Cache miss ratios (percent)				Data traffic	Coherence traffic	Execution time
		L1d	L1i	L2	L3	(MB)	(MB)	(Mcycles)
BARNES	MAI	20.0	0.00264	0.186	46.1	41.5	12.8	408
	STALL	10.5	0.00270	0.232	46.2	41.0	13.1	297
	FUNC	32.2	0.00244	0.161	46.1	42.8	14.0	186
FFT	MAI	11.1	0.0273	1.22	70.0	8.00	1.74	21.1
	STALL	16.4	0.0185	0.913	69.3	6.98	1.60	23.0
	FUNC	9.39	0.00974	1.37	67.8	6.56	1.48	22.4
LU_C	MAI	1.37	0.00158	0.554	53.3	12.1	2.98	97.1
	STALL	0.617	0.00127	0.515	58.9	9.11	2.30	56.7
	FUNC	34.6	0.00139	0.256	45.2	11.2	2.61	36.5
LU_NC	MAI	1.51	0.00224	0.712	53.0	16.8	4.14	119
	STALL	2.70	0.00157	0.551	52.4	12.6	2.84	84.2
	FUNC	22.0	0.00156	0.276	47.4	12.9	2.72	59.0
OCEAN_C	MAI	2.18	0.0133	3.78	52.4	81.0	19.2	211
	STALL	3.47	0.0102	3.40	53.8	77.5	18.8	206
	FUNC	15.4	0.0312	7.18	50.6	81.0	21.5	30.9
RADIX	MAI	6.89	0.0228	0.968	70.3	9.09	2.13	29.8
	STALL	10.2	0.0138	0.800	69.4	8.61	2.17	32.6
	FUNC	30.4	0.0832	2.35	69.4	7.60	1.86	3.06
WATER_S	MAI	6.47	0.00481	0.108	64.8	3.96	1.24	75.0
	STALL	8.26	0.00310	0.0980	64.4	3.45	1.18	71.3
	FUNC	16.5	0.00320	0.0729	58.9	3.37	1.24	40.2
WATER_N	MAI	5.63	0.00397	0.209	66.0	8.56	2.42	96.2
	STALL	7.19	0.00266	0.212	67.5	8.77	2.57	91.0
	FUNC	12.3	0.00258	0.130	56.4	6.65	1.98	64.2
APACHE	MAI	39.3	0.219	12.5	64.4	151	26.6	138
	STALL	41.6	1.22	10.2	66.0	138	24.5	123
	FUNC	43.9	2.10	10.4	63.3	47.4	8.79	4.37
JBB2000	MAI	16.0	0.169	8.51	63.0	36.7	4.31	53.2
	STALL	17.7	0.457	7.81	58.4	42.1	4.91	48.6
	FUNC	19.0	0.439	7.52	57.6	40.6	4.72	12.3

Table 3. Cache miss ratios, data traffic, coherence traffic and execution time for all applications (64 byte coherence unit).

processors and several nodes makes it important to model multiprocessor systems. Simulators with this ability include Simics [3], Talisman [15], PharmSim [16], ASIM [17], TFSim [18], SimOS [19] and RSIM [20].

Some simulators focus on user-level execution of programs without including the effect of the operating system [17, 20, 21, 22, 23, 24]. Other simulators, model operating systems but require modifications to the operating system to be able to run user applications [15, 16, 19]. The SimFlex project [26] at Carnegie Mellon University includes the simulation package Flexus which is another simulator project that uses the Simics Micro Architectural Interface. It is currently capable of modeling CMPs but not SMT processors.

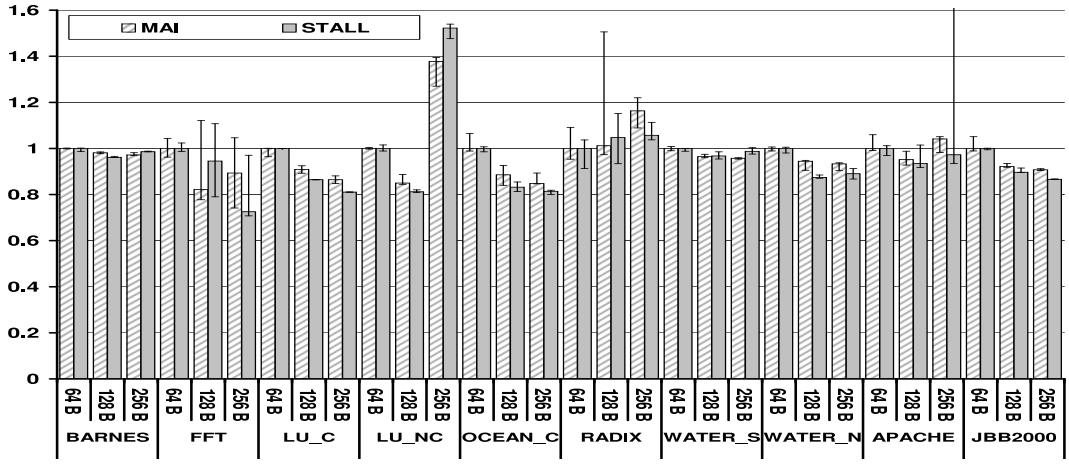
A useful feature in several simulators is the ability to run the simulations with different levels of accuracy and simulation speed [19, 21, 24, 25]. For example, in several of these simulators, a modeled out-of-order processor can be replaced by an in-order processor for improved simulation speed. Other simulation approaches include SimPoint [27] and SMARTS [28] which main contributions are to find statistically more relevant working sets.

9 Conclusion

We have presented Vasa a highly configurable simulation framework. Vasa enables detailed simulation of multiprocessor systems by providing models of modern out-of-order processors, caches, store buffers, interconnects and memory controllers.

Vasa is less complex than many other simulators because it relies on the concepts of downstream timing and instant coherence. Timing is only modeled on the way from the processors to the memory and state changes take place instantly throughout the entire system.

Vasa includes three different fidelity modes, where the level of detail can be traded for simulation speed. We find that the most detailed mode of simulation, which includes a fully pipelined out-of-order processor, in many cases can be avoided. An exception to this is if one specif-



(a) Normalized Execution time

Figure 3. Normalized execution time for all applications when scaling coherence unit.

ically wants to study processor design options or first level cache designs. For architectural studies targeting memory system components further away from the processor, our results show that a less detailed simulation mode yields accurate results. The less detailed modes can also be more useful since the speed increase enables longer simulations.

10 Acknowledgement

We would like to thank the Multifacet group at University of Wisconsin for providing us with the commercial workloads used in this article.

This work is supported in part by Sun Microsystems, Inc., the Parallel and Scientific Computing Institute (PSCI), as well as the PAMP research program, supported by the Swedish Foundation for Strategic Research.

References

[1] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, & D. M. Hawkins, Characterizing and comparing prevaling simulation techniques, *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, Washington, DC, USA, 2005, 266-277.

[2] E. Berg & E. Hagersten, StatCache: A probabilistic approach to efficient and accurate data locality analysis, *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, Austin, TX, USA, 2004, 20-27.

[3] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, & B. Werner, Simics: A full system simulation platform, *IEEE Computer, 35*(2), 2002, 50-58.

[4] M, Karlsson, K. Moore, E. Hagersten, & D. A. Wood, Memory system behavior of Java-based middleware, *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, Anaheim, CA, USA, 2003, 217-228.

[5] *Simics Micro-Architectural Interface*, Reference manual, Virtutech Inc., ver 2.2.10, 2005.

[6] M. Ekman, *Strategies to Reduce Energy and Resources in Chip Multiprocessors*, (Gothenburg, Sweden: Chalmers Institute of Technology, 2004).

[7] J. M. Tendler, J. S. Dodson, J. S. Fields Jr., H. Le, & B. Sinharoy, POWER4 system microarchitecture, *IBM Journal of Research and Development, 46*(1), 2002, 5-25.

[8] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, & R. L. Stamm, Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor, *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, Philadelphia, PA, USA, 1996, 191-202.

[9] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, & A. Gupta, The SPLASH-2 programs: Characterization and methodological considerations, *Proceedings of the 22nd International Symposium on Computer Architecture*, S. Margherita Ligure, Italy, 1995, 24-36.

[10] http://www.spec.org/osg/jbb2000/.

[11] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, X. Min, M. D. Hill, D. A. Wood, & D. J. Sorin, Simulating a $2M commercial server on a $2K PC, *IEEE Computer 36*(2), 2003, 50-57.

[12] J. Nilsson, P. Nandula, & A. Landin, *SUMO - A Generalized Memory Hierarchy Simulator*, Reference manual, Sun Microsystems Inc.

[13] M, Karlsson, K. Moore, E. Hagersten, & D. A. Wood, Exploring Processor Design Options for Java-Based Middleware, *Proceedings of the 34th International Conference on Parallel Processing*, Oslo, Norway, 2005, 59-68.

[14] A. R. Alameldeen & D. A. Wood, Variability in architectural simulations of multi-threaded workloads, *Proceedings of the 30th International Symposium on High Performance Computer Architecture*, San Diego, CA, USA, 2003, 7-18.

[15] R. C. Bedichek, Talisman: Fast and accurate multicomputer simulation, *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, Ottawa, Ontario, Canada, 1995, 14-24.

[16] H. W. Cain, K. M. Lepak, B. A. Schwartz, & M. H. Lipasti, Precise and accurate processor simulation, *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, Cambridge, MA, USA, 2002, 13-22.

[17] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. K. Luk, S Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, & T. Juan, Asim: A performance model framework, *IEEE Computer 35*(2), 2002, 68-76.

[18] C. J. Mauer, M. D. Hill, & D. A. Wood, Full-system timing-first simulation, *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Marina Del Rey, CA, USA, 2002, 108-116.

[19] M. Rosenblum, S. A. Herrod, E. Witchel, & A. Gupta, Complete computer system simulation: The SimOS approach, *IEEE Parallel Distributed Technology 3*(4), 1995, 34-43.

[20] L. Schaelicke & M. Parker, *ML-RSIM Reference Manual*, Technical report 02-10, Dept. of Computer Science and Engineering, Univ. of Notre Dame, 2002.

[21] T. Austin, E. Larson, & D. Ernst, SimpleScalar: an infrastructure for computer system modeling, *IEEE Computer 35*(2), 2002, 59-67.

[22] S. E. Breach, *Design and Evaluation of a Multiscalar Processor*, (Madison, WI, USA: Univ. of Wisconsin, 1998).

[23] E. Larson, S. Chatterjee, & T. Austin, MASE: A novel infrastructure for detailed microarchitectural modeling, *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, Tuscon, AZ, USA, 2001, 1-9.

[24] E. Schnarr & J. R. Larus, Fast out-of-order processor simulation using memoization, *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, 1998, 283-294.

[25] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill & D. A Wood, Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, Submitted to *Computer Architecture News*, 2005.

[26] http://www.ece.cmu.edu/ simflex/.

[27] G. Hamerly, E. Perelman, & B. Calder, How to use SimPoint to pick simulation points, *SIGMETRICS Performance Evaluation Review 31*(4), 2004, 25-30.

[28] E. E. Wunderlich, T. F. Wenisch, B. Falsafi, & J. C. Hoe, SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling, *Proceedings of the 30th Annual International Symposium on Computer architecture*, San Diego, CA, USA, 2003, 84-97.