Chip Multiprocessor Design Space Exploration through Statistical Simulation

Davy Genbrugge and Lieven Eeckhout, *Member*, *IEEE*

Abstract—Developing fast chip multiprocessor simulation techniques is a challenging problem. Solving this problem is especially valuable for design space exploration purposes during the early stages of the design cycle where a large number of design points need to be evaluated quickly. This paper studies statistical simulation as a fast simulation technique for chip multiprocessor (CMP) design space exploration. The idea of statistical simulation is to measure a number of program execution characteristics from a real program execution through profiling, to generate a synthetic trace from it, and simulate that synthetic trace as a proxy for the original program. The important benefit is that the synthetic trace is much shorter compared to a real program trace, which leads to substantial simulation speedups. This paper enhances state-of-the-art statistical simulation: 1) by modeling the memory address stream behavior in a more microarchitecture-independent way and 2) by modeling a program's time-varying execution behavior. These two enhancements enable accurately modeling resource conflicts in shared resources as observed in the memory hierarchy of contemporary chip multiprocessors when multiple programs are coexecuting on the CMP. Our experimental evaluation using the SPEC CPU benchmarks demonstrates average prediction error of 7.3 percent across a range of CMP configurations while varying the number of cores and memory hierarchy configurations.

Index Terms—Performance of systems (modeling techniques, simulation).

1 Introduction

Architectural simulation is a crucial tool in a computer designer's toolbox because of its flexibility, its ease of use, and its ability to drive design decisions early in the design cycle. The downside, however, is that architectural simulation is very time-consuming. Simulating an industry-standard benchmark for a single microprocessor design point easily takes a couple of weeks to run to completion, even on today's fastest machines and simulators. Culling a large design space through architectural simulation of complete benchmark executions thus simply is infeasible. And this problem keeps on increasing over time given Moore's law which projects that the number of cores in chip multiprocessors, also called multicore processors, will double with every new generation. Given the current era of chip multiprocessors, there is a big quest for fast simulation techniques for driving the design process of chip multiprocessors.

Researchers and computer designers are well aware of the multicore simulation problem and have been proposing various methods for coping with it, such as sampled simulation [1], [8], [27], [29], parallelized simulation, and/or hardware-accelerated simulation using FPGAs [5], [22], [30], or analytical modeling [10], [16], [26]. In this paper, we take a different approach through statistical simulation. The idea of statistical simulation is to first measure a statistical profile of a program execution through (specialized) functional simulation or profiling; a statistical profile collects a number of program execution characteristics, such as instruction mix, interinstruction dependency distributions, statistics concerning control flow behavior, branch behavior, and data memory access patterns. These statistics are then used to build a synthetic trace; this synthetic trace exhibits the same execution characteristics as the original program trace, by construction, but is much shorter than the original program trace. Simulating this synthetic trace then yields a performance estimate. Given its short length (on the order of a couple millions of instructions), simulating a synthetic trace is done very quickly.

Previous work has been exploring the statistical simulation paradigm extensively for uniprocessor simulation [6], [11], [18], [20], and one earlier study [19] and one more recent study [13] applied statistical simulation to multithreaded workloads running on shared memory multiprocessor systems. None of this prior work addresses the modeling of shared resources in chip multiprocessors though. This paper extends the statistical simulation methodology to model shared resources in the memory subsystem of chip multiprocessors such as shared caches, off-chip bandwidth, and main memory. This makes statistical simulation a viable fast simulation technique for quickly exploring chip multiprocessor design spaces. We do not envision statistical simulation as a substitute for detailed simulation though. We rather consider statistical simulation as a useful complement to detailed simulation at the earliest stages of the design cycle: the design space can be culled using statistical simulation, and when a region of interest is identified, detailed but slower simulation can be used to explore the region of interest in greater detail, which will reduce the overall design space exploration time.

• The authors are with the Department of Electronics and Information Systems (ELIS), Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium. E-mail: {dgenbrug, leeckhou}@elis.ugent.be.

Manuscript received 14 Jan. 2009; revised 12 Apr. 2009; accepted 29 Apr. 2009; published online 21 May 2009.
Recommended for acceptance by R. Gupta.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-01-0022.
Digital Object Identifier no. 10.1109/TC.2009.77.

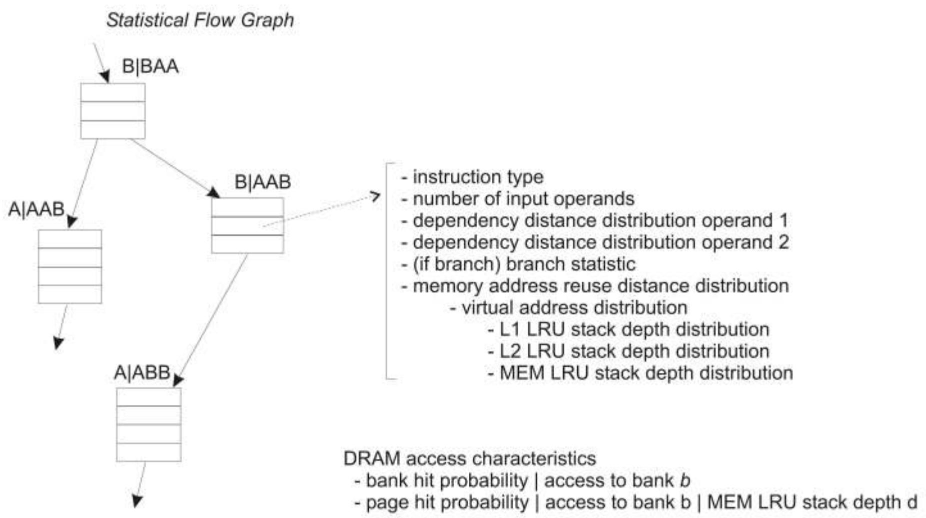
Statistical Flow Graph
B|BAA
A|AAB
B|AAB
A|ABB
- instruction type
- number of input operands
- dependency distance distribution operand 1
- dependency distance distribution operand 2
- (if branch) branch statistic
- memory address reuse distance distribution
- virtual address distribution
- L1 LRU stack depth distribution
- L2 LRU stack depth distribution
- MEM LRU stack depth distribution
DRAM access characteristics
- bank hit probability | access to bank b
- page hit probability | access to bank b | MEM LRU stack depth d

Fig. 1. Illustration of the statistical profile.

This paper makes the following contributions:

- We extend the statistical simulation methodology to chip multiprocessors running multiprogram workloads. To enable the accurate modeling of shared resources in a chip multiprocessor's memory hierarchy, we collect statistics to model memory address patterns such as reuse distances, and LRU stack distance probabilities instead of cache miss rate probabilities as done in prior work. Microarchitecture-independent memory address stream modeling enables modeling conflict behavior by coexecuting synthetic traces on the chip multiprocessor.
- We show that in order to accurately model conflict behavior in shared memory hierarchies, it is important to accurately model time-varying program execution behavior. To this end, we collect a statistical profile and generate a synthetic mini-trace per instruction interval, and then, subsequently coalesce these mini-traces to form the overall synthetic trace.
- The memory address modeling is done in a microarchitecture-independent way. A single statistical profile for the largest cache of interest during the design space exploration can now be used to explore various cache configurations with varying degrees of associativity and number of sets, whereas previous work requires a statistical profile for each cache configuration of interest.
- We demonstrate that the overall framework presented in this paper is accurate and efficient for quickly exploring the chip multiprocessor design space: the performance prediction error is less than 7.3 percent, on average, while achieving a one-order magnitude simulation speedup compared to detailed simulation.

This paper is organized as follows: Section 2 describes the statistical simulation methodology for chip multiprocessors. After detailing the experimental setup in Section 3, we then evaluate its accuracy, speed, and use for CMP design space exploration in Section 4. Finally, we describe related work in Section 5, and conclude and discuss future research directions in Section 6.

2 Statistical CMP Simulation

Statistical simulation is done in three steps: statistical profiling, synthetic trace generation, and synthetic trace simulation. In the following sections, we discuss all three steps in great detail.

2.1 Statistical Profiling

Statistical profiling collects a number of program execution characteristics in a statistical way. This can be done efficiently through specialized functional simulation or through profiling, e.g., using (dynamic) binary instrumentation tools. Fig. 1 illustrates what a statistical profile looks like; we will now discuss each component in more detail.

2.1.1 Statistical Flow Graph

The key structure in the statistical profile is the *statistical flow graph (SFG)* [6] which represents a program's control flow behavior in a statistical manner. In an SFG, the nodes are the basic blocks along with their basic block history, i.e., the basic blocks being executed prior to the given basic block. The order of the SFG is defined as the length of the basic block history, i.e., the number of predecessors to a basic block in each node in the SFG—in this paper, we consider third order SFGs. For example, consider the following dynamic basic block sequence "ABBAABAABBA." The third order SFG then makes a distinction between basic block "A" given its basic block history "ABB," "BBA," "AAB," and "ABA;" this SFG will thus contain the following nodes: "A|ABB," "A|BBA," "A|AAB," and "A|ABA." The edges in the SFG interconnecting the nodes represent transition probabilities between the nodes. Fig. 1 gives an example third order SFG with four nodes "B|BAA," its successors "A|AAB" and "B|AAB," and "A|ABB" which is the successor of "B|AAB."

The idea behind the SFG now is to model all the other program characteristics along the nodes of the SFG. This allows for modeling program characteristics correlated with (or conditionally dependent on) execution path behavior. This means that for a given basic block, different statistics are computed for different basic block histories, i.e., we collect different statistics for basic block "A" given its history "AAB" and "ABB."

2.1.2 Instructions and Their Dependencies

For each instruction in each basic block in the SFG, we record its *instruction type*. We make a distinction between loads, stores, conditional branches, indirect branches, integer ALU operations, integer multiply operations, integer divide operations, floating-point ALU operations, floating-point multiply operations, etc. This distinction is made based on the instruction's semantics and execution latencies. For each instruction, we also record the number of *input registers* or operands.

For each input register, we also compute a distribution of the *dependency distance*. The dependency distance is defined as the number of dynamically executed instructions between the production of a register value (register write) and its consumption (register read). We consider only read-after-write (RAW) dependencies since our focus is on out-of-order architectures in which write-after-write (WAW) and write-after-read (WAR) dependencies are dynamically removed through register renaming as long as enough physical registers are available. We also collect an RAW memory dependency distribution; this is to model store-to-load dependencies. Although very large dependency distances can occur in real program traces, we can limit these register and memory dependency distributions for our purposes to the maximum reorder buffer size of interest. In our study, we limit the dependency distribution to 512.

2.1.3 Branch Characteristics

For each branch in the SFG, we compute the probability for the branch: 1) to be taken; 2) to be fetch redirected (target misprediction in conjunction with a correct taken/not-taken prediction for conditional branches); and 3) to be mispredicted. These branch characteristics are specific to a particular branch predictor.

2.1.4 Memory Address Stream Characteristics

Prior work in statistical simulation models the memory address stream through cache miss statistics, i.e., the statistical profile captures the cache miss rates of the various levels in the cache hierarchy. Although this is sufficient for the statistical simulation of single-core processors, it is inadequate for modeling chip multiprocessors with shared resources in the memory hierarchy, such as shared L2 and/or L3 caches, shared off-chip bandwidth, interconnection network, and main memory. Coexecuting programs on a chip multiprocessor affect each other's performance through conflicts in the shared resources, and the level of interaction between coexecuting programs is greatly affected by the microarchitecture—the amount of interaction can be very different from one microarchitecture compared to another. As such, cache miss rates profiled from single-threaded execution are unable to model conflict behaviors in shared chip multiprocessor resources when coexecuting multiple programs. We, therefore, take a different approach in this work: our aim is to model memory access behavior in the synthetic traces in a way that is independent of the memory hierarchy so that conflict behavior among coexecuting programs can be derived during the simulation of the synthetic traces.

Modeling memory address stream locality behavior requires that we model the correlation between individual memory accesses—in our prior work [11], we found that intermemory reference correlation is necessary for accurately modeling memory-level parallelism as well as delayed hits in statistical simulation. In this prior work, we modeled intermemory reference correlation through correlating hit/miss histories. We now instead use the notion of reuse distances which is a cache hierarchy-independent program characteristic. We, therefore, compute a distribution of the *reuse distance* for each memory access in the SFG; we do this for the instruction addresses, as well as for the load's and store's effective addresses. The reuse distance is defined as the number of memory references between two references to the same memory location. (The reuse distance differs from the LRU stack distance in that the LRU stack distance counts unique memory references only, whereas the reuse distance counts all memory references.) We compute the reuse distance distribution as follows: for each dynamic execution of a given instruction, we compute its memory reference reuse distance and update the corresponding entry in the instruction's reuse distance distribution. We measure this distribution conditionally dependent on the reuse distances of the 50 prior memory references—this is to model memory reference locality [11]. The reuse distance distribution thus captures the temporal locality in the memory address stream. The distribution is limited in size and is measured in buckets (of size power of 2) in order to limit the size of the reuse distance distribution that needs to be stored as part of the statistical profile.

We also compute a distribution of *virtual memory addresses* conditionally dependent on the reuse distance, i.e., for each memory access (instruction pointer and load/store address), we keep track of the memory locations that it touches and how frequently it touches each memory location. (Measuring the virtual memory address distribution conditionally dependent on the reuse distance models correlation among memory references.) Conditionally dependent on the virtual memory address, we then compute three additional memory address stream characteristics, namely, the distributions of the *LRU stack depth* for the L1 cache, L2 cache, and main memory. The LRU stack depth for main memory is computed as the number of unique DRAM page accesses since the last access to that same DRAM page, assuming a single-bank DRAM design. (We will consider multibank DRAM configurations later.) Similarly, the LRU stack depths for the L1 and L2 caches are computed as the number of unique cache block references to the same set since the last reference to that same cache block. (Note that throughout the paper, we refer to the shared cache as the L2 cache; extending our framework to model shared L3 caches is straightforward.) For computing the LRU stack depths for the L1 and L2 caches, we assume the largest L1 and L2 caches one may be potentially interested in during design space exploration. The maximum LRU stack depth kept track of during profiling is $(a+1)$ with a being the associativity of the largest cache of interest. Accessing the LRU stack at

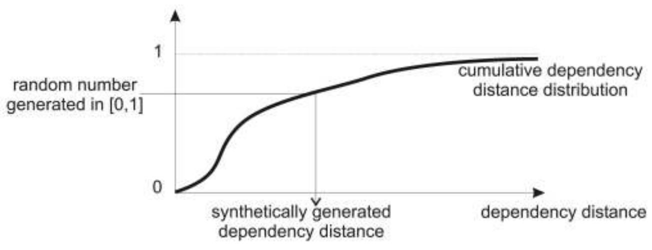

Fig. 2. Illustrating synthetic trace generation using random number generation and the cumulative dependency distance distribution.

depth $(a+1)$ means that a miss occurred in the largest cache of interest. The LRU stack depth profile can be used to estimate cache miss rates for caches that are smaller than the largest cache of interest. In particular, all accesses to an LRU stack depth larger than a will be cache misses in an a-way set-associative cache.

We measure two additional DRAM access characteristics, namely, the *bank hit* and *page hit* statistics. These DRAM access characteristics assume a particular DRAM organization in terms of the number of banks and their organization (interleaved or linear), as well as page size. The bank hit statistics quantify the probability for a bank hit. The page hit statistics quantify the probability for a page hit for each bank.

2.2 Synthetic Trace Generation

The second step in the statistical simulation methodology is to generate a synthetic trace from the statistical profile. The synthetic trace generator takes as input the statistical profile and outputs a synthetic trace that is fed into the statistical simulator. Synthetic trace generation uses random number generation; a random number in the interval $[0, 1]$ is used with the cumulative distribution function to determine the particular value for the program characteristic (see Fig. 2). In particular, synthetic trace generation walks the SFG in a statistical way, i.e., for each node in the SFG, it determines the next node based on the internode transition probabilities. For each node, we output the instructions. For each input operand, we then determine the dependency distance, i.e., we determine on what prior instruction this instruction depends through an RAW dependency. In case of a load, we also determine on what prior store instruction this load depends. In case of a branch, we probabilistically label the branch as a taken branch, fetch-redirected branch, or branch misprediction. For loads and stores as well as for all instruction addresses, we also determine their virtual memory addresses, their LRU stack depths for the L1, and L2 caches as well as main memory, as well as whether they result in a DRAM bank and page hit or miss.

2.3 Synthetic Trace Simulation

Simulating the synthetic trace is done as follows.

2.3.1 Instruction Scheduling and Execution

Instruction scheduling and execution is done in a similar way as conventional architectural simulation. Instructions are scheduled for execution on a functional unit when their dependencies have been cleared, and they are steered toward a specific functional unit based on their instruction type.

2.3.2 Branches

Branches are labeled in the synthetic trace. The label determines whether the branch is taken, fetch redirected, or mispredicted. The label determines the action the statistical simulator should take, similar to conventional architectural simulation. In particular, depending on the aggressiveness of the instruction cache fetch policy, fetch may stop upon a taken branch, or fetch may be redirected. On a branch misprediction, synthetic instructions are fed into the pipeline as if they were from the correct path. When the branch is resolved, the pipeline is squashed and refilled with synthetic instructions from the correct path.

2.3.3 I-Cache Misses

In case of an L1 I-cache miss, the fetch engine stops fetching for a number of cycles equal to the L2 access latency. L2 I-cache and I-TLB misses are handled similarly.

2.3.4 Virtual Address to Physical Address Translation

The virtual addresses that appear in the synthetic traces need to be translated into physical addresses during statistical trace simulation in order to accurately model conflict behavior in physically indexed caches and main memory—the L2/L3 caches are typically physically indexed, whereas the L1 is often virtually indexed to speed up the L1 access time. A naive solution would simply employ the first-come-first-served strategy in statistical simulation as done under detailed simulation, i.e., the next available physical memory page is allocated when a new virtual address page is touched (bump pointer allocation). This, however, leads to inaccurate modeling. The reason is that the synthetic trace is a miniature version of the original program trace and does not touch all memory pages as does the real program trace—this is exactly where the simulation speedup comes from through statistical simulation—and therefore, the virtual to physical address mapping is very different for the synthetic trace than for the original trace. This changes the conflict behavior in the memory hierarchy during statistical simulation compared to detailed simulation, yielding very different performance pictures. To solve this problem, we propose a simple but effective strategy, as illustrated in Fig. 3. Say that the last virtual memory page touched by a program is page x, and the next memory access (for the same program) touches virtual memory page y, see program A in Fig. 3. Then, the virtual to physical address mapper will allocate virtual memory pages $x+1$ up to y in the next available physical memory, i.e., the bump pointer is advanced by $y-x$ memory pages. This assumes that the original program accesses memory pages $x+1$ up to $y-1$ prior to accessing memory page y; we found this simple heuristic to be a reasonable approximation because of spatial locality.

2.3.5 Load and Store Instructions

As mentioned before, we generate a virtual address and L1/L2/DRAM LRU stack depths for each memory reference. Simulating the synthetic trace on a CMP then requires that we effectively simulate the entire memory hierarchy. In statistical simulation for a uniprocessor system, on the other hand, the memory hierarchy does not need to be simulated

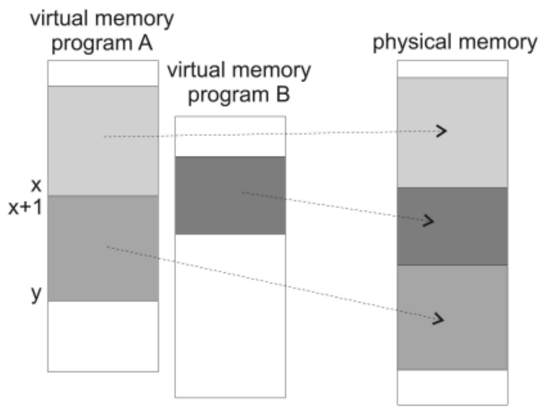

Fig. 3. Illustrating virtual to physical address translation during statistical simulation.

since cache misses are simply flagged as such in the synthetic trace; based on these cache miss flags, appropriate latencies are assigned [6], [11], [18], [20]. Statistical simulation of a CMP with shared memory hierarchy resources, on the other hand, requires that the caches, DRAM, and their interconnections be simulated in order to model conflict behavior.

Each cache line in each cache contains the following information:

- The ID of the program that most recently accessed the cache line; we will refer to this ID as the *program ID*. This enables the statistical simulator to keep track of the program "owning" the cache line.
- The set index of the set in the largest cache of interest that corresponds to the given cache line; we will refer to this set index as the *stored set index*. In case, the cache being simulated has as many sets as the largest cache of interest, the stored set index is the set index of the simulated cache. The stored set index will enable the statistical simulator to model cache lines conflicting for a given set in case the number of sets is reduced for the simulated cache compared to the largest cache of interest.
- A valid bit stating whether the cache line is valid.
- A cold bit stating whether the cache line has been accessed. The cold bit will be used for driving cache warm-up, as will be discussed later.
- In case of a write-back cache, we also maintain a dirty bit stating whether the cache line has been written by a store operation.
- And finally, we also keep track of which instruction in the synthetic trace accessed the given cache line; this is done by storing the position of the instruction in the synthetic trace which we call the *instruction ID*.

Simulating a cache then proceeds as follows assuming that all memory references are annotated with set information s (is obtained from the virtual or physical memory address) and LRU stack depth information d for the largest cache of interest. We first determine the set s' being accessed in the simulated cache; this is done by selecting the $log_2 S$ least significant bits from the set index s with S being the number of sets in the simulated cache. The cache access is considered a cache hit in case there are at least d valid cache lines in set s' for which: 1) the stored set indices equal s and 2) the stored program IDs equal the ID of the program being simulated. In case, the above conditions do not hold, the cache access is considered a cache miss. The most recently accessed cache block is put on top of the LRU stack for the given set.

An appropriate warm-up approach is required for the large caches, such as the unified L2 caches; without appropriate warm-up, the large caches would suffer from a large number of cold misses. Making the synthetic trace longer could solve this problem; however, this would definitely affect the usefulness of statistical simulation which is to provide performance estimates from very fast simulation runs. As such, we take a different approach and use a warm-up approach for warming the L2 cache. The warm-up technique that we use first initializes all cache lines as being cold by setting the cold bit in all cache lines. The warm-up approach then applies a *hit-on-cold* strategy, i.e., upon the first access to a given cache line, we assume that it is a hit and the cold bit is set to zero. In other words, if the cold bit is set, we assume that it is a hit. This hit-on-cold warm-up strategy is simple to implement and is fairly accurate.

During this work, we also found that it is important to model L1 D-cache write-backs during synthetic trace simulation; write-backs can have a significant impact on the conflict behavior in the shared L2 cache. This is done by simulating the L1 D-cache similar to what is described above; L1 D-cache write-backs then access the L2. The L2 cache access is a miss in case all instruction IDs in the given set (with the same program ID) are larger than the instruction ID of the cache line written in the L2; if not, it is a hit.

2.3.6 *Simulation Speed*

The important benefit of statistical simulation is that a synthetic trace is very short, typically, a couple million instructions. The reason for these short synthetic traces is that the performance metrics quickly converge to a steady-state value when simulating a synthetic trace. As such, synthetic traces containing no more than a few million of instructions are sufficient for obtaining stable and accurate performance estimates. We quantify the simulation speedup compared to detailed simulation in the evaluation section.

2.4 Modeling Time-Varying Execution Behavior

A critical issue to the accuracy of statistical simulation for modeling CMP performance is that the synthetic trace has to capture its time-varying execution behavior. The reason is that overall performance is affected by the phase behavior of the coexecuting programs: the relative progress of a program is affected by the conflict behavior in the shared resources. For example, extra cache misses induced by cache sharing may slow down a program's execution. A program running relatively slow because of cache sharing may result in different program phases coexecuting with the other program(s), which, in turn, may result in different cache sharing behavior, and thus, faster or slower relative progress.

To model time-varying behavior, we divide the entire program trace into a number of instruction intervals; an instruction interval is a sequence of consecutive instructions in the dynamic instruction stream. We then collect a statistical profile per instruction interval and generate a synthetic mini-trace. Coalescing these mini-traces yields the overall synthetic trace. The synthetic trace then captures the original trace's time-varying behavior.

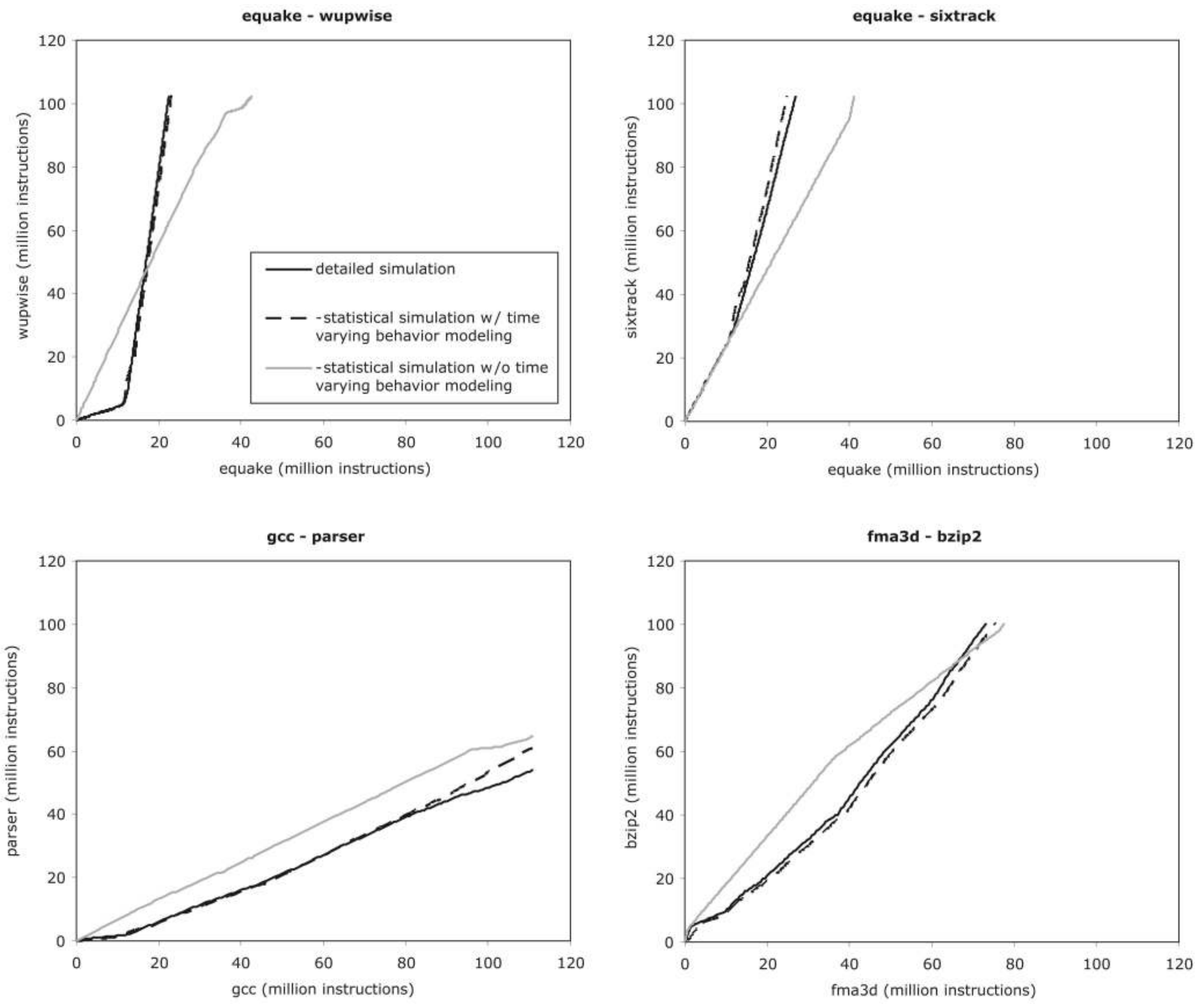

Fig. 4. Relative progress graphs for equake-wupwise, sixtrack-equake, gcc-parser, and fma3d-bzip2.

The importance of modeling a program's time-varying behavior is illustrated in Fig. 4. The four graphs show *relative progress graphs* when coexecuting two programs on a multicore processor: equake-wupwise, sixtrack-equake, gcc-parser, and fma3d-bzip2. A point (x, y) on a relative progress curve denotes that the first program has executed x instructions and the second program has executed y instructions. In other words, a slow slope denotes that the first program makes fast relative progress compared to the second program; a steep slope denotes that the first program makes slow relative progress compared to the second program. All graphs in Fig. 4 demonstrate the importance of modeling a program's time-varying behavior. Without time-varying behavior modeling, statistical simulation is unable to track relative progress rates, which leads to inaccurate multicore processor performance predictions (see Fig. 5). The reason for this inaccuracy is that very different phases are coexecuted under statistical simulation compared to detailed simulation. If a program's time-varying execution behavior is modeled, on the other hand, statistical simulation is capable of accurately tracking relative progress rates, which yields substantially more accurate multicore performance predictions. The important insight here is that modeling the time-varying behavior in statistical simulation does not attribute to the accuracy for single-core processor performance estimation, but it does have a substantial impact on the accuracy when coexecuting programs on a multicore processor.

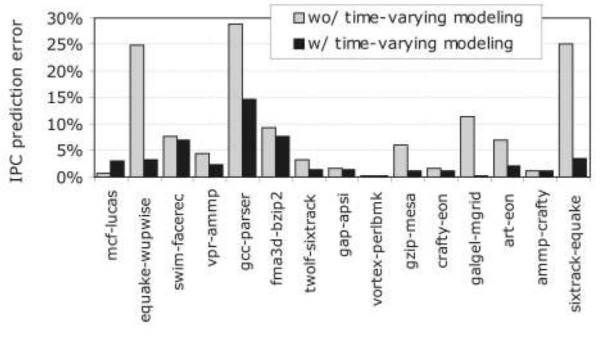

Fig. 5. Prediction error through statistical simulation with and without modeling a program's time-varying behavior.

TABLE 1
Example Microarchitectural Parameters That Do or Do Not Require That a New Statistical Profile Is Computed

a new statistical profile is needed	branch predictor (type and size) cache hierarchy (number of cache levels and cache line size) DRAM configurations (number of banks, interleaved vs. linear, page size)
no new statistical profile is needed	processor width (fetch, decode, dispatch, issue and commit width) pipeline depth ROB size LSQ size fetch buffer size number and types of the functional units instruction execution latencies memory hierarchy (L1, L2 and DRAM) access latencies cache size and associativity off-chip memory bandwidth

2.5 Design Space Exploration Using Statistical Simulation

The statistical profile contains a collection of microarchitecture-dependent and microarchitecture-independent characteristics. This has an important implication in practice: the use of a single statistical profile is limited by the set of microarchitecture-dependent characteristics. For example, given that the branch statistics are specific to one particular branch predictor configuration, a new statistical profile needs to be computed in case the branch predictor is changed. However, a single statistical profile can be used to explore a large range of microarchitecture parameters such as the number of cores, the reorder buffer size, issue width, pipeline depth, etc., because there is no statistic in the statistical profile that is tied to any of these microarchitecture parameters, i.e., the relevant program characteristics are microarchitecture-independent. This is an important property because it implies that a very large fraction of the design space can be explored using a single statistical profile. Table 1 summarizes which microarchitectural parameters can or cannot be changed during design space exploration without the need for recomputing the statistical profile. An important improvement over prior work in statistical simulation [6], [11], [13], [18], [19], [20] is that the cache statistics are largely microarchitecture-independent. As such, we can explore most of the memory hierarchy design space from a single statistical profile. The only parameter that requires a new statistical profile to be computed is the number of cache levels and their line sizes. The number of cache sets, cache associativity, bandwidth, and latencies can be changed without recollection of the statistical profile.

3 Experimental Setup

We use the SPEC CPU2000 benchmarks with the reference inputs in our experimental setup (see Table 2); this table also displays the global L2 cache miss rates for the various benchmarks in our baseline 16MB 16-way set-associative cache. The binaries of the CPU2000 benchmarks are taken from the SimpleScalar Web site. We consider 100M single (and early) simulation points as determined by SimPoint [23], [24] in all of our experiments. The synthetic traces are 10M instructions long, unless mentioned, otherwise—we evaluate the impact of the synthetic trace length on accuracy and simulation speedup in Section 4.2. For measuring the statistical profiles capturing time-varying behavior, we measure a statistical profile per 10M-instruction interval. From these 10 statistical profiles, we then generate 10 1M-instruction mini-traces that are subsequently coalesced to form the 10M-instruction synthetic traces.

We use the M5 simulator [2] in all of our experiments. Our baseline per-core microarchitecture is a four-wide superscalar out-of-order core (see Table 3). When simulating a CMP, we assume that all cores share the L2 cache as well as the off-chip bandwidth for accessing main memory. Simulation stops as soon as one of the coexecuting programs terminates, i.e., as soon as one of the programs has executed 100M instructions in case of detailed simulation, or 10M instructions in case of statistical simulation. We then record how many instructions were executed so far for each coexecuting program, and we compute single-threaded IPC for executing that many instructions. Having obtained IPC numbers under both multicore execution and single-threaded execution enables computing system throughput

TABLE 2
The SPEC CPU2000 Benchmarks, Their Reference Inputs,
the Single 100M Simulation Points Used in This Paper and Their Global L2 Cache Miss Rates

benchmark	input	simpoint	L2	benchmark	input	simpoint	L2	benchmark	input	simpoint	L2
bzip2	program	9	1.4%	twolf	ref	31	5.3%	fma3d	ref	298	2.2%
crafty	ref	0	0.7%	vortex	ref2	57	0.5%	galgel	ref	3,150	3.2%
eon	rushmeier	18	0.0%	vpr	route	71	2.9%	lucas	ref	35	8.9%
gap	ref	2,094	0.7%	ammp	ref	2,130	4.8%	mesa	ref	89	0.1%
gcc	166	99	0.8%	applu	ref	18	4.6%	mgrid	ref	6	3.3%
gzip	graphic	9	2.3%	apsi	ref	46	2.5%	sixtrack	ref	82	0.2%
mcf	ref	316	24.8%	art	ref-110	67	32.8%	swim	ref	5	7.5%
parser	ref	16	2.3%	equake	ref	194	8.3%	wupwise	ref	584	1.3%
perlbmk	makerand	1	0.2%	facerec	ref	136	2.7%				

TABLE 3
Baseline Processor Core Model Assumed in Our Experimental Setup; Simulated CMP Architectures Share the L2 Cache

Processor core	
ROB	80 entries
LSQ	64 entries
store buffer	32 entries
processor width	decode, dispatch, issue and commit 4 wide fetch 8 wide
latencies	load (2), mul (3), div (20)
L1 I-cache	32KB 4-way set-assoc 64B line size
L1 D-cache	32KB 4-way set-assoc 64B line size, 8 MSHRs, 16-entry write buffer
L2 cache	unified, 16MB 16-way set-assoc 64B line size, 12 cycles access latency, 8 MSHRs, 16-entry write buffer
branch predictor	local 1K-entry predictor, 8-way set-assoc 2K-entry BTB, 32-entry RAS
Bus	
bus frequency	666MHz
bus width	16 bytes
SDRAM	
banks	4
page size	4KB
RAS active time	125 CPU cycles
RAS to CAS delay	25 CPU cycles
CAS latency	5 CPU cycles
RAS precharge time	10 CPU cycles
RAS cycle time	70 CPU cycles

(STP) [9], also called weighted speedup [25], which is defined as

$$STP = \sum_{i=1}^{n} \frac{IPC_{i,multi\ core}}{IPC_{i,single\ threaded}},$$

with n the number of coexecuting programs.

4 Evaluation

We now evaluate the statistical simulation methodology proposed in this paper in three dimensions: 1) accuracy both in terms of a single design point as in terms of exploring a wide design space; 2) simulation speed; and 3) storage requirements for storing the statistical profiles on disk.

4.1 Accuracy

4.1.1 Homogeneous Workloads

The top graph in Fig. 6 evaluates the accuracy of statistical simulation for a single program running on a single-core processor. The average IPC prediction error is 2.4 percent; this is in line with previously reported results by Genbrugge and Eeckhout [11]. The other three graphs in Fig. 6 evaluate the accuracy when running homogeneous multiprogram workloads on a multicore with a shared L2 cache, i.e., multiple copies of the same program are executed simultaneously. The average prediction errors for the two-core, four-core, and eight-core machines are 5.6, 6.3, and 7.3 percent, respectively. Statistical simulation is capable of accurately tracking the impact of the shared L2 cache on overall application performance. For some programs, cache sharing has almost no impact, see, for example, mesa: the IPC for mesa remains unaffected by L2 cache sharing. For other programs, on the other hand, cache sharing has a large impact, see, for example, art, mgrid, and swim. Statistical simulation is accurate enough for identifying which programs are susceptible to L2 cache sharing; moreover, statistical simulation yields an accurate prediction of the extent to which cache sharing affects overall performance.

4.1.2 Heterogeneous Workloads

Fig. 7 evaluates the accuracy of statistical simulation for heterogeneous workloads. The four sets of graphs, Figs. 7a, 7b, 7c, and 7d, represent different sets of workloads. The left column shows results through detailed simulation, and the right column shows results through statistical simulation. In each graph, there are four bars for each benchmark. The "one-core" bars represent per-benchmark IPC when run alone. The "two-core" bars represent per-benchmark IPC when corun with another benchmark; the "four-core" bars represent per-benchmark IPC when corun with three other benchmarks, etc. The corun workloads are determined as such, see, for example, the top-left graph: we corun art with applu, mcf with lucas, etc., on a two-core configuration; for the four-core configuration, we corun art, applu, mcf, and lucas, and corun equake, wupwise, swim, and facerec; for the eight-core configuration, we corun all benchmarks in the workload.

Not surprisingly, per-benchmark IPC decreases with increasing multicore processing. This is due to resource conflicts in the shared memory hierarchy, i.e., the more conflicts, the more the coexecuting programs interact and affect each other's performance. The degree to which coexecuting benchmarks affect each other's performance heavily depends on the benchmarks' characteristics, i.e., the more memory-intensive the benchmarks are, the more they affect each other's performance. For example, the coexecuting benchmarks affect each other's performance very heavily in the (a) workload—these benchmarks are all memory-intensive; on the contrary, the benchmarks in workload (c) barely affect each other's performance because none of the benchmarks are memory-intensive. The important observation from these graphs is that statistical simulation accurately tracks the performance trends observed through detailed simulation.

4.1.3 Design Space Exploration

We now demonstrate the accuracy of statistical simulation for driving design space exploration, which is the ultimate goal of the statistical simulation methodology. To do so, we consider a design space of 80 design points with varying L2 cache configurations and a varying number of cores. We vary the L2 cache size from 128 KB to 16 MB with varying associativity from 2- to 16-way set associative; the cache line size is kept constant at 64 bytes. And we vary the number of cores from 1, 2, 4 up to 8. This design space consisting of 80 design points is very small compared to a realistic design space; however, the reason is that we are validating the accuracy of statistical simulation against detailed simulation. The detailed simulation for all those 80 design points was very much time-consuming, which is the motivation for statistical simulation in the first place.

Fig. 8 shows a scatter plot with system throughput through detailed simulation on the horizontal axis versus system throughput through statistical simulation on the vertical axis. The four graphs in Fig. 8 show four different heterogeneous eight-program mixes. The average system throughput prediction error equals 3.5 percent. We observe that the prediction error increases slightly with an increasing number of cores: an average prediction error of 1.9 percent

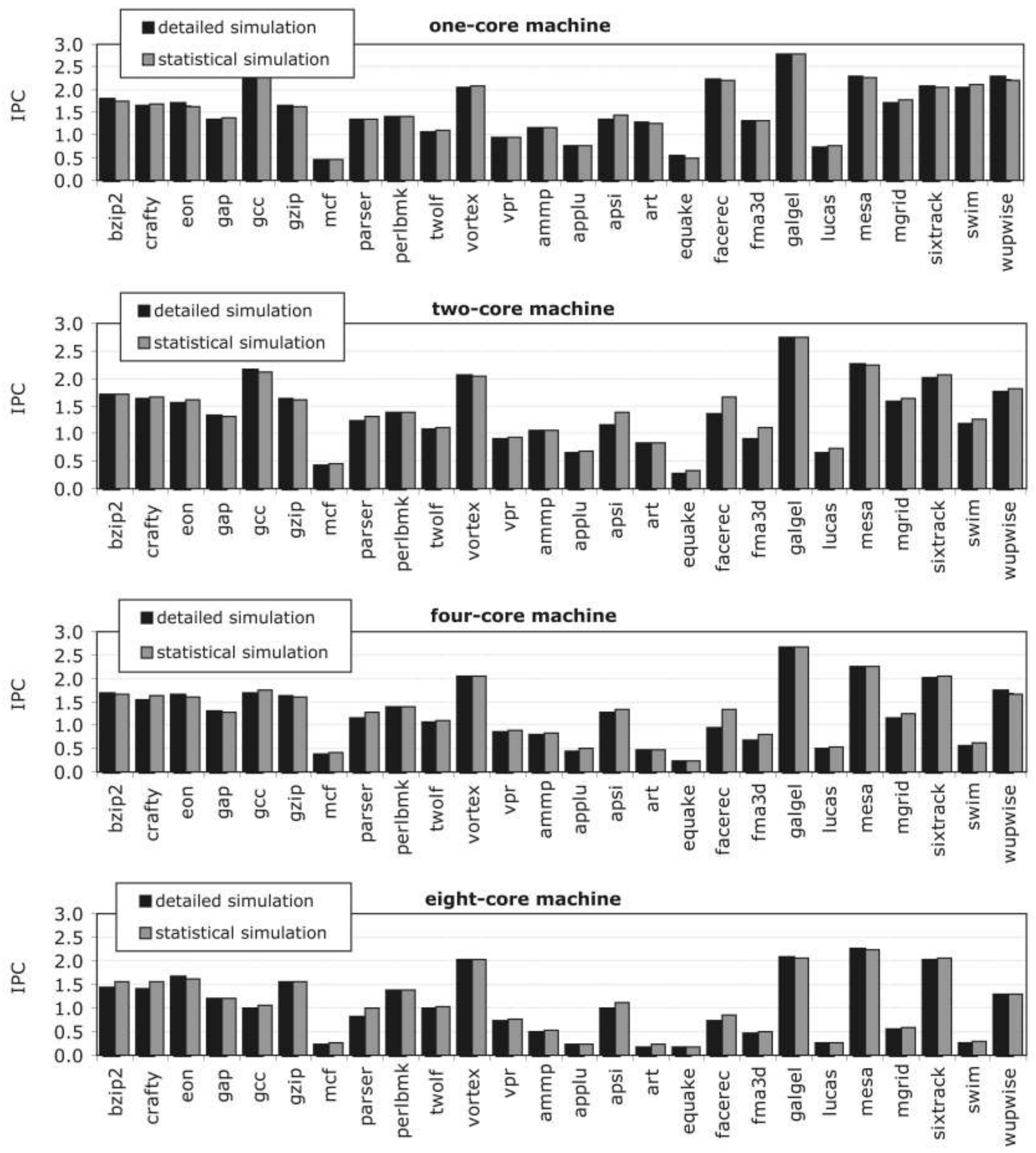

Fig. 6. Evaluating the accuracy of statistical simulation for single-program and homogeneous multiprogram workloads.

for a two-core processor, 3.7 percent for a four-core processor, and 5 percent for an eight-core processor. Overall, we conclude that for all four workload mixes, the system throughput estimates through statistical simulation correlate very closely with the system throughput numbers obtained from detailed simulation.

4.1.4 Cache Design Space Exploration

Fig. 9 illustrates the accuracy of statistical simulation for exploring the shared L2 cache design space. In these graphs, we consider the IPC for a single benchmark, twolf, that we found to be sensitive to both the cache configuration parameters and the amount of parallel processing; we obtained similar results for other benchmarks though, however, less pronounced as for twolf. In these experiments, twolf is run solely on a unicore processor, with sixtrack on the two-core machine, with fma3d-bzip2-sixtrack on the four-core machine, and with vpr-ammp-gcc-parser-fma3d-bzip2-sixtrack on the eight-core machine. Again, the overall conclusion is that statistical simulation accurately tracks performance differences across cache configurations and across a different number of cores. Note that these results were obtained from a single statistical profile, namely, a statistical profile for the largest cache of interest, a 16 MB 16-way set-associative cache. In other words, a single statistical profile is sufficient to drive a cache design space exploration.

4.1.5 3D Stacking Case Study

For demonstrating the value of statistical simulation for exploring new architecture paradigms, we now consider a case study in which we evaluate performance of a multicore processor in combination with 3D stacking [15]. In this case study, we compare the performance of a four-core processor with a 16 MB L2 cache connected to external DRAM memory through a 16-byte wide memory bus against an eight-core processor with integrated on-chip DRAM memory (through 3D stacking) and no L2 cache and a

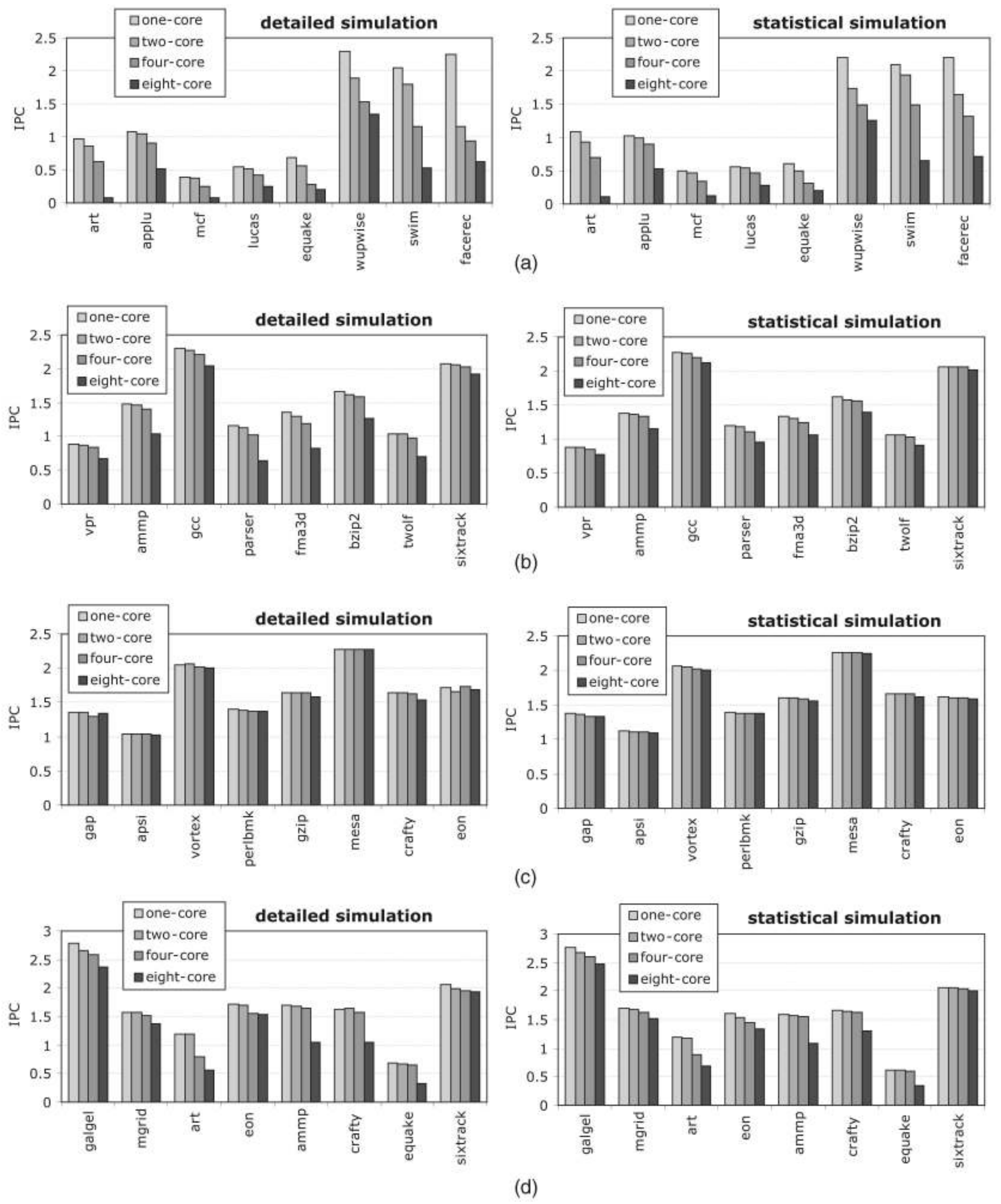

Fig. 7. Evaluating the accuracy of statistical simulation for four heterogeneous workload mixes. The two-core configuration results show per-benchmark IPC when corun with another benchmark, e.g., in the top-left graph, we corun art with applu, mcf with lucas, etc. For the four-core configuration, we corun art, applu, mcf, and lucas, and we corun equake, wupwise, swim, and facerec; for the eight-core configuration, we corun all benchmarks in the workload.

128-byte wide memory bus. We assume a 150-cycle access time for external memory and a 125-cycle access time for 3D-stacked memory. Fig. 10 quantifies system throughput for these two design points for four different eight benchmark mixes. The eight-core processor with 3D-stacked memory achieves substantially higher system throughput than the four-core processor with the on-chip L2 cache. The improvement in system throughput varies across workload mixes, and statistical simulation can accurately track performance differences between both design alternatives: the maximum error in predicting the system throughput delta between the four core with on-chip L2 versus the eight core with 3D-stacked DRAM is 12 percent.

4.2 Simulation Speed

Having shown the accuracy of statistical simulation for CMP design space exploration, we now evaluate the simulation speed. Fig. 11 shows the average IPC prediction error as a function of the synthetic trace length. For a single-program workload, the prediction error stays almost flat, i.e., increasing the size of the synthetic trace beyond 1M instructions does not increase prediction accuracy. For multiprogram workloads, on the other hand, the prediction accuracy is sensitive to the synthetic trace length, and sensitivity increases with the number of programs in the multiprogram workload. This can be understood intuitively: the more programs there are in the multiprogram workloads, the longer it takes before the shared caches are warmed up and

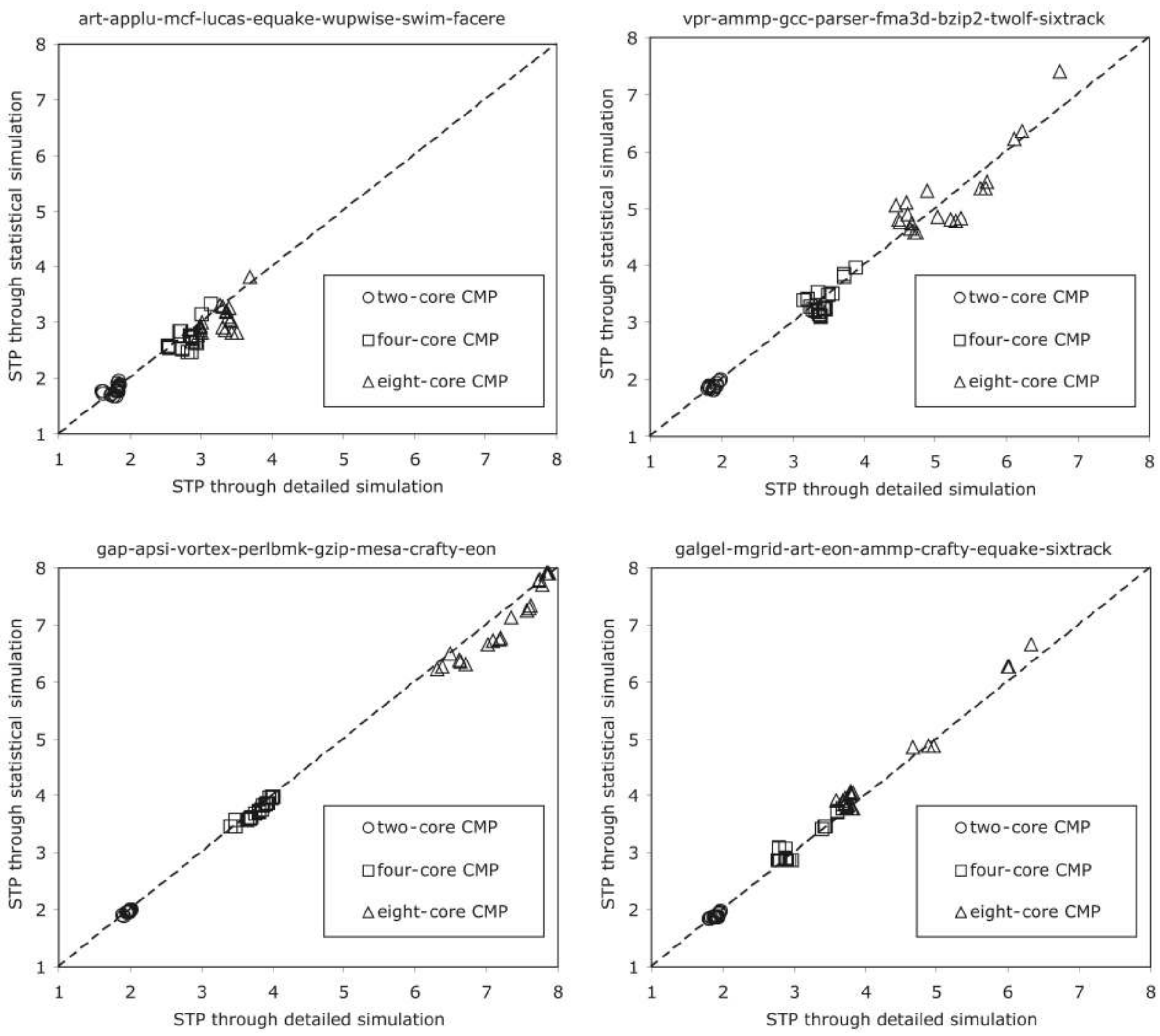

Fig. 8. Evaluating the accuracy of statistical simulation for exploring CMP design spaces: measured system throughput through detailed simulation versus estimated system throughput through statistical simulation. The four graphs represent four eight-program workload mixes.

the longer it takes before the conflict behavior is appropriately modeled between the coexecuting programs. The results in Fig. 11 demonstrate that 10M-instruction synthetic traces yield accurate performance predictions, even for eight-core processors. In our experiments, we, therefore, went from 100M instruction real program traces to 10M instruction synthetic traces. This is a $10\times$ decrease in the dynamic instruction count which yields an approximate $10\times$ reduction in the overall simulation time.

4.3 Storage Requirements

As a final note, the storage requirements are modest for statistical simulation. The statistical profiles when compressed on disk are 87 MB, on average, per benchmark.

5 Related Work

We now discuss related work in statistical modeling and fast multithreaded processor simulation techniques.

5.1 Statistical Modeling

Statistical simulation for modeling uniprocessors has received more and more interest over the last few years. Noonburg and Shen [17] model a program execution as a Markov chain in which the states are determined by the microarchitecture and the transition probabilities by the program. Iyengar et al. [14] use a statistical control flow graph to identify representative trace fragments; these trace fragments are then coalesced to form a reduced program trace. The statistical simulation framework considered in this paper is different in its setup: we generate a synthetic trace based on a statistical profile. The initial models proposed along this line were fairly simple [3], [7]: the entire program characteristics in the statistical profile are typically aggregate metrics, averaged across all instructions in the program execution. Oskin et al. [20] propose the notion of a graph with transition probabilities between the basic blocks while using aggregate statistics. Nussbaum and Smith [18] correlate various program characteristics to the basic block size in order to improve accuracy. Eeckhout et al. [6] propose the SFG which models the control flow in a statistical manner; the various program characteristics are then correlated to the SFG. In our own prior work [11], we further improve the overall accuracy of the statistical simulation framework through accurate memory data flow modeling: we model

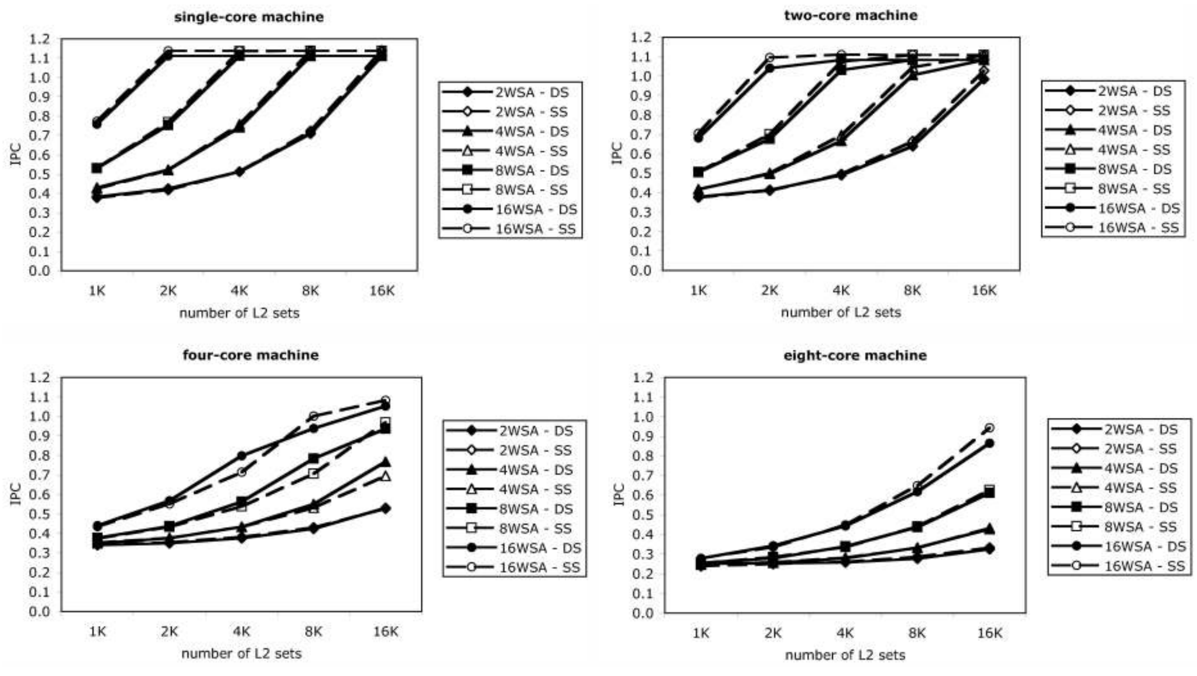

Fig. 9. Evaluating the accuracy of statistical simulation for tracking shared cache performance as a function of the cache configuration (number of sets and associativity) and the number of cores on the CMPs; the example benchmark here is twolf; "DS" denotes detailed simulation, and "SS" denotes statistical simulation.

cache miss correlation, store-load dependencies, and delayed hits, and report an average IPC prediction error of 2.3 percent for a wide superscalar out-of-order processor compared to detailed simulation.

Nussbaum and Smith [19] extend the uniprocessor statistical simulation method to multithreaded programs running on shared-memory multiprocessor (SMP) systems. To do so, they extended statistical simulation to model synchronization and accesses to shared memory. Hughes and Li [13] more recently introduced synchronized statistical flow graphs that incorporate interthread synchronization. Cache behavior is still modeled based on cache miss rates though; by consequence, they are unable to model shared caches as observed in modern CMPs.

Chandra et al. [4] propose performance models to predict the impact of cache sharing on coscheduled programs. The output provided by the performance model is an estimate of the number of extra cache misses for each thread due to cache sharing. These performance models are limited to predicting cache sharing effects and do not predict overall performance. Moreover, the performance models assume that coscheduled programs make fixed progress, i.e., the models ignore the effect that cache sharing may have on how programs affect each other's performance.

5.2 Fast Multithreaded Processor Simulation

The approaches that have been proposed for speeding up multithreaded processor simulation can basically be classified in two main categories: sampled simulation and parallelized simulation, which we discuss below.

Van Biesbrouck et al. [27], [28], [29] propose the cophase matrix for guiding sampled simultaneous multithreading (SMT) processor simulation running multiprogram workloads. The idea of the cophase matrix is to keep track of the

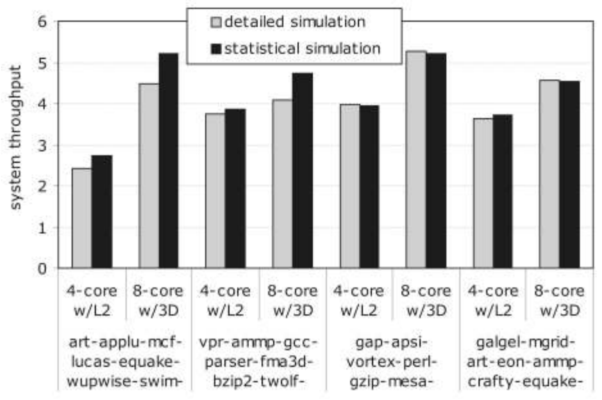

Fig. 10. 3D stacking case study: comparing system throughput for a four-core CMP with L2 cache and external DRAM memory versus an eight-core CMP with on-chip DRAM memory (through 3D stacking) and without an L2 cache.

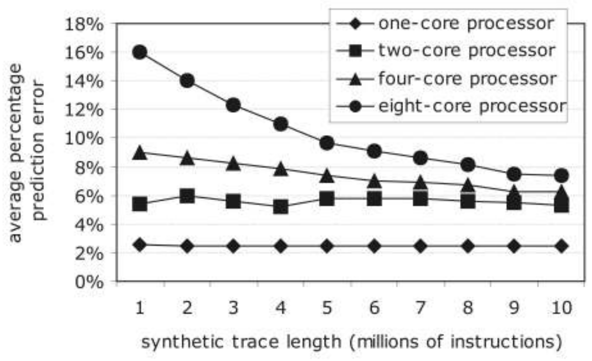

Fig. 11. Percentage average IPC prediction error as a function of synthetic trace length for single-program and multiprogram homogeneous workloads.

relative progress of the programs on a per-phase basis when executed together. By doing so, cophases need to be simulated only once; the performance of recurring cophase executions can then simply be read from the cophase matrix, which speeds up simulation.

Ekman and Stenström [8] use random sampling to speed up multiprocessor simulation. They observe that the variability of the overall system throughput decreases with an increasing number of processors when running multiprogram workloads. This means that fewer random samples need to be taken to estimate overall performance for larger MP systems than for smaller MP systems in case one is interested in aggregate performance only. Wenisch et al. [31] obtained similar conclusions for throughput server workloads.

Barr et al. [1] propose the Memory Timestamp Record (MTR) to store microarchitecture state (cache and directory state) at the beginning of a sample as a checkpoint. This checkpoint can then be used to quickly restore and estimate hardware state at the beginning of each sample for different microarchitecture configurations.

Penry et al. [22] build a structural model of a CMP that enables them to automatically parallelize the simulator. The individual components in the structural CMP model are designed to execute concurrently in hardware and are thus candidates to run in parallel in simulation. Penry et al. also simulate components in hardware using FPGAs. FPGA-based simulation acceleration has received increased attention over the recent years, see, for example, [5], [21], [30].

6 Conclusion and Future Work

Simulating chip multiprocessors is extremely time-consuming. This is especially a concern in the earliest stages of the design cycle where a large number of design points need to be explored quickly. This paper proposed statistical simulation as a fast simulation technique for chip multiprocessors running multiprogram workloads. In order to do so, we extended the statistical simulation paradigm: 1) to collect cache set access and per-set LRU stack depth profiles and 2) to model time-varying behavior in the synthetic traces. These two enhancements enable the accurate modeling of the conflict behavior observed in shared caches. Our experimental results showed that statistical simulation is accurate with average IPC prediction errors of less than 7.3 percent over a broad range of CMP design points, while being one order of magnitude faster than detailed simulation. This makes statistical simulation a viable fast simulation approach to CMP design space exploration.

There are several avenues along which we can take this research for future work. First, we plan on extending the statistical simulation methodology to multithreaded workloads. This paper considered multiprogram workloads only and showed that given the enhancements proposed in this paper, CMP resource sharing can be modeled accurately in statistical simulation for multiprogram workloads. Combining it with the Nussbaum and Smith [19] and Hughes and Li [13] approaches will make statistical simulation viable for modeling multithreaded workloads running on CMPs with shared resources. Second, this paper made a first but important step toward making the statistical profile microarchitecture-independent. The cache statistics are independent of the number of sets in the cache and the cache's associativity. They are dependent on the cache line size though; also, the branch prediction statistics are dependent on a particular branch predictor configuration. Making the statistical profile completely, microarchitecture-independent would make the framework even more efficient and applicable. For example, statistically simulating SMT processors would then be possible to do. Third, we found the accuracy of the shared cache performance estimation through statistical simulation to be subject to warm up in the shared cache. This paper assumed hit-on-cold. In future work, we will consider potentially more accurate and efficient cache warm-up strategies.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable feedback. The original version of this paper was published at the 2007 International Conference on Computer Design (ICCD) [12]. They are also grateful to Jos Delbar who did some preliminary experiments for this paper. Lieven Eeckhout is a postdoctoral fellow with the Fund for Scientific Research-Flanders (Belgium) (FWO Vlaanderen). This research is also partially funded by the FWO projects G.0232.06 and G.0255.08, and the UGent-BOF project 01J14407.

This paper is an extended version of the paper "Statistical Simulation of Chip Multiprocessors Running Multiprogram Workloads" by D. Genbrugge and L. Eeckhout published at the 2007 International Conference on Computer Design. The journal paper extends the conference paper in the following ways:

- The conference paper assumes a simplified DRAM model, i.e., it assumes a fixed memory access latency. This paper models a realistic DRAM configuration in its experimental setup with nonconstant access latencies, and the statistical simulation framework is extended accordingly to accurately model DRAM accesses in a statistical way.
- The conference paper models the L1 cache behavior through cache miss rates. This paper models the memory address stream in a microarchitecture-independent way which enables exploring a large memory hierarchy design space from a single statistical profile.
- This paper provides an extended description of the framework and an extended evaluation on more and larger chip multiprocessor design spaces, including a 3D stacking case study.

References

[1] K.C. Barr, H. Pan, M. Zhang, and K. Asanovic, "Accelerating Multiprocessor Simulation with a Memory Timestamp Record," *Proc. 2005 IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS)*, pp. 66-77, Mar. 2005.

[2] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, no. 4, pp. 52-60, July/Aug. 2006.

[3] R. Carl and J.E. Smith, "Modeling Superscalar Processors via Statistical Simulation," *Proc. Workshop Performance Analysis and Its Impact on Design (PAID), Held in Conjunction with the 25th Ann. Int'l Symp. Computer Architecture (ISCA)*, June 1998.

[4] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip-Multiprocessor Architecture," *Proc. 11th Int'l Symp. High-Performance Computer Architecture (HPCA),* pp. 340-351, Feb. 2005.

[5] D. Chiou, D. Sunwoo, J. Kim, N.A. Patil, W. Reinhart, D.E. Johnson, J. Keefe, and H. Angepat, "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," *Proc. Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO),* pp. 249-261, Dec. 2007.

[6] L. Eeckhout, R.H. Bell Jr., B. Stougie, K. De Bosschere, and L.K. John, "Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies," *Proc. 31st Ann. Int'l Symp. Computer Architecture (ISCA),* pp. 350-361, June 2004.

[7] L. Eeckhout and K. De Bosschere, "Hybrid Analytical-Statistical Modeling for Efficiently Exploring Architecture and Workload Design Spaces," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* pp. 25-34, Sept. 2001.

[8] M. Ekman and P. Stenström, "Enhancing Multiprocessor Architecture Simulation Speed Using Matched-Pair Comparison," *Proc. 2005 IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS),* pp. 89-99, Mar. 2005.

[9] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multi-Program Workloads," *IEEE Micro,* vol. 28, no. 3, pp. 42-53, May/June 2008.

[10] S. Eyerman, L. Eeckhout, T. Karkhanis, and J.E. Smith, "A Mechanistic Performance Model for Superscalar Out-of-Order Processors," *Proc. ACM Trans. Computer Systems (TOCS),* May 2009.

[11] D. Genbrugge and L. Eeckhout, "Memory Data Flow Modeling in Statistical Simulation for the Efficient Exploration of Microprocessor Design Spaces," *IEEE Trans. Computers,* vol. 57, no. 10, pp. 41-54, Jan. 2007.

[12] D. Genbrugge and L. Eeckhout, "Statistical Simulation of Chip Multiprocessors Running Multi-Program Workloads," *Proc. 2007 Int'l Conf. Computer Design (ICCD),* pp. 464-471, Oct. 2007.

[13] C. Hughes and T. Li, "Accelerating Multi-Core Processor Design Space Evaluation Using Automatic Multi-Threaded Workload Synthesis," *Proc. IEEE Int'l Symp. Workload Characterization (IISWC),* pp. 163-172, Sept. 2008.

[14] V.S. Iyengar, L.H. Trevillyan, and P. Bose, "Representative Traces for Processor Models with Infinite Cache," *Proc. Second Int'l Symp. High-Performance Computer Architecture (HPCA),* pp. 62-73, Feb. 1996.

[15] T. Kgil, S. D'Souza, A. Saidi, B. Nathan, R. Dreslinski, S. Reinhardt, K. Flautner, and T. Mudge, "PicoServer: Using 3D Stacking Technology to Enable a Compact Energy Efficient Chip Multiprocessor," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 117-128, Oct. 2006.

[16] B. Lee, J. Collins, H. Wang, and D. Brooks, "CPR: Composable Performance Regression for Scalable Multiprocessor Models," *Proc. 41st Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO),* Nov. 2008.

[17] D.B. Noonburg and J.P. Shen, "A Framework for Statistical Modeling of Superscalar Processor Performance," *Proc. Third Int'l Symp. High-Performance Computer Architecture (HPCA),* pp. 298-309, Feb. 1997.

[18] S. Nussbaum and J.E. Smith, "Modeling Superscalar Processors via Statistical Simulation," *Proc. 2001 Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* pp. 15-24, Sept. 2001.

[19] S. Nussbaum and J.E. Smith, "Statistical Simulation of Symmetric Multiprocessor Systems," *Proc. 35th Ann. Simulation Symp.,* pp. 89-97, Apr. 2002.

[20] M. Oskin, F.T. Chong, and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Design," *Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA),* pp. 71-82, June 2000.

[21] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J.S. Emer, "Quick Performance Models Quickly: Closely-Coupled Partitioned Simulation on fpgas," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS),* pp. 1-10, Apr. 2008.

[22] D.A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D.I. August, and D. Connors, "Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multi-Processors," *Proc. 12th Int'l Symp. High-Performance Computer Architecture (HPCA),* pp. 27-38, Feb. 2006.

[23] E. Perelman, G. Hamerly, and B. Calder, "Picking Statistically Valid and Early Simulation Points," *Proc. 12th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* pp. 244-256, Sept. 2003.

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 45-57, Oct. 2002.

[25] A. Snavely and D.M. Tullsen, "Symbiotic Jobscheduling for Simultaneous Multithreading Processor," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 234-244, Nov. 2000.

[26] D.J. Sorin, V.S. Pai, S.V. Adve, M.K. Vernon, and D.A. Wood, "Analytic Evaluation of Shared-Memory Systems with ILP Processors," *Proc. 25th Ann. Int'l Symp. Computer Architecture (ISCA),* pp. 380-391, June 1998.

[27] M. Van Biesbrouck, L. Eeckhout, and B. Calder, "Considering All Starting Points for Simultaneous Multithreading Simulation," *Proc. Int'l Symp. Performance Analysis of Systems and Software (ISPASS),* pp. 143-153, Mar. 2006.

[28] M. Van Biesbrouck, L. Eeckhout, and B. Calder, "Representative Multiprogram Workloads for Multithreaded Processor Simulation," *Proc. IEEE Int'l Symp. Workload Characterization (IISWC),* pp. 193-203, Oct. 2007.

[29] M. Van Biesbrouck, T. Sherwood, and B. Calder, "A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation," *Proc. Int'l Symp. Performance Analysis of Systems and Software (ISPASS),* pp. 45-56, Mar. 2004.

[30] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J.C. Hoe, D. Chiou, and K. Asanovic, "RAMP: Research Accelerator for Multiple Processors," *IEEE Micro,* vol. 27, no. 2, pp. 46-57, Mar. 2007.

[31] T.F. Wenisch, R.E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J.C. Hoe, "SimFlex: Statistical Sampling of Computer System Simulation," *IEEE Micro,* vol. 26, no. 4, pp. 18-31, July 2006.

Davy Genbrugge received the MS degree in computer science from Ghent University in 2005. He is currently working toward the PhD degree at the Electronics and Information Systems Department, Ghent University, Belgium. His research interests include computer architecture, in general, and simulation technology more in particular.

Lieven Eeckhout received the PhD degree in computer science and engineering from Ghent University in 2002. He is an associate professor in the Electronics and Information Systems Department at Ghent University, Belgium. His research interests include computer architecture, virtual machines, performance analysis and modeling, and workload characterization. He is a member of the IEEE and the IEEE Computer Society.