The Multi-Program Performance Model: Debunking Current Practice in Multi-Core Simulation

Kenzo Van Craeynest Lieven Eeckhout
ELIS Department, Ghent University, Belgium

Abstract

Composing a representative multi-program multi-core workload is non-trivial. A multi-core processor can execute multiple independent programs concurrently, and hence, any program mix can form a potential multi-program workload. Given the very large number of possible multi-program workloads and the limited speed of current simulation methods, it is impossible to evaluate all possible multi-program workloads.

This paper presents the Multi-Program Performance Model (MPPM), a method for quickly estimating multi-program multi-core performance based on single-core simulation runs. MPPM employs an iterative method to model the tight performance entanglement between co-executing programs on a multi-core processor with shared caches. Because MPPM involves analytical modeling, it is very fast, and it estimates multi-core performance for a very large number of multi-program workloads in a reasonable amount of time. In addition, it provides confidence bounds on its performance estimates. Using SPEC CPU2006 and up to 16 cores, we report an average performance prediction error of 2.3% and 2.9% for system throughput (STP) and average normalized turnaround time (ANTT), respectively, while being up to five orders of magnitude faster than detailed simulation.

Subsequently, we demonstrate that randomly picking a limited number of multi-program workloads, as done in current pactice, can lead to incorrect design decisions in practical design and research studies, which is alleviated using MPPM. In addition, MPPM can be used to quickly identify multi-program workloads that stress multi-core performance through excessive conflict behavior in shared caches; these stress workloads can then be used for driving the design process further.

1 Introduction

Simulation and modeling are at the foundation of processor design and computer architecture research, i.e., research and development is driven by the careful evaluation of design alternatives in order to make correct design decisions. A key aspect of experimental evaluation is the workload that serves as input to simulation and modeling. A workload that is unrepresentative of a processor's target workload may lead to a suboptimal design, hence, it is absolutely crucial to have a representative workload.

Building a representative workload is very challenging, especially for multi-core processors. A multi-core processor has multiple hardware thread contexts, and each hardware thread context can execute a different program. As a result, a multi-core processor workload may consist of a mix of multiple independent programs. In fact, multi-program workloads are a very important and significant fraction of today's workloads. Given the limited amount of multi-threading in current desktop applications [3], multi-program workloads are predominant in today's computer systems, including mobile devices, laptops, desktops and even servers.

Evaluating multi-program workloads is non-trivial though. For a given number of programs, the number of multi-program workloads quickly explodes. For N programs and M hardware contexts, there are M combinations with repetition out of N programs, or $\binom{N+M-1}{M}$ multi-program workloads in total. This means there are 435 possible multi-program workload mixes for 29 SPEC CPU2006 benchmarks on a dual-core processor; 35,960 workload mixes for a quad-core processor; and more than 30.2 million workload mixes for an eight-core processor. Assuming detailed cycle-accurate processor simulation at a speed of 300 KIPS and assuming 1B instruction workloads—as in our setup—this would result in 54 days for simulating all possible two-program workloads. For four- and eight-program workloads, total simulation time would need to be counted in years. Clearly, simulating all possible multi-program workloads using detailed simulation is completely infeasible in practice.

Hence, current practice in computer architecture research and development is to pick a limited number of multi-program workload mixes, typically a dozen or a couple tens that are randomly chosen. Often, architects com-

pose classes of multi-program workload mixes with each class representing a particular set of multi-program workloads. For example, one class may comprise combinations of memory-intensive programs, another class may comprise mixes of compute-intensive and memory-intensive programs, and yet another class may comprise a set of compute-intensive workloads. Within each class, architects then select a number of random multi-program workload mixes. It is unclear though whether a limited number of randomly chosen multi-program workloads is representative for the very large set of multi-program workloads.

In this paper, we propose the Multi-Program Performance Model (MPPM), a method for quickly estimating multi-program multi-core performance from single-core simulation runs; this allows for quickly estimating multi-core performance for a large number of multi-program workloads in a reasonable amount of time. We collect a profile during single-core simulation that captures a program's memory behavior as well as its phase behavior; this is a one-time cost. We then employ an iterative method that models the performance entanglement between the co-executing programs on a multi-core processor with shared caches: the iterative method captures how per-program performance affects the amount of resource sharing, and, vice versa, how resource sharing in its turn affects per-program performance. Since this iterative method involves an analytical model, it is very fast, while being accurate. Using this powerful technique, we demonstrate that current practice of simulating a limited number of multi-program mixes may lead to incorrect design decisions. Instead, we advocate MPPM which allows for evaluating performance for a very large number of multi-program workload mixes in a reasonable amount of time. The method can also be used to identify workload mixes that stress multi-core performance due to excessive conflict behavior in shared resources.

More specifically, we make the following contributions in this paper.

- We propose the Multi-Program Performance Model (MPPM) for estimating multi-core processor performance for multi-program workloads. MPPM uses single-core simulation profiles and estimates multi-core processor performance while taking into account resource sharing in shared caches when running multi-program workloads. The performance entanglement between co-executing programs due to resource sharing is solved through an iterative approach that estimates the amount of resource sharing and how it affects per-core performance, and vice versa. We report an average performance prediction error of 2.3% and 2.9% for system throughput (STP) and average normalized turnaround time (ANTT), respectively, compared to detailed simulation across SPEC CPU2006 using the x86 CMP$im simulator [9] and up to 16 cores.
- We demonstrate that MPPM can quickly estimate multi-core processor performance from single-core simulation runs, which enables estimating multi-core performance for a large number of multi-program workload mixes in a reasonable amount of time. Indeed, the key feature of the proposed method is that it decouples per-core simulation from multi-core simulation, yielding a multi-core processor simulation and modeling approach for multi-program workloads with only linear time complexity in the number of programs. Our method is shown to be up to five orders of magnitude faster than detailed multi-core processor simulation.
- We demonstrate that current practice of randomly choosing a limited number of multi-program workloads may lead to incorrect design decisions. Instead, a more accurate approach is to use MPPM for evaluating all, or at least a very large number of, multi-program workload mixes. In addition, MPPM provides confidence bounds on its performance estimates.
- We demonstrate that MPPM can identify multi-program workloads that yield very poor multi-core processor performance due to excessive conflict behavior in shared resources. Architects can then focus on these stress workloads and fine-tune the design to yield better performance.

This paper is organized as follows. Section 2 presents MPPM in great detail, after which we present the experimental setup in Section 3. We evaluate MPPM in Section 4, debunk current practice in Section 5, and use MPPM for identifying multi-program stress workloads in Section 6. Finally, we discuss related work and how MPPM differs from prior work in this area in Section 7, and we conclude and describe future work in Section 8.

2 Multi-Program Performance Model

Figure 1 gives a general overview of the Multi-Program Performance Model (MPPM). We first perform single-core simulation profiling runs for all the benchmarks in the benchmark suite. This is a one-time cost. Once these single-core profiles are collected, they serve as input to the multi-program performance model. In other words, MPPM estimates multi-program performance from single-core simulation profiling runs. Because the single-core simulation runs are a one-time cost only, and because MPPM is an analytical approach, the proposed method is very effective at estimating multi-program performance on multi-core processors. In fact, estimating multi-program performance for an arbitrary set of benchmarks is done very quickly — typically

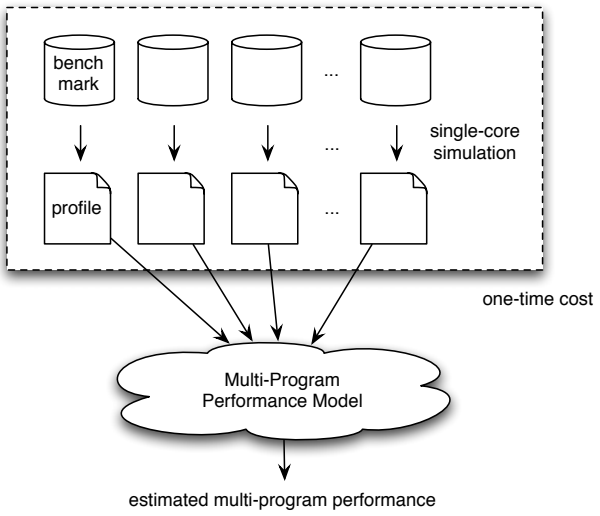

Figure 1. General overview of MPPM.

around a couple tenths of seconds per multi-program workload — hence, MPPM enables predicting multi-core performance for a large set of multi-program workloads in a reasonable amount of time. As an example, MPPM estimates multi-core performance for 5,000 four-program workloads in half an hour.

MPPM assumes a particular multi-core processor architecture of interest for the single-core simulation runs. In other words, if one were to predict performance for a multi-core processor with out-of-order processor cores and a cache hierarchy with three levels of cache, this same or derived processor architecture needs to be considered during the single-core simulations as well. This means that the single-core simulations need to be performed with the same core architecture and the same or derived cache hierarchy, however, the benchmark is run in isolation, i.e., there are no co-executing programs. (In fact, in our setup, we can run single-core simulations and derive performance models for cache hierarchies with reduced associativity at each level of the hierarchy. For example, we can run single-core simulations and collect the single-core profiles for a 16-way set-associative cache, and derive single-core simulation profiles for an 8-way set-associative cache without having to run additional single-core simulations.) Although this may seem like a limitation of MPPM — single-core simulation runs need to be performed for different multi-core architectures of interest — it is not a major limitation in practice: it requires single-core simulation runs only. There are no time-consuming multi-core simulations required, and the MPPM model makes multi-core performance predictions quickly for a large set of multi-program workloads. Further, once the single-core simulation profiles are obtained, MPPM can estimate performance for a varying number of cores, different cache associativities, and different combinations of co-executing programs very quickly.

We now detail on the two major steps in MPPM: single-core simulation profiling and the performance model itself.

2.1 Single-core Simulation Profiling

Single-core simulation profiling collects three characteristics:

- **Single-core CPI** is the number of Cycles Per Instruction (CPI) when running the single-core workload in isolation, i.e., there are no co-executing programs. CPI is easily obtained by dividing cycle count with the number of dynamically executed instructions.

- **Memory CPI** is the fraction of the single-core CPI waiting for memory. There are two ways for computing the memory CPI. One way is to employ the counter architecture proposed by Eyerman et al. [8] for computing CPI stacks on out-of-order processors; implementing this counter architecture in the simulator enables computing the memory CPI component from a single simulation run. Alternatively, the memory CPI can be computed from two simulation runs: one run with a perfect Last-Level Cache (LLC), i.e., all accesses to the LLC are hits and there are no memory accesses, versus one run with an imperfect LLC, i.e., LLC misses go off to memory. The CPI obtained from the latter minus the CPI obtained from the former then is the memory CPI.

- **Stack Distance Counters (SDCs)** capture a program's temporal memory access behavior in set-associative (or fully associative) caches [12]. We collect SDCs for each program on the LLC without cache sharing, i.e., by running the program in isolation. An SDC for an A-way set-associative cache involves $A+1$ counters, $C_1, C_2, \ldots, C_A, C_{>A}$, and is computed as follows. On each access, one of the counters is incremented. If the access is to the ith position in the LRU stack for that set, the ith counter C_i is incremented. If the cache access involves a miss, then the $C_{>A}$ counter is incremented.

Each of these performance characteristics are measured on a per-interval basis. The reason is to be able to model the impact of time-varying phase behavior on resource contention in multi-core processors. In our setup, we measure these characteristics for every interval of 20 million (dynamically executed) instructions. For a 1 billion instruction trace, this implies 50 intervals in total per benchmark with the above characteristics measured for each interval. This is done for each benchmark in the benchmark suite.

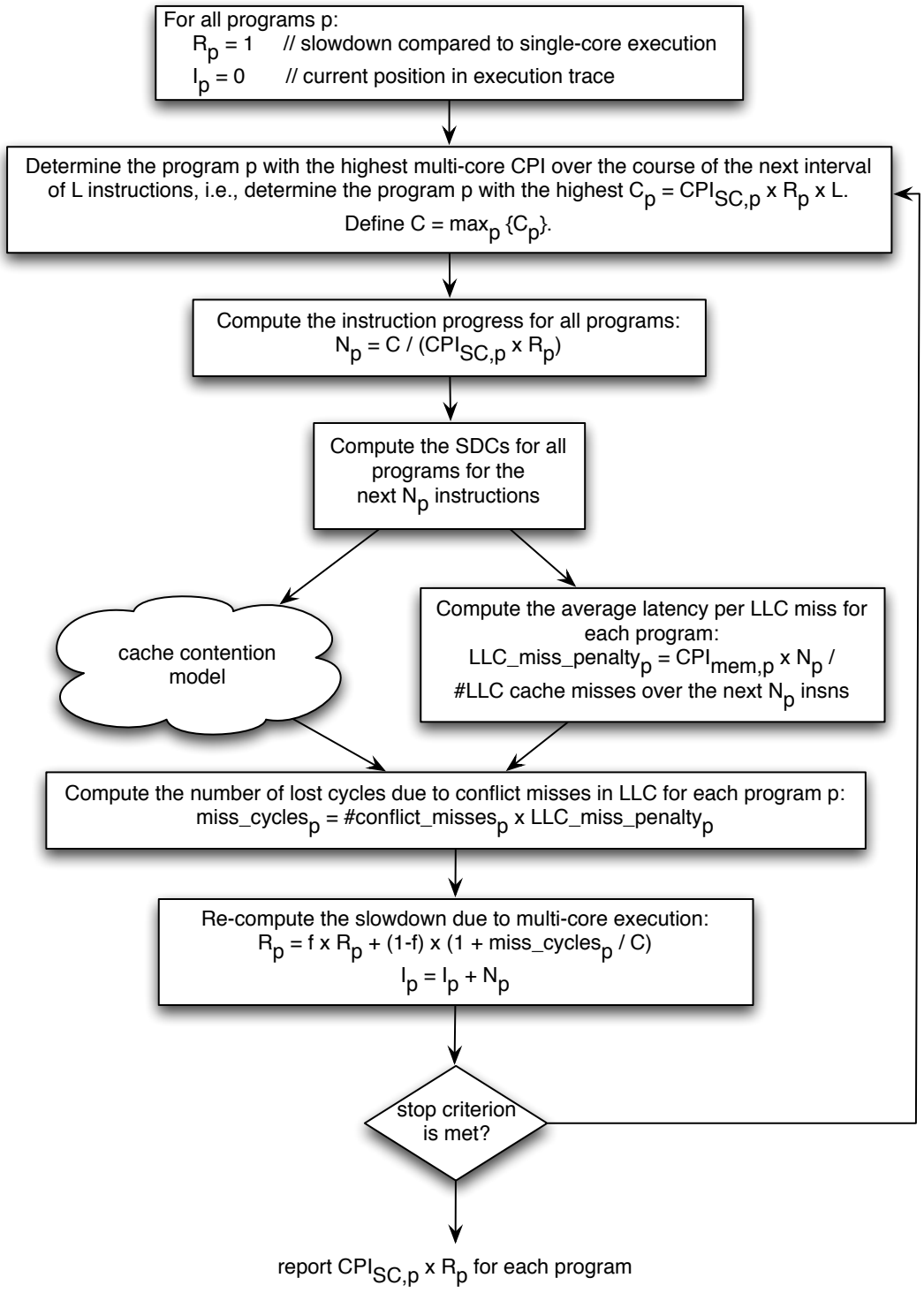

Figure 2. The Multi-Program Performance Model.

2.2 MPPM

The single-core performance characteristics as mentioned in the previous section then serve as input to the Multi-Program Performance Model (MPPM). The concept of the MPPM is to initially start from the single-core performance measurements and then iteratively converge on how resource contention in shared resources affects per-core performance in a multi-core processor. The reason for the iterative process is the tight performance entanglement between per-core performance and resource contention, i.e., per-core performance affects the amount of resource contention, and vice versa, resource contention affects per-core performance. In order to model this tight performance entanglement, the model initially estimates the amount of resource contention assuming each program makes progress as per the single-core simulations; however, the amount of resource sharing affects per-core progress, which in its turn affects resource sharing. Hence, in the next iteration, per-core progress is adjusted to incorporate how resource contention affects per-core progress. This, in its turn, may again affect the amount of resource contention seen, which leads to the second iteration, etc. This iterative process continues until convergence.

Figure 2 gives a schematic overview of the MPPM model. We define R_p as the slowdown for a program p in the multi-program workload mix relative to isolated execution. We assume all programs experience the same relative slowdown of $R_p = 1$ and execute at single-core speed initially. Further, we assume that all programs start at the beginning of the execution trace, i.e., the instruction pointer is set to zero: $I_p = 0$. Once these initial conditions are set, the iterative process starts.

At each step in the iterative process, we first determine the slowest program in the workload mix, or the program in the multi-program workload mix with the highest multi-core CPI over the next L instructions. L is 200M instructions in our setup. A program's multi-core CPI is computed as the single-core CPI ($CPI_{SC,p}$) multiplied with its relative slowdown R_p. We define C to be the number of cycles it takes for the slowest program to execute L instructions. We then determine how much progress each program in the workload mix can make during the next C cycles. We define N_p as the number of instructions each program p can execute during the next C cycles.

Once we know how many instructions each program will execute during the next time interval of C cycles, we can compute the SDCs for each of the programs over this time interval. This is done by adding the per-interval SDCs. As mentioned in the previous section, the single-core performance characteristics are measured on a per-interval basis of 20M instructions. Computing the SDCs for the next time interval of C cycles is done by simply adding the per-interval SDCs for the next N_p instructions. The SDCs serve two needs. First, it serves as input to a cache contention model that estimates the additional number of conflict misses due to cache sharing in the LLC. There exist several cache contention models [4, 5, 10]. We use the Frequency of Access (FOA) model proposed by Chandra et al. [4] because it is a fairly simple model and we found it to be accurate enough for our needs. Second, the SDCs allow for estimating the average penalty per LLC miss. This is done by dividing the number of cycles lost due to memory accesses with the number of LLC misses; we assume here that the average LLC miss penalty is the same under multi-core execution versus single-core execution. The number of cycles waiting for memory is computed as the memory CPI component ($CPI_{mem,p}$) multiplied with the number of instructions in the next time interval N_p. The number of LLC misses is obtained from the SDC's $C_{>A}$ counter, as mentioned above. We estimate the number of cycles lost due to conflict misses in the LLC by multiplying the number of additional conflict misses due to cache sharing (obtained from the cache contention model) and the average penalty per LLC miss (computed as described above).

We can now estimate the (current) relative slowdown for each program in the workload mix due to resource contention. This is done using an exponential moving average of the average slowdown observed so far and the current slowdown according to the above model. The reason for taking an exponential moving average is to include a smoothing effect, which we found to be important for achieving good accuracy, especially for programs with significant time-varying execution behavior. We also compute the current position in the execution for each program. This is done by simply advancing the instruction pointer by N_p instructions for each program.

This iterative process is repeated multiple times until a stop criterion is met. Each iteration involves 200M instructions for the slowest running program, and the iterative process continues until the slowest running program in the workload mix has executed 5B instructions in total. Given that our instruction traces are 1B instructions in size, this means that the slowest program needs to iterate over its entire trace five times. Faster running programs may iterate over their trace more than five times. We found that the performance numbers converged given this stop criterion.

2.3 Discussion

We want to emphasize again that MPPM itself does not involve detailed cycle-accurate multi-core simulation. The process as explained above only involves 'analytical' simulation in which we employ analytical models for estimating multi-core performance. This yields a very fast multi-core performance estimation technique: MPPM makes a multi-

ROB	128 entries
pipeline	8-stage, 4-wide
ld/st	max of two loads & one store per cycle
L1 I-cache	32KB, 4WSA, LRU, 1 cycle
L1 D-cache	32KB, 8WSA, LRU, 1 cycle
L2 cache	private, 256KB, 8WSA, 10 cycles
L3 cache	shared, see Table 2
memory	200 cycles
branch prediction	perfect

Table 1. Baseline processor configuration.

	size	*assoc*	*latency*
config #1	512KB	8	16
config #2	512KB	16	20
config #3	1MB	8	18
config #4	1MB	16	22
config #5	2MB	8	20
config #6	2MB	16	24

Table 2. Last-level cache (LLC) configurations.

core performance estimate in less than one second, provided that the single-core simulation runs were done beforehand.

It is also important to stress that MPPM is independent of the cache replacement and/or partitioning strategy employed in the shared cache. In fact, the cache contention model is an integral part of the approach, and if the cache contention model supports multiple cache replacement and/or partitioning strategies, so does MPPM. The cache contention model used in this paper is the FOA model [4], as mentioned before. FOA assumes that the effective cache space for a program is proportional to its access frequency. The intuition is that a program that has a high access frequency tends to bring in more data into the cache, and hence it effectively occupies a larger fraction of the caches. We found FOA to be accurate enough for our purpose. Part of our future work will focus on other cache contention models that can potentially model other cache replacement and/or partitioning strategies.

3 Experimental Setup

We use a multi-core processor simulator based on CMP$im [9], which is an x86 simulator built on top of Pin. CMP$im is a user-level simulator and allows for simulating multi-core processor architectures. Our version of the CMP$im simulator is the one available from the Cache Replacement Championship[1]. The processor architecture that we simulate is detailed in Tables 1 and 2. We consider 4-wide out-of-order processor cores with private L1 instruction and data caches. Each core has a private L2 cache. The L3 cache is shared among the cores and is the last-level cache (LLC) in our setup: we apply our method to the L3 cache. All caches implement an LRU replacement policy. We consider different LLC configurations, as shown in Table 2. If not explicitly mentioned, we report performance results for configuration #1 which has the smallest LLC; this is to stress our model.

We consider all the SPEC CPU2006 benchmarks with their reference inputs. All the benchmarks were compiled with the GNU C compiler version 4.3.4 and optimization level `-O2`. We use SimPoint [13] to pick representative simulation points of one billion instructions each.

When quantifying multi-core processor performance we consider two performance metrics, namely system throughput (STP) and average normalized turnaround time (ANTT) [7]. STP measures multi-core performance from a system perspective and quantifies the accumulated progress by all the programs in the multi-program workload mix. STP equals weighted speedup by Snavely and Tullsen [14], and is a higher-is-better metric:

$$STP = \sum_{p=1}^{n} \frac{CPI_{SC,p}}{CPI_{MC,p}}.$$

ANTT focuses on user-perceived performance and quantifies the average slowdown during multi-core execution relative to single-core, isolated execution. ANTT is the reciprocal of the hmean metric proposed by Luo et al. [11]:

$$ANTT = \frac{1}{n} \sum_{p} \frac{CPI_{MC,p}}{CPI_{SC,p}}.$$

4 Model Evaluation

We evaluate MPPM along two criteria: accuracy and speed. However, before doing so, we first quantify performance variability across random sets of multi-program workloads — this will demonstrate that obtaining tight confidence bounds requires a sufficiently large number of workload mixes.

4.1 Variability

Figure 3 shows the variability in STP and ANTT as a function of the number of random sets of multi-program workloads on a four-core processor. These graphs clearly illustrate that selecting a limited number of random multi-program workloads yields limited confidence in the overall performance measurements. For example, selecting 10 workload mixes yields a 10% and 18% confidence interval for STP and ANTT, respectively. Doubling the number of

[1] `http://www.jilp.org/jwac-1/`

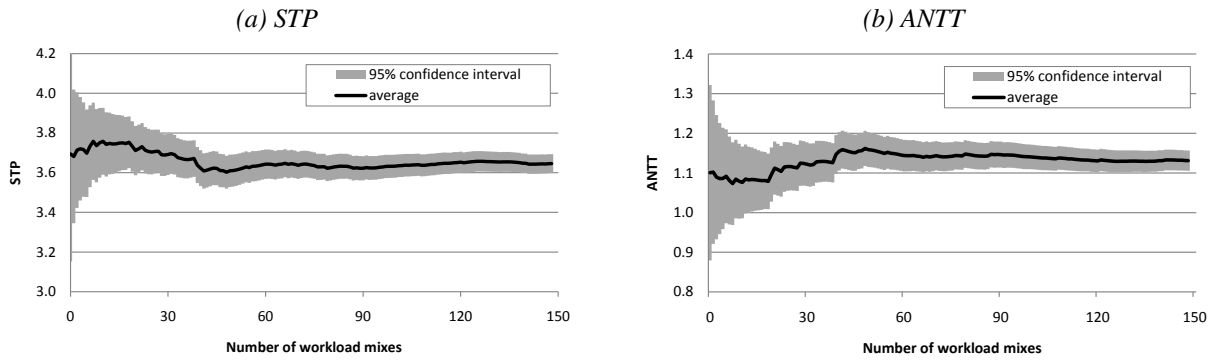

Figure 3. Variability in (a) STP and (b) ANTT as a function of the number of multi-program workload mixes.

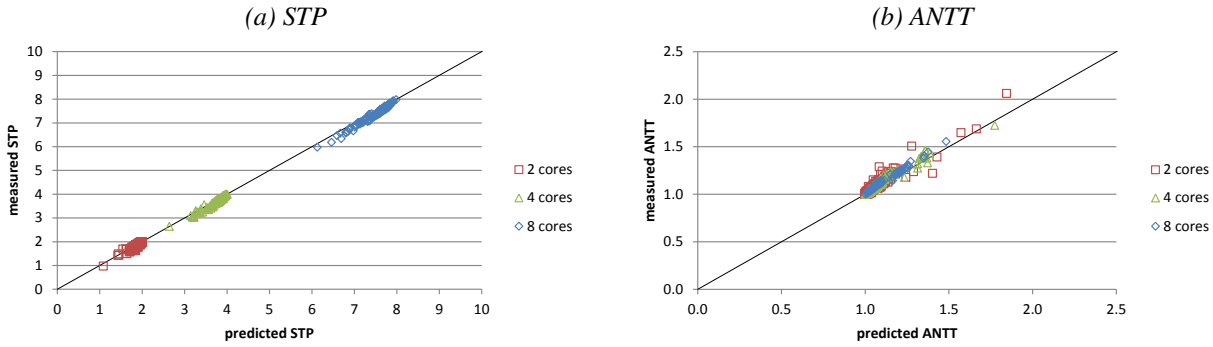

Figure 4. Accuracy of MPPM for predicting (a) STP and (b) ANTT; measured STP/ANTT on vertical axis versus predicted STP/ANTT on the horizontal axis.

workload mixes to 20 does not increase confidence dramatically: the confidence intervals are still around 7% and 13% for STP and ANTT, respectively. Such confidence intervals may be too large for many practical research and design studies. Comparing design alternatives that differ in the percent range with a performance evaluation method with this large confidence intervals may be problematic. At 150 workload mixes, the confidence bounds are down to 2.6% and 4.5% for STP and ANTT, respectively; hence, we report performance numbers for 150 workload mixes in the remainder of the paper.

4.2 Accuracy

Figure 4 shows scatter plots for STP and ANTT. These graphs assume the baseline processor configuration with 2, 4 and 8 cores, and 150 multi-program workload mixes. We simulate these workload mixes through detailed simulation using CMP$im; this yields the 'measured' STP and ANTT metrics. We also estimate multi-core performance using MPPM which yields the 'predicted' metrics. The scatter plots show the predicted metrics versus the measured metrics. Each dot represents one of the workload mixes. Perfect prediction would imply all dots to lie on the bisector. We observe a strong correlation between the measured and predicted performance metrics, i.e., all the dots lie around and are close to the bisector. The average error across these workload mixes equals 1.4%, 1.6% and 1.7% for STP and 2, 4 and 8 cores, respectively; and 1.5%, 1.9% and 2.1% for ANTT and 2, 4 and 8 cores, respectively.

We also ran a number of experiments for 16 cores using 25 16-program workload mixes; here, we consider a larger 1MB LLC (configuration #4). We were unable to run more than 25 16-program workload mixes because of time constraints — the simulations took extremely long, which is exactly the problem we are addressing with MPPM. (These results are not shown in Figure 4.) The average error equals 2.3% and 2.9% for STP and ANTT, respectively.

The fact that MPPM is accurate for predicting STP and ANTT suggests that it is also effective for predicting the relative per-program slowdown, or by how much a program gets slowed down due to multi-program execution on a multi-core processor. Figure 5 reports both the measured

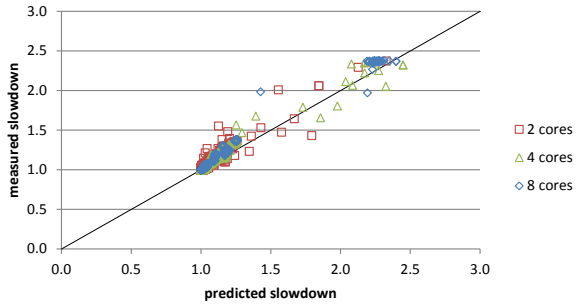

Figure 5. Measured versus predicted relative per-program slowdown due to multi-core execution.

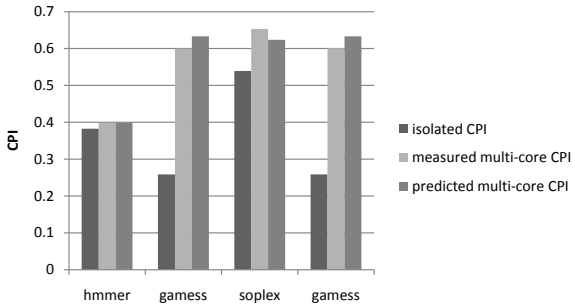

Figure 6. Tracking the performance of individual programs in a multi-program workload consisting of two copies of gamess **along with** hmmer **and** soplex**: isolated execution CPI, measured multi-core execution CPI (through simulation), and predicted multi-core execution CPI (through MPPM).**

and the predicted relative slowdown for each program. Correlation is good and the average error equals 7% across the 150 workloads for 2, 4 and 8 cores; for the 25 multi-program workloads on the 16-core processor (not shown in Figure 5), we obtain an average error of 4.5%. Figure 6 makes this more concrete and shows an example for the 4-program workload with the worst STP; this workload consists of two copies of gamess along with hmmer and soplex. The gamess copies are slowed down substantially through multi-core execution (more than 2×), whereas soplex is slowed down somewhat only, and hmmer is barely affected by multi-core execution. MPPM predicts these slowdowns accurately.

It is worth noting that the error for predicting per-program performance, although low, is larger than the error for predicting STP and ANTT. The reason is that STP and ANTT measure total system performance and average per-program performance, respectively, and given how STP and ANTT are computed, see Section 3, positive and negative errors in predicting per-program performance get smoothed, which leads to more accurate overall STP/ANTT predictions.

4.3 Speed

MPPM is substantially faster than detailed simulation. As mentioned before, MPPM requires single-core simulations, but this is a one-time cost only. In our setup with 1B instruction simulation points this takes around 1 hour of simulation time per benchmark on CMP$im, or slightly more than an entire day for the entire SPEC CPU2006 benchmark suite on a single machine. The MPPM model itself is very fast because it does not involve detailed simulation. Instead, it uses analytical modeling for estimating multi-core performance. MPPM takes less than one second per multi-program workload; typically, a couple tenths of seconds. In contrast, simulating a multi-program workload on a detailed cycle-accurate simulator is extremely time-consuming. For example, simulating one multi-program workload for an eight-core processor takes around 12 hours in our setup. In summary, depending on the scenario, MPPM is up to 5 orders of magnitude faster than detailed simulation. Considering 150 multi-program workloads on an 8-core processor, MPPM (including the cost of single-core simulations) is 62× faster than detailed simulation. Assuming that the single-core simulations were done beforehand, MPPM is more than 53,000× faster.

5 Debunking Current Practice

Now that we have demonstrated that MPPM is both accurate and fast for estimating multi-program workload performance on multi-core processors, we leverage MPPM to evaluate (and debunk) current practice in simulating multi-program workloads. One of the critical concerns when simulating multi-program workloads is which workloads to pick out of the very large set of possible multi-program workloads. Current practice is to pick a limited number of multi-program workloads. The reason for limiting the number of workloads is that simulation is extremely time-consuming. Hence, researchers pick a limited number of workloads at random out of the very large set of possible multi-program workloads. Alternatively, researchers often classify their benchmark programs in a number of classes, e.g., compute-intensive versus memory-intensive programs, and then randomly pick multi-program workloads from these classes to form multi-program workload categories. For example, one category may be workload mixes consisting of compute-intensive programs only; another category may be workload mixes with memory-intensive pro-

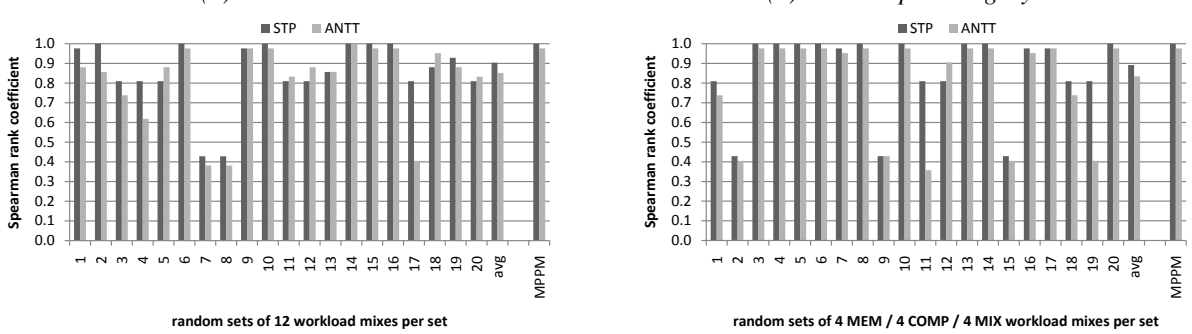

Figure 7. Evaluating current practice of selecting random workload mixes: Rank correlation coefficient for 20 sets of 12-program workloads versus MPPM. (a) Random selection of programs; and (b) Random selection of programs within program categories.

grams only; a third category may contain mixed compute- and memory-intensive programs.

To evaluate whether current practice is adequate, we consider the following setup. We compare six multi-core configurations that differ in their LLC configuration in terms of size, associativity and access time, see Table 2. Without detailed experimental evaluation, it is unclear which configuration yields the best overall performance. For example, configuration #2 has a higher associativity than configuration #1, and thus a lower miss rate, but its access latency is also higher. Similar trade-offs are possible between all pairs of configurations. So, it is unclear which configuration yields the best overall performance without detailed analysis. We now evaluate how well current practice and MPPM can rank these six configurations. The results are shown in Figure 7. These graphs show the rank correlation coefficients for current practice assuming 12 randomly selected multi-program workloads on a quad-core processor. This experiment is repeated 20 times, hence the 20 bars on the left-hand side of the two graphs in Figure 7. The second but last bars on the right-hand side, labeled 'avg' reports the average correlation coefficient for current practice. We then compare against MPPM while considering 5,000 multi-program workloads, see the right-most bars in Figure 7. The Spearman rank correlation coefficient quantifies how well two rankings compare to each other, or more formally, how well the relationship between two rankings can be described using a monotonic function; a Spearman rank correlation coefficient of one means a perfect match in the rankings. Our point of reference is the ranking obtained from detailed simulation with 150 multi-program workloads. MPPM is clearly more accurate than current practice. For some of the randomly picked workload mixes, the rank correlation coefficient is as low as 0.5 and below, see for example mixes 7 and 8 in Figure 7(a) and mixes 2, 9 and 15 in Figure 7(b). MPPM on the other hand

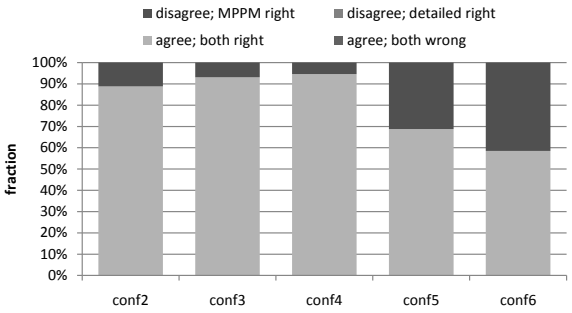

Figure 8. Fractions of when current practice agrees or disagrees with MPPM, and when MPPM is correct and current practice is not.

achieves a rank correlation coefficient of 1 and 0.93 for STP and ANTT, respectively.

In order to make it more concrete we now pairwise compare design points, namely configuration #1 versus all the other configurations #2 through #6. Figure 8 quantifies how often current practice (assuming multi-program categories) disagrees with MPPM, and, when they disagree, how often MPPM is correct compared to the reference of detailed cycle-accurate simulation of 150 multi-program workloads, and thus by consequence, how often current practice leads to incorrect conclusions. In the most extreme comparison between configuration #1 and #6 we observe that in approximately 40% of the cases current practice disagrees with MPPM, and current practice leads to an incorrect conclusion with respect to which configuration yields the best performance.

These experiments collectively illustrate that current practice of selecting a limited number of multi-program mixes may lead to misleading and incorrect conclusions in practical research and design studies. MPPM on the other

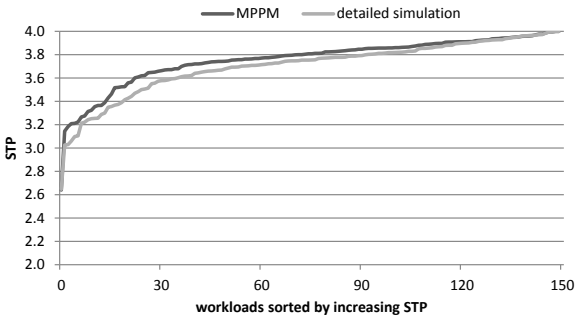

Figure 9. Identifying 4-program workloads with the worst STP.

hand leads to correct conclusions because it considers a very large number of multi-program mixes. In addition, MPPM does so in only a small fraction of the time needed through current practice.

6 Identifying Stress Workloads

As mentioned before, MPPM's unique ability is to quickly estimate multi-program performance on multi-core processors. One important application of the MPPM framework is to identify workload mixes that stress the multi-core processor in the sense that performance is substantially lower for these workloads compared to the other workloads. Once these stress workloads are identified, they can be analyzed in more detail in order to understand why the multi-core processor fails to deliver good performance for these workloads, which may ultimately lead to an improved design. We find MPPM to be accurate in ranking the multi-program workloads with respect to how severe these workloads stress the processor architecture. Figure 9 shows sorted STP values obtained through detailed simulation and MPPM; more precisely, the workloads on the horizontal axis are sorted by increasing value of the STP values obtained through detailed simulation. This graph once again illustrates that MPPM is accurate compared to simulation, and in addition, it illustrates that MPPM can identify the worst-case workloads. MPPM is able to identify the top-23 worst-case workloads out of the 25 worst-case workloads obtained through detailed simulation. Further analysis showed that one particular benchmark, namely gamess, is very sensitive to multi-core execution: we found that gamess gets a slowdown by a factor 2.2×, whereas the other benchmarks experience a slowdown of at most 1.3× (gokmk) and 1.2× (soplex, omnetpp, h264 and xalan); the remaining benchmarks are less sensitive to cache sharing.

7 Related Work

There exists some work in simulating and modeling multi-program workloads on multicore processors. These techniques differ from MPPM in several ways, as decribed below.

The work most similar to ours is the work by Eklöv et al. [5]. They present StatCC which is a cache contention model that, given the relative CPIs of the co-scheduled programs and their individual reuse distance distributions, estimates the reuse distance distribution of the interleaved access stream to the shared cache. The reuse distance is defined as the number of memory references between two consecutive accesses to the same cache line. Once they have the reuse distance distribution of the interleaved access stream, they leverage their prior StatCache [2] and StatStack [6] work to estimate the cache miss rates for the co-scheduled programs. They use an equation solver to solve the interdependence between the programs' CPIs and cache miss rates. There are a number of key differences between Eklöv et al.'s approach and this work. We use an iterative method for solving the CPI versus cache miss rates interdependence while taking into account time-varying workload behavior, instead of the general-purpose equation solver by Eklöv et al. In addition, we use a stack distance counter distribution instead of a reuse distance distribution. Finally, we evaluate our approach for up to 16 cores; Eklöv et al. limited their evaluation to 2 cores only.

Lee et al. [10] use (spline-based) regression modeling to build multi-core processor performance models. They build three regression models: a core model, a contention model for the shared resources (i.e., the shared memory hierarchy), and a model that combines the core and contention models to form an overall multi-core processor performance model. This approach does not address the challenge of having to deal with the explosion in the number of multi-program workload mixes, because the contention model needs to be trained for each workload mix. Although the contention model can be trained through cache simulation, which is much faster than multi-core simulation, it is fundamentally limited by the explosion in the number of multi-program workload mixes. Our method on the other hand, addresses this fundamental challenge through analytical modeling. Further, our method takes into account time-varying execution behavior for determining the amount of contention through shared resources and its impact on performance.

Chandra et al. [4] propose three cache contention models. The input to the models is the shared cache stack distance distribution or a circular sequence profile for each program. The output is the number of additional cache misses due to cache sharing for each of the threads. The three models proposed by Chandra et al. include the Frequency of Ac-

cess (FOA) model, the Stack Distance Competition (SDC) model and the Inductive Probability (Prob) model. The key difference between our work and Chandra et al.'s work is that we predict overall multi-core processor performance, in contrast to Chandra et al.'s method which estimates cache miss rates only. Further, they do not take into account time-varying phase behavior, and their evaluation is limited to two-program workload mixes only.

Van Biesbrouck et al. [18] propose the co-phase matrix as a method to quickly simulate multi-program workloads on multi-threaded architectures. The co-phase matrix takes into account a program's time-varying execution behavior and basically keeps track of the performance for all the co-phases in a two-program workload mix. The key idea here is that each co-phase needs to be simulated only once, its performance is stored in the co-phase matrix, and whenever the same co-phase is encountered again, the relative progress for each of the co-executing programs is simply picked from the co-phase matrix, and does not to be simulated again. This greatly reduces overall simulation time. In follow-on work, Van Biesbrouck et al. [16] consider multiple starting points for each of the programs in the multi-program workload mix. The co-phase matrix does not address the challenge of having to deal with an explosion in the number of multi-program workloads, i.e., the co-phase matrix needs to computed for each multi-program workload mix of interest.

Van Biesbrouck et al. [17] propose a method for picking a representative set of multi-program workloads. They profile each program using a set of micro-architecture independent characteristics, and they then apply statistical analysis techniques such as Principal Component Analysis and Cluster Analysis to pick a limited but representative set of multi-program workload mixes. Similar to our work, they aim at addressing the explosion in the number of multi-program workload mixes, however, their solution differs from ours in a fundamental way. Their approach comes up with a limited set of multi-program workload mixes that need to be simulated in detail, whereas our approach estimates the performance of a multi-program workload mix through analytical modeling. This implies that we can estimate performance for a large number of multi-program workload mixes much more quickly, and in addition, in contrast to Van Biesbrouck et al., we compute confidence bounds on performance by considering a very large number of workload mixes.

Tuck and Tullsen [15] propose a methodology for quantifying performance on multi-threaded architectures, which is also applicable to multi-core processors. A challenging problem when evaluating multi-threaded processor performance is that the relative progress of independent co-executing programs may differ across processor architectures, and hence, the effective multi-program workload may be different across architectures, leading to biased and incorrect design decisions. Tuck and Tullsen propose to re-iterate the execution of a program in a multi-program workload execution as soon as it reaches the end of its execution. This process is re-iterated until convergence. Whereas Tuck ad Tullsen apply this approach on real hardware experiments, Vera et al. [19] propose a similar approach for simulation purposes. Both approaches run either real hardware or simulation experiments to evaluate multi-program performance; instead, MPPM employs an analytical model which allows for evaluating a large number of multi-program workload mixes in limited time.

Alameldeen and Wood [1] study non-determinism when evaluating multi-threaded programs on multi-processor processors. Non-determinism refers to the fact that small timing variations can cause executions that start from the same initial state to follow different execution paths. They propose adding non-determinism in deterministic simulators to model this effect, and they report confidence bounds when simulating multi-threaded programs. Our work is orthogonal and targets multi-program workloads. MPPM quantifies how variability in multi-program workload mixes affects performance; this is done by computing confidence bounds.

8 Conclusion and Future Work

Current practice in multi-core simulation is to consider a limited number of multi-program workloads. In this paper, we have shown that tens of multi-program workloads are not representative, and may lead to incorrect decisions in practical design and research questions. Instead, we advocated and proposed the Multi-Program Performance Model (MPPM), which is an analytical approach for estimating multi-program multi-core performance from single-core simulation runs. MPPM incorporates a program's time-varying execution behavior, and accurately estimates the tight performance entanglement between co-executing programs because of resource contention in shared caches. MPPM was shown to be accurate within 2.3% and 2.9% on average for STP and ANTT, respectively, for SPEC CPU2006 and up to 16 cores, while being up to five orders of magnitude faster than detailed simulation. Hence, MPPM estimates multi-program performance for a very large number of multi-program workloads in a reasonable amount of time, while providing confidence bounds on its performance estimates. An appealing usage of MPPM is to identify multi-program workloads that yield poor performance due to excessive conflict behavior in shared resources.

There is ample room for further evaluation, improvements and extensions to MPPM. Multi-threaded workloads not only incur negative interference among co-executing threads but also positive interference, i.e., one thread fetch-

ing data that is later accessed by other threads. We believe that MPPM can accurately model the impact of positive interference on overall performance because MPPM models the impact of cache sharing behavior of per-thread performance, but this needs to be further evaluated using multi-threaded workloads. Other avenues for future research include using MPPM for modeling heterogeneous multi-core performance and exploring the heterogeneous multi-core design space, as well as modeling resource sharing in simultaneous multi-threading (SMT) cores. Yet other avenues are to improve the modeling of sources of contention other than cache sharing, such as bandwidth sharing, TLB sharing and the impact of prefetching.

Acknowledgments

We thank the anonymous reviewers for their constructive and insightful feedback. Kenzo Van Craeynest is supported through a doctoral fellowship by the Agency for Innovation by Science and Technology (IWT). Additional support is provided by the FWO projects G.0255.08 and G.0179.10, the UGent-BOF projects 01J14407 and 01Z04109, and the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295.

References

[1] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. In *HPCA*, pages 7–18, Feb. 2003.

[2] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *SIGMETRICS*, pages 169–180, June 2005.

[3] G. Blake, R. G. Dreslinski, T. N. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In *ISCA*, pages 302–313, June 2010.

[4] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip-multiprocessor architecture. In *HPCA*, pages 340–351, Feb. 2005.

[5] D. Eklöv, D. Black-Schaffer, and E. Hagersten. Fast modeling of cache contention in multicore systems. In *HiPEAC*, pages 147–158, Jan. 2011.

[6] D. Eklöv and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *ISPASS*, pages 55–65, Mar. 2010.

[7] S. Eyerman and L. Eeckhout. System-level performance metrics for multi-program workloads. *IEEE Micro*, 28(3):42–53, May/June 2008.

[8] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *ASPLOS*, pages 175–184, Oct. 2006.

[9] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), held in conjunction with ISCA*, June 2008.

[10] B. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *MICRO*, pages 270–281, Nov. 2008.

[11] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, pages 164–171, Nov. 2001.

[12] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, June 1970.

[13] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, pages 45–57, Oct. 2002.

[14] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for simultaneous multithreading processor. In *ASPLOS*, pages 234–244, Nov. 2000.

[15] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *PACT*, pages 26–34, Sept. 2003.

[16] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Considering all starting points for simultaneous multithreading simulation. In *ISPASS*, pages 143–153, Mar. 2006.

[17] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Representative multiprogram workloads for multithreaded processor simulation. In *IISWC*, pages 193–203, Oct. 2007.

[18] M. Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *ISPASS*, pages 45–56, Mar. 2004.

[19] J. Vera, F. J. Cazorla, A. Pajuelo, L. J. Santana, E. Fernández, and M. Valero. FAME: Fairly measuring multithreaded architectures. In *PACT*, pages 305–316, Sept. 2007.