BADCO: Behavioral Application-Dependent Superscalar Core Model

Ricardo A. Velasquez, Pierre Michaud, André Seznec

BADCO : Behavioral Application-Dependent Superscalar Core Model

Ricardo A. Velásquez
IRISA/INRIA
Rennes, France
ricardo-andres.velasquez@inria.fr

Pierre Michaud
IRISA/INRIA
Rennes, France
pierre.michaud@inria.fr

André Seznec
IRISA/INRIA
Rennes, France
andre.seznec@inria.fr

***Abstract*—Microarchitecture research and development rely heavily on simulators. The ideal simulator should be simple and easy to develop, it should be precise, accurate and very fast. But the ideal simulator does not exist, and microarchitects use different sorts of simulators at different stages of the development of a processor, depending on which is most important, accuracy or simulation speed. Approximate microarchitecture models, which trade accuracy for simulation speed, are very useful for research and design space exploration, provided the loss of accuracy remains acceptable. Behavioral superscalar core modeling is a possible way to trade accuracy for simulation speed in situations where the focus of the study is not the core itself. In this approach, a superscalar core is viewed as a black box emitting requests to the uncore at certain times. A behavioral core model can be connected to a cycle-accurate uncore model. Behavioral core models are built from cycle-accurate simulations. Once the time to build the model is amortized, important simulation speedups can be obtained. We describe and study a new method for defining behavioral models for modern superscalar cores. The proposed Behavioral Application-Dependent Superscalar Core model, BADCO, predicts the execution time of a thread running on a superscalar core with an error less than 10 % in most cases. We show that BADCO is qualitatively accurate, being able to predict how performance changes when we change the uncore. The simulation speedups we obtained are typically between one and two orders of magnitude.**

I. Introduction

Modern high-performance processors have a very complex behavior which reflects the complexity of the microarchitecture and the applications running on it. Models are necessary to understand this behavior and take decisions.

Various sorts of models are used at different stages of the development of a processor, and for different purposes. For instance, analytical models are generally used for gaining insight. Fast performance models are useful in research studies and, in early development stages, for comparing various options. As we take decisions and restrict the exploration to fewer points in the design space, models become more detailed. In general, there is a tradeoff between accuracy and simplicity. A "heavy" model, e.g., a RTL description, gives accurate performance numbers, but requires a lot of work and is not appropriate for research and design space exploration. A "light" model, e.g., a trace-driven performance simulator, can be used for research and exploration but provides approximate numbers. Moreover, it is possible to use different levels of detail for different parts of the microarchitecture, depending on where we focus our attention.

In this study, what we call an application-dependent core model, or *core model* for short, is an *approximate* model of a superscalar core (including the level-1 caches) that *can* be connected to a cycle-accurate uncore model, where the uncore is everything that is not in the superscalar core (memory hierarchy including the L2 cache and beyond, communication network between cores in a multicore chip, etc.). It must be emphasized that a core model is not a complete processor model. A complete processor model provides a global performance number, while a core model emits requests to the uncore (e.g., level-1 cache miss requests) and receives responses to its requests from the uncore. The request latency may impact the emission time of future requests. The primary goal of a core model is to allow reasonably fast simulations for studies where the focus is not on the core itself, in particular studies concerning the uncore.

Core models may be divided in two categories : structural models and behavioral models. Structural core models try to emulate the *internal* behavior of the core microarchitecture. Simulation speedups in this case come from not modeling all the internal mechanisms but only the ones that are supposed to most impact performance.

Behavioral core models try to emulate the *external* behavior of the core, which is mostly viewed as a black box. Unlike structural models, behavioral models are derived from cycle-accurate simulations, which is a disadvantage in some cases. But in situations where model building time can be amortized, behavioral core models are potentially faster and more accurate than structural models. Yet, behavioral core models have received little attention so far.

To the best of our knowledge, the work by Lee et al. is the only previous study that has focused specifically on behavioral superscalar core modeling [1]. They found that behavioral core models could bring important simulation speedups with a reasonably good accuracy. However the cycle-accurate simulator that they used, SimpleScalar *sim-outorder* [2], does not model precisely all the mechanisms of a modern superscalar processor. We present in this paper an evaluation of Lee et al.'s Pairwise Dependent Cache Miss (PDCM) core modeling

This work was partially supported by the European Research Council Advanced Grant DAL No 267175 and a PhD fellowship funded by Region Bretagne

method using the Zesto cycle-accurate simulator, a detailed model of a modern superscalar microarchitecture [3]. We implemented a core model based on the PDCM approach with a reasonably good accuracy. Still, we identified some opportunities to improve the accuracy.

This has led us to propose a new method for behavioral application-dependent superscalar core modeling, BADCO, inspired by but different from PDCM. Unlike PDCM, which uses a single cycle-accurate simulation to build the core model, BADCO uses two cycle-accurate simulations. The first cycle-accurate simulation, identical to the one performed for PDCM, provides timing information for μops when all level-1 (L1) miss requests have a null latency. For the second information, we force a long latency on all L1 miss requests. Unlike PDCM, which uses a structural approach to find the dependences between requests, BADCO infers dependences from the timing information provided by the second cycle-accurate simulation.

The accuracy of BADCO is on average better than that of PDCM on all the configurations we have tested. We have studied not only the ability of BADCO to predict raw performance but also its ability to predict how performance changes when we change the uncore. Our experiments demonstrate a good qualitative accuracy of BADCO, which is important for design space exploration. The simulation speedups we obtained for PDCM and BADCO are in the same ranges, typically between one and two orders of magnitude.

This paper is organized as follows. In Section II, we discuss previous work on core modeling. Section III illustrates the limits of approximate core modeling. Section IV briefly describes PDCM and how we adapted it for the Zesto simulator. We describe the proposed BADCO modeling method in Section V. Section VI presents an experimental evaluation of the accuracy and simulation speed of PDCM and BADCO.

II. Previous work on superscalar core modeling

Trace-driven simulation is a classical way to implement approximate processor models. Trace-driven simulation does not model exactly (and very often ignores) the impact of instructions fetched on mispredicted paths and it cannot simulate certain data mispeculation effects. The primary goal of these approximations is not to speed up simulations but to decrease the simulator development time. A trace-driven simulator can be more or less detailed : the more detailed, the slower. We focus here on modeling techniques that can be used to implement a core model and that can potentially bring important simulation speedups.

A. Structural core models

Structural models speed up superscalar processor simulation by modeling only "first order" parameters, i.e., the parameters that are supposed to have the greatest performance impact in general. Structural models can be more or less accurate depending on how many parameters are modeled. Hence there is a tradeoff between accuracy and simulation speedup.

Loh described a time-stamping method [4] that processes dynamic instructions one by one instead of simulating cycle by cycle as in cycle-accurate performance models. A form of time-stamping had already been implemented in the DirectRSIM multiprocessor simulator [5], [6]. Loh's time-stamping method uses scoreboards to model the impact of certain limited resources (e.g., ALUs). The main approximation is that the execution time for an instruction depends only on instructions preceding it in sequential order. This assumption is generally not exact in modern processors.

Fields et al. used a *dependence graph* model of superscalar processor performance to analyze quickly the microarchitecture performance bottlenecks [7]. Each node in the graph represents a dynamic instruction in a particular state, e.g., the fact that the instruction is ready to execute. Directed edges between nodes represent dependences, e.g., the fact that an instruction cannot enter the reorder buffer (ROB) until the instruction that is ROB-size instructions ahead is retired.

Karkhanis and Smith described a "first-order" performance model [8], which was later refined [9], [10], [11]. Instructions are (quickly) processed one by one to obtain certain statistics, like the CPI (average number of cycles per instruction) in the absence of miss events, the number of branch mispredictions, the number of non-overlapped long data cache misses, and so on. Eventually, these statistics are combined in a simple mathematical formula that gives an approximate global CPI. Recently, a method called *interval simulation* was introduced for building core models based on the first-order performance model [12], [13]. Interval simulation permits building a core model relatively quickly from scratch.

Another recently proposed structural core model, called *In-N-Out*, achieves simulation speedup by simulating only first-order parameters, like interval simulation, but also by storing in a trace some preprocessed microarchitecture-independent information (e.g., longest dependence chains lengths), considering that the time to generate the trace is paid only once and is amortized over several simulations [14].

B. Behavioral core models

Kanaujia et al. proposed a behavioral core model for accelerating the simulation of multicore processors running homogeneous multi-programmed workloads [15] : one core is simulated with a cycle-accurate model, and the others cores mimic the cycle-accurate core approximately.

Li et al. used a behavioral core model to study multicores running heterogeneous multi-programmed workloads [16]. Their behavioral model simulates not only performance but also power consumption and temperature. The core model consists of a trace of level-2 (L2) cache accesses annotated with access times and power values. This per-application trace is generated from a cycle-accurate simulation of a given application, in isolation and assuming a fixed L2 cache size. Then, this trace is used for fast multicore simulations. The model is not accurate because the recorded access times are different from the real ones. Therefore the authors do several multicore simulations to refine the model progressively, the L2 access times for the next simulation being corrected progressively based on statistics from the previous simulation.

In the context of their study, the authors found that 3 multicore simulations were enough to reach a good accuracy.

The ASPEN behavioral core model was briefly described by Moses et al. [17]. This model consists of a trace containing load and store misses annotated with timestamps [17]. Based on the timestamps, they determine whether a memory access is blocking or non-blocking.

Lee et al. proposed and studied several behavioral core models [18], [1]. These models consist of a trace of L2 accesses annotated with some information, in particular timestamps, like in the ASPEN model. They studied different modeling options and found that, for accuracy, it is important to consider memory-level parallelism. Their most accurate model, Pairwise Dependent Cache Miss (PDCM), simulates the effect of the reorder buffer and takes into account dependences between L2 accesses. We describe in Section IV our implementation of PDCM for the Zesto microarchitecture model.

C. Behavioral core models for multi-core simulation

Behavioral core models can be used to investigate various questions concerning the execution of workloads consisting of multiple independent tasks [16], [19]. Once behavioral models have been built for a set of independent tasks, they can be easily combined to simulate a multi-core running several tasks simultaneously. This is particularly interesting for studying a large number of combinations, as the time spent building each model is largely amortized.

Simulating accurately the behavior of parallel programs is more difficult. Trace-driven simulation cannot simulate accurately the behavior of non-deterministic parallel programs for which the sequence of instructions executed by a thread may be strongly dependent on the timing of requests to the uncore [20]. Some previous studies have shown that trace-driven simulation could reproduce somewhat accurately the behavior of *certain* parallel programs [20], [12], and it may be possible to implement behavioral core models for such programs [18], [21]. Nevertheless, behavioral core modeling may not be the most appropriate simulation tool for studying the execution of parallel programs. The rest of this study focuses on single-thread execution.

III. The limits of approximate microarchitecture modeling

The curves on Figure 1 demonstrate the complex behavior of an OoO superscalar core. These curves, one for *h264ref* and one for *libquantum*, were obtained with the Zesto simulator [3] and show the normalized execution time as a function of the L1 miss latency, assuming that the miss latency is uniform and constant. One would expect these curves to be monotonically increasing and convex (see the Appendix) : as the miss latency is increased, there should be more and more misses on the critical path (the chain of dependent events that determines the overall execution time [22]). The curve for *h264ref* is nearly convex, as are the curves for a majority of our benchmarks. However, some benchmarks like *libquantum* have a clearly non-convex curve. This shows that the critical path, though

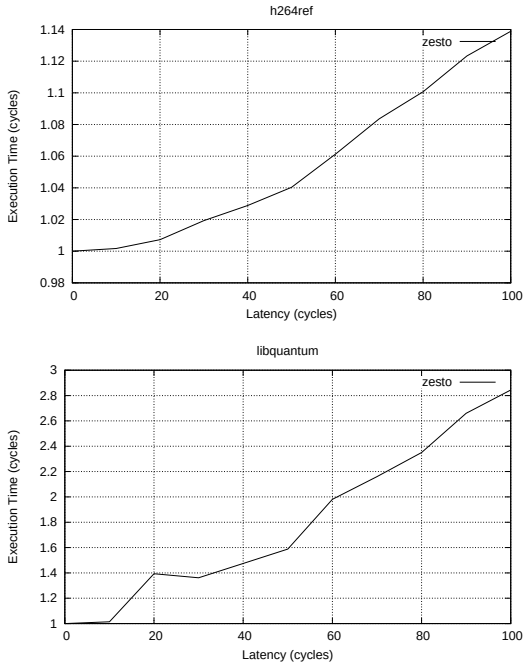


Figure 1. Normalized execution time for *h264ref* and *libquantum* as a function of the L1 miss latency, assuming a constant and uniform miss latency, using the Zesto simulator.

a convenient conceptual tool, does not reflect completely what happens in a OoO microarchitecture. This illustrates the inherent difficulty of defining approximate microarchitecture performance models. The behavior of an OoO core depends on many mechanisms interacting in a complex way and impacting performance. Such complex behavior is difficult to reproduce with a simplified model, be it structural or behavioral. With this limitation in mind, the aim of approximate microarchitecture modeling is to find a good trade-off between simulation accuracy and simulation speed.

IV. The PDCM behavioral model

Lee et al. recently introduced the PDCM behavioral core model [1]. During the model building phase, a per-application trace is generated from a cycle-accurate microarchitecture simulator, assuming an ideal L2 cache, i.e., forcing an L2 cache hit on each L1 cache miss. Each trace item represents an L1 miss. The trace item information contains **(1)** the access type (read, write, instruction, etc.), **(2)** the *instruction delta*, i.e., the number of instructions between this L1 miss and the next L1 miss, **(3)** the *time delta*, i.e., the number of cycles elapsed between this L1 miss and the next L1 miss, and **(4)** a *data dependence*, i.e., on which previous L1 miss this L1 miss depends, directly or indirectly. This data dependence is found by analyzing register and memory dependences during trace generation, taking into account the indirect dependences caused by delayed L1 hits [1].

[1]If an L1 miss Y is data-dependent on a delayed L1 hit which is waiting for a cache line requested by a previous L1 miss X, then Y is considered data-dependent on X [10].

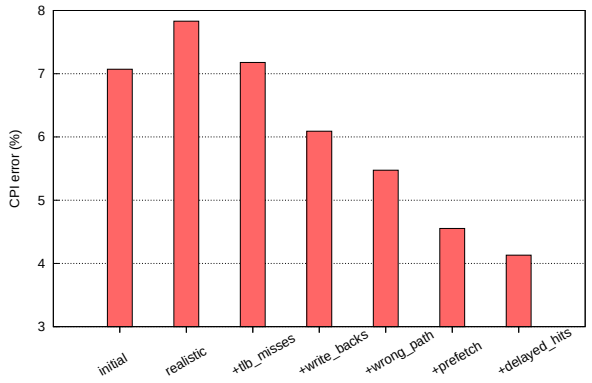


Figure 2. Our efforts to adapt the PDCM method to the Zesto microarchitecture model and decrease the average CPI error.

During the trace-driven simulation, the time deltas and the dependences are used to compute the issue time of each L1 miss. Dependences include both the data dependences and the structural dependences induced by the limited reorder buffer (ROB) and MSHRs. In particular, the instruction deltas are used to simulate the effect of the limited ROB and determine whether or not independent L1 misses can overlap.

It should be noted that PDCM is a behavioral model as the time deltas are obtained from a cycle-accurate microarchitecture simulation. Because the time deltas correspond to an ideal L2 cache, PDCM is very accurate when L2 misses are few.

However PDCM uses a structural approach to model the impact of L2 misses : it is assumed that modeling the effect of the ROB, MSHRs and data dependences is sufficient to reproduce accurately the performance impact of L2 misses. Yet, core resources other than the ROB and MSHRs may impact performance significantly, for instance the limited number of ALUs, L1 cache ports, reservation stations, etc. Even considering an unlimited ROB and MSHRs, the time deltas between consecutive and data-independent L1 misses may depend on the miss latency, e.g., because of resource conflicts happening differently (the miss latency may impact the order in which instructions are executed and how many times instructions are rescheduled), or because a mispredicted branch is data-dependent on an L1 miss.

The original PDCM was tested with SimpleScalar sim-outorder microarchitecture model assuming 100% correct branch predictions [1]. Zesto is more detailed than sim-outorder, and we had to spend substantial effort adapting PDCM for Zesto in order to improve the accuracy. Figure 2 illustrates our efforts. The first bar (leftmost) shows the accuracy obtained with our initial implementation of PDCM, based on what is explicitly described in the original PDCM paper, taking into account the limited MSHRs and assuming a perfect branch prediction. The second bar shows the impact of having a realistic branch predictor and activating hardware prefetchers : unsurprisingly, the accuracy degrades. Then we improved the accuracy, keeping the general principles of the PDCM approach : we have introduced in the model TLB misses (third bar), write backs (fourth bar), wrong-path L1 misses, which we attach to the mispredicted branch (fifth bar), L1 prefetch requests (sixth bar), and more precise modeling of delayed hits (last bar). The numbers shown for PDCM in the remaining of this study were obtained with our optimized version.

V. BADCO : A NEW BEHAVIORAL CORE MODEL

The new behavioral model we propose, BADCO, is inspired from PDCM. However BADCO uses a behavioral method to find dependences between requests to the uncore, unlike in PDCM where an explicit data-dependence analysis is performed. Unlike PDCM which uses a single cycle-accurate simulation to build the core model, BADCO uses two cycle-accurate simulations.

For the first cycle-accurate simulation, we force the latency of each request to zero. This simulation is identical to the one done for PDCM. From this first simulation, we obtain a trace T0. Then we perform a second simulation by giving a long latency to each request. We set the request latency to a value greater than or equal to L, where L is typically greater than the greatest latency that may be experienced when using the core model, e.g., $L = 1000$ cycles. Certain requests have a latency greater than L : we set the latencies so as to force the completion times of successive data requests to be separated by L cycles or more. We obtain from this second simulation a trace TL. Both T0 and TL contain some timing information for each retired μop.

A BADCO model is then built from the information contained in T0 and TL. The information in TL is used to find (direct and indirect) dependences between requests. Dependences include not only data dependences, but also branch mispredictions, limited resources (reservation stations, MSHRs, ...), etc. We do not perform any detailed analysis of these dependences during trace generation. Instead, dependences are found indirectly by analyzing the timing information in TL. We use the fact that, if a request R2 is issued before a previous request R1 is completed, R2 does not depend on R1. If R2 depends only on R1, R2 is often issued a few cycles after R1 completes. That is basically how we detect dependences. Forcing successive requests in TL to occur at intervals no less than 1000 cycles is for disambiguation : R1 is the request whose completion time is closest to the issue time of R2. Of course, this method is not 100 % reliable, but it works well in practice.

A. The BADCO machine

A BADCO machine is an abstract core that fetches and executes *nodes*. A node N_i represent a certain number S_i of retired μops (not necessarily contiguous in sequential order). S_i is the node *size*. The sum of all nodes sizes, $\sum_i S_i$, is equal to the total number of μops executed. As the BADCO machine works on nodes instead of μops, the bigger the nodes, the greater the expected simulation speedup. The next section explains how we build the nodes. A node N_i also has a certain latency in clock cycles, called the node *weight* W_i.

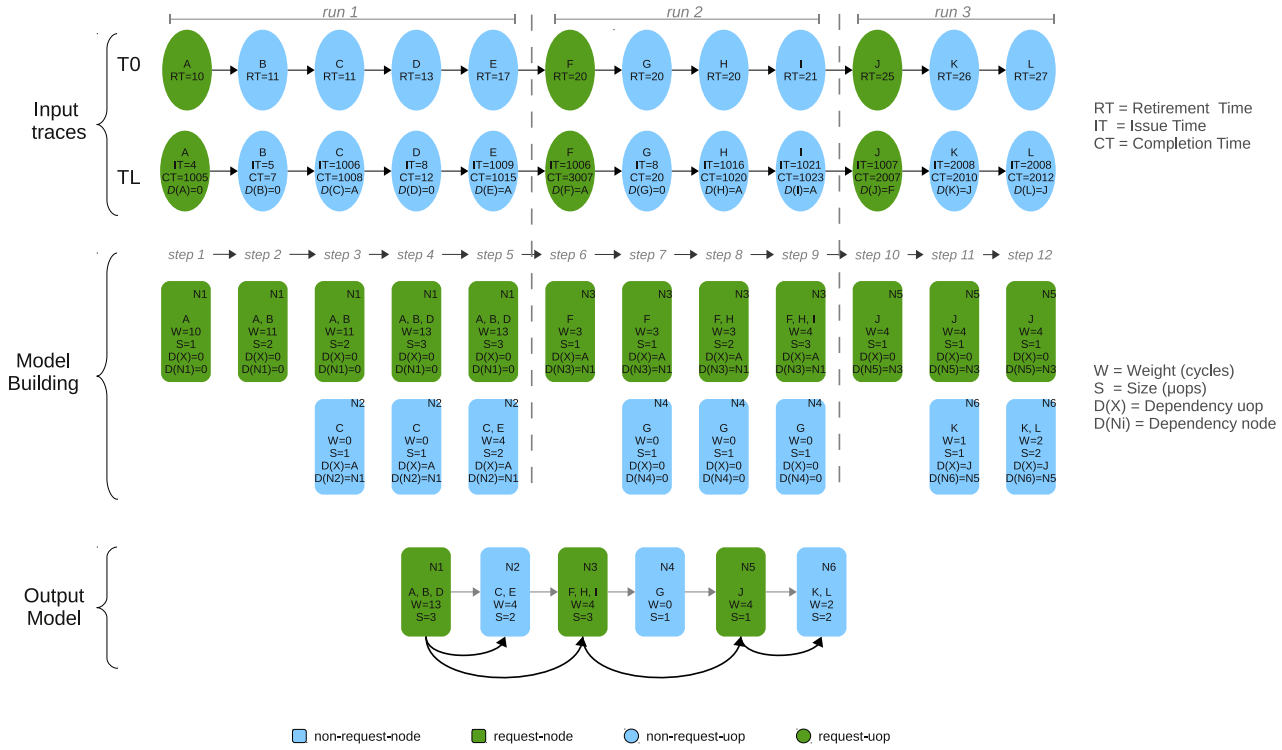


Figure 3. Example of BADCO model building : Input traces T0 and TL containing the same 12 dynamic μops in sequential order at the top, μop-by-μop processing of the traces at the center, and the final BADCO model featuring 6 nodes at the bottom.

Some nodes, called *request nodes*, carry one or several requests to the uncore. There are three sorts of request nodes : I-nodes, L-nodes and S-nodes. An I-node may carry three sorts of requests : IL1 miss, ITLB miss or instruction prefetch requests. An L-node (or S-node) carries the requests attached to one load (or store) μop (DL1 miss, DTLB miss, write-back, DL1 prefetch [2]). An L-node or S-node can also be an I-node. In the BADCO model, a node may be dependent on one older request node, called the *dependency node*.

During the trace-driven simulation, the BADCO machine fetches nodes and inserts them in the BADCO *window* in sequential order. I-nodes send their requests to the uncore at fetch time. Node fetching imitates what the real core does [3]. The BADCO window emulates the real core reorder buffer (ROB). When the sum of nodes sizes inside the window does not exceed the ROB size, the next node can be fetched. Otherwise node fetching is stalled. Once in the window, nodes can start executing. An L-node may send its requests as soon as its dependency node is completed. An L-node is considered completed when all its requests are finished. Other nodes are considered completed when their dependency node is completed. Nodes are retired from the window in the order they were fetched. A node is ready for retirement when it is completed and it is the oldest node in the window. The retirement of a node N_i from the window actually happens exactly W_i cycles after the node is ready for retirement. After being retired from the window, an S-node is sent to a post-retirement store queue, imitating what the real core does with stores. The requests carried by an S-node are issued to the uncore after retirement. The BADCO machine models the occupancy of the MSHRs inside the core. It imitates, to the extent possible, how the real core manages the MSHR. In particular, a request requiring an MSHR entry must wait until there is a free MSHR entry before being sent to the uncore.

[2]We attach a DL1 miss request to the first μop (load or store) accessing that cache line. We attach a DL1 prefetch to the μop triggering the prefetch. We attach a write-back request to the same μop to which the request causing the write-back is attached.

[3]The Zesto model implements next-line prefetching for the instructions, but does not pipeline the instruction misses. Node fetching mimics this behavior.

B. BADCO model building

The BADCO model building phase consists in grouping μops with the same dependencies in nodes, and defining the dependencies among these nodes. Traces T0 and TL provide the information for this process.

Both traces T0 and TL in the top part of Figure 3 represent the same sequence of dynamic μops in program order. The μops in T0 are annotated with their retirement time *"RT"*. The μops in TL are annotated with their issue time *"IT"* and completion time *"CT"*. Some μops carry one or several requests, they are called *request* μops[4]. All other μops are called *non-request* μops. A request μop and the non-request μops following it untill the next request μop form a *run*.

For each μop X, we define its *dependency* μop D(X) as follows : D(X) is the request μop before X and closest to

[4]Each request to the uncore is attached to a single μop

X whose CT is less than the IT of X. For example, μop H in Figure 3 has $IT = 1016$, the closest request μop with $CT < 1016$ is μop A with $CT = 1005$, then $D(H) = A$.

We process traces T0 and TL simultaneously and μop by μop, in lockstep fashion. For each μop, we determine if the μop starts a new node or if it is attributed to an existing node. Every request μop X starting a run creates a new node N_j to which it is attributed. The dependency node $D(N_j)$ of N_j is the node to which D(X) has been attributed. *All subsequent μops attributed to the same node must have the same dependency μop.* In particular, all the μops in the run with the same dependency are attributed to node N_j. If a non-request μop cannot be attributed to any of the nodes already created for that run, we create a new node for the μop.

Attributing a μop to a node N_i means incrementing the node size S_i and adding to the node weight W_i the difference between the retirement time of the μop in T0 and that of the previous μop. By doing so, the sum of all nodes weights, $\sum_i W_i$, equals the total execution time when all the requests to the uncore have a null latency.

The central part of Figure 3 presents step by step the building process of nodes. Step 1 processes μop A; A is a request μop and starts the node N1 with $W = 10$, $S = 1$, and $D(N1) = 0$. Step 2 processes μop B; B is a non-request μop with $D(B) = 0$, and as consequence, it is attributed to N1 with $D(N1) = 0$. The properties of N1 are updated, the size S is incremented, and 1 cycle is added to the weight W because $RT_B - RT_A = 1$. In Step 3, we start a new node N2 for the non-request μop C with $W = RT_C - RT_B = 0$, $S = 1$ and $D(N2) = N1$ (A attributed to N1). The μop C cannot be attributed to the node N1 because all μops in N1 have a null dependency and C depends on A. Steps 5 and 6 attribute μops D and E to nodes N1 and N2 respectively. Step 6 processes the request μop F and starts the processing of the second run of μops. We create a new node N3 with $W = RT_F - RT_E = 3$, $S = 1$ and $D(N3) = N1$ (A attributed to N1). Step 7 processes the non-request μop G; G starts a new node N4 because $D(G) = 0$ and cannot be attributed to N3. Note that G cannot be attributed to N1 either because N1 belongs to the previous run. The building process continues in a similar fashion for the subsequents μops. The bottom part of Figure 3 presents the final BADCO model.

VI. Experimental evaluation

The cycle-accurate simulator used for this study is Zesto [3]. Some of the characteristics of the core and uncore configurations we consider are given in tables I and II respectively. We consider 3 different core configurations : "small", "medium" and "big". The L2, LLC and memory bus each can have a low or high value. This defines up to 8 different uncore configurations. For instance, the configuration denoted "010" has a small L2, a big LLC, and a narrow memory bus. The "big" core is the default core configuration. The default uncore configuration is "001". We will not present results for configurations "100" and "101" since they are not realistic.

core type	**small**	**medium**	**big**
decode/issue/commit	3/4/3	3/5/3	4/6/4
RS/LDQ/STQ/ROB	12/12/8/32	18/18/12/64	36/36/24/128
DL1/DTLB MSHRs	4/2	8/4	16/8
clock	3 GHz		
IL1 cache	2 cycles, 32 kB, 4-way, 64-byte line, LRU, next-line prefetcher		
ITLB	2 cycles, 128-entry, 4-way, LRU, 4 kB page		
DL1 cache	2 cycles, 32 kB, 8-way, 64-byte line, LRU, write-back, IP-based stride + next line prefetchers		
DTLB	2 cycles, 512-entry, 4-way, LRU, 4 kB page		
Branch predictor	TAGE 4 kB, BTAC 7.5 kB, indirect branch predictor 2 kB, RAS 16 entries		

Table I
CORE CONFIGURATIONS. THE DEFAULT CONFIGURATION IS THE "BIG" CORE.

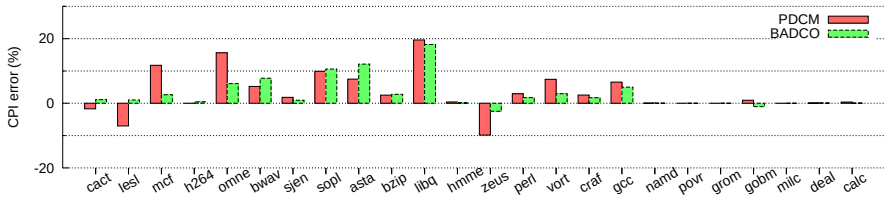
	low ("0")	**high ("1")**
L2 size/latency	256 kB / 6 cycles	1 MB / 8 cycles
LLC size/latency	2 MB / 18 cycles	16 MB / 24 cycles
FSB width	2 bytes	8 bytes
DL1 write buffer	8 entries	
L2	64-byte line, 8-way, LRU, write-back, 8-entry write buffer, 16 MSHRs, IP-based stride + next line prefetchers	
LLC	64-byte line, 16-way, LRU, write-back, 8-entry write buffer, 16 MSHRs, IP-based stride + stream prefetchers	
FSB clock	800 MHz	
DRAM latency	200 cycles	

Table II
UNCORE CONFIGURATIONS. THE L2, LLC AND MEMORY BUS EACH CAN HAVE A LOW OR HIGH VALUE, WHICH DEFINES UP TO 8 DIFFERENT CONFIGURATIONS. FOR INSTANCE, THE CONFIGURATION DENOTED "010" HAS A SMALL L2, A BIG LLC AND A NARROW MEMORY BUS. THE DEFAULT CONFIGURATION IS "001".

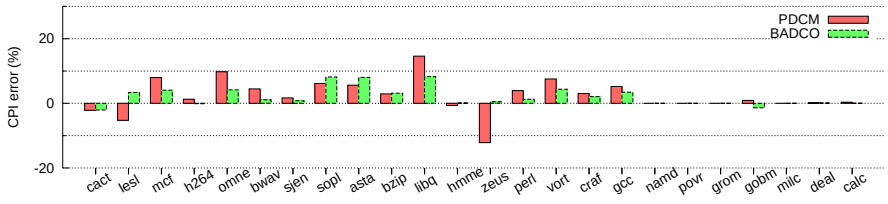
For generating traces T0 and TL, we skip the first 40 billions instructions of each benchmark, and the trace represents the next 100 millions instructions (no cache warming was performed). We assume that simulations are reproducible, so that T0 and TL represent exactly the same sequence of dynamic μops. We used SimpleScalar EIO tracing feature [2], which is included in the Zesto simulation package. We present results for the SPEC CPU2006 benchmarks that we are able to run with Zesto. We have also included two SPEC CPU2000 benchmarks, *vortex* and *crafty*. We have chosen these two benchmarks because they experience a relatively high number of instruction misses and branch mispredictions, which is interesting for testing the models. All the benchmarks were compiled with gcc-3.4 using the "-O3" optimization flag.

A. Metrics

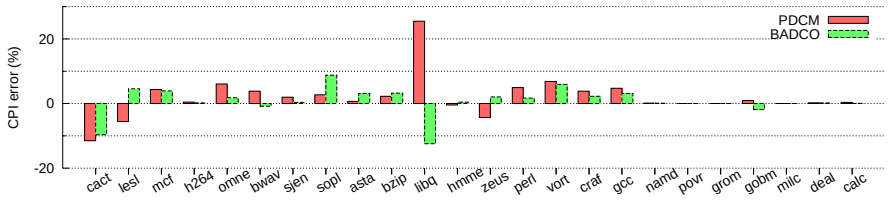
The primary goal of behavioral core modeling is to allow fast simulations for studies where the focus is not on the core itself, in particular studies concerning the uncore. Ideally, a core model should strive for *quantitative* accuracy. That is, it should give *absolute* performance numbers as close as possible to the performance numbers obtained with cycle-



(a) "big" core



(b) "medium" core



(c) "small" core

Figure 4. CPI error of PDCM and BADCO for the "small", "medium" and "big" cores, with the uncore configuration "001".

accurate simulations. Nevertheless, perfect quantitative accuracy is difficult, if not impossible to achieve in general with a simple model.

Yet, *qualitative* accuracy is often sufficient for many purposes. Qualitative accuracy means that if we change a parameter in the uncore (i.e., memory latency), the model will predict accurately the *relative* change of performance. Indeed, if we use behavioral core modeling in a design space exploration for example, more important than being accurate in the final cycle count is being able to estimate relative changes in performance among the different configurations in the design space. Therefore we use several metrics to evaluate the PDCM and BADCO core models. The *CPI error* for a benchmark is defined as

$$\text{CPI error} = \frac{CPI_{ref} - CPI_{model}}{CPI_{ref}}$$

where CPI_{ref} is the CPI (cycles per instruction) for the cycle accurate simulator Zesto, and CPI_{model} is the CPI for the behavioral core model (PDCM or BADCO). The CPI error may be positive or negative. The smaller the absolute value of the CPI error, the more *quantitatively* accurate the behavioral core model. The *average CPI error* is the arithmetic mean of the *absolute value* of the CPI error on our benchmarks set.

For a fixed core, we define the relative performance variation **RPV** of an uncore xyz as

$$RPV = \frac{CPI_{001} - CPI_{xyz}}{CPI_{001}}$$

where CPI_{001} is the CPI of the uncore configuration "001" and CPI_{xyz} is the CPI of uncore configuration xyz (see Table II). The model *variation error* is defined as

$$\text{Variation error} = |RPV_{ref} - RPV_{model}|$$

where RPV_{ref} is the RPV as measured with the cycle-accurate core model and RPV_{model} is the RPV obtained with the behavioral core model (PDCM or BADCO). The smaller the variation error, the more *qualitatively* accurate the behavioral core model. When the variation error is null, it means that the behavioral core model predicts for uncore xyz the exact same performance variation relative to the reference uncore as the cycle-accurate core model. The *average variation error* is the arithmetic mean of the variation error on our benchmarks set.

B. Quantitative accuracy

Figure 4 shows for each benchmark the CPI error of PDCM and BADCO for the "small", "medium" and "big" cores, with the uncore configuration "001". The maximum error is on *libquantum*, both for PDCM and BADCO and for the three core configurations. This is consistent with the non-convex curve of *libquantum* shown in Section III, indicating an inherent modeling difficulty. The table below gives the average CPI error of PDCM and BADCO :

	average CPI error		
	"small"	"medium"	"big"
PDCM	3.8 %	4.0 %	4.7 %
BADCO	3.3 %	2.4 %	2.8 %

BADCO is on average more accurate than PDCM for each of the three core configurations.

C. Qualitative accuracy

Figure 5 shows the Relative Performance Variation (RPV) of Zesto, PDCM and BADCO for the six uncore configurations "000", "010","011", "110" and "111" (see Table II), assuming a "big" core. The baseline uncore is "001".

Both PDCM and BADCO exhibit a reasonably good qualitative accuracy, i.e., they predict approximately how performance changes when we change the uncore. Neither PDCM nor BADCO are very good at predicting tiny performance changes (RPV of a few percents), but they are relatively good at predicting important performance changes. This makes PDCM and BADCO suitable for design space exploration, e.g., for selecting some "interesting" uncore configuration for which more detailed simulations will be done. The table below gives the average variation error of PDCM and BADCO :

	average variation error				
	"000"	"010"	"011"	"110"	"111"
PDCM	4.6 %	4.0 %	1.3 %	4.1 %	1.2 %
BADCO	2.6 %	2.2 %	0.7 %	2.5 %	0.8 %

BADCO is on average more accurate than PDCM for each of the 5 uncore configurations.

D. Simulation speed

We did all the simulation speed measurements on the same machine, which features an Intel Xeon W3550 (Nehalem microarchitecture, 8 MB L3 cache, 3.06 GHz) with Turbo Boost disabled and 6 GB of memory. All the simulation input files, including the traces for PDCM and BADCO, were stored on the local disk of that machine. Zesto, PDCM and BADCO were compiled with gcc-4.1 using the "-O3" optimization flag. We simulated the "big" core configuration and two different uncore configurations : one is the Zesto uncore configuration "001", the other is a simplistic uncore forcing all requests latencies to a null value. With the simplistic uncore, what we measure is essentially the simulation time for the core alone. Figure 6 shows the simulation time in millions of instructions simulated per second for Zesto, PDCM and BADCO.

The simulation speedup achieved with PDCM or BADCO, in comparison with Zesto, is typically between one and two orders of magnitude. Benchmarks with the greatest speedups are the ones with the fewest L1 misses. The table below gives the harmonic mean on our benchmarks of the simulation speed in millions of instructions simulated per second (MIPS) :

	simulation speed (MIPS)		
	Zesto	PDCM	BADCO
with Zesto uncore	0.17	2.91	2.52
core alone	0.19	13.04	8.82

PDCM is generally faster than BADCO because a BADCO nodes represents about 50 μops on average (harmonic mean on our benchmarks), whereas a PDCM trace item represents on average 90 μops. Hence PDCM works at a larger granularity.

The PDCM and BADCO models we have implemented can be connected to a cycle-accurate uncore model. This means that the core does not know the request latency when it sends a request to the uncore. Hence the core model inspects each clock cycle in case an event occurs, which limits the simulation speedup. We believe that higher speedups might be achieved. We are currently investigating the possibility to use information from the uncore that could allow the core model to not inspect every clock cycle.

VII. Conclusion

We introduced BADCO, a new behavioral application-dependent model of superscalar cores. A behavioral core model is like a black box emitting requests to the uncore at certain times. A BADCO model can be connected to a cycle-accurate uncore model for studies where the focus is not the core itself, e.g., design space exploration of the uncore or study of multiprogrammed workloads. A BADCO model is built from two cycle-accurate simulations. Once the time to build the model is amortized, important simulation speedups can be obtained. We have compared the accuracy of BADCO with that of PDCM, a previously proposed behavioral core model. From our experiments, we conclude that BADCO is on average more accurate than PDCM, essentially because it is based on two cycle-accurate simulations instead of a single one for PDCM. With BADCO, the simulated performance error is less than 10% for most of the configurations and benchmarks we have tested. Moreover, we have demonstrated that BADCO offers a good qualitative accuracy, being able to predict how performance varies when we change the uncore.

So far, the simulation speedups we have obtained with BADCO are typically between one and two orders of magnitude compared with Zesto. Nevertheless, we are still working on trying to obtain higher speedups.

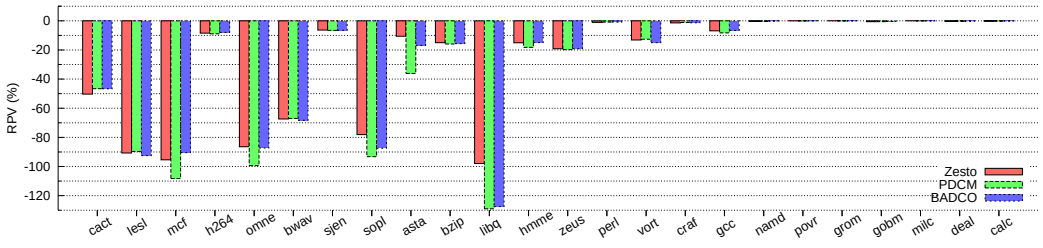
Appendix

Let us assume that the execution of a program by a superscalar processor can be modeled as a graph, where nodes represents certain events and edges represent dependences between events [22], [7]. Each edge is annotated with a latency. Let us assume that requests to the uncore are a subset of the graph edges, and that all the requests have the same latency X. We enumerate all the possible paths (i.e., dependence chains) in the graph and denote N_k the number of requests on path k. The length of path k is

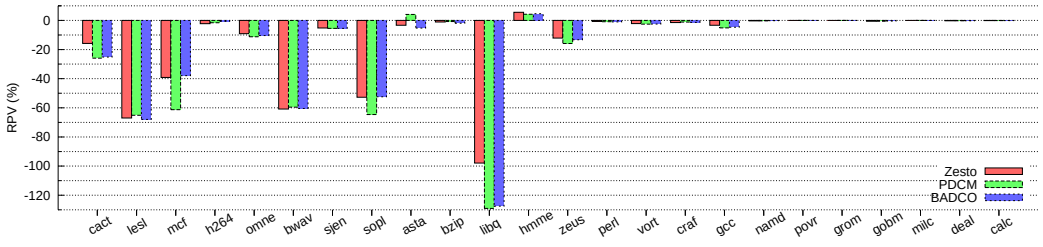
$$T_k(X) = L_k + N_k X$$

and the total execution time is the length of the longest path

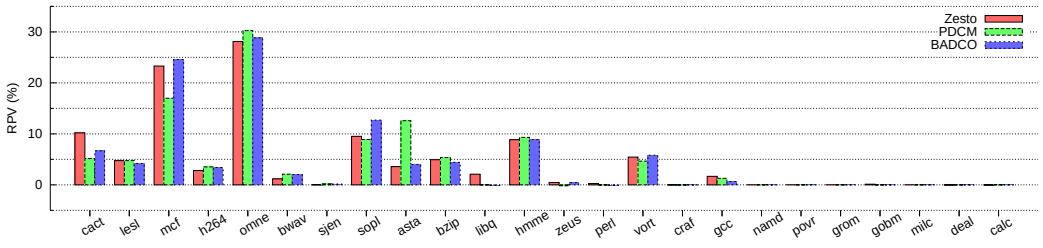
$$T(X) = \max_k T_k(X) = T_{p(X)}(X)$$



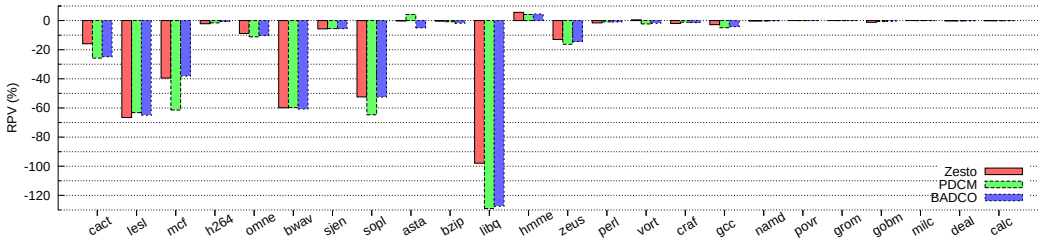
(a) uncore "000"



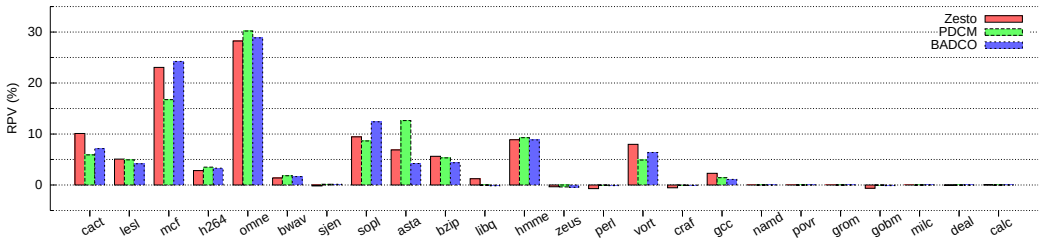
(b) uncore "010"



(c) uncore "011"

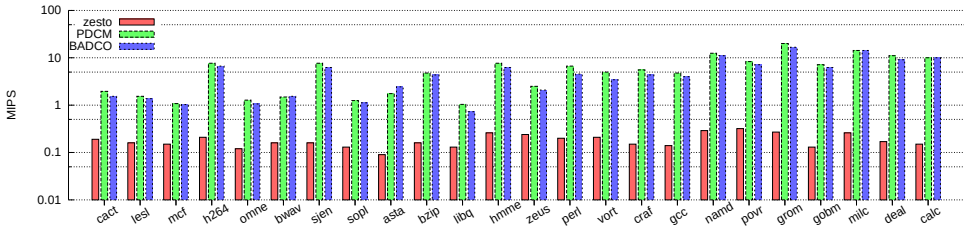


(d) uncore "110"

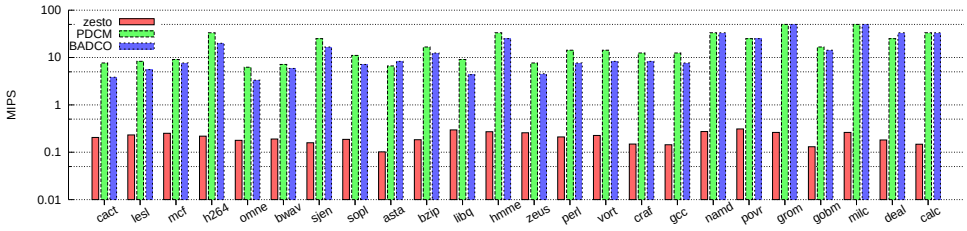


(e) uncore "111"

Figure 5. Relative performance variation (RPV) of Zesto, PDCM and BADCO for the uncore configurations "000", "010","011", "110" and "111", assuming a "big" core. The baseline uncore is "001".



(a) core + Zesto uncore



(b) core alone

Figure 6. Simulation speed in millions of instructions simulated per second (MIPS) with and without considering the impact of the Zesto uncore (logarithmic scale).

where $p(X)$ is the longest path. $N_{p(X)}$ is the slope of $T(X)$ at X. Let us consider $X < Y$. We have

$$T_{p(Y)}(X) \leq T(X)$$
$$T_{p(X)}(Y) \leq T(Y)$$

This implies $(N_{p(Y)} - N_{p(X)})X \leq (N_{p(Y)} - N_{p(X)})Y$, which is possible only if $N_{p(Y)} \geq N_{p(X)}$. The slope of $T(X)$ increases with X, hence $T(X)$ is convex.

REFERENCES

[1] K. Lee, S. Evans, and S. Cho, "Accurately approximating superscalar processor performance from traces," in *Proc. of the Int. Symp. on Performance Analysis of Systems and Software*, 2009.

[2] T. Austin, E. Larson, and D. Ernst, "SimpleScalar : an infrastructure for computer system modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002, http://www.simplescalar.com.

[3] G. Loh, S. Subramaniam, and Y. Xie, "Zesto : a cycle-level simulator for highly detailed microarchitecture exploration," in *Proc. of the Int. Symp. on Performance Analysis of Systems and Software*, 2009.

[4] G. Loh, "A time-stamping algorithm for efficient performance estimation of superscalar processors," in *Proc. of the ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems*, 2001.

[5] M. Durbhakula, V. S. Pai, and S. Adve, "Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors," in *Proc. of the 5th Int. Symp. on High-Performance Computer Architecture*, 1999.

[6] 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," in *Proc. of the 25th Int. Symp. on Computer Architecture*, 1998.

[7] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn, "Using interaction costs for microarchitectural bottleneck analysis," in *Proc. of the 36th Int. Symp. on Microarchitecture*, 2003.

[8] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proc. of the 31st Int. Symp. on Computer Architecture*, 2004.

[9] S. Eyerman, J. E. Smith, and L. Eeckhout, "Characterizing the branch misprediction penalty," in *Proc. of the Int. Symp. on Performance Analysis of Systems and Software*, 2011.

[10] X. E. Chen and T. M. Aamodt, "Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs," in *Proc. of the 41st Int. Symp. on Microarchitecture*, 2008.

[11] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Transactions on Computer Systems*, vol. 27, no. 2, May 2009.

[12] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation : raising the level of abstraction in architectural simulation," in *Proc. of the 16th Int. Symp. on High-Performance Computer Architecture*, 2010.

[13] F. Ryckbosch, S. Polfliet, and L. Eeckhout, "Fast, accurate, and validated full-system software simulation on x86 hardware," *IEEE Micro*, vol. 30, no. 6, pp. 46–56, Nov. 2010.

[14] K. Lee and S. Cho, "In-N-Out : reproducing out-of-order superscalar processor behavior from reduced in-order traces," in *Proc. of the IEEE Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2011.

[15] S. Kanaujia, I. E. Papazian, J. Chamberlain, and J. Baxter, "FastMP : a multi-core simulation methodology," in *Workshop on Modeling, Benchmarking and Simulation*, 2006.

[16] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron, "CMP design space exploration subject to physical constraints," in *Proc. of the 12th Int. Symp. on High Performance Computer Architecture*, 2006.

[17] J. Moses, R. Illikkal, R. Iyer, R. Huggahalli, and D. Newell, "ASPEN : towards effective simulation of threads & engines in evolving platforms," in *Proc. of the 12th IEEE / ACM Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2004.

[18] S. Cho, S. Demetriades, S. Evans, L. Jin, H. Lee, K. Lee, and M. Moeng, "TPTS : a novel framework for very fast manycore processor architecture simulation," in *Proc. of the 37th Int. Conf. on Parallel Processing*, 2008.

[19] L. Zhao, R. Iyer, J. Moses, R. Illikkal, S. Makineni, and D. Newell, "Exploring large-scale CMP architectures using ManySim," *IEEE Micro*, vol. 27, no. 4, pp. 21–33, Jul. 2007.

[20] S. R. Goldschmidt and J. L. Hennessy, "The accuracy of trace-driven simulations of multiprocessors," in *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 1993.

[21] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero, "Trace-driven simulation of multithreaded applications," in *Proc. of the Int. Symp. on Performance Analysis of Systems and Software*, 2011.

[22] B. Fields, S. Rubin, and R. Bodik, "Focusing processor policies via critical-path prediction," in *Proc. of the 28th Int. Symp. on Computer Architecture*, 2001.