BADCO: Behavioral Application-Dependent superscalar Core Models

Ricardo A. Velasquez, Pierre Michaud, André Seznec

▶ To cite this version:

Ricardo A. Velasquez, Pierre Michaud, André Seznec. BADCO: Behavioral Application-Dependent superscalar Core Models. [Research Report] RR-7795, INRIA. 2011, pp.21. hal-00641446

HAL Id: hal-00641446
https://hal.inria.fr/hal-00641446

Submitted on 15 Nov 2011

BADCO: <u>B</u>ehavioral Application-<u>D</u>ependent superscalar <u>CO</u>re models

Ricardo A. Velásquez, Pierre Michaud, André Seznec

BADCO: <u>B</u>ehavioral <u>A</u>pplication-<u>D</u>ependent superscalar <u>CO</u>re models

Ricardo A. Velásquez, Pierre Michaud, André Seznec

Project-Team ALF

Research Report n° 7795 — November 2011 — 21 pages

Abstract: Microarchitecture research and development relies heavily on simulators. The ideal simulator should be simple and easy to develop, it should be precise, accurate and very fast. As the ideal simulator does not exist, 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 detailed 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 <u>b</u>ehavioral <u>a</u>pplication-<u>d</u>ependent superscalar <u>co</u>re model (**BADCO**) predicts the execution time of a thread running on a superscalar core with an error typically under 5%. We show that BADCO is qualitatively accurate, being able to predict how performance changes when we change the uncore. The simulation speedups obtained with BADCO are typically greater than 10.

Key-words: processor, microarchitecture, superscalar core, fast simulation, behavioral model

Modèles comportementales dépend de l'application pour superscalaires cœur

Résumé : La recherche et développement en microarchitecture est en grande partie basée sur l'utilisation de simulateurs. Le simulateur idéal devrait être simple, facile à développer, précis, et très rapide. Comme le simulateur idéal n'existe pas, les microarchitectes utilisent différentes sortes de simulateurs à différentes étapes du développement d'un processeur, en fonction de ce qui est le plus important, la précision ou la vitesse de simulation. Les modèles approchés de microarchitecture, qui sacrifient de la précision afin d'obtenir une plus grande vitesse de simulation, sont très utiles pour la recherche et pour l'exploration d'un espace de conception, pourvu que la perte de précision reste acceptable. La modélisation comportementale de coeur superscalaire est une méthode possible de définition de modèle approché dans les cas où l'objet de l'étude n'est pas le coeur lui-même. Cette méthode considère un coeur superscalaire comme une boite noire émettant des requêtes vers le reste du processeur à des instants déterminés. Un modèle comportemental de coeur peut être connecté à un modèle de hiérarchie mémoire précis au cycle près. Les modèles comportementaux sont construits à partir de simulations détaillées. Une fois le temps de construction du modèle amorti, des gains importants en temps de simulation peuvent être obtenus. Nous décrivons et étudions une nouvelle méthode pour la définition de modèles comportementaux de coeurs superscalaires. La méthode que nous proposons, BADCO, prédit le temps d'exécution d'un programme sur un coeur superscalaire avec une erreur typiquement inférieure à 5%. Nous montrons que la précision d'un modèle BADCO est aussi qualitative et permet de prédire comment la performance change lorsqu'on modifie la hiérarchie mémoire. Les gains en temps de simulation obtenus avec BADCO sont typiquement supérieurs à 10.

Mots-clés : processeur, microarchitecture, coeur superscalaire, simulation rapide, modèle comportemental

Contents

1   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 with 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 : they can be either structural or behavioral. Structural 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 models try to emulate the *external* core behavior : the core is mostly viewed as a black box generating requests to the uncore. In general, and unlike structural models, behavioral models are derived from detailed 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 and are not well understood. 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 [15]. They found that behavioral core models could bring important simulation speedups with a good accuracy. However the detailed simulator that they used as reference, namely SimpleScalar *sim-outorder* [1], does not model precisely all the mechanisms of a modern superscalar processor. We present in Section 3 an evaluation of Lee et al.'s *p*airwise *d*ependent *c*ache *m*iss **PDCM** core modeling method using Zesto, a detailed superscalar processor simulator

[18]. Despite our efforts, the maximum error with PDCM is still significant. This led us to propose a new method for defining *b*ehavioral *a*pplication-*d*ependent superscalar *co*re (**BADCO**) models, inspired by, but different from Lee et al.'s method.

A BADCO model is built from two cycle-accurate traces (*T0* and *TL*). *T0* is generated by forcing a zero latency to every L2 access. The purpose of this trace is to obtain the contribution of each μop to the total cycle count independent from the uncore configuration. *TL* is generated by forcing a latency of L cycles to every L2 access. *TL* is used to extract the data and memory dependencies among μops, and to capture L2 accesses. The timestamps annotated in these traces are used to construct a *coarse-grained dependence graph*. The nodes in the graph define groups of μops. Nodes are annotated with the size in μops, the weight in cycles, and the L2 accesses of its associated μops. The edges define data/memory dependencies between nodes. During simulation, the BADCO model emulates the processor's reorder buffer (ROB) and the level-1 miss status holding registers (MSHRs), and honors dependencies between nodes. BADCO performs a differential processing of instruction-request, load-request and store-request that mimics the real processor.

We have compared the accuracy of BADCO with that of PDCM. Compared to PDCM, BADCO is better able to model the effect of delayed L1 data cache hits. BADCO is on average more accurate than PDCM and can estimate the thread execution time with an error typically under 5%. From our experiments, we found that the maximum error of BADCO is less than half the maximum error of PDCM. 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 very good qualitative accuracy of BADCO, which is important for design space exploration. The simulation speedups obtained with BADCO are typically greater than 10..

This paper is organized as follows. Section 2 discusses previous work on core modeling. Section 3 presents our implementation of Lee et al.'s the (**PDCM**) modeling method using the Zesto simulator. We describe the proposed **BADCO** modeling method in Section 4. Section 5 describes the experimental setup. Finally, Section 6 presents an experimental evaluation of results.

2 Previous work on superscalar core modeling

Trace-driven simulation is a classical way to implement approximate processor models. Trace-driven simulation is approximate because it does not model exactly (and very often ignores) the impact of instructions fetched on mispredicted paths and because 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 in this section on modeling techniques that can be used to implement a core model and that can potentially bring important simulation speedups.

2.1   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 [17] 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 [4, 26]. 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 constraints, e.g., the fact that an instruction cannot be dispatched until the instruction that is ROB-size instructions ahead is retired.

Karkhanis and Smith described a "first-order" performance model [11], which was later refined [6, 2, 5]. Instructions are (quickly) processed one by one to obtain certain statistics, like the CPI 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 performance. The model assumes that limited resources, like the issue width, either are large enough to not impact performance or are completely saturated (in a balanced microarchitecture, this assumption is generally not true [22]). Nevertheless, this model provides interesting insights. Recently, a method called *interval simulation* was introduced for building core models based on the first-order performance model [8, 24]. Interval simulation permits building a core model relatively quickly from scratch.

Another structural core model, called *In-N-Out*, was described recently [14]. 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 dependency chains lengths), considering that the time to generate the trace is paid only once and is amortized over several simulations.

2.2   Behavioral core models

Kanaujia et al. proposed a behavioral core model for accelerating the simulation of the execution of homogeneous multi-programmed workloads on a multicore processor [10] : one core is simulated with a detailed model, and the others cores mimic the detailed core approximately.

Li et al. have used a behavioral core model to simulate the execution of multi-programmed workloads on a multicore [16]. They have shown that behavioral core modeling can be used to simulate power consumption and temperature.

The core model consists of a trace of L2 cache accesses annoted with access times and power values. This per-application trace is generated from a detailed 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 are enough to reach a good accuracy.

The ASPEN behavioral core model was briefly described by Moses et al. [20]. This model consists of a trace containing load and store misses annotated with timestamps [20]. Based on the timestamps, they determine whether a memory access is blocking or non-blocking. ASPEN takes the trace as input and simulates the actual delays and events through a finite state machine that models the states that a thread would go through its execution: idle, ready, executing or stalled.

Lee et al. have proposed and studied several behavioral core models [3, 15]. 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 *pairwise dependence cache miss* model simulates the effect of the reorder buffer and takes into account dependences between L2 accesses. We present in Section 3 a description of our implementation of a PDCM model for the Zesto core.

2.3   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 (e.g., [16, 27]).

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 multi-threaded programs is more difficult. Trace-driven simulation (functional-first simulation in general) cannot simulate accurately the behavior of non-deterministic multi-threaded programs for which the sequence of instructions executed by a thread may be strongly dependent on the timing of requests to the uncore [9]. Some previous studies have shown that functional-first simulation could reproduce somewhat accurately the behavior of *certain* parallel programs (e.g., [9, 8]), and it may be possible to use behavioral core models to model accurately the execution of such programs [3, 23]. Nevertheless, behavioral core modeling may not be the most appropriate simulation tool for studying the execution of multi-threaded programs. The rest of this study focuses on single-thread execution.
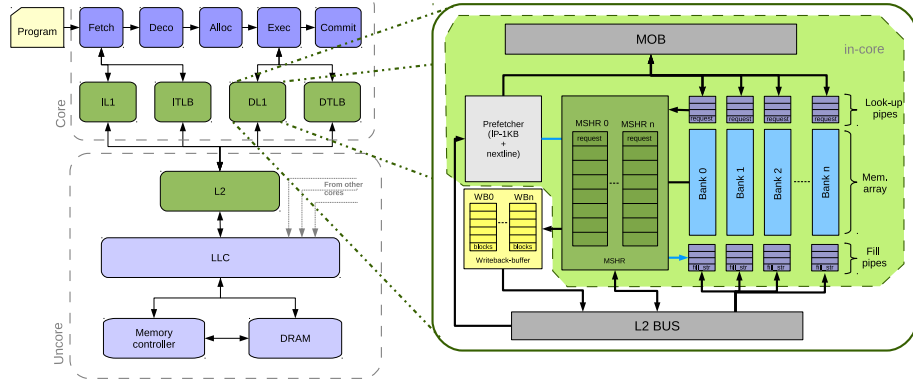
Figure 1: Core and uncore

3 PDCM model as reference

Zesto is a highly detailed x86 cycle-accurate simulator. In particular the memory
hierarchy is more detailed than in SimpleScalar. Figure 1 shows a basic block
diagram of the Zesto CPU specifying the core and uncore parts. What we call
core in this study includes the CPU pipeline stages, the level-1 caches and level-1
TLBs. It also includes the MSHRs and prefetchers, but not the DL1 write-back
buffer. What is not in the core is in the uncore.

The PDCM model was originally implemented and tested with SimpleScalar
sim-outorder [15]. The authors assumed a perfect branch prediction and no
hardware cache prefetchers. We have ported the PDCM strategy to a more
detailed x86 simulator, namely Zesto [18]. Moreover, we have extended PDCM
to support machine configurations with realistic branch predictors and hardware
prefetchers. To obtain with Zesto the accuracy level demonstrated by Lee et al.
on Simplescalar sim-outorder, we had to modify PDCM. We started from the
original PDCM model and added features progressively to improve the accuracy
as much as we could. In this section we make a brief review of the PDCM
simulation kernel and describe the changes introduced to the original model.

3.1 ROB occupancy analysis

The PDCM model is a behavioral core modeling strategy where trace items are
μops with attributed L2 accesses. The trace is generated with a cycle-accurate
simulator modeling a L2 cache having 100% hit rate. Trace items are annotated
with the number of cycles elapsed ($cyc_{elapsed}$) since the previous committed μop,
and the number of μops (μops_{num}) committed in that interval. During trace
generation, the data dependency chains among μops are analyzed to determine
dependencies among trace items. Of all possible dependencies, only the closest
is annotated.

The ROB occupancy analysis simulates the ROB behavior in order to control
which trace items can issue requests to memory. Only the items inside the ROB
can emit requests to memory. A trace item is enqueued into the ROB if its
μops_{num} is less than or equal to the number of free positions in the ROB. Once
a trace item is admitted into the ROB and the annotated item dependency has

been fulfilled, the associated requests can be issued to the memory hierarchy. The item dependencies act as an extra condition to issue memory requests. If an item B depends on another item A, A must complete the processing of all its requests before B can start processing its own. When an item reaches the top of the ROB, their associated μops start committing. The μops commit with a speed of $\mu ops_{num}/cyc_{elapsed}$. An item is committed when all its associated μops have committed and when all its associated requests have been processed.

3.2 PDCM Implementation

In order to adapt PDCM to Zesto, some changes were required. The changes include the kind of requests captured during the trace generation phase. Unlike SimpleScalar, Zesto models the translation of virtual to physical addresses. This includes the modeling of TLBs. Due to the non-negligible effect of I/D-TLB misses, we need to capture this kind of L2 accesses into the trace. We also capture write-back requests and prefetch requests. In addition, the use of realistic branch predictors makes necessary the capture of requests in the wrong path. In this context, we attribute all the requests in a mispredicted path to the mispredicted branch.

As a consequence of the new request types, extra complexity of inter-request dependencies arise. For instance, a trace item with data TLB and read requests must process first the TLB request before issuing the read requests to memory. A similar dependency can be established between read/write/prefetch requests and write-back requests.

Another important change introduced in Lee's model is related to the handling of instruction requests. The PDCM model makes no difference between the handling of instruction and data requests. In our PDCM implementation, trace items process the instruction requests before being enqueued into the ROB. The changes discussed so far concern our implementation of PDCM for the Zesto simulator. The accuracy of PDCM, which we evaluate in Section 6, is reasonably good *on average*. However, despite our efforts, the maximum error can be quite significant. This led us to propose a new modeling method, BADCO.

4 A new behavioral application-dependent superscalar core model

The BADCO approach consists in attaching every μop to a single μop accessing the uncore. We call the μops generating requests to the uncore *requests μops* or simply *r-μops*. We call the μops without requests *non request μops* or *nr-μops* for short. For a given μop A, the *r-μop* B to which the μop A is attached will be called the *uncore parent μop* (*up-μop* for short) of A. A will be called the *uncore child μop* (*uc-μop* for short) of B. In order to perform this attachment, we run a cycle-accurate simulation, forcing a long latency (e.g. 1000 cycles) for each uncore request. Such a long latency ensures that a μop A, posterior to an *r-μop* B is independent from B if A starts executing before B completes. Hence, we attach A to the last *r-μops* that completed before A.

The underlying assumption of our model is that, for any memory latency, the interval between the completion of an *up-μops* and the completion of any
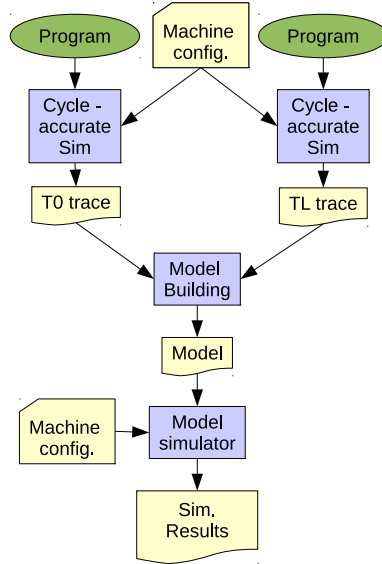
Figure 2: BADCO methodology

of its uc-μops will remain approximately constant.

When the uncore is simulated in conjunction with our model, the uncore provides the timing for the *r-μops* completion. When *r-μops* completes, our model starts issuing the group of its *uc-μops* .

A restriction of our model is that we assume that, independent of the uncore behavior, the same instructions are executed on the wrong path and the same uncore accesses are always performed, including misses generating on the wrong path and prefetch requests.

The model is further detailed below.

4.1   Trace generation

Figure 2 shows an overview of the BADCO methodology. The model building phase requires two detailed simulations. The detailed simulator used for this study is Zesto [18]. For both simulations, we assume an unlimited number of MSHRs [12]. In the first simulation, we force a null latency for all the requests to the uncore (i.e., we assume an ideal L2 cache). In the second simulation, we force a latency of L cycles for all the requests to the uncore, where L is a fixed and arbitrary value. Typically, we take L greater than the longest latency that the core may experience when connected with a cycle-accurate uncore, e.g., 1000 clock cycles.

We have instrumented Zesto to generate a trace [1]. The trace generated by the first and second simulations are called T0 and TL respectively. We assume that simulations are reproducible [2], so that T0 and TL correspond exactly to

---

[1] For generating the trace, we skip the first 40 billions instructions of each benchmark, and the trace represents the next 100 millions instructions. A practical use of the BADCO methodology may use sampling to obtain a representative set of traces [25].

[2] We used SimpleScalar EIO tracing feature [1], which is included in the Zesto simulation package. Other known methods for reproducible simulations include for instance System-

the same sequence of instructions. There is one trace item per retired μop. In T0, each μop is annotated with its retirement cycle. The retirement cycle is used to estimate the cycle contribution of a μop to the global cycle count independent of the uncore configuration. In TL, each μop is annotated with its issue cycle and completion cycle. These timestamps are used to attach *uc-μops* to *up-μops* .

In the TL trace, we also attribute memory access events to μops. By memory access events, we mean instruction misses, data misses, write backs events but also instruction and data TLB misses and prefetch requests. We do not trace the μops on the wrong path. However, an uncore request generated on the wrong path is attached to the first μop on the correct path that uses the block. Moreover we found that due to specific features of the architecture modeled in Zesto, it is necessary to model delayed hits : when several miss requests are pending on the same block, Zesto allocates an entry in the MSHR per request, therefore delayed hits have a non-negligible impact on performance. In order to handle this, delayed hits are also recorded in the trace.

4.2   Model building

BADCO model building is illustrated on an example in Figure 3. Model building starts from traces T0 and TL in Figure 3(a). The final BADCO model is shown in Figure 3(c).

A BADCO model is a *coarse-grained "dependence" graph* (CGDG). A node in such graph represents a set of *uc-μops* with the same *up-μop* . By dependence, we do not mean effective dependence, but the observation of a completion dependence, meaning that, in TL, the *uc-μops* complete after their *up-μop* and before the next *r-μops* .

In a node, we group the *uc-μops* associated with a given *up-μop* that belongs to an interval between two *r-μops* . This node will be simulated as a group. When a node features a *r-μop* , this *r-μop* , will always be the first μop in the node; there is at most one *r-μop* per node.

The node is represented by several parameters. The *node size S* is the number of *uc-μops* represented by the node. The *node weight W* is the sum of the individual contribution of each *uc-μops* to the total cycle count when the total penalty of the node requests is zero, i.e using trace T0. A node *carries* the memory accesses and delayed hits that are attributed to the μops represented by the node.

Nodes carrying requests are called *request nodes* or *r-nodes* for short. Other nodes, called *non-request nodes* or *nr-nodes* can carry one or several delayed hits. A single incoming edge is associated with each node from the *r-node* it depends: it is the node of their *up-μop* .

4.3   Simulation

The simulation kernel consists of three processing stages. The stages mimic the order that a superscalar core uses to process memory requests. Instruction requests are issued to the uncore during the fetch stage, load requests are issued during out-of-order execution, and store requests are issued after commit.
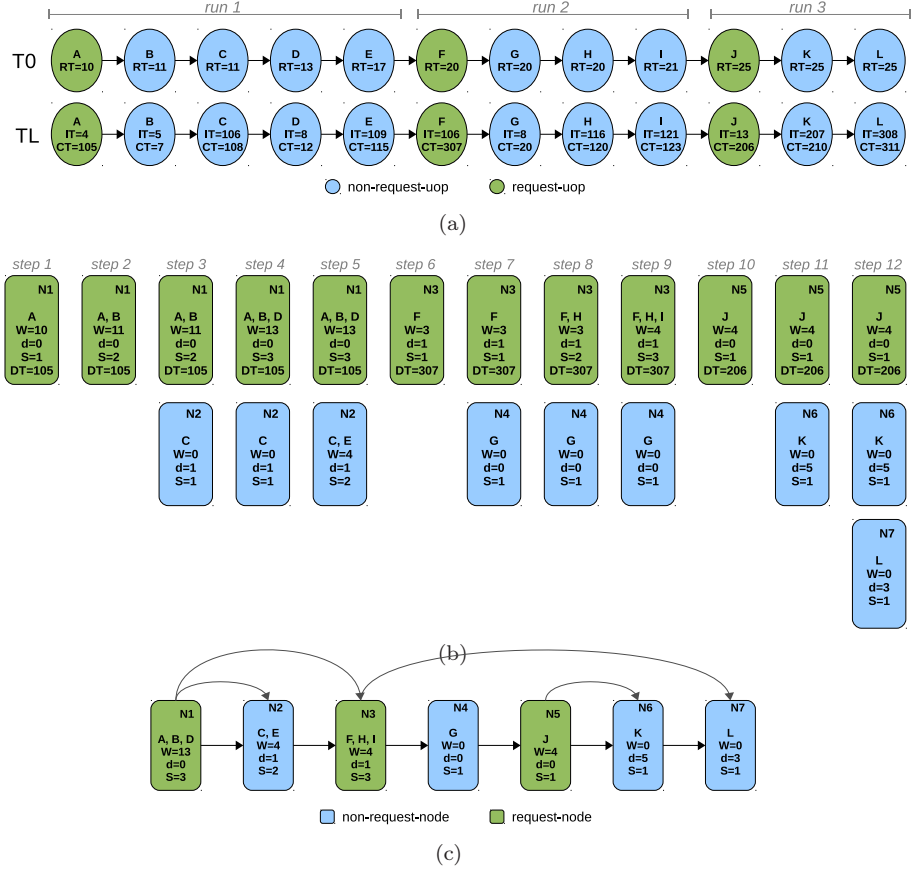

Effect Logs [21].

Figure 3: Example of BADCO model building : (a) Traces T0 and TL containing the same 12 dynamic μops in sequential order, (b) μop-by-μop processing of the traces, (c) final BADCO model featuring 7 nodes (RT=retirement time, IT=issue time, CT=completion time, DT=dependence time, S=node size, W=node weight, d=origin of the dependence edge).

We found that some benchmarks accuracy is very sensitive to the distinction between load requests and store requests.

In-order fetch request processing. The first stage fetches sequentially the nodes from the trace and then processes the instruction requests. When all instruction requests have been processed, the node is ready to be enqueued into the ROB. A fetched node can be enqueued into the ROB, if the number of in-flight μops is smaller than the ROB size. The number of in-flight μops is equal to $\sum_{j}^{x=i} S_x$ where i and j are the IDs of the head and tail nodes inside the ROB, and S_x is the size of the node in μops. If the ROB is full, the fetch stage stalls until the fetched node is enqueued.

Out-of-order load request processing. The second stage is a modified version of the ROB occupancy analysis proposed by Lee et al. [15]. The ROB

size controls which nodes can issue load requests to the uncore. Inside the ROB, the processing of different nodes requests is done out-of-order. Before a node issues requests to the uncore, it must have its dependencies fulfilled. That is, the parent node, i.e., the node from which the dependence edge d originates, must complete the processing of all its requests.

During out-of-order processing, nodes must compete for the limited resources to access the *uncore*. The number of outstanding requests is limited according to the size of DL1-MSHR for DL1-requests and to the size of DTLB-MSHR for DTLB requests.

Nodes are retired from the ROB in order. A node is ready to commit W cycles after all its load requests are completed and actually commits when it has reached the head of the ROB.

Note that W guarantees that, in case a request has a null latency, the execution time is the same as recorded in T0. Retired nodes with pending store requests are enqueued in a post-retirement queue where they remain until all the store request are processed. We call this queue the *store queue*.

In-order store request processing. This stage processes in order the nodes in the *store queue*. Just the node in the head of the queue can issue its requests to the uncore. The store-requests compete with load-request to access the uncore. Note that a node with store-request will complete execution after commit. This is accounted to resolve the dependencies among nodes.

5 Experimental Setup

As explained before, we use the Zesto simulator to obtain the traces T0 and TL for BADCO, and the PDCM trace. Zesto was instrumented to capture all the accesses to the L2 cache and perform the attribution of requests to μops. We also modified Zesto to perform the model simulation. In this context the model simulation uses the cycle accurate simulation of the Zesto uncore. We replaced the code of the core by the simulation kernel of the model: BADCO or PDCM. The integration is totally transparent for the uncore, and it allows to compare the same collected statistics in the L2 cache, the last-level cache (LLC), etc. Zesto is also used as the reference cycle-accurate simulator in all the experiments.

5.1 Machine Model

Table 1 presents the baseline Zesto configuration used during most of the experiments. This configuration is an approximation for the Nehalem microarchitecture. All the configurations use a real branch predictor, data and instruction prefetchers.

Section 6.1 evaluates the quantitative accuracy of PDCM and BADCO models for different ROB sizes. The configuration for such experiment is given in Table 2. The 128 entries ROB configuration is identical to the baseline configuration in table 1. The 64 and 32 entries configuration are derived from the baseline by scaling buffers sizes linearly with the ROB size and by keeping the dispatch, issue and commit width approximately proportional to the square root of the ROB size [19]. Section 6.2 analyzes the qualitative accuracy of PDCM

Decode/issue/commit width	4/6/4
RS/LDQ/STQ/ROB	36/36/24/128 entries
ALU/IMUL/IDIV	3/1/1 units, 1/3/24 cyc. lat.
FADD/FMUL/FDIV/CPLX	1/1/1/1 units, 3/5/32/58 cyc. lat.
IL1 cache	2 cyc.s, 32KB, 4-way, 64B line size, LRU, nextline prefetcher
ITLB	2 cyc., 128 sets, 4-way, LRU
DL1 cache	2 cyc., 32KB, 8-way, 64B line size, 8 entries MSHR, 8 entries WB-buffer, LRU, write-alloc., write-back, IP + nextline prefetchers
DTLB	2 cyc., 256 sets, 4-way, LRU, 4 entries MSHR
Branch direction pred.	TAGE 4KB
Branch target pred.	BTAC 7.5KB
Indirect branch target pred.	2LEVBTAC 2KB
Return address pred.	Stack 16 entries

Table 1: Baseline Machine Setup

and BADCO models changefor 5 different uncore configurations. Those configurations are listed in Table 3. Configuration 1 is the uncore baseline. For all configurations, the L2 cache has the following features :8 way associative, 8 entries MSHR, 8 entries write-back buffer, LRU replacement policy and 2 prefetchers (IP + nextline). The LLC has the following features : 16 way associative, 16 entries MSHR, 8 entries write-back buffer, LRU replacement policy and 2 prefetchers (IP + stream).

parameter	rob=128	rob=64	rob=32
rs_size	36	18	12
ldq_size	36	18	12
stq_size	24	12	8
dl1_mshrs	16	8	4
dtlb_mshrs	8	4	2
decode_width	4	3	3
alloc_width	4	3	3
exec_width	6	5	4
commit_width	4	3	3

Table 2: Microarchitecture setup for ROB sizes of 128, 64 and 32 entries

parameter	cfg1	cfg2	cfg3	cfg4	cfg5	cfg6
L2_size(KB)	256	1024	256	256	1024	256
L2_lat(cyc)	2	8	2	2	8	2
LLC_size(MB)	2	16	2	2	16	2
LLC_lat(cyc)	12	24	12	12	24	12
LLC_BW(B/cyc)	64	64	4	64	64	4
fsb_width(B)	8	8	2	8	8	2
dram_lat(cyc)	200	200	200	500	500	500

Table 3: Uncore configurations

bench.	*input*	*bench.*	*input*
cactusADM	benchADM.par	leslie3d	leslie3d.in
mcf	inp.in	h264ref	fore. ref base.
omnetpp	omnetpp.ini	bwaves	bwaves.in
sjeng	ref.txt	soplex	ref.mps
astar	BigLakes2048	bzip2	input.source
libquantum	ref	hmmer	nph3.hmm
zeusmp	zmp_inp	perlbench	diffmail
vortex	lendian1.raw	crafty	crafty.in
gcc	200.i		

Table 4: Inputs used for SPEC benchmarks

5.2 Benchmarks

For all the experiments, we use 15 of the 29 SPEC CPU2006 benchmarks and 2 SPEC CPU2000 benchmarks (vortex and crafty). The criterion to select the benchmarks subset was that each benchmark must run in Zesto without problems for the duration of the simulation, with at least one of the reference input datasets. 22 CPU2006 benchmarks fulfilled the requirement. We also exclude from the statistics 7 benchmarks, namely calculix, namd, gromacs, gobmk, dealII, milc and povray, for the reason that they are mostly CPU bound, with very few level-1 misses. Actually, BADCO models these 7 benchmarks performance with excellent accuracy ($< 0.5\%$) and provides very high simulation speedups ($> 100x$), which is not surprising. We added vortex and crafty in our list of benchmarks because they experience a relatively high number of instruction misses and branch mispredictions, which is interesting for testing the model. All benchmarks were compiled using GCC-3.4 and optimization flag -O3. The benchmarks that we have used for the statistics are listed in Table 4. We have used Simplescalar EIO traces (also featured in Zesto) to obtain deterministic simulations. The EIO traces have been generated by skipping the first 40 Billion instructions and simulating the next 100 Million instructions. No cache warming was done.

5.3 Metrics

As noted previously, the primary goal of behavioral core modeling is to allow reasonably 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-accurate core simulations. Nevertheless, perfect quantitative accuracy is difficult, if not impossible to achieve in general. 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 configuration in the design space. We use several metrics to evaluate the PDCM and BADCO core models :

CPI error.   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 is proportional to the cycle count difference between Zesto and the behavioral model. The smaller the CPI error in absolute value, the more *quantitatively* accurate the behavioral core model. The *average CPI error* is the arithmetic mean of the *absolute value* of the CPI error for the benchmarks listed in Table 4. The CPI error may be positive or negative, while the *average CPI error* is always positive.

Relative performance variation and variation error   . The relative performance variation (**RPV**) of an uncore configuration $cfgX$ is defined as

$$RPV = \frac{CPI_{cfg1} - CPI_{cfgX}}{CPI_{cfg1}}$$

where CPI_{cfg1} is the CPI of the baseline uncore configuration $cfg1$ (see Table 3) and CPI_{cfgX} is the CPI of uncore configuration $cfgX$. A positive RPV means that we have increased the performance compared to the baseline uncore configuration. The model *variation error* for configuration $cfgX$ is defined as

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

where RPV_{ref} is the RPV as measured with the Zesto cycle-accurate core 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 configuration $cfgX$ the exact same change in performance compared to the baseline as the cycle-accurate core model. The *average variation error* is the arithmetic mean of the variation error on all the benchmarks listed in Table 4 and on all the uncore configurations listed in Table 3 but $cfg1$. The *maximum variation error* is the maximum of the variation errors on all benchmarks and all uncore configurations.

6 Experimental evaluation

We evaluate the PDCM model and our BADCO model in two aspects. The first is the CPI error for different core microarchitectures, and the second is the relative performance variation due to changes in the uncore.

6.1 Quantitative accuracy

Figure 4 shows the CPI error for PDCM and for BADCO. The CPI error is presented for 3 different ROB sizes : 32, 64 and 128 entries. For PDCM, the average CPI error is 8.1%, 6.1% and 6.6% for a ROB size of 32, 64 and 128 entries respectively. On the other hand, the average CPI error for BADCO is 4.7%, 4.9% and 3.8% respectively. The maximum CPI error we have observed with PDCM is 36.3% (on libquantum with a 32-entry ROB). For BADCO, the maximum CPI error is 15.9% (on zeusmp with a 32-entry ROB). BADCO is more accurate than PDCM not just in average error but also in maximum error.

The PDCM model seems to be more accurate for benchmarks with a high misprediction rate, such as perlbench, gcc and crafty. The reason for this may be the way requests in the wrong path are attached to μops. PDCM captures all the requests in the wrong path and attaches them to the mispredicted branch. On the other hand, BADCO captures only the requests on the wrong path that are actually reused by the correct path. For instance, BADCO overestimates the performance on gcc, which may indicate that pollution in the L2 cache (and perhaps in the LLC) is significant for gcc [3].

On the other hand, BADCO is clearly more accurate than PDCM on libquantum and hammer. These two benchmarks have as common characteristic that they exhibit a huge number of delayed hits in the L1 data cache. PDCM does not model correctly the impact of delayed hits. Indeed, the null miss penalty for uncore requests during trace generation prevents the capture of all possible delayed hits that may occur in a long latency penalty. BADCO overcomes this problem because the requests (including delayed hits) are captured in trace TL, for which the request latency was set to a high value.

6.2 Qualitative accuracy

A comparison among the RPVs of Zesto, PDCM model and BADCO model is presented in figure 5 for five different uncore configurations, and for a 128-entry ROB. The uncore configurations are listed in Table 3. Configuration $cfg1$ is used as the reference, i.e. all changes in performance are relative to this baseline configuration. It must be noted that both PDCM and BADCO predict the sign of the performance change almost perfectly. For PDCM, the average variation error is 6.3% and the maximum variation error is 53.8%. On the other hand, for BADCO, the average variation error is 2.8% and the maximum variation error is 16.1%. For both models, we observed the worst accuracy on configurations $cfg3$ and $cfg6$, which both have a small L2, a small LLC, and a small memory bandwidth. These experiments demonstrate the power of behavioral core models for estimating changes in performance for various uncore configurations.

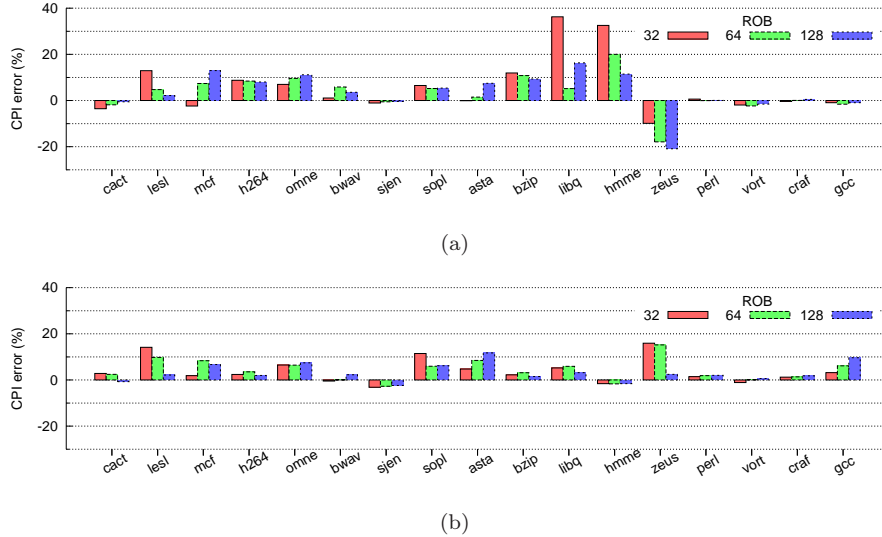[3] Some solutions for decreasing cache pollution have been proposed [13].

(a)

(b)

Figure 4: CPI error for a ROB size of 32, 64 and 128 entries : (a) PDCM model, (b) BADCO model.

6.3   Simulation speed

The main purpose of a behavioral core model is to accelerate simulations. We measure simulation speedups with the help of a profiling tool. First we measure the fraction of the total simulation time corresponding to the core in the cycle-accurate simulation. Then we measure the fraction of the total simulation time corresponding to the BADCO kernel in the BADCO simulation. The *core* simulation speedup is the ratio of these two values. We are still working on optimizing the BADCO kernel. Nevertheless, in its current state we observe core simulation speedups in the range of 3.2x to 42.5x. The average speedup is 17.2x.

It is also possible to define a global speedup, i.e., the total simulation time with Zesto divided by the total simulation time with the BADCO kernel. Of course, the global speedup depends on the uncore. When including the contribution of the Zesto uncore in the simulation time, the average speedup is 13.6x.

7   Conclusion

We introduced BADCO, a new behavioral application-dependent model of a modern superscalar core. 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. Compared to PDCM, BADCO

is better able to model the effect of delayed L1 data cache hits. BADCO is on average more accurate than PDCM and can estimate the thread execution time with an error typically under 5%. From our experiments, we found that the maximum error of BADCO is less than half the maximum error of PDCM. 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 very good qualitative accuracy of BADCO, which is important for design space exploration. The simulation speedups obtained with BADCO are typically greater than 10.

References

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar : an infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002. http://www.simplescalar.com.

[2] X. E. Chen and T. M. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. In *Proceedings of the 41st International Symposium on Microarchitecture*, 2008.

[3] 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 *Proceedings of the 37th International Conference on Parallel Processing*, 2008.

[4] M. Durbhakula, V. S. Pai, and S. Adve. Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, 1999.

[5] 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*, 27(2), May 2009.

[6] S. Eyerman, J. E. Smith, and L. Eeckhout. Characterizing the branch misprediction penalty. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2011.

[7] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *Proceedings of the 36th International Symposium on Microarchitecture*, 2003.

[8] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation : raising the level of abstraction in architectural simulation. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, 2010.

[9] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1993.

[10] S. Kanaujia, I. Esmer Papazian, J. Chamberlain, and J. Baxter. FastMP : a multicore simulation methodology. In *Workshop on Modeling, Benchmarking and Simulation*, 2006.

[11] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st International Symposium on Computer Architecture*, 2004.

[12] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, 1981.

[13] C. J. Lee, H. Kim, O. Mutlu, and Y. N. Patt. Performance-aware speculation control using wrong path usefulness prediction. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, 2008.

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

[15] K. Lee, S. Evans, and S. Cho. Accurately approximating superscalar processor performance from traces. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2009.

[16] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. CMP design space exploration subject to physical constraints. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, 2006.

[17] G. Loh. A time-stamping algorithm for efficient performance estimation of superscalar processors. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2001.

[18] G. Loh, S. Subramaniam, and Y. Xie. Zesto : a cycle-level simulator for highly detailed microarchitecture exploration. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2009.

[19] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *pact*, page 2. Published by the IEEE Computer Society, 1999.

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

[21] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2006.

[22] D. B. Noonburg and J. P. Shen. Theoretical modeling of superscalar processor performance. In *Proceedings of the 27th International Symposium on Microarchitecture*, 1994.

[23] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero. Trace-driven simulation of multithreaded applications. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2011.

[24] F. Ryckbosch, S. Polfliet, and L. Eeckhout. Fast, accurate, and validated full-system software simulation on x86 hardware. *IEEE Micro*, 30(6):46–56, November 2010.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[26] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ILP processors. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[27] L. Zhao, R. Iyer, J. Moses, R. Illikkal, S. Makineni, and D. Newell. Exploring large-scale CMP architectures using ManySim. *IEEE Micro*, 27(4):21–33, July 2007.
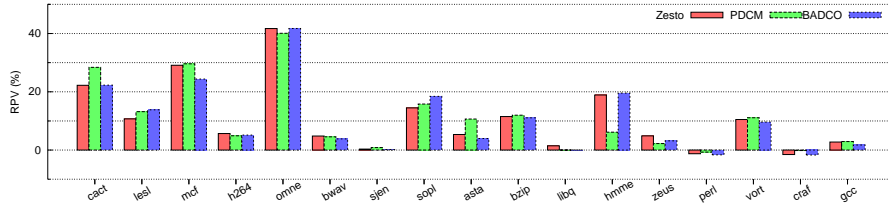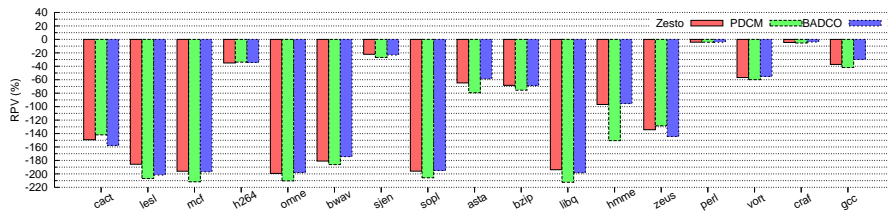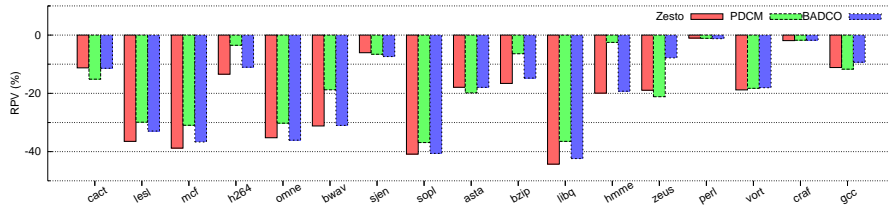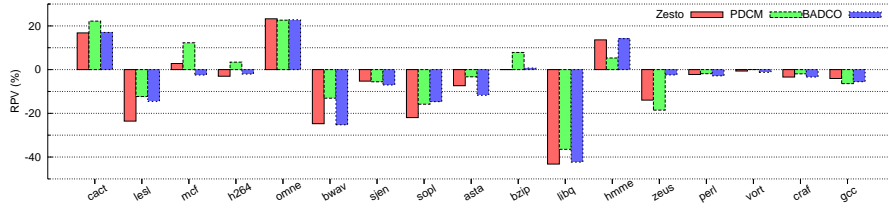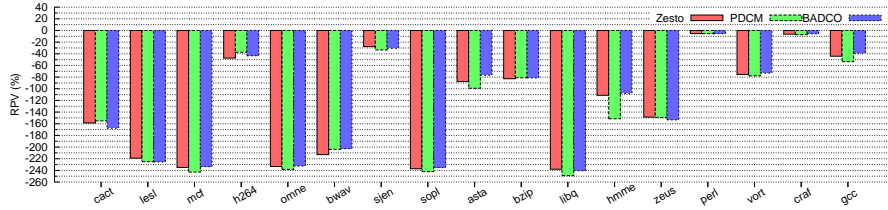
Figure 5: Relative performance variation (RPV) of Zesto, PDCM and BADCO for uncore configurations in Table 3 (ROB size is 128, baseline uncore is cfg1) : (a) configuration cfg2, (b) configuration cfg3, (c) configuration cfg4, (d) configuration cfg5, and (e) configuration cfg6.