Sharing the Instruction Cache Among Lean Cores on an Asymmetric CMP for HPC Applications

Ugljesa Milic[*][†], Alejandro Rico[‡], Paul Carpenter[*] and Alex Ramirez[§]

[*]Barcelona Supercomputing Center
[†]Universitat Politècnica de Catalunya
{first.last}@bsc.es

[‡]ARM Inc.
alejandro.rico@arm.com

[§]Google
alexramirez@google.com

***Abstract*—High performance computing (HPC) applications have parallel code sections that must scale to large numbers of cores, which makes them sensitive to serial regions. Current supercomputing systems with heterogeneous or asymmetric CMPs (ACMP) combine few high-performance big cores for serial regions, together with many low-power lean cores for throughput computing. The low requirements of HPC applications in the core front-end lead some designs, such as SMT and GPU cores, to share front-end structures including the instruction cache (I-cache). However, little work exists to analyze the benefit of sharing the I-cache among full cores, which seems compelling as a solution to reduce silicon area and power.**

This paper analyzes the performance, power and area impact of such a design on an ACMP with one high-performance core and multiple low-power cores. Having identified that multiple cores run the same code during parallel regions, the lean cores share the I-cache with the intent of benefiting from mutual prefetching, without increasing the average access latency. Our exploration of the multiple parameters finds the sweet spot on a wide interconnect to access the shared I-cache and the inclusion of a few line buffers to provide the required bandwidth and latency to sustain performance. The projections with McPAT and a rich set of HPC benchmarks show 11 % area savings with a 5 % energy reduction at no performance cost.

I. Introduction

High performance computing (HPC) needs more energy-efficient designs to reach the Exascale milestone. Future chip multiprocessors (CMP) used in HPC have to increase performance per power and area unit by exploiting intrinsic characteristics of HPC code.

Running mostly in parallel, HPC applications favor throughput-oriented CMPs. Relying on the abundant TLP, and for a given area or power budget, it is more beneficial to implement many low-power cores instead of a few heavyweight ones, as in case of Intel's Xeon Phi [1] and IBM's BlueGene/Q [2] architectures. Still, with more cores on a chip and increasing available TLP, the sequential part of the code eventually becomes the bottleneck [3]. To address this problem, HPC clusters usually employ a latency-optimized CMP to handle serial code and an accelerator (GPU or many-core) to run parallel sections. More tightly coupled, recent proposals considered an Asymmetric CMP (ACMP) architecture with at least one high-performance core for running in sequential and many lean cores for throughput computing [4], [5].

The differences between HPC workloads and typical desktop applications lead to a different core front-end design and sizing. Heavyweight cores support large instruction footprints and complex branch behavior with private instruction caches (I-cache) and sophisticated branch predictors. On the other hand, HPC applications have small(er) code footprint, long(er) basic blocks, and (more) predictable branches [6]. Moreover, all parallel threads in HPC applications execute the same code approximately at the same time. This makes sharing the core front-end structures a potentially beneficial solution. Simultaneous multithreading (SMT) [7], GPU [8], and AMD's Bulldozer [9] cores provide shared front-end structures to their running threads to increase efficiency.

In this paper, we evaluate an ACMP design with a private L1 I-cache for the high-performance core and a shared I-cache for the set of lean cores. The potential benefits of sharing the I-cache across multiple cores start with savings in chip area and static power, and they extend to improved I-cache hit rates due to constructive cross-thread instruction prefetching. Potential drawbacks include increased I-cache access latency and contention due to the shared access interconnection network. We study different parameters to explore the limitations and find an optimal design considering performance and hardware cost.

The contributions of this work are the following:

1) We characterize 24 workloads from three HPC benchmark suites distinguishing serial and parallel code regions. Parallel parts of the code have longer basic blocks compared with serial ones and the number of misses in a standard-size 32 KB I-cache is negligible. These findings motivate tailoring an ACMP design for HPC by sharing a smaller I-cache among worker threads and leaving a standard-size I-cache private for the master thread.
2) To reduce the congestion and access latency to a shared I-cache, we analyse the tradeoff in increasing the bandwidth of the interconnect and adding more line buffers per core. We find that increasing the bandwidth of an interconnect is more beneficial than adding line buffers.
3) We estimate the performance, area, and energy savings of our proposal using McPAT [10]. For ACMPs built from one big and eight lean cores, we measure 11 % savings in area and 5 % in energy, without performance loss. In cases where initial I-cache miss ratio is high, we even observe a performance improvement due to code prefetching among lean cores that share the I-cache.

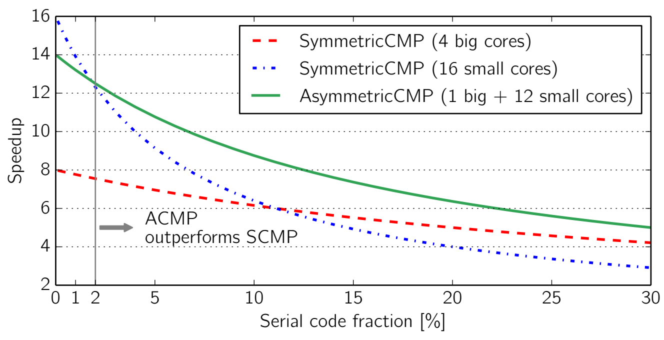


Fig. 1: Potential speedup obtained by different CMP designs depending on the serial code fraction. One big core spends $4\times$ more resources for $2\times$ more performance compared with a small core.

4) With the large core executing both serial and parallel code, we study a configuration where a single I-cache is shared among all cores. Although attractive from the perspective of additional area and power savings, we show that such a front-end organization degrades performance, especially as the serial code fraction increases.

To the best of our knowledge, this is the first paper exploring different aspects of a shared I-cache among cores based on the specific characteristics of HPC workloads from a performance, area, and energy perspective.

II. Motivation

A. The advantage of an ACMP design

For any parallel application, the serial code fraction limits the speedup over the sequential execution [3]. There is a need to efficiently execute both parallel and sequential parts of the code with an appropriate CMP design. To do that, previous work suggested an Asymmetric CMP design, where multiple single-ISA cores exist on a chip, but with different power, area, and performance characteristics [4]. Serial code can thus be executed on a single heavyweight core, while many lean cores can execute the parallel code sections.

Figure 1 shows the potential speedup that different CMP designs can provide depending on the serial code fraction for a parallel workload. The cost model for multicore chips, assumptions about core performance, constant cache and interconnect cost, are all taken from previous work [4]. The CMP designs we present on the figure have the same hardware budget, equal to 16 base core equivalents or small cores. We compare two symmetric CMPs, one with four big and one with 16 small cores, to an asymmetric CMP with one big and 12 small cores. With the serial code fraction above 2 %, an ACMP outperforms both symmetric CMP designs. As the number of cores on a chip increases, the amount of time spent inside the serial code becomes larger. An ACMP design stands out as a solution capable of efficiently executing both parallel and sequential code regions, combining a latency-oriented core with a set of throughput-oriented cores.

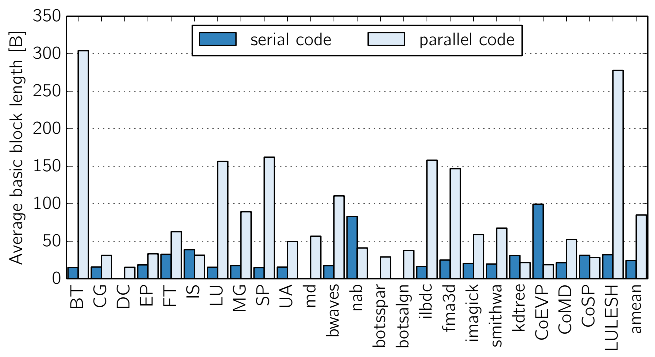


Fig. 2: The average dynamic basic block length in serial and parallel parts of the code.

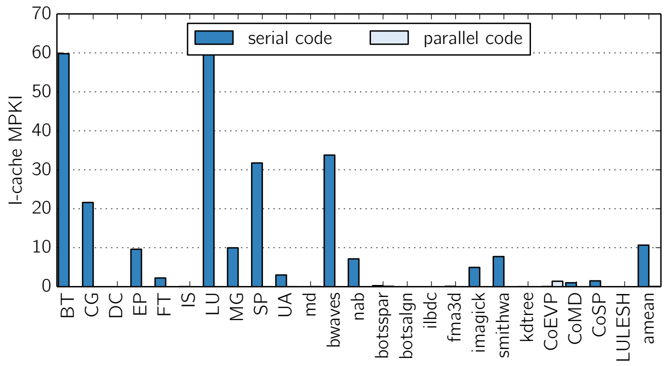


Fig. 3: The I-cache MPKI values in serial and parallel parts of the code using a 32 KB, 8-way associative I-cache with 64 B lines, and LRU replacement policy. The I-cache MPKI values in parallel code are very low.

B. The difference between sequential and parallel code

On an ACMP, the large core executes sequential code and it joins the workers executing parallel code regions. Using Pin [11] as an instrumentation library, we instrument only the master thread and characterize the HPC applications separating the serial and parallel sections by looking at the average basic block size and the I-cache MPKI values.

Figure 2 shows the average dynamic basic block size for each workload we use in our evaluation[1]. HPC applications have $3\times$ longer basic blocks in parallel than in sequential code. This means that HPC benchmarks, while executed in parallel, provide high usefulness of the I-cache lines, increasing the fetch bandwidth without any techniques such as trace cache or multiple branch prediction per cycle [12], [13]. A single I-cache line fetched inside the parallel region contains more instructions to feed the core back-end than an I-cache line from a serial region. Still, there are benchmarks, such as *nab* and *CoEVP*, where basic blocks are longer in serial sections. We will refer to these interesting cases later in Section VI-E.

Figure 3 gives the I-cache MPKI values for each benchmark obtained in serial and parallel code regions. Not just that

[1]We describe the benchmarks and the input sets in Section V-C

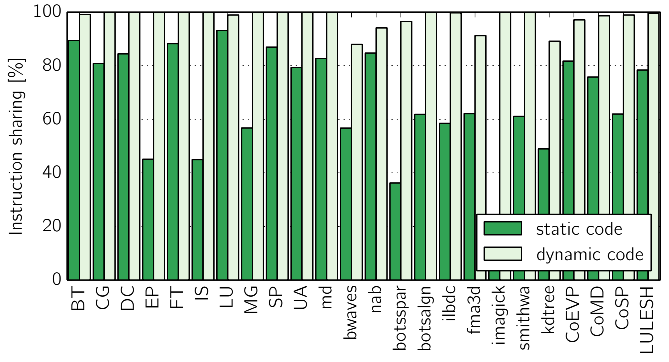


Fig. 4: Percentage of instruction sharing across all threads running on an eight-core CMP per HPC benchmark (parallel sections only).

sequential code sections miss more in a standard-size 32 KB I-cache, but parallel code sections have I-cache MPKI values far below 1 (except for *CoEVP*). HPC applications spend most of their time inside loops, so few basic blocks are fetched over and over again, resulting in a few I-cache misses.

These findings point out the difference between sequential code executed by the large, master core and parallel code executed by all cores. With its aggressive back-end and short basic blocks, the large core needs quick access to the instruction memory to deliver enough instructions every cycle. On the other hand, lean cores have less demanding back-ends and $3\times$ longer basic blocks, so a prolonged I-cache access latency is less likely to introduce additional stall cycles. Moreover, parallel code sections have negligible I-cache MPKI.

C. Lean cores and the code they execute

The total area of a lean core is small, so private core front-end structures, such as instruction cache, contribute significantly to the area budget. Instruction supply spends 42 % of energy in embedded processors [14] or around 15 % of the total power in an ARM Cortex-A15 core [15]. McPAT shows that ARM's Cortex-A9 and Sun's Niagara2 spend around 15 % of the core area and power on I-caches [10]. ARM's lean cores have similar area footprint compared to those in Intel's Xeon Phi or IBM's BlueGene and recent works consider ARM a potential player in this market [16], [17].

Figure 4 shows an intrinsic property of HPC applications: inside the parallel regions, most of the threads execute the same code. It gives the percentage of instruction footprint shared among *all* the threads running the application. Instruction sharing is extremely high for HPC workloads. On average, around 99 % of dynamically executed instructions are the same for all running threads. Different threads work on different sets of data but the same set of instructions, as in parallel loops and parallel tasks, which results in a large amount of duplication across private I-caches.

These facts motivate our study on sharing the I-cache among lean cores in an ACMP. The potential benefits include improved I-cache hit rates due to constructive cross-thread instruction prefetching, as well as savings in chip area and static power. For example, factoring out around 15 % of per-core private real estate for an eight lean core cluster, opens an opportunity to spend that saving on an additional core. The main potential drawback is a larger I-cache access latency due to the introduction of a shared-access interconnection network. The goal of this work is to evaluate this tradeoff and to provide an optimal solution tuned to increase performance per power and area.

III. Related Work

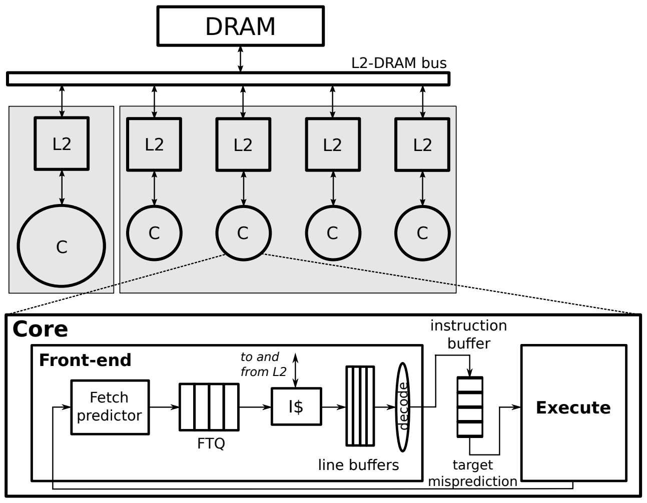
Asymmetric processors have been proposed as a heterogeneous, single-ISA multicore design to reduce the execution time of a parallel application for a given hardware budget [4], [6], [18]–[20]. The large core (latency sensitive) would be used to execute serial bottleneck, while many small cores (throughput oriented) run parallel code. Our work shows how HPC applications benefit from an ACMP by executing the master thread on a large core and worker threads on small cores. We further try to improve the performance per area by evaluating the benefits of sharing an I-cache among lean cores on an ACMP.

As soon as we start sharing resources among cores in a CMP, we enter the blurred space between multicore and multithreaded processors. The first papers dealing with simultaneous multithreaded (SMT) processors already identified the shared front-end as one of the major bottlenecks [7]. There have been proposals and products for multithreaded processors with a lower resource sharing degree than SMT. Conjoined cores [21], CASH [22], IBM Cyclops64 [23], and AMD's Bulldozer module [9] propose a CMP where adjacent cores share some of the hardware structures such as the I-cache, the data cache, and the floating point unit.

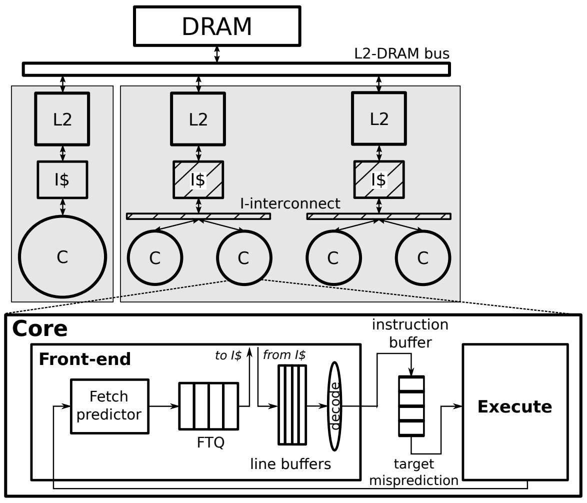
All of the previous proposals focus on sharing resources among two adjacent heavyweight cores, while our intention is to provide a thorough analysis on sharing only the I-cache among many worker cores on an ACMP. Since the rest of the core front-end is not shared, this design improves scaling and it allows sharing among more than two cores. Our work points the limiting factors with more cores sharing an I-cache, with the main objective of increasing performance for the same hardware budget.

The I-cache sharing has also been studied for OLTP workloads [24], which have instruction footprints that exceed the capacity of the I-cache in general-purpose processors. Their design advocates for sharing a larger capacity I-cache to reduce the number of misses in the I-cache. We show that a single shared I-cache, smaller than a private one, reduces the number of I-cache misses due to inter-thread prefetching, and also leads to area (and power) savings. In their work, the authors focus only on miss analysis not concerning the implication of the proposed design on execution time, as we do here.

Sharing the I-cache among many low-power embedded processors has also been evaluated [25]. Their work is focused on embedded micro-kernels and caches of 1 KB in size. They observe performance improvements up to 60 %, and identify conflicting accesses to the shared I-cache as a potential source



(a) baseline architecture



(b) shared I-cache architecture

Fig. 5: Schematic representation of baseline and shared I-cache architectures, including the instruction part of memory hierarchy. Heavyweight core is not modified.

of problems. In this paper, we evaluate mechanisms to hide the extra latency involved in conflicting accesses to the shared I-cache and interconnect.

Finally, in the context of HPC workloads, NVIDIA GPU accelerators [8] already use a shared I-cache for all CUDA cores in a streaming multiprocessor (SM). Threads running on an SM (a warp) fetch and execute the same instruction in lock-step mode every cycle, which prevents conflicting I-cache accesses and latency variations. We evaluate this approach in a more general way focusing on ACMPs where each thread has its own program counter and executes a separate instruction stream without any constraints.

IV. Shared I-cache Architecture

For a baseline configuration, we consider an ACMP composed of one large and eight lean cores with private L1 and L2 caches, connected to an on-chip memory controller giving access to off-chip memory. Figure 5 shows the instruction side of the baseline and proposed ACMP architectures. It presents four worker cores for simplicity. In our study, we use a configuration with one big and eight small cores. We first detail the core model, based on a decoupled front-end architecture. After that, we present the evaluated ACMP architecture with a shared I-cache among lean cores.

A. Core Front-End

Figure 5a shows the baseline architecture. The core model decouples the I-cache from the branch predictor with a *fetch target queue* (FTQ) [26]. With the objective of increasing fetch bandwidth, the branch predictor and FTQ work with *fetch blocks* (FB) instead of basic blocks. An FB is a sequence of instructions that ends at a taken branch and, thus, it may contain multiple basic blocks if their instructions are consecutive.

The Fetch Predictor (which is actually the branch predictor) generates the fetch address for the next fetch request and stores it in the FTQ. An FTQ entry contains the starting address and the length of the FB. The private I-cache is then accessed using the FB starting address at the front of the FTQ. If the instructions to be requested to the I-cache happen to be already in one of the line buffers, no request is made to the I-cache, and the contents of that line buffer are reused instead. With more line buffers, the front-end is capable of having more outstanding requests to its I-cache, one request per line buffer. When the requested I-cache line is returned from the cache, it is stored in one of the line buffers, which act as prefetch buffers. Using shift and rotate logic, instructions are extracted from the line buffer and stored in the instruction queue. From that point, the back-end, representing the rest of the pipeline, executes and retires those instructions. In case of a branch misprediction, the pending I-cache requests are discarded and all front-end stages of the pipeline flushed.

Figure 5b details the shared I-cache architecture. The FB predictor, FTQ, line buffers and decode logic are as in the baseline architecture. The main difference is that the I-cache is placed outside of the core and connected to multiple cores. Depending on the sharing degree, more or less cores may share one I-cache. In the figure, two lean cores share one I-cache thus, there are two I-caches for four cores.

B. Shared I-cache and Interconnect

Multi-banked caches consist of several cache banks, providing multiple accesses in the same cycle, up to one access per bank. This technique is attractive for last-level caches since they are usually shared among cores. The same logic can be applied to a shared I-cache. Instead of serializing core accesses to the I-cache, multiple requests can be served as long as they fetch from different banks.

To fully utilize a multi-banked cache, all cores must be connected to all banks, which means using a crossbar switch as interconnect or multiple buses. Although crossbar and multi-buses provide higher bandwidth and reduce the congestion,

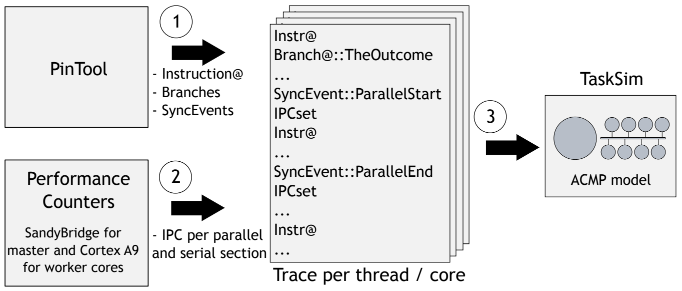


Fig. 6: Temporal flow of a simulation process.

they are expensive in area and power. According to previous work [27], the area cost of a crossbar increases quadratically with the number of cache banks, whereas the area of a bus increases linearly. Given our main objective of reducing area and power without hurting the performance, we evaluate this tradeoff in Section VI-B.

V. Simulation Methodology

A. Simulation Framework

Our simulation framework is based on TaskSim [28], a trace-driven cycle-level simulator for parallel architectures running multithreaded applications. We use Pin [11] as the instrumentation framework for tracing the benchmarks. At runtime, our PinTool creates a trace file per thread storing the sequence of executed instruction addresses. For branch instructions, beside the address, it also stores its outcome (taken or not-taken) as well as branch target address. That way, we store all the information needed for reproducing the instruction stream. To resolve the weaknesses of trace-driven simulation such as inter-thread ordering and synchronisation, we introduce synchronisation events inside the trace files. We implement five events that cover all OpenMP primitives in the evaluated workloads: parallel start, parallel end, wait and signal on critical sections and semaphores, and barrier. The simulation framework thus has a double role. First, it models the entire ACMP as shown on Figure 5, reads the trace files, and sends the requests to the I-caches for every fetch block. Second, it mimics the run-time system by managing the state of every thread according to the synchronization events in order to reproduce the same static scheduling of the application running in the real machine.

The simulation framework models the core front-end of both the baseline and shared I-cache architectures in detail following the description given in Section IV. We implement the core back-end so it can simulate processors with different levels of performance. Each cycle, the back-end attempts to commit up to a given number of instructions (commit rate) from its instruction queue. The capability of keeping the maximum commit rate in the back-end every cycle directly depends on front-end performance. This way we analyse the performance of our shared I-cache architecture avoiding back-end design artefacts. To distinguish master and worker cores, we apply the IPC values measured on a separate run using performance counters [29] on two different platforms. Running each benchmark with a single thread, we obtained IPC values for each serial and parallel code section so that TaskSim can change the commit rate of the cores depending on which part of the application it currently simulates.

Figure 6 illustrates the simulation process. PinTool produces the traces, one per thread, capturing the instruction stream (step 1). Using performance counters from two different platforms (one for master and other for worker threads/cores), we add IPC values to the traces, for each parallel and sequential code section (step 2). Finally, TaskSim reads the traces and models the entire ACMP (step 3).

We surveyed a set of existing simulators and did not find one that had a front-end pipeline modelled at such level of detail that allowed us to reason about the baseline and shared I-cache organizations described in Section V-A. For example, having a pipelined front-end implementation is crucial in our analysis since an access to the shared I-cache can take multiple cycles. Also, the core front-end includes a set of line buffers that behave as a micro-cache or loop-buffer [30], [31] reducing the number of accesses to the I-cache (private or shared). Our core implementation in TaskSim models these features together with all the other hardware components shown in Figure 5 in a cycle-accurate way.

B. Simulation Setup

Table I shows the configuration parameters for the simulated ACMP. The cache hierarchy, fetch predictor, shared I-interconnect, memory controller, and off-chip memory are modelled in detail. Cores-per-cache or *cpc* stands for the number of worker cores that share one I-cache. For example, with eight worker cores in total and $cpc = 4$ there are two groups of four cores where each group share one I-cache. The I-cache size, line width, associativity and latency remain the same for any degree of sharing. We focus our evaluation on the parameters that most affect the impact of our proposal: different degrees of sharing (*cpc*), the number of line buffers, and the I-interconnect bandwidth.

C. Benchmark Suites

We evaluate our proposal using three HPC benchmark suites: NAS Parallel Benchmarks (NPB suite), SPEC OMP 2012 (SPECOMP suite), and ExMatEx Applications. We run all of the 10 benchmarks from NPB suite with *input set C*, and 10 benchmarks from SPECOMP suite with *reference* inputs.[2] We also use four ExMatEx Applications (CoEVP, CoMD, CoSP, and LULESH) with default input parameters. Our evaluation is based on 24 HPC workloads in total, all of them implemented using the OpenMP programming model.[3] The input size for each benchmark was chosen so that the instrumentation takes an acceptable amount of time, but always executing at least 20 billion instructions in total.

We evaluate OpenMP applications in this paper but our conclusions are also applicable to other HPC programming

[2]SPEC OMP benchmark suite has three more applications which are identical to the corresponding ones from NPB suite.

[3]ExMatEx workloads are implemented using MPI+OpenMP programming model and we run them with one process in our experiments.

Parameter	Value(s)
ACMP	1 master and 8 worker cores
master core	IPC values from an Intel's i7 core
worker core	IPC values from an ARM's Cortex-A9 core
Cores-per-cache (*cpc*)	[1, 2, 4, 8] 1 stands for a baseline (private I-caches)

I-cache	size = 32KB, 8-way
	latency = 1 cycle
	line width = 64B

Line buffers	[2, 4, 8] width = 64B

I$-interconnect	type = single or double bus
	latency = 2 cycles + contention[4]
	width = 32B
	arbitration = round-robin

Fetch predictor	16KB gshare + 256-entry loop predictor

L2 cache	size = 1MB, 32-way
	latency = 20 cycles
	line width = 64B

L2-DRAM bus	latency = 4 cycles + contention
	width = 32B

DRAM	size = unlimited timing parameters = standard[5]

TABLE I: Configuration parameters for the simulated ACMP.

models, including distributed memory models like MPI. Although MPI tasks run on separate processes, they still run the same executable. In such case, the OS maps all the code regions to the same physical page, since code pages are read only. The same applies for shared libraries. This means that multiple processes in MPI applications, running on a single node, share the same code as they access the same physical code pages.

VI. Evaluation

In this section we present the evaluation of sharing the I-cache among lean cores on an ACMP. We start by checking how simple I-cache sharing affects the performance. Increasing the I-cache access latency by putting a shared bus between worker cores and the I-cache, we measure the performance loss for some workloads especially with the higher degrees of sharing. We evaluate how adding more line buffers and doubling the bandwidth of a shared bus overcomes this problem as a tradeoff between the performance and energy consumption. At the end, we find the scalability limits of this proposal and answer the question if a single I-cache can be shared among all cores on an ACMP, including the master core.

A. Naive I-cache Sharing

First, we evaluate sharing a $32\,\mathrm{KB}$ I-cache among two, four, and eight small cores, and compare with the baseline architecture (private, $32\,\mathrm{KB}$ I-caches). Figure 7 shows the normalized execution time with respect to the baseline architecture for

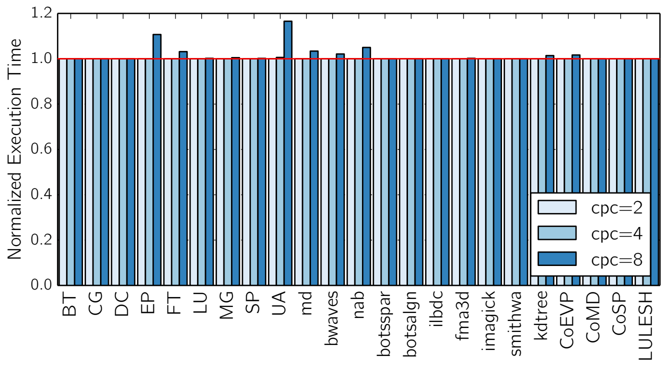


Fig. 7: Naive scaling. Execution time for different levels of sharing a $32\,\mathrm{KB}$ I-cache among worker cores. We use four line buffers and a single bus as the interconnection network.

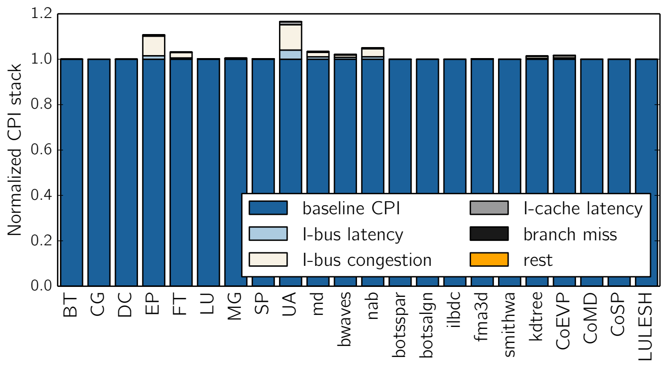


Fig. 8: Naive scaling. Normalized CPI stack per benchmark for the highest level of sharing ($cpc = 8$).

different levels of sharing. For some benchmarks, a single I-cache shared among eight cores increases execution time, up to $18\,\%$ in the case of *UA*. Figure 8 gives the normalized CPI stack per benchmark when a single I-cache is shared among all eight cores. Very few additional stall cycles are caused by the latencies from I-cache misses, branch misses, and fetch requests to the upper levels of memory hierarchy. HPC applications have predictable branches and a simulated $16\,\mathrm{KB}$ gshare augmented with a loop predictor provides a low number of branch mispredictions (with $3.8\times$ higher branch MPKI values in serial code than in the parallel sections). The majority of stall cycles are due to the extra latency brought by the intermediate shared bus. Most stall cycles are caused by contention on the I-bus. We explore two potential features to overcome these stall cycles: putting more line buffers or increasing the bandwidth of the shared interconnect.

B. Scalable I-cache Sharing

With more line buffers, the front-end is capable of having more outstanding requests to its I-cache, one request per line buffer. Every time the starting address of the current fetch block exists in a line previously brought into one of the line buffers, the front-end reuses that line buffer and does not issue

[4]Contention refers to the number of cycles waiting on a busy bus due to its utilization by some other core.

[5]Values for DRAM timing parameters match the Micron DDR3-1600 specification.

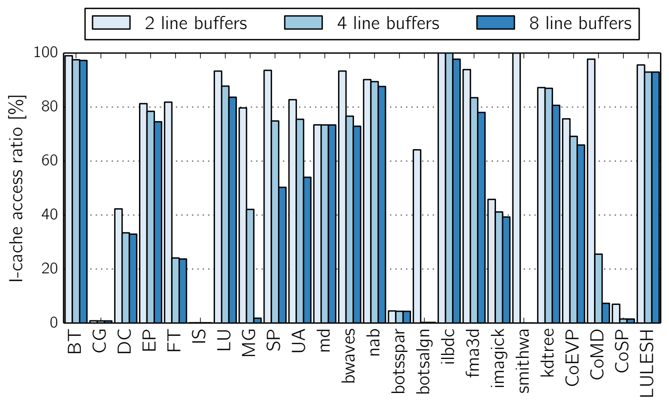


Fig. 9: I-cache access ratio for different number of line buffers. More than eight line buffers does not reduce the I-cache access ratio significantly.

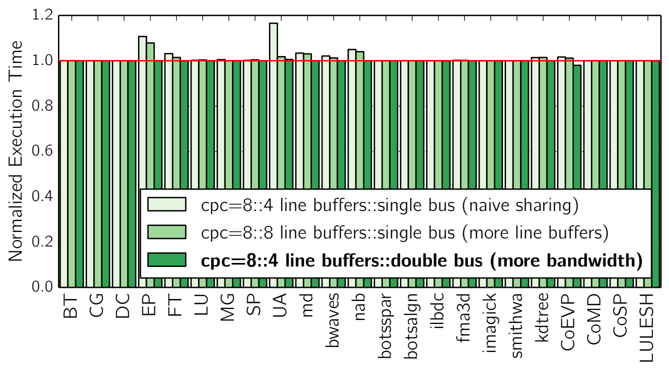


Fig. 10: Trade-off between adding more line buffers and doubling the interconnection bandwidth when a single 16 KB I-cache is shared ($cpc = 8$). The execution times are normalized to the baseline architecture (private, 32 KB I-caches).

a request to the I-cache. This reduces the number of accesses to the shared I-cache and contention on the shared bus.

Figure 9 shows how using more line buffers reduces the I-cache access ratio, defined as the number of lines fetched from the I-cache divided by the total number of fetch requests. This is expected due to high temporal locality that is present in the code. It is interesting how this temporal locality complements our analysis on average basic block length (see Figure 2). For almost all of the benchmarks where the average basic block length is small, the I-cache access ratio is also low (*CG, IS, botsalgn, botsspar, CoSP*). On the other side, when the basic blocks are long, almost all the accesses are to the I-cache (*BT, LU, ilbdc* and *LULESH*).

Another way of reducing the contention on a shared interconnection is to increase its bandwidth. Instead of a single bus, we use a shared multi-banked I-cache so that each bank now has its own bus connected to all worker cores. For example, having an I-cache with two banks, one with even and one with odd cache lines, we connect a separate bus for each bank, so that the I-cache requests of even cache line addresses route through the first bus, and the requests with odd line addresses route through the second bus. That way, a shared multi-banked I-cache is able to provide two cache lines per cycle as long as they are found in different cache banks. Doubling the number of buses increases the area of the I-interconnect by $4\times$ compared to a single bus proposal. With the cost of dedicating more area and power budget to this solution, we reduce the contention on the shared I-interconnect.

Figure 10 shows how these two techniques affect the total execution time. Adding more line buffers is beneficial for some benchmarks where it reduces the I-cache access ratio, such as *UA*. But, in most cases, the baseline with four line buffers already captures most executed basic blocks and hot loops, thus adding more line buffers to this set has a limited effect. On the other hand, doubling the bandwidth of the interconnection network between the lean cores and the shared I-cache completely removes the stall cycles caused by prolonged I-cache access latency. By using two I-buses instead of one, we halve the number of cores requesting the I-cache line per bus, and reduce the contention.

C. Miss Analysis

Figure 11 shows how sharing an I-cache among all worker threads ($cpc = 8$) affects the number of misses per kilo instruction (MPKI). The numbers above the bars represent the absolute MPKI values obtained with a set of private, 32 KB I-caches. As we have seen before on Figure 3, HPC applications miss very few times accessing an I-cache in parallel regions. On average, sharing the I-cache reduces the number of misses by 50 %, and up to 90 % in case of *LU* and *SP*, compared to a baseline architecture (private I-caches). Even a smaller I-cache shared among all lean cores ($cpc = 8 :: 16KB$) provides fewer misses than the set of per core 32 KB I-caches. This is a direct consequence of the code sharing among threads in HPC workloads. Threads prefetch instructions for each other in a shared I-cache and we have observed in some cases *a complete absence of cold misses for some threads*. Sharing the I-cache increases the number of non-compulsory misses for some benchmarks due to the lower overall capacity (*botsalgn, smithwa*). In those cases the MPKI values are still reduced, which implies that compulsory misses are dominant. In some other cases (*SP, imagick, LULESH*), even non-compulsory misses are reduced due to almost perfect time alignment among threads accessing the same line in the shared I-cache.

The most interesting case is the *CoEVP* benchmark. That is the only HPC workload we analyse for which the I-cache MPKI value is above 1 for a private, 32 KB I-cache. Sharing a single I-cache among all worker cores halves the number of misses, and with a double I-bus we provide enough bandwidth so that congestion does not introduce additional stall cycles. With these two things combined, we even observe a 2 % performance improvement, as shown on Figure 10. For HPC applications where I-cache misses introduce a significant performance degradation, our proposal of sharing the I-cache among lean cores stands not just as an area and power saving technique, but also to increase the performance.

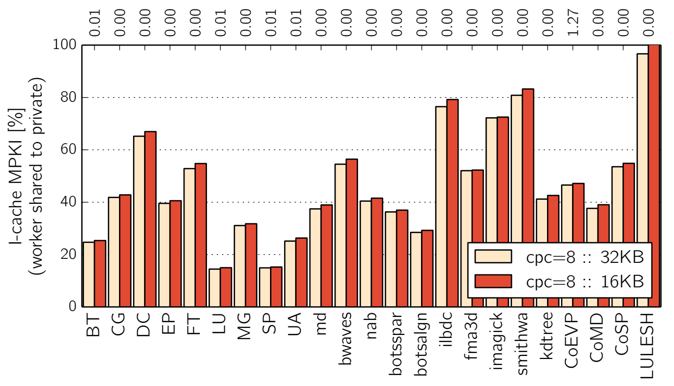


Fig. 11: MPKI values for an I-cache shared among all eight lean cores in its two sizes, 32 KB and 16 KB, normalized to a baseline ACMP (private 32 KB I-caches). Numbers above the graph represent absolute MPKI values for each benchmark with private I-caches.

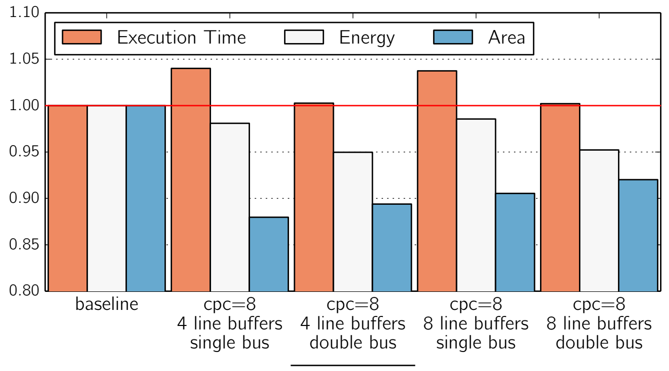


Fig. 12: Energy and area savings adding more line buffers and doubling the interconnection bandwidth when a single 16 KB I-cache is shared ($cpc = 8$). All the values are normalized to the baseline architecture and averaged across the benchmarks.

D. Area and Power Savings

We estimate the area and power savings relative to a set of lean cores with private I-caches as the baseline. Master core, LLC and NoC are not included in this analysis. Sharing an I-cache among cores reduces the occupied area and total power but at the same time the additional shared bus introduces overheads.

We use McPAT [10] and CACTI [32] to estimate the area and energy consumption of cores, I-caches, I-buses, and line buffers. We have selected the ARM Cortex-A9 configuration file from the McPAT bundle because it has been validated against real silicon and is representative of lean cores. We run McPAT for different ACMP configurations and I-cache sizes and use statistics from simulation outputs and performance counters. Then, we obtain the area and power numbers and compare them with the baseline values.

Both wires and logic of the shared bus contribute to interconnection overhead. When a bus is wired without array structures underneath, logic can be placed under the bus without additional area overhead [27]. The area occupied by a bus is determined by the number of wires, the wire pitch and length. In our model, bus width is the same as the I-cache line width, which determines the number of wires plus address lines. The wire pitch for a 45 nm technology is 205 nm [33]. The length of the bus is estimated as the number of cores times the bus width [34]. This gives a quadratic dependence of bus area on line width. For power estimation we use the power-to-area relation taken from the McPAT values of the NoC component (bus). It gives a linear dependence of total power on area. With previously obtained area values for the bus, we apply this coefficient to get its total power numbers. For dynamic power, we set the number of transactions on the NoC as the number of accesses to the shared I-cache and apply the same dynamic-to-total power ratio, once we calculate the total power.

Figure 12 presents execution time, energy, and area consumption of eight worker cores for different design points, averaged across the benchmarks and normalized to the baseline. For the highest level of sharing ($cpc = 8$), we focus on the trade-off between using more line buffers or doubling the bandwidth. Compared to the baseline, sharing an I-cache reduces the area and static power. The number of accesses to the shared I-cache increases $8\times$ but since we share smaller, 16 KB I-cache, its dynamic power is also lower compared to a set of private I-caches. We calculate the energy as the product of total power (dynamic and static) and execution time. Configurations with only one I-bus have the highest area savings but modest energy savings, mostly due to the increased execution time. With the methodology explained in the previous paragraph, we estimate that the area budget of a double I-bus is around 45 % of a 16 KB I-cache. More line buffers brings less activity on the bus and less accesses to the I-cache but more area and energy for a line buffer access.

Figure 12 also presents optimal designs for different metrics. In case we are mostly interested in area savings, sharing the I-cache among eight cores with four line buffers and single bus, stands as the optimal design. Unfortunately, it also brings 4 % of performance degradation on average. If hurting the performance is not an option, the best configuration is an I-cache shared among eight lean cores with four line buffers and a double I-bus that provides savings of 5 % in energy and 11 % of area.

These savings can be used to increase performance for the same power and area budget. A shared I-cache architecture among worker cores allows adding an extra core for the same area. This can be attractive for many-core designs such as Xeon Phi, configuring the processing elements in octa-core clusters each with a single shared I-cache. Another possibility is to increase other hardware structures, such as data cache and SIMD execution unit. HPC codes benefit from additional thread- and data-level parallelism, therefore leading to higher CMP performance per unit of area and energy efficiency.

E. A single I-cache shared among all cores on an ACMP

Besides executing serial parts of the code, the master core acts as an additional worker core during parallel code sections.

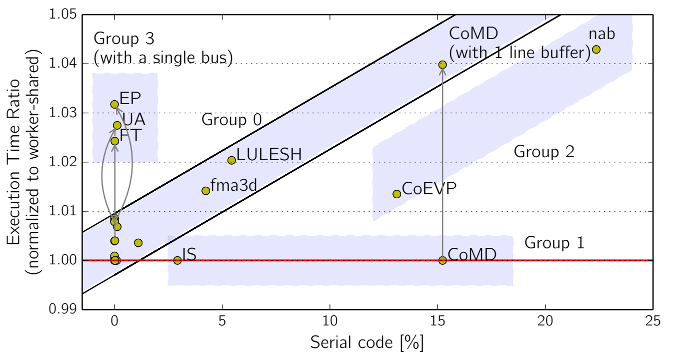


Fig. 13: Execution time ratio dependence on the serial code fraction.

Here, we analyse whether the master core can be joined to the set of worker cores sharing a single I-cache. That way, it can benefit from inter-thread code prefetching and contribute further to the area and energy savings by discarding its private I-cache. In this analysis we use shared 32 KB I-cache, so that we do not hurt the master core execution by reducing the I-cache size. Configuring the I-interconnect as a double bus, we compare *all-share* proposal (master and workers share a single I-cache) with the previously evaluated *worker-shared* proposal (the I-cache is shared only among worker cores).

Figure 13 explains why for some benchmarks it is harmful to share a single I-cache among all cores. It shows how the performance ratio between all-shared and worker-shared proposals depends on the fraction of serial code. In general, with higher percentage of serial code, all-shared needs more time to complete the same job compared to worker-shared configuration. The master core has more aggressive back-end (heavyweight core) and it runs alone in serial code parts that have shorter basic blocks on average. Sharing an I-cache, every time it fetches the sequential code it has to send the request through the I-interconnect bringing back fewer instructions. With the increased I-cache access latency and shorter basic blocks, the master core does not provide enough instructions to its back-end, introducing stall cycles and hurting the performance. We estimate this dependency with the area between two diagonal black lines on Figure 13. Still, there are few outliers that we break down into groups, each with different reasons being distant from the general dependency:

- **Group 0 - default behavior:** Most of the benchmarks belong to this group as they have a negligible amount of instructions executed in serial parts. Benchmarks like *fma3d* and *LULESH* show the general trend, for every 5 % of serial code fraction the performance degrades for 1 % compared to worker-shared configuration.
- **Group 1 - code locality in serial code:** Although with significant amount of instructions executed in serial by master core (especially for *CoMD*) the execution time is the same as in worker-shared setup. The reason is high code locality of serial code. For example, when we configure the core front-end with four line buffers (baseline), *CoMD* rarely accesses the shared I-cache when executing sequential code. Only one line buffer is not enough to exploit the serial code locality, thus *CoMD* moves to Group 0.
- **Group 2 - long basic blocks in serial code:** Figure 2 shows that HPC applications have short basic blocks in sequential code regions, with two benchmarks as exceptions, *nab* and *CoEVP*. That is the reason why these two benchmarks do not belong to Group 0. With longer basic blocks, the master core behaves like worker cores in parallel regions.
- **Group 3 - scalability limitations:** If we use a single I-bus, *EP, FT,* and *UA* benchmarks show performance degradation when the master core also shares the I-cache. This time, the stall cycles are not caused due to prolonged I-cache access latency in serial code sections, but in parallel ones. Adding one more core to a single I-bus increases the congestion and the execution time. This finding exposes the scalability limits. Sharing an I-cache among more than eight cores introduces additional stall cycles which can not be mitigated with a double bus interconnect and four line buffers. With higher interconnection bandwidth and line buffers, the performance degradation can be reduced, but the extra area and energy cost do not justify such an investment, leading to a design with the same performance and the same area and power budget as the baseline ACMP.

This final analysis further stresses the difference between parallel code commonly run on HPC systems and serial bottleneck that exists in every parallel application. There is a need to tailor the cores on a CMP differently, depending on the parts of the code they execute. Although attractive with the additional energy and area savings, sharing an I-cache among all cores on an ACMP shows performance degradation as the amount of serial code increases. Our findings suggest that an I-cache can be shared among worker cores providing energy and area savings for the same performance, but the master core should be left with its private I-cache.

VII. Conclusion

In this paper we have analysed sharing the I-cache among multiple worker cores to provide a more balanced ACMP architecture for HPC workloads, based on their intrinsic code characteristics. The parallel code regions are executed by worker threads running the same code with long basic blocks. Due to initially low I-cache MPKI values and with the mutual code prefetching among threads, the shared and smaller I-cache feeds instructions to lean worker cores using a simple double bus as an I-interconnect, and a standard, small set of prefetch buffers.

Sharing an I-cache, the arbitration policy on an I-bus becomes the fetching policy, previously evaluated in the context of SMT cores. We believe that the evaluation of these policies can further reduce the impact of contention on a shared I-bus, and maybe increase the performance in some cases. Moreover, customizing the rest of the multicore front-end and sharing both the iTLB and branch predictor may also provide benefits from similar cross-thread prefetching and constructive interference effects. This creates opportunities to further improve the energy efficiency of HPC multicores exploiting the specific properties of "single program, multiple data" workloads to be explored in future work.

Our results on 24 workloads from three HPC benchmark suites show considerable area and energy savings of around 11 % and 5 %, respectively, without performance loss. The analysis suggests that constructive interference between threads reduces the number of I-cache misses and almost eliminates cold I-cache misses. In cases where the initial I-cache MPKI values were high, sharing an I-cache among worker cores even increases the performance.

VIII. Acknowledgements

The research was supported by European Unions 7th Framework Programme [FP7/2007-2013] under project Mont-Blanc (288777), the Ministry of Economy and Competitiveness of Spain (TIN2012-34557, TIN2015-65316-P, and BES-2013-063925), Generalitat de Catalunya (2014-SGR-1051 and 2014-SGR-1272), HiPEAC-3 Network of Excellence (ICT-287759), and finally the Severo Ochoa Program (SEV-2011-00067) of the Spanish Government.

References

[1] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[2] R. A. Haring, M. Ohmacht, T. W. Fox, M. K. Gschwind, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski *et al.*, "The IBM Blue Gene/q compute chip," *Micro, IEEE*, 2012.

[3] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967.

[4] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, 2008.

[5] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, 2005.

[6] U. Milic, P. Carpenter, A. Rico, and A. Ramirez, "Rebalancing the Core Front-End through HPC Code Analysis," in *International Symposium on Workload Characterization (IISWC)*. IEEE, 2016.

[7] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *ACM SIGARCH Computer Architecture News*. ACM, 1996.

[8] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE micro*, 2010.

[9] M. Butler, L. Barnes, D. D. Sarma, and B. Gelinas, "Bulldozer: An approach to multithreaded compute performance," *IEEE Micro*, 2011.

[10] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*. ACM, 2005.

[12] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1996.

[13] T.-Y. Yeh, D. T. Marr, and Y. N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache," in *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, 2014.

[14] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield, "Efficient embedded computing," *Computer*, 2008.

[15] N. Corp., "NVIDIA Tegra 4 Family CPU Architecture," Tech. Rep., 2013.

[16] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero, "Supercomputing with commodity cpus: Are mobile socs ready for hpc?" in *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, 2013.

[17] L. Gwennap, "Nvidia's First CPU Is a Winner," *Microprocessor Report*, August 2014.

[18] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguade, "Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors," *Computer Architecture Letters*, 2006.

[19] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *ACM SIGARCH Computer Architecture News*. ACM, 2009.

[20] S. Eyerman and L. Eeckhout, "Modeling critical sections in amdahl's law and its implications for multicore design," in *ACM SIGARCH Computer Architecture News*. ACM, 2010.

[21] R. Kumar, N. P. Jouppi, and D. M. Tullsen, "Conjoined-core chip multiprocessing," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2004.

[22] R. Dolbeau and A. Seznec, "Cash: Revisiting hardware sharing in single-chip parallel processor," 2002.

[23] G. Almási, C. Caşcaval, J. G. Castanos, M. Denneau, D. Lieber, J. E. Moreira, and H. S. Warren Jr, "Dissecting Cyclops: A detailed analysis of a multithreaded architecture," *ACM SIGARCH Computer Architecture News*, 2003.

[24] P. Kundu, M. Annavaram, T. Diep, and J. Shen, "A case for shared instruction cache on chip multiprocessors running oltp," in *ACM SIGARCH Computer Architecture News*. ACM, 2003.

[25] D. Bortolotti, F. Paterna, C. Pinto, A. Marongiu, M. Ruggiero, and L. Benini, "Exploring instruction caching strategies for tightly-coupled shared-memory clusters," in *System on Chip (SoC), 2011 International Symposium on*. IEEE, 2011.

[26] G. Reinman, T. Austin, and B. Calder, "A scalable front-end architecture for fast instruction delivery," in *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 1999.

[27] R. Kumar, V. Zyuban, and D. M. Tullsen, "Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling," in *Proceedings. 32nd International Symposium on Computer Architecture, 2005*. IEEE, 2005.

[28] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero, "Trace-driven simulation of multithreaded applications," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2011.

[29] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, 1999.

[30] J. Turley, "Cortex-A15 "Eagle" flies the coop," *Microprocessor Report*, November 2010.

[31] B. Solomon, A. Mendelson, R. Ronen, D. Orenstien, and Y. Almog, "Micro-operation cache: A power aware frontend for variable instruction length isa," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2003.

[32] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," Technical Report 2001/2, Compaq Computer Corporation, Tech. Rep., 2001.

[33] J. Lee, C. Nicopoulos, S. J. Park, M. Swaminathan, and J. Kim, "Do we need wide flits in networks-on-chip?" in *VLSI (ISVLSI), 2013 IEEE Computer Society Annual Symposium on*. IEEE, 2013.

[34] S. Niar, L. Eeckhout, and K. De Bosschere, "Comparing multiported cache schemes." in *PDPTA*, 2003.