Rebalancing the Core Front-End through HPC Code Analysis

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

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

‡ARM Inc.
alejandro.rico@arm.com

§Nvidia Corp.
aramirez@nvidia.com

***Abstract*—There is a need to increase performance under the same power and area envelope to achieve Exascale technology in high performance computing (HPC). The today's chip multiprocessor (CMP) design is tailored by traditional desktop and server workloads, different from parallel applications commonly run in HPC. In this work, we focus on the HPC code characteristics and processor front-end which factors around 30% of core power and area on the emerging lean-core type of processors used in HPC. Separating serial from parallel code sections inside applications, we characterize three HPC benchmark suites and compare them to a traditional set of desktop integer workloads. HPC applications have biased and mostly backward taken branches, small dynamic instruction footprints, and long basic blocks. Our findings suggest smaller branch predictors (BP) with the additional loop BP, smaller branch target buffers (BTB), and smaller L1 instruction caches (I-cache) with wider lines. Still, the aforementioned downsizing applies only to the cores meant to run parallel code. The difference between serial and parallel code sections in HPC applications points to an asymmetric CMP design, with one baseline core for sequential and many HPC-tailored cores designed for parallel code. Predictions using Sniper simulator and McPAT show that an HPC-tailored lean core saves 16% of the core area and 7% of power compared to a baseline core, without performance loss. Using the area savings to add an extra core, an asymmetric CMP with one baseline and eight tailored cores has the same area budget as a symmetric CMP composed out of eight baseline cores demanding 4% more power and providing 12% shorter execution time on average.**

I. Introduction

Chip multiprocessor (CMP) power consumption is one of the first class constraints on exascale system performance and effectiveness [1], [2]. We need more energy-efficient CMP designs to reach the exascale milestone, providing more performance within the same power and area budget.

Today's high performance computing (HPC) nodes are made of CMPs tailored for desktop and server applications. They usually have few heavy-weight cores capable of exploiting the available instruction-level parallelism through wide superscalar out-of-order execution. The core front-end is designed to support large instruction footprints and to predict the outcomes of branches in complex control flows. On the other side, HPC workloads are different, running in parallel, thus demanding throughput-oriented CMPs. Keeping the same power and area budget, a handful of heavy cores are replaced by many lean ones, integrated as an alternative to exploit thread-level parallelism. For example, Intel's Xeon Phi [3] and IBM's BlueGene/Q [4] CMP architectures integrate many power-efficient lean cores targeting parallel HPC workloads.

With the current configuration of the front-end structures, an embedded processor spends 42% of its energy on instruction supply [5]. Instruction fetches and branch predictions consume 30% of the total power in the ARM Cortex-A15 core [6]. McPAT also shows that lean cores, such as ARM's Cortex-A9 and Sun's Niagara2, spend 25% of the total core area, and 30% of the total core power on instruction delivery [7]. Therefore, it is important to evaluate microarchitectural optimizations to lessen front-end activity and area which can have significant impacts on overall power consumption.

In this paper, we evaluate three HPC benchmark suites and compare them with traditional desktop applications, found in SPEC CPU INT. We analyse architecture independent code properties, followed by architecture dependent implications. Our results show that HPC applications expose different code characteristics, quantifying each of them. With those findings, we give recommendations on how to adequately dimension the core front-end structures of lean cores for HPC workloads to get maximal area and power savings without performance impact. Moreover, this work analyses the difference between serial and parallel code sections inside HPC workloads. Not just that HPC cores should be tailored differently from desktop, but also the master core has to be tailored differently from worker cores. With the serial section gaining relevance in new applications, tailoring the master core separately from workers may become critical.

The main contributions of this paper are:

- A characterization of 29 applications from three HPC benchmark suites. Compared to SPEC CPU INT, we find that HPC workloads have fewer branch instructions, more biased and backward taken conditional branches, smaller instruction footprints, and longer basic blocks. Those findings suggest the use of smaller I-caches with wider lines, smaller BPs with loop BPs, and smaller BTBs.
- When analysing HPC applications, we separate code characteristics for serial and parallel code sections. We find them different, with serial parts being more close to desktop applications in most of the cases. This points

that as the serial bottleneck increases inside parallel applications, we need an asymmetric CMP with many throughput-oriented cores and one (or few) latency-optimized core for serial code sections.
- Using McPAT and the Sniper as a multi-core simulator [8], we configure an asymmetric octa-core CMP with one baseline and seven cores with downsized front-end structures. We estimate that 16 KB I-cache (compared to 32 KB I-cache), 2.5 KB branch predictor (compared to 16 KB BP), and 256-entry BTB (compared to 2K-entry BTB) reduce the total lean core area budget by 16% and core power by 7% with no impact on performance. For the same area budget as a CMP made out of eight baseline cores, we can add an extra core. An asymmetric CMP with one baseline and eight tailored cores provides 12% performance improvement on average.

The rest of the paper is organized as follows. In Section II we explain our methodology and introduce the benchmark suites. Architecture independent and dependent characteristics of HPC workloads are presented in Sections III and IV. We provide our recommendations on core front-end rebalancing and performance evaluation in Section V. Related work is given in Section VI. Finally, we summarize in Section VII.

II. Methodology and Evaluation

In this Section, we explain the methodology of our work. Workload analysis is based on a set of pintools that collect data related to code characteristics at runtime. After that, we present benchmark suites evaluated in this work.

A. Workload analysis

We use Pin [9] as a tool for dynamic instrumentation of application binaries. It provides an infrastructure for writing program analysis tools called *pintools*. For architecture independent characterization, they only collect the statistic in their analysis routines. For example, in the case of branch instruction analysis, a pintool counts the number of branches, checks for each one if it is taken or not taken, etc. For architecture dependent characterization, pintools simulate specific HW structures. In the case of instruction fetching, a pintool configures an I-cache and instruments each instruction analysing if its address hits or misses in the I-cache.

For each upcoming analysis, we provide a detailed explanation of the particular pintool used to obtain the results.

Performance and power estimations are done using the Sniper multi-core simulator [8] in combination with the McPAT library [7]. We provide more information on their setup and usage in Section V.

B. Benchmark suites

We use 29 workloads from three benchmark suites to evaluate the benefits of tailoring the core front-end structures for HPC. Additionally, we analyse a set of 12 desktop applications to point the difference between them and HPC workloads.

ExMatEx applications. This suite stands as a recent set of eight HPC applications with real scientific workloads,

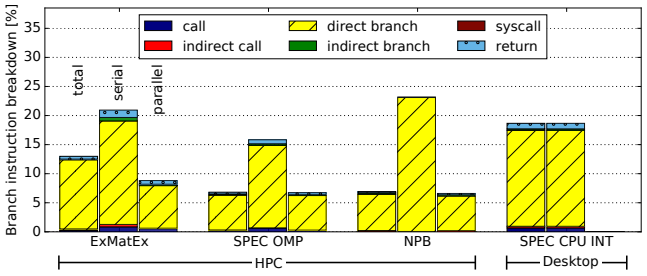


Fig. 1: Dynamic branch instruction breakdown for each benchmark suite as the percentage of total instructions.

numerically intensive kernels and kernels bounded by memory, I/O, and communication [10]. We use default input parameters for each of these workloads. Half of the applications from this suite are implemented using OpenMP while the other half combine MPI and OpenMP.

SPEC OMP 2012 benchmark suite. An objective and representative set of HPC workloads for measuring the performance of shared-memory CMPs [11]. The suite covers 11 scientific and engineering applications, from computational fluid dynamic to image manipulation[1]. These benchmarks were analysed using the *reference* input set.

NAS Parallel Benchmark suite. NPB suite is a set of 10 pseudo-applications derived from computational fluid dynamics apps [12]. Developed and maintained by NASA, it is a widely-used and standard set of HPC workloads. We use the *C* input set for analysing this suite.

SPEC CPU INT 2006 suite. These benchmarks represent a standard set of applications for measuring the system's processor and memory performance [13]. This suite is included as a comparison between HPC and desktop applications. Using *reference* input set, we analyse only integer benchmarks (all 12 of them) since floating point workloads are derived from scientific applications and many of them are already covered by the SPEC OMP suite.

We compile each workload with the *gcc 4.9* compiler version and the -O3 optimization flag. Instrumentation is performed on an Intel SandyBridge CMP, running eight and analysing only the first processing element ($thread_0$ or $process_0$). This is acceptable because our focus is on the private front-end structures per core and the impact of shared resources (such as last-level cache and interconnect network) through interference among cores in a CMP is known to be negligible on the instruction side [14]. Moreover, the first processing element is the one that executes both serial and parallel code sections.

III. Microarchitecture Independent Workload Characterization

Here, we provide intrinsic code characteristics of HPC applications. We focus on aspects affecting the core front-end: branches, instruction footprints, and basic blocks. We

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

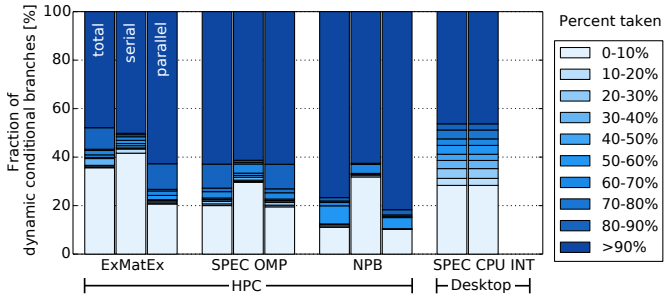


Fig. 2: Distribution of branch directions. Conditional branches are dominantly decided in one direction, either taken or not taken. Desktop applications have more evenly distributed directions of conditional branches.

TABLE I: The average percentage of backward and forward taken branches per benchmark suite.

Suite	Serial code		Parallel code	
	backward	forward	backward	forward
ExMatEx	72%	28%	69%	31%
SPEC OMP	73%	27%	74%	26%
NPB	71%	29%	80%	20%
SPEC CPU INT	56%	44%		

also point to a difference between sequential and parallel code sections we observed among workloads.

A. Branch Instructions

We start our analysis determining the dynamic mix of branch instructions. The pintool inspects every branch instruction and counts its frequency and type.

Figure 1 shows the dynamic branch instruction breakdown. All branch instructions factor out 13% of the total dynamic instruction mix for ExMatEx suite and around 7% for SPEC OMP and NPB, compared to 19% on average for SPEC CPU INT workloads (*total* bars). This indicates that HPC workloads probe branch predictors less often. The number of system calls is negligible. Indirect jumps (both branches and calls) are rare except for *EP*, *UA*, *md*, *kdtree*, and *CoEVP*. On average, they are less than 0.5% of all branches, and up to 2.5% in case of *CoEVP*. The majority of dynamic branch instructions are conditional and unconditional direct branches. This figure also shows a big difference between serial and parallel code sections inside the HPC applications. Closer to SPEC CPU INT, sequential parts have almost 3× more branch instructions than parallel parts on average.

Characteristic 1. HPC workloads, specially the parallel sections, have almost 3× lower ratio of branch instructions than typical desktop benchmarks.

Figure 2 presents a more detailed analysis of dynamic conditional direct branches. It gives a stacked bar for each suite showing the distribution of branches depending on the percentage of times they are taken. HPC workloads have between 90% (in case of NPB) to 80% (in case of ExMatEx) of dynamic branches dominantly either taken or not taken. On the other side, SPEC CPU INT applications have more distributed directions of conditional branches. Interestingly, serial and parallel code sections have similar behaviour, except that not-taken branch instructions are more frequent in sequential code. Additionally, Table I breaks down all taken branches on backward and forward ones. While for HPC suites around 75% of taken branch instructions jump backwards, SPEC CPU INT benchmarks have almost the same number of forward and backward taken branches. Again, serial and parallel code sections show similar '75-to-25' ratio between backward and forward taken branches. These results show bias in direction with the potentially high predictability of branches among HPC workloads. They suggest that the use of a simple and smaller branch predictor would yield a low misprediction ratio. We analyse this assumption in Section IV-A.

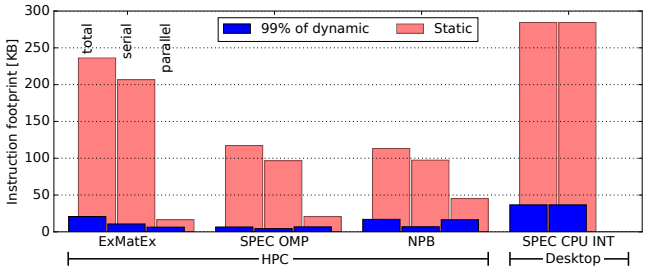


Fig. 3: Static instruction footprint and memory we need to store 99% of dynamic instructions per benchmark suite.

Characteristic 2. Compared to desktop benchmarks, branch instructions in HPC applications are more biased (predominantly decided in the same direction). Most of the taken branches are the backward ones.

B. Instruction Footprint

To find out the required I-cache design, we analyse the sizes of both static and dynamic instruction footprints. We use a pintool that stores the size of each basic block in bytes and counts how many times that basic block has been executed. That way, we find the static and dynamic instruction footprints per basic block and, thus, for the whole application.

Figure 3 shows the total static instructions footprint and the amount of memory needed to fit 99% of dynamic instructions, averaged per benchmark suite. Workloads from SPEC OMP and NPB suites have small static code size, up to 252 KB (for *UA*) and around 121 KB on average. Workloads from ExMatEx suite have larger static instruction footprint, up to 800 KB for *VPFFT* and 242 KB on average. These benchmarks are more recent, close representatives of real applications, linked with external libraries (such as LAPACK, BLAS, FFTW) that increase total instruction footprint. Among HPC workloads, sequential code has larger static instruction footprint, but still smaller than desktop applications. There is high spatial code locality. Most of the HPC workloads (17 out of 29) fetch almost 100% of instructions from one or

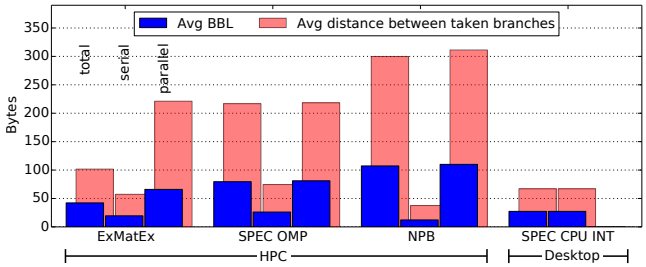


Fig. 4: Average basic block length and distance between taken branches for each benchmark suite.

two KB of memory. Still, cases such as *LULESH* and *CoGL* from ExMatEx or *BT* from NPB suite, fetch between 60% and 90% of instructions from 16 KB of memory. On average, HPC workloads in parallel execute 99% of instructions from just 14 KB of memory. Serial code sections also show high spatial locality, even higher compared to parallel sections for SPEC OMP and NPB suites. Among these benchmarks, the total number of instructions executed sequentially is small, thus the existence of any loop(s) (the majority of taken branches is backward-taken according to Table I) results in high code spatial locality. In this case, the serial code inside parallel HPC applications show different behaviour from SPEC CPU INT.

These results point that most of the HPC benchmark's dynamic code fit in less than 32 KB of memory. For many of them, even 4 KB of memory is enough to store almost every instruction. HPC applications spend most of their time inside loops, so few basic blocks are fetched and executed over and over again. Nevertheless, to know exactly how these characteristics impact the number of I-cache misses, we need to observe temporal behaviour as well. We cover this analysis later, in Section IV-C.

Characteristic 3. HPC workloads have small instruction footprint.

C. Basic blocks

Due to the low frequency of branch instructions, we expect to find long basic blocks in HPC benchmarks. Traditional desktop and server applications are known to have short basic blocks [15], [16]. Our analysis confirms this. Many complex features are implemented in today's CPUs to overcome the problem of short basic blocks and increase fetch bandwidth, such as multiple branch predictions per cycle, instruction alignment, and a trace cache. Tailoring an HPC core, these may not be needed.

Figure 4 shows the average basic block length and the average distance between taken branches per benchmark suite. The average basic block size for HPC applications is around 78 bytes. Some of them have very long basic blocks, for example, *BT* (312 B), *swim* (152 B), and *LULESH* (126 B). The distance between taken branches is even longer. It suggests the usage of wider I-cache lines that would still have high usefulness and keep fragmentation low. For those benchmarks where basic blocks are not long, reuse distance is short. That is the case with *CoHMM*, *CoSP*, *botsspar*, *CG*, and *IS*, where the average basic block size is around 32 B, but the majority of them are executed with a reuse distance between one and two basic blocks. Once fetched, a wide cache line would be frequently reused without sending new fetch requests to the I-cache, acting as a prefetch buffer [17]. Compared with SPEC CPU INT applications, HPC workloads have around 4× longer basic blocks and 5× longer distance between taken branches with parallel code sections. Sequential parts are similar to desktop applications. These results are important for a design of the I-cache, as we show in Section IV-C.

Characteristic 4. Parallel sections of HPC workloads have 4× longer basic blocks and 5× longer distance between taken branches compared to SPEC CPU INT benchmarks.

D. Difference between sequential and parallel code sections in HPC workloads

Our previous measurements demonstrate that ExMatEx benchmarks have slightly bigger code sizes, less biased branches, and shorter basic blocks compared to SPEC OMP and NPB. There are two reasons for such a behaviour.

First, it includes benchmarks with a considerable amount of instructions executed in sequential regions bringing its characteristics overall closer to SPEC CPU INT. Run on an eight-core CMP, *CoEVP* has a master thread that executes around 35% of its instructions sequentially, between parallel sections. The similar behaviour is observed for *CoMD* (8% of instructions is executed in sequential parts), *CoSP* (9%), and *LULESH* (11%) applications, all from the ExMatEx suite. Comparing basic block lengths, *CoMD* and *LULESH* have 2× and 3× longer basic blocks in parallel than in sequential code sections, respectively. Among SPEC OMP and NPB workloads, master thread executes less than 1% of all instructions in sequential regions, except for *nab* and *fma3d* (around 4% in sequential parts). On the previous graphs, the *total* bars are always between *serial* and *parallel* for ExMatEx, while for SPEC OMP and NPB *total* is almost the same as *parallel*.

Second, ExMatEx benchmarks include many external libraries which increase their instruction footprint. This consequently increases the number of branch instructions, and as we shall see, complexity in predicting them, increasing the number of misses in the appropriate front-end structures. If our analysis were done only with SPEC OMP and NPB benchmarks, we could have ignored the impact of these facts, leading us to some wrong conclusions and findings.

Characteristic 5. Sequential and parallel code sections in HPC applications show different behaviour. With a comparable number of instructions executed sequentially, the analysis of ExMatEx workloads is essential for this work while the use of SPEC OMP and NPB benchmarks alone can be misleading for some metrics.

On the other side, the amount of instructions executed in serial directly depends on the number of threads running the application and the size of input set. For example, running *fma3d* and *nab* benchmarks from SPEC OMP with *train* inputs and eight threads gives around 25% of all instructions executed in serial by master thread. With the same inputs as we use here (*reference*) but running 64 threads, the fraction of serial code increases to 18% and 19% (from 4% with eight threads). Today, Intel's Xeon Phi cards and IBM's Power8 CMPs support around 200 threads. As the number of cores and/or hardware threads per CMP increases, handling the serial parts of parallel applications may become crucial. Our analysis so far shows not just that the HPC benchmarks are different from desktop, but that also serial code sections are different from parallel inside an HPC application.

IV. Microarchitecture Dependent Workload Characterization

Driven by the observations in the previous section, we analyse how we can accommodate the core front-end structures for HPC applications. We focus on branch predictors, branch target buffers, and instruction caches.

A. Branch Predictor

Characteristics 2 and 3 demonstrates the existence of a small number of biased branches in HPC applications. It suggests the high predictability and use of simpler and smaller branch predictors that can provide the same performance as the ones we can currently find in today's CPUs. To evaluate this idea, we implement a pintool with three different branch predictors:

- **gshare** - branch prediction that uses one global history table and branch history register (BHR) XORed with branch address to index the history table [18].
- **tournament** - the branch predictor implemented in the Alpha 21264 processor [19]. It has two branch predictors, one based on private and the other on global history tables, and the one which is currently more successful predicts the outcome of a branch.
- **TAGE** - a modern branch predictor that relies on tagging the entries and capturing different global history lengths [20]. It uses a base predictor (bimodal) and a set of tagged tables indexed using different history lengths that form a geometric series.

To make a fair comparison between branch predictors, we evaluate configurations that have the same hardware cost. Table II gives an overview of the parameters used for different branch predictors, so they have approximately the same size. We define two sets of configuration parameters, *small* with a 2 KB hardware budget and *big* with 16 KB. We take this as a reasonable assumption in a lean-core design given that on the 2nd Championship Branch Prediction competition [21], 4 KB and 32 KB budgets are used for heavy-weight cores.

Since HPC workloads spend most of their time inside loops and the majority of taken branches are backwards, we also check how a loop branch predictor (LBP) affects mispredictions when it is added to the small predictors analysed here.

TABLE II: Size parameters and hardware cost of evaluated branch predictors. Parameter n stands for the number of address bits used to index the tables, and parameter m stands for branch history length.

Predictor	Hardware cost (bits)	Size parameters ~2 KB (small)	Size parameters ~16 KB (big)
gshare	2^{m+1}	$m = 13$	$m = 16$
tournament	$2^n(m+2) + 2^{m+2}$	$n = 10, m = 8$	$n = 12, m = 14$
TAGE	according to [21][2]	2 tables	12 tables

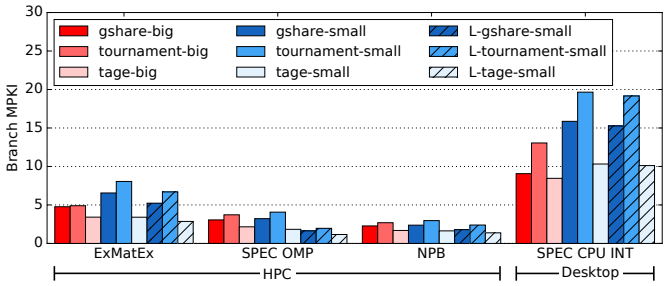


Fig. 5: Branch MPKI for different branch predictor configurations and benchmark suites.

An LBP tries to identify loops with a constant number of iterations. The prediction is by default provided by the base predictor, but, in cases where high confidence is achieved, the prediction from an LBP is taken as the final decision. We implement a 64-entry LBP with an approximate hardware budget of 512 B.

Figure 5 shows the number of branch mispredictions per kilo instructions (branch MPKI) for every branch predictor and three configurations per BP: big, small, and small with an LBP (bars with prefix L on a graph). There are several things to observe here. First, Figure 5 demonstrates the difference between desktop and HPC workloads for the same configurations and types of branch predictors. As pointed by Characteristics 1 and 2, HPC workloads have fewer branch instructions per execution and more biased branches. This results in SPEC CPU INT applications having around 3× higher branch MPKI compared to ExMatEx ones, and around 5× compared to SPEC OMP and NPB ones. For every HPC benchmark suite, sequential parts have higher branch MPKI than its parallel parts, but lower than SPEC CPU INT.

Second, it is clear that TAGE outperforms the other two branch predictors for all cases. This holds not just on per suite, but also, on per benchmark level. TAGE is much better in reducing the interference or aliasing which occurs when different branch instructions point to the same prediction bits. With the simple usage of lower address bits or XORing them with a history register, different branch instructions can map to the same prediction entry which reduces the effective usage of a prediction table. It can even be destructive if the branch

[2]The original proposal considered 32KB hardware budget and 12 tagged components. For our *big* configuration, we use the same number of components with half the number of entries. For *small* configuration, we use just two tables (for history lengths of 4 and 16) with approximately 3× fewer entries per table.

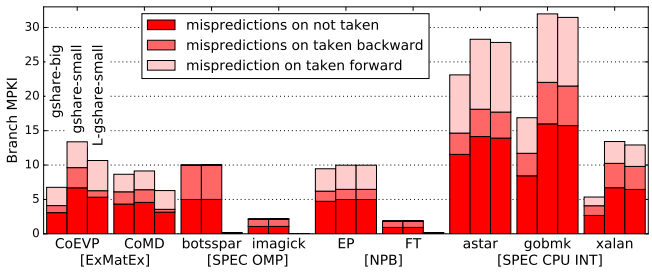


Fig. 6: Branch MPKI breakdown for gshare branch predictor and a subset of workloads. We distinguish mispredictions on not taken, taken backward, and taken forward branches.

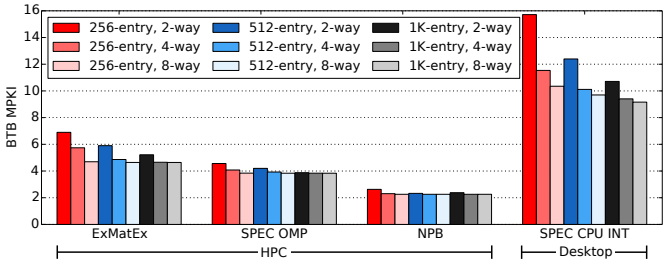


Fig. 7: BTB MPKI for different number of entries and associativity. We use branch address to index BTB.

instructions take different directions. By (partially) tagging its entries, TAGE eliminates (reduces) this effect. Also, TAGE has multiple components each for a different global history length from very short to very long. Compared to the other two branch predictors and for HPC benchmarks, TAGE provides almost the same branch MPKI values regardless of its size. With just a 2 KB hardware budget, small TAGE is better than 16 KB big gshare or tournament predictor.

Figure 5 also demonstrates how desktop and HPC applications are different in exploiting the LPB. For each benchmark suite, it shows the reduction of branch MPKI values when an LPB is implemented together with a small base predictor. It barely reduces the number of misses for desktop applications. On the other hand, HPC applications, both sequential and parallel code sections, benefit from an LPB. Still, reducing the size of branch predictor increases the MPKI values in serial parts. These results show that a core used to execute parallel HPC code should have a branch predictor tuned differently than the one used to run desktop applications. With biased and mostly backward taken branches, long basic blocks, and low fraction of branch instructions in the instruction mix, smaller and properly configured branch predictors can be used in HPC cores without performance loss.

Implication 1. HPC and desktop applications express different behaviour, thus require different tuning of a branch predictor. A loop BP is essential for HPC applications, while desktop workloads benefit from bigger BPs.

Figure 6 breaks down the branch MPKI values obtained with gshare predictor for a subset of workloads. A branch misprediction can be caused by a not taken, a taken backward, or a taken forward branch instruction. As expected, the presence of a loop BP reduces the number of branch mispredictions on taken backward branches, especially for HPC workloads. While it has a moderate effect on branch MPKI values for benchmarks like *CoEVP* and *CoMD*, in cases of *botsspar* and *imagick*, an LBP eliminates the branch mispredictions, not just the taken backward, but also the not taken ones. After the last loop iteration when the branch should not be taken, a two-bit entry in a gshare table would miss because the saturating counter is in a *strongly-taken* prediction state, while an LBP knows exactly how many iteration that loop will execute. Still, there are some HPC benchmarks where the presence of a loop BP has no effect on a branch MPKI value such as in case of *EP*. Looking at the SPEC CPU INT benchmarks, taken backward misses exist but they are not reduced by an LBP since those are not part of loop structures. It is interesting to note that the majority of all mispredictions comes from the not taken branches, for all benchmark suites.

TAGE branch predictor shows similar behaviour as gshare on Figure 6 but with lower branch MPKI values and without the difference between *big* and *small* configurations for HPC benchmarks. An LBP is also beneficial for TAGE but mostly reducing mispredictions on not taken branches. When the control flow inside the loop is regular, TAGE predictor is able to predict loops with constant number of iterations, just as an LBP. On the other hand, when the control flow in the loop body is changeable, TAGE predictor may fail to correctly predict the exit of the loop [21].

B. Branch Target Buffer

The branch predictor provides information about whether the next branch will be taken or not taken. Still, it does not supply the target address in case the branch is predicted as taken. For that, we use the branch target buffer (BTB) which is implemented as a cache that stores the branch target address for taken branches. We use the current instruction address to index and lookup in the BTB and, if the address is found, then the instruction being fetched is a taken branch, and the data part of the entry contains the next PC after the branch. In the BTB, we store only branches predicted as taken since not-taken ones will continue fetching from the next sequential instruction. With a correctly predicted outcome of a branch and a correct target address stored in the BTB, we have a zero branch penalty.

Figure 7 shows how the number of misses in the BTB depends on its size and associativity. Changing the BTB size from 256 to 1024 entries, has no impact on the number of misses for HPC applications. High associativity is needed to reduce the aliasing problem, especially for ExMatEx benchmarks, mostly due to a simple modulo indexing to the BTB. Characteristics 1 and 2 state that HPC benchmarks have a small number of branches and they are strongly biased. Once when BTB stores a destination address for a taken branch, that branch is probably going to be taken the next time it occurs

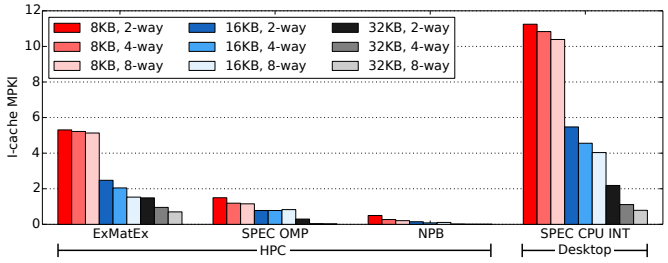


Fig. 8: The average I-cache MPKI values for all benchmark suites. The cache line is 64 B.

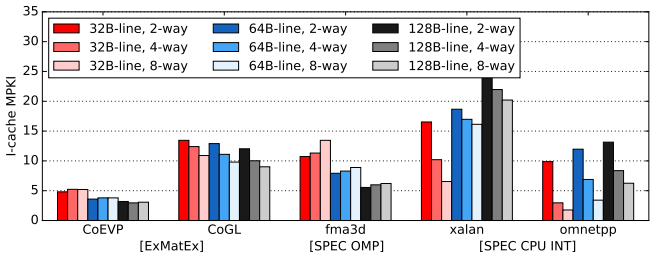


Fig. 9: I-cache MPKI values for some specific benchmarks. The cache size is 16 KB.

with the same destination address. This results in the same MPKI values regarding the size of the BTB. Similar to the findings in the previous Section, desktop applications show higher BTB MPKI values for the same BTB configurations compared to HPC applications. Bigger branch target buffers provide significantly lower BTB MPKI, such as in cases of *gcc*, *gobmk*, and *sjeng*.

Implication 2. A low(er) number of entries in the BTB is enough for HPC applications.

C. Instruction cache

Characteristic 3 says that HPC applications have a small dynamic instruction footprint. Moreover, most of the execution time is spent in loops where only a few basic blocks are executed many times. Figure 3 shows that for parallel parts of HPC benchmarks about 99% of all instructions are fetched from less than 16 KB of memory. On the other side, an I-cache factors out a considerable part of power and area on lean cores. We check how different I-cache sizes and associativities impact the number of misses.

Our pintool simulates the I-cache behaviour throughout the execution. In the beginning, it creates a cache structure with the specified characteristics such as cache size, line width, and associativity. We implement LRU replacement policy.

Characteristic 4 points that HPC applications have long basic blocks. Once we fetch an I-cache line, we extract the instructions sequentially, without accessing the I-cache again, until we reach the end of a line or a taken branch. Due to the long distance between taken branches (see Figure 4) and without any alignment techniques, even 128B-wide I-cache lines have a high usefulness, 71% on average. We define usefulness as the number of different bytes accessed in a line out of the total line size. For the same line width, SPEC CPU INT has only 33% usefulness. Besides that, wider cache lines also reduce the number of accesses to the I-cache.

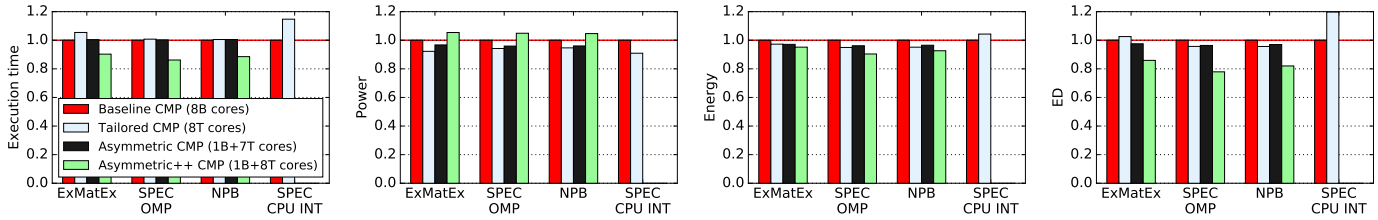
Figure 8 shows how the number of misses in the I-cache depends on its size and associativity, averaged per benchmark suite. Desktop applications, with their large static code footprints, need larger I-caches. Reduction in size is not an option for them since the use of a 16 KB I-cache increases the MPKI 2.5× compared to a 32 KB I-cache. For all benchmarks in SPEC OMP (except the *fma3d*) and NPB suites, even an 8 KB I-cache provides MPKI values bellow 1. ExMatEx applications stress more the I-cache. They have larger static and dynamic instruction footprints, as we saw in Figure 3. For them, an 8 KB I-cache is not an option. With high associativity and 128 B lines, a 16 KB I-cache increases the MPKI from 1 to 2 on average, compared with a standard-size 32 KB I-cache. Instructions executed in sequential regions miss by 50% more on average, and in case of *CoSP* up to 2×, compared to instructions from parallel regions.

There is an interesting observation analysing the impact of the I-cache line width on the MPKI values. Figure 9 shows these dependencies for a subset of benchmarks. While wider lines reduce the number of misses in the I-cache for HPC applications, for SPEC CPU INT workloads is the opposite. For a fixed size and associativity, HPC applications miss by 16% less in a 128B-line than in a 32B-line I-cache. For the same comparison, SPEC CPU INT benchmarks have 19% more I-cache misses on average. This is a direct consequence of our Characteristic 4. With short basic blocks and short distance between taken branches, wider cache lines used with desktop applications have low usefulness and reduce the number of total cache lines available in a fixed-size I-cache. On the other side, HPC workloads benefit from wider I-cache lines, not just due to reducing the number of accesses to the I-cache but also from the high usefulness of wide lines. The existence of hot code regions, such as loops, forces the running thread to execute a few basic blocks multiple times. No matter how large static code is (due to external library linking or any other reason), dynamic instruction footprint remains small and able to fit in less than 32 KB of cache memory.

Implication 3. With small dynamic instruction footprints and long basic blocks, HPC benchmarks can use smaller I-caches with wider lines without performance loss.

V. Impact on Performance, Power and Area

As our Section III highlights, HPC workloads have specific code characteristics. They have a low number of biased, and mostly backward taken branches. Dynamic instruction footprints are small and basic blocks long. Those results suggest a redimensioning of the core front-end structures for HPC, such as I-cache, branch predictor, and BTB. In this section, we evaluate power and area savings of downscaling those



(a) Execution time (b) Power (c) Energy (d) Energy-Delay

Fig. 10: Normalized execution time, power, energy, and energy-delay (ED) product for different CMP configurations and averaged per benchmark suite. We analyse only cores and L2 caches since the rest of CMP is shared and same for all configurations. Asymmetric++ CMP has the same area budget as Baseline CMP.

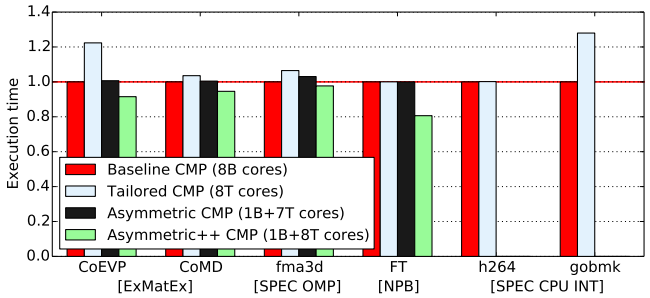


Fig. 11: Execution time for a subset of benchmarks, normalized to a baseline CMP configuration.

TABLE III: I-cache, BP, and BTB share in total area and power budget on a Cortex-A9 core level. BP has 12-cycle miss penalty. The numbers are obtained using McPAT and CACTI tools with processing technology of 40 nm.

		Area[mm^2]	Power[W]
Baseline	**Total core**	**2.49 (100%)**	**0.85 (100%)**
	I-cache (32 KB, 64 B line)	0.31	0.075
	Big BP (16 KB)	0.14	0.032
	BTB (2K entries)	0.125	0.017
Tailored	**Total core**	**2.11 (84%)**	**0.79 (93%)**
	I-cache (16 KB, 128 B line)	0.14	0.049
	Small BP with LBP (2.5 KB)	0.04	0.011
	BTB (256 entries)	0.022	0.002

structures to a configuration that may not impose performance decrease based on our characterization.

We use Sniper [8] and McPAT [7] to simulate the performance impact and project power and area savings by tailoring the core front-end for HPC applications. 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. It also has similar area footprint compared to an IBM BlueGene/Q core and recent research works consider ARM a potential player in the HPC market [22]. In Sniper, we configure an eight-core CMP with Cortex-A9-like cores, private 256 KB L2, and 4 MB shared L3 cache. HPC applications are run with eight threads (or processes) while SPEC CPU INT are run sequentially. Here, we simulate up to 100 billion instructions starting from the first parallel region. For our baseline core model, we use 32 KB, 64B-line I-cache, 16 KB tournament BP, and 2K-entry BTB. Based on the results from our previous Sections, for the alternative HPC-core design we simulate 16 KB, 128B-line I-cache, 2 KB tournament predictor with loop BP, and 256-entry BTB. We refer to it as a tailored core model.

Figure 10a shows the average execution time normalized to the baseline CMP configuration. Figure 11 is similar, presenting the same metric for a subset of benchmarks. As expected, reducing the sizes of front-end structures is not acceptable for desktop applications, although, in some particular cases, it provides no performance degradation (like for *h264*). They need large branch prediction structures to handle complex branch instructions and large I-cache to store the code. SPEC OMP and NPB benchmarks increase their execution time by less than 1% when they are executed on a CMP made out of tailored cores compared to the baseline ones. Among NPB workloads, there is no a single benchmark with more than 3% of execution time increase, while for SPEC OMP, only *fma3d* demonstrates a significant performance loss of 6%, mostly due to the I-cache misses. Running ExMatEx benchmarks on an eight-core tailored CMP increases the execution time by almost 6% on average, hurting four out of eight workloads (*CoHMM*, *CoEVP*, *CoMD*, and *CoGL*). The highest is *CoEVP*, whose execution time goes up by 22%.

As we mentioned before, *CoEVP* benchmark with default inputs and running eight threads spends around 35% of its time inside the serial code. Binding the master thread to run on a baseline core, while the rest of threads run on tailored cores (Asymmetric CMP) provides the same performance as running this benchmark on an eight-core Baseline CMP. This shows that heterogeneity which already exists in HPC systems with accelerators, should be implemented even deeper, on a CMP level. Used as a stand-alone component, accelerators as Intel Xeon Phi may suffer from either under-dimensioning the master core (and slowing down the serial part), or over-dimensioning the workers (and wasting resources on them). Asymmetric CMP designs are already present in different markets (IBM's Cell or ARM's big.LITTLE), and our results show that a similar approach has an advantage in HPC as well.

Implication 4. CMPs used in HPC should be composed out of many tailored cores optimized for parallel, and one (or few) baseline cores designed for sequential executions.

In the baseline configuration, a 32 KB I-cache with 64 B lines contributes with 12% and 9% of the total area and power core budget, respectively. A 16 KB branch predictor, implemented as a tournament predictor in McPAT and thus in Sniper for consistency, factors out around 5% in area and 4% of core power. A 2K-entry BTB in baseline contributes with 5% and 2% of the total core area and power budget. Table III provides these numbers in absolute values.

As results suggest the use of a smaller I-cache, smaller BP with LBP, and a BTB with less entries, we compare the baseline numbers with the ones obtained configuring a 128B-line 16 KB I-cache, 2 KB BP, and a 256-entry BTB. Reducing the sizes of these hardware structures gives 16% savings in area and 7% savings in power at the core level.

Saving this amount of area per core on an asymmetric eight-core CMP, opens the opportunity to add an extra core. With the abundant TLP, we can scale performance with core count under the same area budget. Figures 10a and 11 compare the execution times of HPC workloads on an Asymmetric++ CMP composed out of one baseline and eight tailored cores to a Baseline CMP composed out of eight baseline cores. For the same area budget, Asymmetric++ CMP provides 12% time reduction on average and up to 20% for *FT*.

The rest of plots on Figure 10 show normalized power, energy, and energy-delay (ED) product for different CMP configurations. There is an interesting tradeoff present on Figure 10b. Power is estimated as a sum of the static and dynamic power of private structures: cores and L2 caches. Downsizing the front-end structures we save static power and reduction of dynamic power comes mostly due to the increased execution time for a Tailored CMP. With an additional core, Asymmetric++ CMP increases the power budget by 4% compared to Baseline CMP, on average. With 12% performance improvement and within the same area, this translates into 8% of energy savings and reduction of ED product by 18%.

VI. Related Work

Understanding the workload characteristics is important in the design of efficient computer architectures. Our work targets HPC workload characteristics affecting the core front-end. In contrast to todays fat-core CMPs used in HPC, emerging lean-core ones spend a considerable part of the area and power budget on the front-end hardware structures. There is a need to perform a deep study on HPC and scientific applications at the core level to avoid overprovisioning and increase the performance per power and area. This paper addresses this need, provides a comprehensive characterization and, based on it, a proposal to downscale the core front-end of lean cores.

There is a broad scope of previous application performance analyses run in HPC systems. Most of them evaluate inter-node communication overheads, scalability, bandwidth requirements, and data access behaviour [23], [24]. Our work targets HPC workload characteristics affecting the core front-end. Opposite to todays heavy-core CMPs used in HPC, emerging lean-core ones spend a considerable part of the core area and power on the front-end hardware structures. There is a need to perform a deep study on HPC and scientific applications at the core level to avoid overprovisioning and increase the performance per power and area. This paper addresses this need, provides a comprehensive characterization and, based on it, a proposal to downscale the core front-end.

Vector processors are particularly useful for HPC applications. A single vector instruction replaces an entire loop, and so the instruction fetch and decode bandwidth needed to keep multiple functional units busy is reduced. Many aspects of the analysis performed in this paper were done in the past in the context of vector machines [25], [26]. Scientific and engineering applications have small instruction footprints, long basic blocks, and low control divergence which makes them suitable for SIMD execution. Nowadays, Intel's Xeon Phi cores [27] and Fujitsu's SPARC64 series of chips [28] implement wide vector units to exploit these code characteristics and gain performance. Our work revisits these findings considering modern HPC workloads and in the context of CMPs made out of light-weight out-of-order cores.

Current GPU cards used as accelerators in HPC systems have completely redesigned front-end compared to current CPUs. Power-hungry branch prediction structures are not implemented, and pipeline stalls caused by prolonged branch resolution are leveraged by running many threads concurrently. A programmer has to be aware of reducing the control divergence among threads in a warp since they all execute the same instructions in any given cycle. A recent study shows that about 95% of branches executed on GPUs can be correctly predicted either with a bimodal or a branch predictor based on local history tables [29]. The cache hierarchy that services the instruction supply is well suited for HPC applications. The first-level I-cache is small (4 KB) with wide cache lines (256 B) [30]. Moreover, all of the scalar cores in a single streaming multiprocessor share one I-cache, with threads fetching and executing the same instruction in lock-step mode every cycle [31]. We believe that similar front-end organization should be designed for future CMPs used in HPC, allowing each core to run its thread independently.

There are examples where commodity CPUs have been redesigned to suit better an application domain. A recent study calls for a change in future core design identifying the key micro-architectural needs for emerging scale-out workloads as the opposite of traditional scale-up applications used in data centres [32]. Server applications have large instruction footprints and most stall cycles come from I-cache misses [33]. Because of that, ARM's Cortex-A57 cores, used in micro-servers, have a larger 48 KB I-cache to reduce the impact of I-cache misses [34]. An Intel Xeon Phi core has 512-bit wide vector processing unit so it can exploit the SIMD characteristics of scientific codes [27]. Our findings suggest that a similar core tailoring can be applied to lean-core CMPs used in HPC by redimensioning the existing structures based on application demands.

Asymmetric CMP design refers to a number of cores that use the same instruction set architecture but deliver different performance, and thus, have different power and area demands.

Properly distributing a given hardware budget, an asymmetric CMP implements different types and number of cores to suit various needs of running code. It outperforms a traditional symmetric CMP by running sequential code on one (few) heavy-weight cores and parallel code on many lean ones [35]. HPC applications show similar behaviour, with the sequential code being different from parallel, thus requiring different core front-end tailoring on a single CMP. Downsizing the front-end structures of cores that execute the parallel code, we provide enough space for adding an extra core which translates into higher performance.

VII. Conclusion

Characterizing 29 applications from three generations of benchmark suites, and comparing them with desktop benchmarks, we found a set of valuable insights about the HPC workloads regarding their requirements and effect on the core front-end hardware structures.

HPC applications have a fewer branch instructions, highly biased, and mostly backward taken. The dynamic code footprint of HPC applications is small, and most of them fit in 16 KB. Basic blocks are long, and distance between taken branches even longer, which enables the usage of wider I-cache lines. Branch predictors should be tailored for HPC applications and augmented with a loop branch predictor. Moreover, the results show that HPC benchmarks are not sensible to the size of the BTB (due to the small number of branch instructions) as long as BTB associativity is high.

Compared to traditional desktop and server applications, we find that the demands of HPC applications are lower with regards to the core front-end structures. The conclusion is that the front-end is over dimensioned for these applications and, therefore, we propose a downscaling to save area and power while maintaining the same performance. This holds for cores running the parallel regions of the code but not for the one that runs the sequential sections. Our tailored core front-end configuration requires 16% less area and 7% less power in a lean-core design. On an octa-core CMP, these savings can be used to add additional core and obtain 12% performance improvement for the same area budget.

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] P. Kogge et al, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[2] J. Shalf et al, "Exascale computing technology challenges," in *VECPAR'10*. Springer.

[3] A. Duran et al, "The Intel® many integrated core architecture," in *HPCS'12*. IEEE.

[4] R. A. Haring et al, "The IBM Blue Gene-Q compute chip," *MICRO'12*.

[5] W. J. Dally et al, "Efficient embedded computing," *Computer Society*, 2008.

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

[7] S. Li et al, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO'09*. IEEE.

[8] T. E. Carlson et al, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011.

[9] C.-K. Luk et al, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm SIGPLAN Notices*. ACM, 2005.

[10] T. C. Germann et al, "Exascale co-design center for materials in extreme environments," 2013.

[11] V. Aslot et al, "SPEComp: A new benchmark suite for measuring parallel computer performance," in *OpenMP Shared Memory Parallel Programming*. Springer, 2001.

[12] H. Jin et al, "The OpenMP implementation of NAS parallel benchmarks and its performance," Technical Report NAS-99-011, NASA Research Center, Tech. Rep., 1999.

[13] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, 2006.

[14] L. Spracklen et al, "Effective instruction prefetching in chip multiprocessors for modern commercial applications," in *International Symposium on High-Performance Computer Architecture*. IEEE, 2005.

[15] S. J. Patel et al, "Increasing the size of atomic instruction blocks using control flow assertions," in *MICRO'00*. IEEE.

[16] A. Ramirez et al, "Code layout optimizations for transaction processing workloads," in *SIGARCH Computer Architecture News*. ACM, 2001.

[17] G. Reinman et al, "Fetch directed instruction prefetching," in *MICRO'99*. IEEE.

[18] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, Tech. Rep., 1993.

[19] R. E. Kessler et al, "The Alpha 21264 microprocessor architecture," in *International Conference on Computer Design: VLSI in Computers and Processors*. IEEE, 1998.

[20] A. Seznec et al, "A case for (partially) TAgged GEometric history length branch prediction," *Journal of Instruction Level Parallelism*, 2006.

[21] A. Seznec, "The L-TAGE branch predictor," *Journal of Instruction-Level Parallelism*, 2007.

[22] N. Rajovic et al, "Supercomputing with commodity Cpus: are mobile SoCs ready for HPC?" in *SC'13*. ACM.

[23] S. R. Alam et al, "Characterization of scientific workloads on systems with multi-core processors," in *IISWC*. IEEE, 2006.

[24] R. Cheveresan et al, "Characteristics of workloads used in high performance and technical computing," in *ICS*. ACM, 2007.

[25] L. Kontothanassis et al, "Cache performance in vector supercomputers," in *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 1994.

[26] D. J. Kuck et al, "Measurements of parallelism in ordinary FORTRAN programs," *Computer*, 1974.

[27] A. Ramachandran et al, "Performance evaluation of NAS parallel benchmarks on Intel Xeon Phi," in *ICPP'13*. IEEE.

[28] T. Maruyama et al, "Sparc64 VIIIfx: A new-generation octocore processor for petascale computing," *IEEE micro*, 2010.

[29] M. Arora et al, "Redefining the role of the CPU in the era of CPU-GPU integration," *Micro, IEEE*, 2012.

[30] H. Wong et al, "Demystifying GPU microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE.

[31] E. Lindholm et al, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, 2008.

[32] M. Ferdman et al, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ACM SIGPLAN Notices*. ACM, 2012.

[33] P. Tözün et al, "From a to e: analyzing TPC's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored," in *EDPT'13*. ACM.

[34] J. Bolaria, "Cortex-A57 Extends ARM's Reach," *Microprocessor Report*, 2012.

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