Fast, Accurate Processor Evaluation through Heterogeneous, Sample-based Benchmarking

Pablo Prieto, Pablo Abad, Jose Angel Gregorio, Valentin Puente

Abstract— Performance evaluation is a key task in computing and communication systems. Benchmarking is one of the most common techniques for evaluation purposes, where the performance of a set of representative applications is used to infer system responsiveness in a general usage scenario. Unfortunately, most benchmarking suites are limited to a reduced number of applications, and in some cases, rigid execution configurations. This makes it hard to extrapolate performance metrics for a general-purpose architecture, supposed to have a multi-year lifecycle, running dissimilar applications concurrently. The main culprit of this situation is that current benchmark-derived metrics lack generality, statistical soundness and fail to represent general-purpose environments. Previous attempts to overcome these limitations through random app mixes significantly increase computational cost (workload population shoots up), making the evaluation process barely affordable. To circumvent this problem, in this paper we present a more elaborate performance evaluation methodology named BenchCast. Our proposal provides more representative performance metrics, but with a drastic reduction of computational cost, limiting app execution to a small and representative fraction marked through code annotation. Thanks to this labeling and making use of synchronization techniques, we generate heterogeneous workloads where every app runs simultaneously inside its Region Of Interest, making a few execution seconds highly representative of full application execution.

1 Introduction

Reaching nowadays the 50th anniversary of the commercialization of the first CPU-on-a-chip [1], we have witnessed technology evolution that has turned computing devices into the core component of nearly any activity in our everyday life. Currently, despite the recent emergence of domain-specific processors [2] (led by GPU computing for deep-learning applications), the general-purpose computing model still constitutes a relevant fraction of the semiconductor market. In this computing model, the processor runs applications (often concurrently) with quite dissimilar characteristics. Under these conditions, measuring (and defining) the expected processor behavior (performance) is challenging. Each piece of application code can interact in a different way with processor microarchitecture and concurrency might introduce "unwanted" cross-effects, affecting overall system behavior negatively. Benchmarking is the predominant methodology employed for performance evaluation, providing a standardized way to measure and compare alternative processors. A meticulous selection process is usually performed in order to define a reduced set of applications that are sufficiently representative of a much broader usage scenario, corresponding to a specific target environment (scientific [3], NoSQL serving [4], Machine Learning [5], etc.) or one closer to the "general purpose" scenario [6][7]. Unfortunately, many of these benchmarking suites present two important drawbacks. First, the number of applications under evaluation is usually limited to a few tens. Even when considered a representative sample, if we want to model performance as a random variable, the available number of values is usually below the recommended limit to reach a reasonable confidence margin in the evaluation process. Second, most of the CPU market share corresponds to environments (desktop, cloud computing) where there is limited control of the kind of applications that run on the same processor chip simultaneously. Current benchmarking metrics (latency, rate-mode throughput) might not suffice to gain insight into the consequences of this resource sharing. Therefore, this makes the re-design of the "representative workload" and "representative metric" concepts necessary.

A straightforward technique to increase the benchmark size (and hence the statistical soundness of the results), targeting both heterogeneous and concurrent environments, consists of a random mix of benchmark applications running in parallel inside the same computer [8][9][10]. To the best of our knowledge, this technique is usually employed with a single benchmark suite, and parallel execution relies merely on launching every application in a synchronous way. Despite partly solving traditional benchmarking limitations, this methodology significantly increases the computational cost of the evaluation process (to the point of being impractical in certain conditions). Relying on the same principle of random mixing, in this paper we propose a much more elaborate methodology to avoid these increased costs through the following features:

- Computational resource usage is limited to a small fraction of application code, belonging to its Region of Interest (ROI). Our preliminary explorations demonstrate that many applications from different benchmarks show a similar loop-based ROI structure that has repetitive behavior from the microarchitectural viewpoint.

All Authors are with the Computer Engineering Group of the University of Cantabria, Av. de los Castros SN, 39005, Santander, Spain. E-mail: [prietop, abadp, monaster, vpuente]@ unican.es.

- To ensure that every application runs its ROI while performance is being measured a fine-grain synchronization process is used. Additionally, automated hardware event counting during evaluation increases the variety of information available about execution, given the profuse list of events available in state-of-the-art processors.
- The methodology is generalized to any application, independently of its benchmark suite. This allows a sort of Meta-benchmarking methodology to be created, which can increase metric coverage. To do this, we formally define the code and execution conditions that must be met by a new application to be part of the random mixes.

Following the proposed methodology, we can increase performance metric representativeness, yet under constrained time. This enables the concurrent exploration of alternative performance metrics (such as fairness) and the study of diverse microarchitectural behaviors.

This work expands on previous work [11] by generalizing our methodology to multiple benchmark suites and enhancing evaluation features. In this work, we make the following contributions:

- We develop a multi-benchmark tool for exhaustive and accurate system evaluation. Thanks to the automated workload generation, execution and monitoring process, the user will gain insight into performance issues transparently and in a feasible amount of time.
- We define and standardize the process to add new benchmarks to the initial application pool. Conditions that must be met by any candidate application are defined. Around 50 applications have been profiled and employed in this work to test the methodology.
- We carry out a raw performance evaluation of two counterpart server architectures from the two main CPU vendors, AMD and Intel. Our evaluation is compared to a "conventional" one, such as the one performed through the SPEC CPU17 benchmark [6].
- Direct access to hardware counters during the ROI execution enables elaborate performance evaluation methodologies such as Top-Down [12] and more subtle microarchitectural analysis. We extend processor evaluation of micro-architectural parametrization (SMT and hardware prefetching), to prove that the technique is suitable to enhance understanding of the effect of these techniques.

2 Motivation

As mentioned in the previous section, computational cost can hinder the evaluation process when it moves from a few workloads to several hundreds. This problem has been widely addressed for simulation-based research, where the entire execution of an application is, in most cases, unattainable. To circumvent the problem, sampling techniques (i.e., measuring performance only in a relevant fraction of the original application) are usually employed [13][14][15]. Our proposal follows the same approach in a different context: evaluation of real systems when the number of workloads to be considered is impractical for full execution.

The core operation of *BenchCast* is based on a well-known observation about the execution structure found in many programs. As described in [16], computationally bound applications go through different stages of execution. They usually start with an initialization phase where data structures are set up, moving next to a stage corresponding to the bulk of the execution and ending up with a phase devoted to presenting the application's results. The central stage of the three described is usually labeled as the Region of Interest (ROI), because it corresponds to the largest fraction of execution time and is devoted to the resolution of the main tasks. For this reason, a program's ROI is the most relevant stage in terms of performance. This stage usually has a marked periodical behavior [16] because it tends to be implemented as a set of hierarchical procedures contained in a main loop. Analyzed in detail, this arrangement implies non-uniform behavior from a performance viewpoint, making it difficult to find an execution phase that is representative of the whole program's execution.

Figure 1 shows an example of this time-varying behavior for the *505.mcf* application from the SPEC CPU 2017 benchmark [6]. In both graphs, we measured the temporal evolution of alternative performance metrics (instructions per cycle, branch prediction accuracy and L1D Cache miss rate) making use of two different granularities. In the upper graph, performance metrics were collected through the Linux *perf* command [17], with a fixed period of 100 milliseconds. In contrast, for the lower graph, events were measured at the end of each ROI iteration (variable period), modifying source code to perform this task.

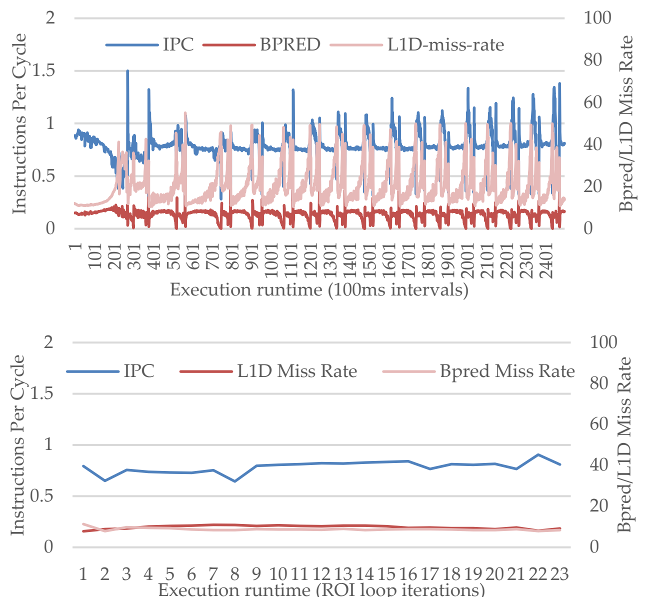


Fig. 1. Time-varying behavior for the SPEC17 application *505.mcf*. Results are shown for IPC, BPRED accuracy, L1D miss rates. (up) 100ms X-axis interval, (down) X-axis interval 1 loop iteration.

The obvious differences between the two graphs reveal a special feature of the aforementioned periodic behavior. When the sampling period is "randomly" selected as a constant time interval, the high variability makes it hard to find a single representative execution phase. In contrast, when the sampling period is somehow adapted to the internal structure of the program (fitting in this case the

length of a ROI iteration), the performance metrics become much steadier, and average metrics are close to the global ones. According to this observation, we hypothesize that the execution of a single ROI iteration can represent the whole application with accuracy.

The next step in this process consisted of the exploration of a large set of applications to verify our hypothesis. We extended this kind of analysis (see Section 3.4 for system configuration) to all the applications from three different benchmarks focused on stressing the system's processor and memory subsystem: SPEC CPU17 [6], Parsec [7] and NAS Parallel Benchmark [3]. SPEC CPU is an industry-standardized suite with 23 benchmarks (rate mode) organized in two different suites (int and float), representative of very different application areas (from desktop to scientific). Similarly, the Parsec suite contains 13 applications focused on emerging fields (computer vision, animation physics, financial analytics, etc.), attempting to be representative of next-generation software. Finally, the eight workloads from the NAS Parallel Benchmarks have a more specific target, all of them being derived from computational fluid dynamics.

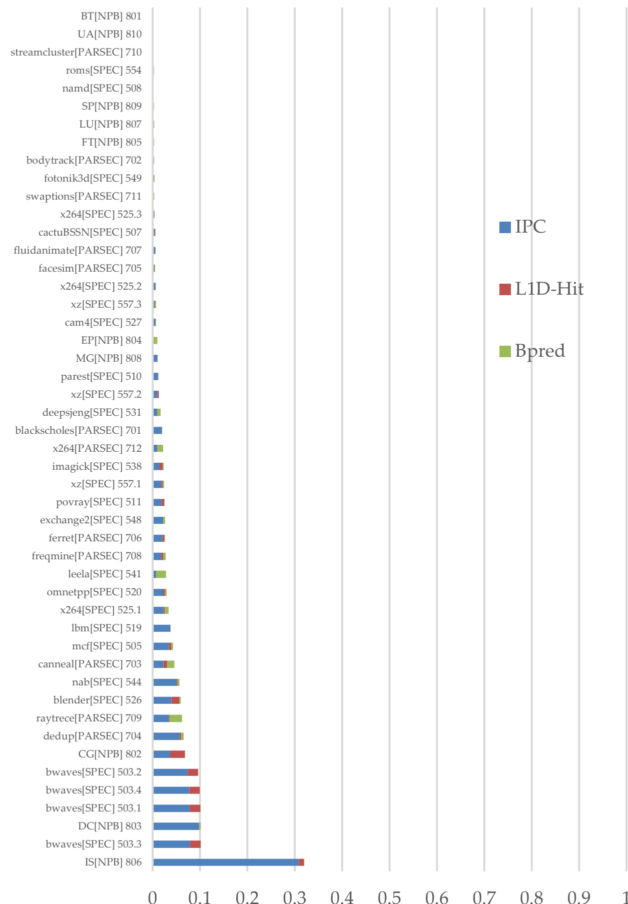


Fig. 2. Relative error comparing ROI iteration (average value) to global execution. Error was estimated for three different performance metrics: IPC, L1D Hit rate and Branch predictor accuracy.

Figure 2 summarizes the results obtained in our exploration. We were able to identify a loop-based ROI in 47 out of 50 applications (94%). For each of these 47 applications, we measured performance values for each ROI iteration, calculating average and standard deviation of each metric dataset. Next, average values were compared to full-execution results, calculating their relative error, which was the value represented by the horizontal bars in Figure 2. The relative error of each performance metric formed the graph and, as can be seen, almost every application presented a total value below 10%.

Therefore, in most cases, it seems accurate to consider that a single iteration of the main loop inside the ROI represents the whole execution with a high degree of confidence. This means it could be possible to reduce the computational effort required to evaluate heterogeneous workloads. If ROI execution can be sychronized, then simply running one (or a few) iteration of the ROI loop of each application simultaneously would be enough to characterize system performance for each workload. This is the cornerstone of the proposed methodology, mixing smart sampling and synchronization to build a computationally feasible and statistically sound evaluation methodology. Through the rest of the paper the proposal is thoroughly described (Section 3) and alternative evaluation procedures are presented (Section 4). To facilitate access to the tool by other researchers and simplify the adoption of their own modifications, a public source code repository and project management tools have been made available (https://github.com/prietop/BenchCast).

3 Methodology (BenchCast)

The three main features of *BenchCast* are described in detail in the following subsections.

3.1 Application Profiling & ROI Evaluation

Despite being found in most applications analyzed, not every workload code corresponds to the loop-ROI structure, or the observed steady state between iterations. For this reason, every new application proposed as part of *BenchCast* must fulfill the set of requirements defined in this section. The profiling process was standardized to guarantee minimal deviation between the fraction of ROI executed and the whole application. Unfortunately, given the heterogeneous nature of the methodology (multi-language, multi-benchmark, etc.), the complete automatization of this profiling process was nearly impossible, and minimal manual work was required to identify and label the ROI.

In summary, this preliminary process involves the following steps:

- **ROI identification**: the application is profiled to identify those functions consuming the largest fraction of execution time.

- **Loop labeling**: previous functions are analyzed looking for the outer loop structure. Code is annotated to measure the fraction of time spent in that loop, considering only applications returning values over 70%.
- **Variability analysis**: several performance metrics are measured for every loop iteration. Variability is measured to find out the sample size (number of iterations) required for a pre-defined error and confidence interval.
- **Execution time**: the time required to execute the number of iterations calculated previously is estimated. Only those applications with a value below a certain threshold (in this case the maximum ROI is set to 20 seconds) are eligible. Total time required to perform the measurement is highly sensitive to this parameter (and hence, the chosen threshold is relatively small).

To gain insight into this process, we will walk through a specific example in detail. Figure 3 describes the steps for the *505.mcf* application, corresponding to the SPEC CPU2017 benchmark.

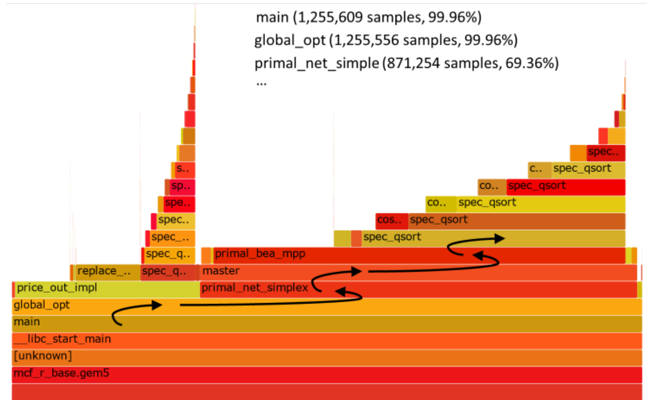
The process starts with hot code-paths exploration (ROI candidates). Stack traces are captured to generate their associated call-graph (calling relations between code functions) and the later profiling can be performed with the scripting tools provided with perf (*stackcollapse*) or by generating a graphical representation called *Flame Graph* [18]. Both solutions provide alternative representations of equivalent information. Figure 3 shows the Flame Graph for *505.mcf* (Step 1), where we can identify the function chain consuming the largest fraction of execution time. It corresponds to the following stack: *main→ global_opt→ primal_net_simplex→ master→ primal_bea_mpp→ spec_qsort*. This part of the process, automated in BenchCast, finishes by locating the source code files where these functions are defined. The information generated in this process facilitates the manual annotation performed in the next step.

Main loop identification is the only supervised action in this process. This search is performed from bottom to top of the flame graph (the bottom functions in Figure 3 are those consuming a larger fraction of time). Every function is examined looking for the outermost loop structure. In this case, *main* and *global_opt* functions can be found in the same file, *mcf.c*. It is easy to identify a while declaration inside *global_opt* consuming about 90% of the execution time. Once located, it is necessary to verify that this loop consumes a significant fraction of total execution time. In our experiments, only those loops consuming more than 70% of total execution time are considered as a suitable ROI. The loop code is annotated to measure both execution time and performance metrics for every iteration. As Figure 3 (Step 2) shows, we make use of the PAPI C interface to obtain a precise event count every iteration.

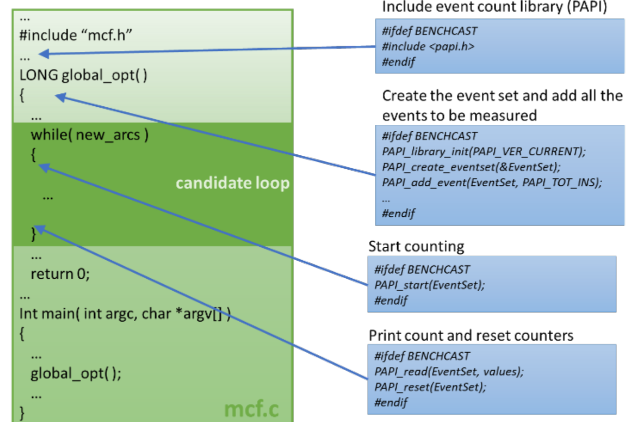
Once identified as a valid ROI loop, the next step consists of a variability analysis of performance metrics across loop iterations. The mean and standard deviation of IPC, Bpred accuracy and L1Cache hit rate values are obtained (Step 3). Making use of these values, the sample size (number of iterations to be executed) can be estimated for a pre-defined error rate and confidence level.

Previous evaluations [11] show that a 10% Error with a 95% confidence interval is enough to ensure the representativeness of the workloads generated. In the *505.mcf* application, the required number of iterations is 2 (the maximum of the three Ns obtained). As a final step, the execu-

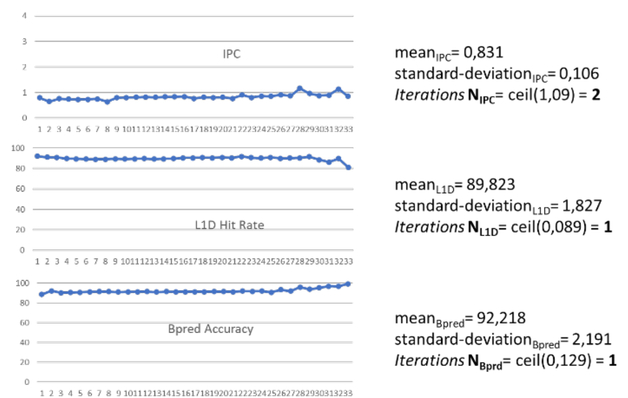
STEP 1: call-graph exploration looking for the main Loop (ROI)



STEP 2: loop identification and code annotation



STEP 3: Variability analysis and final decission



T_{EVAL} = MAX[N_{IPC}, N_{L1D}, N_{bpred}] * AVG[$T_{iteration}$] = 2 * 9,205 = **18,41**

Fig. 3. Graphic description of the profiling process required for every application to be part of BenchCast workloads (exemplified with *505.mcf*).

tion time required to run N iterations is calculated, and only when this value is below the 20-second threshold, the application is included as part of BenchCast.

A similar process to the one described here was done for every application in Figure 2. The values in Table 1 summarize the results of this profiling process, showing information about ROI fraction of total execution time (second column), per-iteration average and standard deviation of main performance metrics (IPC, L1D, L1I, Bpred columns) and N-iterations execution time (last column).

TABLE 1. APPLICATION PROFILING RESULTS

		IPC		L1D		L1I		Bpred		
App	ROI	Mean	Stdev	Mean	Stdev	Mean	Stdev	Mean	Stdev	T_{EVAL}
503.1	99.3	1.649	0.733	93.4	2.578	100.0	0.001	99.8	0.036	25.36
503.2	99.5	1.511	0.616	92.9	2.178	100.0	0.001	99.9	0.031	18.62
503.3	99.3	1.559	0.670	93.0	2.420	100.0	0.001	99.8	0.025	19.86
505	100.0	0.935	0.103	81.8	5.258	100.0	0.003	92.1	2.220	18.41
507	99.1	1.367	0.008	78.0	0.025	91.4	0.028	99.9	0.003	3.46
508	100.0	2.689	0.005	95.5	0.022	100.0	0.002	95.2	0.260	3.68
510	79.0	1.934	0.058	90.4	0.674	99.9	0.139	98.0	0.227	71.05
511	99.5	2.416	0.152	93.1	1.422	95.0	1.906	99.2	0.279	29.11
519	99.3	1.836	0.004	84.1	0.239	100.0	0.000	99.7	0.003	0.08
520	99.3	0.741	0.021	88.5	0.268	98.8	0.252	96.8	0.114	1.06
521	99.0	1.235	0.112	95.1	0.315	99.0	0.236	98.9	0.059	0.57
525.1	96.7	2.796	0.329	97.9	1.180	97.8	1.143	94.3	2.934	0.05
525.2	99.7	3.269	0.153	98.8	1.368	97.2	0.580	97.1	0.635	0.20
525.3	97.1	3.244	0.137	98.9	0.249	97.7	0.649	97.4	0.702	0.27
526	92.5	0.907	0.092	91.6	0.524	8.3	7.331	99.3	0.540	3.43
527	93.3	1.847	0.054	91.9	1.503	96.5	0.566	97.9	0.254	4.59
531	99.8	1.836	0.081	99.0	0.151	97.6	1.530	95.1	0.614	1.80
538	38.9	1.970	0.023	100.0	0.001	100.0	0.000	98.9	0.120	0.07
541	99.8	1.264	0.059	98.3	0.381	99.9	0.068	86.1	1.818	0.61
544	97.4	1.622	0.148	96.3	0.684	100.0	0.003	98.8	2.286	1.68
548	99.5	2.145	0.102	100.0	0.001	99.5	0.417	98.3	0.306	3.13
549	96.9	1.095	0.063	88.8	0.007	100.0	0.005	99.9	0.018	0.20
554	99.5	1.920	0.028	92.0	0.367	99.9	0.010	99.8	0.021	2.56
557.1	88.0	0.775	0.056	92.8	0.609	100.0	0.003	91.7	0.292	5.33
557.2	96.1	2.133	0.049	95.8	0.699	100.0	0.006	98.2	0.132	4.44
557.3	95.6	1.428	0.281	96.3	0.675	100.0	0.020	93.8	1.625	10.28
701	85.8	1.726	0.000	99.6	0.003	100.0	0.001	99.4	0.001	1.19
703	73.9	0.260	0.012	68.7	0.694	99.9	0.009	91.0	0.876	0.02
704	45.2	1.821	0.013	97.7	0.018	97.8	0.726	94.1	0.087	3.36
705	99.5	2.603	0.006	97.3	0.011	97.2	0.066	98.7	0.029	2.80
706	99.4	1.492	0.114	95.6	1.575	99.8	7.833	95.3	0.555	0.10
707	99.5	1.976	0.232	98.8	0.264	100.0	0.000	94.2	1.367	0.51
708	96.6	1.897	0.230	97.9	0.731	99.7	5.443	97.3	2.116	0.52
709	75.1	2.353	0.012	99.7	0.005	100.0	0.003	88.7	0.170	0.64
710	79.9	0.853	0.001	95.8	0.007	100.0	0.000	99.5	0.095	80.80
711	99.1	2.349	0.006	98.4	0.047	100.0	0.002	98.2	0.015	1.58
712	96.5	2.082	0.111	94.7	1.905	88.9	1.424	91.7	3.393	0.20
801	89.5	2.762	0.002	93.2	0.000	100.0	0.000	99.4	0.020	4.01

The literature does not provide a formal definition of which fraction of execution is required to establish that a portion of code makes up a ROI. To select an appropriate value for this threshold, we decided to guarantee that the IPC measured for the whole ROI and the whole application should have a relative error below 5%. A 70% ROI keeps the applications listed in Figure 2 below this error rate. Similarly, the 20-second threshold for ROI execution fulfills two conditions. First, it is small enough to ensure an evaluation process at least one order of magnitude faster than a whole one (in this case, the average execution time of whole applications is ~300 seconds). Second, it is large enough to fulfill the representativeness error and confidence interval margins for most applications.

For some applications, a large variability was observed between iterations (high Stdev values), caused mainly by a variable ROI behavior across different execution phases. In many of these cases we observed that phases respond to simple patterns, it being feasible to split the application into multiple workloads, one for each phase [11].

After this analysis, only 5 applications were ruled out. Three of them with a ROI execution fraction below the 70% limit (538, 704, 806) and the remaining two exceeding the 20-second threshold imposed for ROI execution (510, 710). Relative error and 20-second ROI are mutually related. Relaxing error-related values could lead to a smaller number of discarded applications if ROI length is maintained or to an even shorter ROI execution for the same applications.

3.2 ROI Annotation & Synchronization

Once the ROI of the selected application is known, heterogeneous workloads will be defined. This will increase the available number of samples on our evaluation mechanism. All applications running in any workload will be executing their region of interest simultaneously. Each application should execute at least one iteration of the main loop. To achieve this, we create a master application launcher that executes each application of the workload and synchronizes them at the beginning of their ROI. *BenchCast* uses a POSIX thread barrier mapped onto a shared memory region through a POSIX shared memory object. The barrier and the shared memory object are created by *BenchCast* master launcher. We append barrier calls within the ROI annotation code located in the previous section. The *BenchCast* master launcher then creates child processes for each application to be executed in the workload, attaching each process to a different core (or hardware context) of the system under evaluation, using Linux sched setaffinity system call. *BenchCast* master and the applications wait at the same barrier until all the applications reach their ROI. This process can be repeated as many times as needed, and in our experiments, workloads usually begin after all applications have executed at least one ROI loop (so the workload starts the second time the barrier is reached). Then, the barrier is raised and disabled, and measurements can begin with all the applications executing their ROI concurrently.

BenchCast comes with code annotations for SPEC17, PARSEC and NPB applications. BenchCast includes the necessary information to launch the applications of these benchmarks as well as the PATH to the local installation.

To add a new application to the pool (provided it complies

with the previous section's requirements), some information must be provided to the master launcher program, such as the PATH to the new application and its launch command.

3.3 Workload Generation and Execution

BenchCast both creates workloads and evaluates their behavior during execution. Making use of the PAPI library and attaching PAPI events to the applications executing on the system, *BenchCast* can measure any performance counter available through the PAPI interface. The PAPI library and PAPI event initialization is performed by the *BenchCast* master launcher, and the event list is provided through an easy to modify configuration file. Examples for top-down analysis and basic performance analysis configuration files are provided.

To perform an evaluation using *BenchCast*, we dynamically generate sufficient variety of workloads so the results are statistically significant. Workloads are generated choosing randomly among the available applications in the pool (SPEC2017, PARSEC and NPB out of the box). By default, *BenchCast* launches one application per available core in the system under test. If the number of selected applications is fewer than the number of available cores, multiple copies of each application are launched until all hardware contexts are allocated.

Once the applications reach the synchronization point, at the beginning of their ROIs, they start running simultaneously. The master launcher then starts the PAPI measurement, for the duration of at least one loop of the ROI (at least 20 seconds). Once the execution completes, *BenchCast* stops the measurement and stops all the applications, so the next workload execution can be initiated. The results obtained through the performance counters are written in a results file when each workload ends.

Among others, *BenchCast* provides the following parameters to perform an evaluation of a system:

- Number of cores: number of cores to use on the system. By default, the number of cores available, but a lower number can be provided and some of the cores of the system under test will not be used for the evaluation. These include simultaneous multithreading hardware contexts.
- Number of applications: number of different applications that will be used in each workload. Multiple copies of each application are launched until the selected number of cores has one application each.
- Number of workloads: number of different workloads that will be generated for the complete evaluation.
- Event list: A file containing the list of PAPI events that will be measured for each application in each workload.
- Measurement time: the execution time each application runs for the evaluation. Typically, 20 seconds, to guarantee at least one iteration of the ROI loop.

3.4 Methodology Validation

For the experiment in this section, we used a desktop-like computer configuration, an Intel i5-7500 4-Core chip running at 3.4GHz with 6MB of cache and a main memory of 16GB. The software stack corresponds to Debian 9 distribution (Linux kernel 4.9.0). 1000 random combinations are generated, enough to guarantee that variables follow a normal distribution. For TOTAL workloads, each core runs a single application of the combination in an "infinite loop" and execution is terminated when every application completes at least one execution. *BenchCast* results are obtained executing 20 seconds of their synchronized ROIs. For this number of applications and execution-time values (20 sec. ROI vs. 300 seconds average app execution time), BenchCast can reduce the computational cost from more than a week to only 20 hours. These savings remain constant for each experiment performed, meaning that all the data included in this paper were obtained in less than 7 days, in contrast to the multiple months that would have been necessary without the proper methodology.

Figure 4 shows the IPC histogram for both experiments. The degree of similarity between the two measurements, suggests that the performance figures of *BenchCast* are equivalent to full application, at a fraction of the computational cost. This postulation is statistically supported through a two-sample Kolmogorov-Smirnov (henceforth KS) test [19]. This is a nonparametric test used to compare the equality (probability distribution fit test) of two data samples. The KS statistic is based on the largest vertical difference between the cumulative distribution function (CDF) of both samples and is defined as,

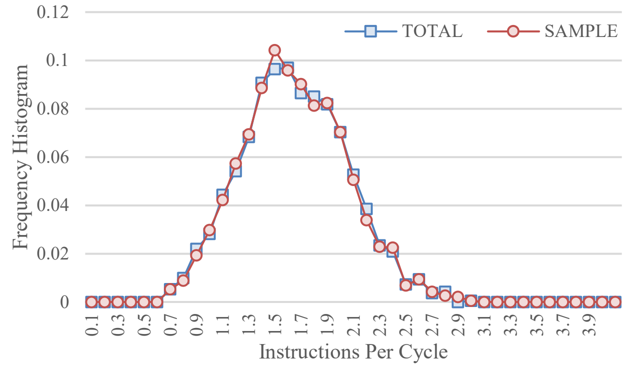


Fig. 4. Distribution comparison with a 1000-workload sample. Histograms for ITER (*BenchCast*) and TOTAL executions.

$$D = max_{1 \le i \le N} |CDF_{ITER}(i) - CDF_{TOTAL}(i)|$$

where CDF_{ITER} and CDF_{TOTAL} are the samples under test and N the number of observations. This KS statistic is meant for testing the (null) hypothesis of both samples coming from a common distribution. The hypothesis regarding the distributional form is rejected if the test statistic D is greater than a critical value obtained from a table [19]. In this case, with a number of samples larger than 40 and a 1% significance level, the critical value can be calculated as,

$$C = \frac{1.63}{\sqrt{N}}$$

According to the data collected for both samples, the maximum difference is 0.0109, which is less than the critical value. Therefore, we would accept, at the 1% significance level, the hypothesis that both sample distributions come from the same population.

To gain even more insight into similarity we evaluate the

random variable $e(w)$ defined as

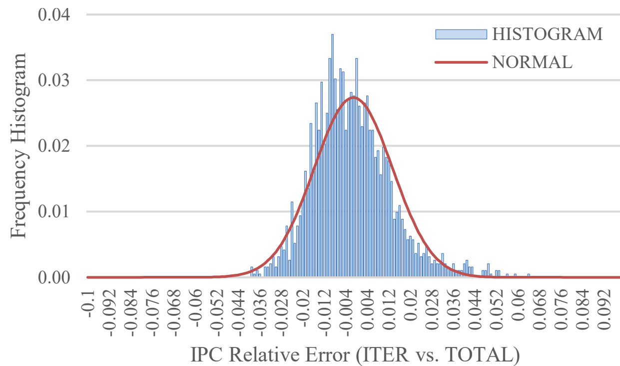


Fig. 5. Histogram and estimated normal distribution for the "relative error" random variable.

$$e(w) = 1 - \frac{IPC_{iter}(w)}{IPC_{total}(w)}$$

In words, $e(w)$ is the per-workload IPC relative error between TOTAL and ITER results. Both IPC_{sample} and IPC_{total} can be approximated by a normal distribution [11], and as results in Figure 5 show, the generated error variable $e(w)$ seems to fit into a similar kind of distribution. As can be seen, the average value of error distribution is 0.0012, while the standard deviation is 0.014. These values mean that error remains below 5% (4.374% exactly) with a 99% confidence level.

4 System Evaluation through BenchCast

In this section, we will describe the versatility of the BenchCast methodology to carry out alternative performance evaluations. It should be noted that all these evaluations would also be possible running complete applications, but at a prohibitive computation cost.

The flexibility of hardware performance counters enables the evaluation of many events, which provides not only performance metrics, but also enables the analysis of the hidden architectural causes explaining these results. For this reason, each of the experiments presented provides both the raw performance numbers and additional information about microarchitectural behavior, which leads to a much more consistent discussion of results.

The number of potential experiments is nearly as large as the number of events available in the performance monitoring unit. In this work we limit the content to three basic experiments that provide a reasonable idea of the strengths of our tool. These experiments are described in the next three subsections.

4.1 System-wide Performance

The first experiment will compare two alternative commercial processors using the proposed methodology. Basic performance evaluation with *BenchCast* is carried out measuring the total number of instructions retired during the 20-second ROI interval (IPC is not a valid latency metric in this case due to the different frequency of operation in the processors evaluated and potential power scaling across measurements). Two similar servers with AMD and Intel processors will be used. The first server configuration is a two-socket with two Intel Xeon Silver 4216 chips running at 2.10 GHz, with 22MB of cache and a main memory of 110GB. The counterpart server configuration corresponds to a two-socket AMD EPYC 7352 with a 24-Core processor per socket (48 cores in total) at 1.5 GHz (up to 3.2 GHz) with 128MB of cache and 110GB of main memory. The software stack is the same in both systems. The same 1000 workloads are generated, executed and profiled to obtain final results in both systems. Our performance results are compared to the SPECrate metrics collected through the "official" procedure described in the "Run and Reporting Rules" section of SPECCPU documentation [6].

Both SPECrate (above) and BenchCast (below) results are presented in Figure 6. In both cases, the performance metrics are represented with a frequency histogram (bars) and

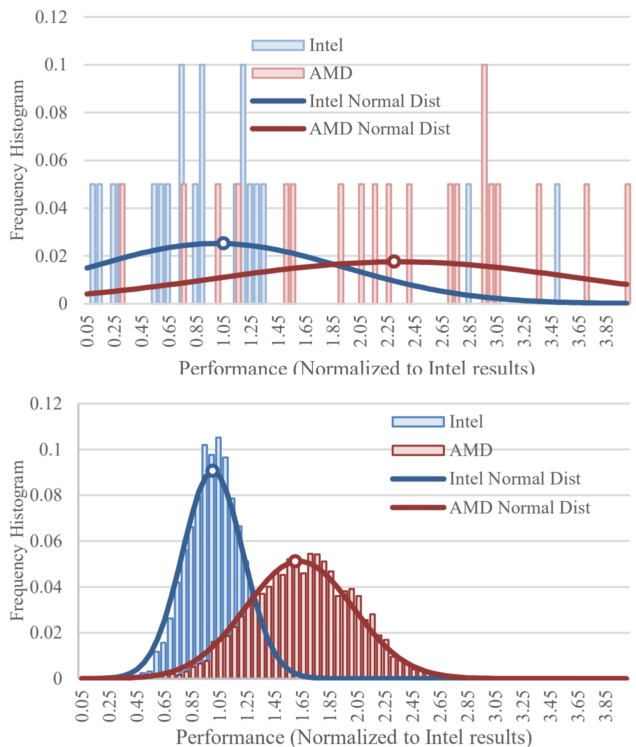


Fig. 6. Execution comparison of AMD-based and Intel-based server configurations. (Above) SPECrate methodology and (Below) BenchCast methodology.

its estimated probability distribution function (lines). Results are normalized to Intel's Average value. As observed in Figure 6.above, SPECrate evaluation estimates a 2.33 times better average performance of the AMD server compared to the Intel one. With 1.5 times the core count (32 vs. 48), AMD seems to obtain a better per-core performance than its counterpart. Unfortunately, the small number of workloads employed by SPECrate provides two probability distributions with a large standard deviation, reducing the confidence interval below 50%, which is far from statistical standards.

Comparing SPECrate results to *BenchCast* ones, we observe two significant differences. First, the number of workloads evaluated with *BenchCast* enables a drastic reduction in standard deviation. Second, the margin of AMD is reduced from 2.33 to 1.62 in this case. This result indicates that when a large, heterogeneous number of workloads is

evaluated, per-core performance becomes nearly equal in both server configurations and the only advantage of the AMD server comes from the number of cores. This substantial difference between the two evaluation methodologies could be a determining factor in a tradeoff metric such as performance-cost in certain multitenancy environments, such as cloud providers.

BenchCast enables us to move one step further in the performance comparison process. Thanks to event counting tools, we can explore in detail the divergence observed in

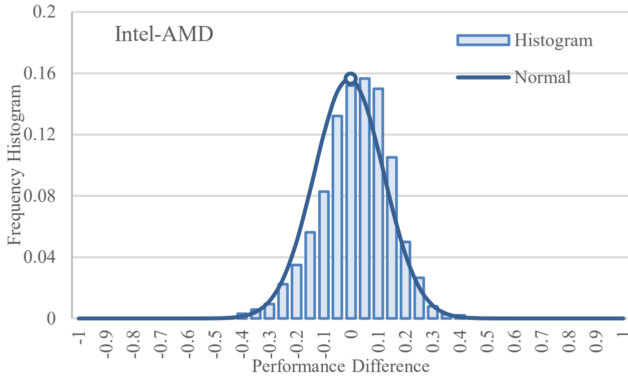


Fig. 7. Per-core Performance difference between Intel and AMD servers. Frequency histogram.

SPECrate and *BenchCast* results. To do so, we will explore the per-core performance of both processors defining the random variable *D(w)* as:

$$D(w) = Perf_{Intel}(w) - Perf_{AMD}(w)$$

D(w) is the Performance difference between AMD-based and Intel-based servers running the same workload *w*. Since both performance variables can be approximated by a normal distribution [11], *D(w)* is also normal. Since we are interested in the performance of a single core (not a hardware context, SMT is disabled for this experiment), we divide the performance results by the number of cores of each processor chip. We show the results in Figure 7. In this graph, the x-axis indicates which processor performs better (Intel for positives, AMD for negatives), confirming the similar behavior pointed out in Figure 6 (BenchCast). The estimated distribution mean is -0.01, which represents a marginal advantage of AMD cores over Intel ones. With this value, we can conclude that on average, both cores perform similarly, and AMD-server benefit is derived nearly exclusively form core count.

Despite the near-zero mean, the standard deviation indicates the presence of many non-zero values where processors perform differently. This dataset can be useful to continue obtaining relevant performance information, dividing workloads executed into two groups, depending on their side of the x-axis. Thus, we could determine whether the workloads from each side have different features which could indicate the strengths and weaknesses of each processor microarchitecture.

For this group analysis we will employ a performance analysis methodology known as Top-Down [12]. It is a practical method to identify true bottlenecks in out-of-order processors, built on top of existing performance counters in Intel microarchitectures. From total pipeline slots (number of instructions that can be issued/retired per cycle), Top-Down estimates which fraction is utilized by "good instructions" and which fraction remains empty due to stalls from different parts of the processor pipeline. Processor stalls are classified following a hierarchical approach. At the top of this hierarchy four major categories are defined:

- *Frontend Bound*: fraction of slots wasted because the frontend undersupplies instructions to the backend, fetching and decoding issues mainly
- *Backend Bound*: fraction of slots wasted because no uops are delivered at the issue pipeline, due to a lack of required resources, memory hierarchy or functional unit issues.
- *Bad Speculation*: fraction of slots wasted due to incorrect speculations associated with branch prediction.
- *Retiring*: issued uops that get retired. Slots utilized by "good instructions" (a 100% Retire corresponds to the maximal IPC of the given microarchitecture).

In this experiment we will evaluate the behavior of the two application groups (x-axis sides) analyzing whether Top-Down results differ. We limit this evaluation to the Intel server, which is enough to provide a preliminary idea

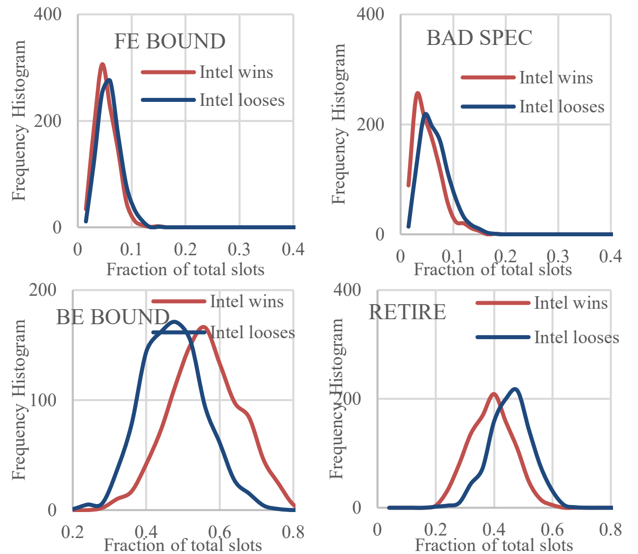


Fig. 8. Frequency histogram for each Top-Down First Level Category. AMD-winning vs. Intel-winning workloads.

about what makes each processor core better/worse from a software perspective. For the same number of workloads employed throughout this section we obtained the probability distribution function of each of the four categories, shown in Figure 8. With one graph per category, we pair the results of both groups, in order to check for any observable difference. Both *Frontend Bound* and *Bad Speculation* categories seem to obtain quite similar results, which means that no particular difference is noted for these parts of the pipeline between both groups. In contrast, there is a significant difference in the *Backend Bound* category, where we can observe that those applications with a more relevant bottleneck in the backend seem to behave better in intel processors than in AMD. Those applications pressuring core backend seem to be better suited to the Intel-based server. To understand

the source of inefficiencies of the AMD backend, this exploration should analyze the lower levels of top-down hierarchy. However, this is beyond the scope of this work, which is limited to demonstrating that this kind of systematic analysis is feasible through the *BenchCast* tool.

4.2 Simultaneous Multithreading

Simultaneous Multithreading (SMT) is a performance enhancement technique present in almost every modern general-purpose processor. It basically consists of the splitting of a physical core into multiple (usually two) virtual cores known as hardware threads or hardware contexts. This organization allows two instruction streams to run simultaneously through the same pipeline, improving aggregate ILP by improving processor resource utilization (especially if some of the threads have a clear bottleneck in some of the stages of the execution pipeline). Final performance improvement is usually far from the theorical upper limit of adding the IPC of each thread in the aggregate thanks to the second thread. The hardware resources available are shared with a "rival" and this has significant impact, which is dependent on the nature of each thread (even to the point of being detrimental under certain scenarios or resource sharing policies). Using *BenchCast* it is possible to estimate the actual benefit of enabling SMT in a general scenario, as well as understanding how each shared resource can impact on performance.

For this exploration, we limit our experiments to the Intel-based server configurations employed in the previous section. 1000 randomly generated applications were executed, first with SMT activated, then deactivating SMT through EFI settings. The results of both executions are shown in Figure 9, represented by a frequency histogram (bars) and

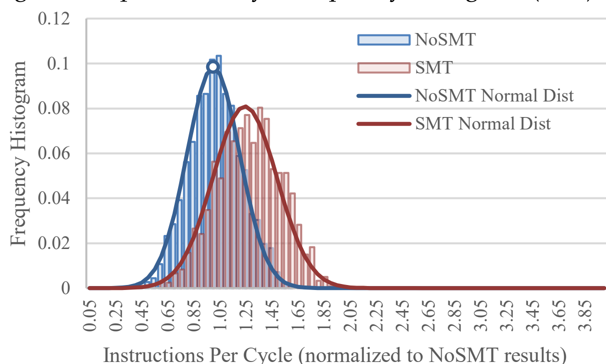


Fig. 9. Performance effect of SMT. IPCT histogram and estimated normal distribution.

its estimated probability distribution function (lines). Performance values were normalized to NoSMT results (mean=1 for NoSMT distribution). According to the graph, SMT improves raw processor performance by 25% on average. The SMT measured benefit is far below the theoretical upper limit, which means that vCPUs perform 40% worse than physical cores.

As mentioned previously, the access to every performance counter available allows us to look for the multiple sources of inefficiency and their contribution to the observed performance gap. Concerning SMT, we can distinguish between two kinds of shared resources: core-level and processor-level resources. Core-level resources correspond to those shared inside each physical core, such as L1 cache, branch prediction, issue queue, etc. Processor-level resources are those shared among all cores, such as Last Level Cache or memory bandwidth. Through the appropriate selection of hardware events, *BenchCast* enables the fast exploration of the performance effect of SMT on different Core-level and processor-level resources. For this work we limit our experiments to the most performance sensitive elements, branch prediction and cache hierarchy (Both L1 and LLC).

First, we evaluate the different behavior of three core-level resources in the presence and absence of SMT: branch predictor, L1 data and instruction caches. Results for these metrics are presented in Figure 10, as the cumulative frequency graph of misses/misspredictions per kilo instruction (MPKI). In this kind of graph, the x-axis represents the

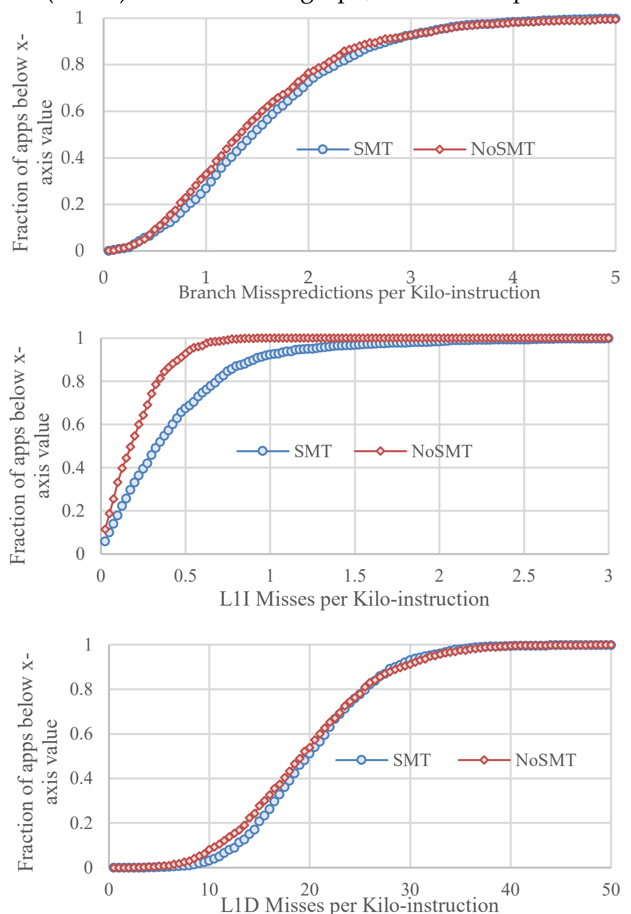


Fig. 10. SMT Effect (cumulative frequency distribution) on different hardware resources: Branch Prediction (above), L1I MPKI (mid) and L1D MPKI (below).

parameter under test while the y-axis indicates the fraction of workloads that are below that MPKI value. The presence of two instruction streams doubles, on average, the number of misses per kilo-instruction. Contrary to the intuitive idea of sharing the instruction cache between two threads, the final impact of this degradation on overall performance seems to be insignificant, because in both cases very low miss rates are observed. The negative impact of SMT is more subtle on both branch prediction and L1D performance. In the case of branch prediction, the low values on the x-axis indicate that degradation might have a minimal

impact on performance. In contrast, L1D MPKI results are one order of magnitude larger, which could indicate that the pressure on it has more impact on performance results. Concerning uncore resources, we close this section analyzing the impact of SMT on L3 performance. As can be seen in Figure 11, SMT produces a similar degradation to that observed in L1D. While LLC capacity remains constant, the number of active working-sets doubles with SMT, increasing the number of misses per kilo instruction. The exploration enabled by event counting in BenchCast allows us to

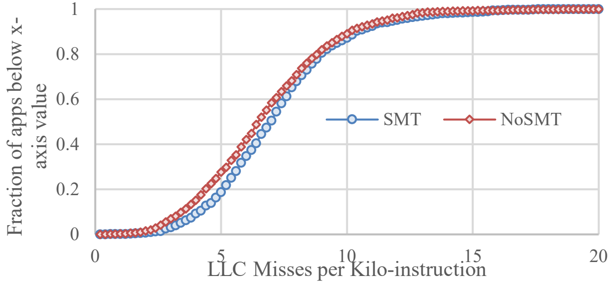


Fig. 11. SMT Effect (cumulative frequency distribution) on L3 MPKI.

conclude that the pressure imposed by application datasets on the memory hierarchy has a more noticeable effect on performance than doubling Instruction working sets or branch patterns.

4.3 Hardware Prefetching

The last *BenchCast* use-case example is focused on Hardware prefetching, which is a fundamental technique to tolerate cache miss latency in state-of-the-art processors [20][21]. Proactively fetching data from slower locations to a faster cache level in advance might significantly reduce average memory access time. Nearly every modern processor includes some hardware prefetching support, exploiting simple and regular access patterns. This is the case of many Intel microarchitectures (starting with Nehalem), where four different types of data prefetchers are implemented in hardware. Two of these prefetchers, known as DCU, are associated with L1 Data cache, where prefetching is triggered by load instructions when certain conditions are met [22]. The streaming prefetcher is triggered by an ascending access to recent loads, assuming that it is part of a streaming algorithm, automatically fetching the next line. A PC-based prefetcher keeps track of individual load instructions looking for a regular stride. When found, a prefetch is sent to the next address which is the sum of the current address and the stride. The two remaining prefetchers are associated with L2 cache. The L2 Spatial Prefetcher strives to complete every cache line fetched to the L2 cache with the pair line that completes it to a 128-byte aligned chunk. The L2 Streamer prefetcher monitors read requests (loads, stores, L1 prefetches and code fetches) from the L1 cache for ascending and descending sequences of addresses. When a stream of requests is detected, the anticipated cache lines are prefetched.

Again, the purpose of this section is to demonstrate the versatility of BenchCast, carrying out a detailed analysis of the performance effect of prefetching. The Model Specific Register (MSR) with address 0x1A4 will be used to control the activation/deactivation of these prefetchers. We define different combinations of enabled/disabled prefetchers, analyzing performance metrics for each of them. Figure 12 shows the IPC distribution of every prefetcher enabled (ALL), L1 prefetching disabled and L2 enabled (L2), L2 disabled and L1 enabled (L1) and every prefetcher disabled (NONE). In this graph, all performance values were normalized to NONE mean.

As expected, the absence of prefetching has a negative effect on performance, and the average IPC decreases from 1.6 to 1.29, which corresponds to a 20% performance degradation. Another observable result is the unbalanced contribution of L1 and L2 prefetchers to performance improvement. The activation of prefetching at each level has a positive effect in both cases but seems to be more relevant in the case of L2. The reason for this result might be the

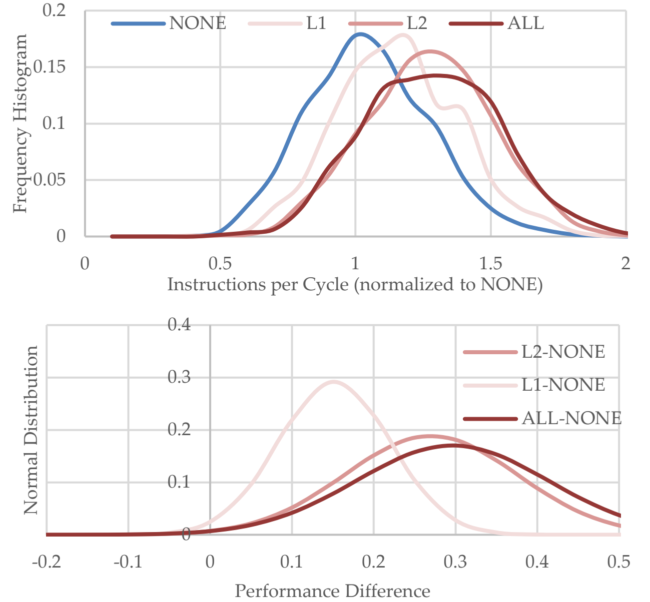


Fig. 12. (above) Performance effect of prefetching. IPCT distribution, normalized to NONE mean value. (below) Performance difference observed after prefetching activation, normal distribution.

large penalty of LLC misses and the larger LLC cache size minimizing the negative pollution effects caused by prefetching. A noteworthy result is that when both prefetchers are combined, there is no benefit when compared to L2 only prefetching.

In order to establish which fraction of workloads undergo a performance degradation caused by prefetching, we also define the following performance-difference variables:

$$ALL - NONE(w) = Perf_{ALL}(w) - Perf_{NONE}(w)$$
$$L2 - NONE(w) = Perf_{L2}(w) - Perf_{NONE}(w)$$
$$L1 - NONE(w) = Perf_{L1}(w) - Perf_{NONE}(w)$$

The normal distribution of these three variables is shown in Figure 12. In this graph, all values below zero represent those workloads with poorer performance after prefetching activation. As can be seen, the activation of both or L2 prefetchers improves performance in a consistent way. According to measured mean (0.296, 0.27) and stdev (0.117, 0.105), less than 1% applications will suffer from performance degradation. In the case of L1 prefetch, this fraction grows to 3% applications (0.151 mean, 0.068 stdev). It still represents a small fraction, but combined with the lower IPC improvement on average, explains its worse results

when compared to L2 prefetching.
Again, the access to event counting allows us to get a sense of the result. In this case, we analyzed how different hierarchy levels react to the changes in the prefetcher. As an initial step, we focused our attention on the L2 Cache, as it is the closest-to-processor level where prefetches are stored [22]. Figure 13 shows the cumulative frequency distribution of the MPKI for each prefetcher combination. These results are consistent with the performance ones, showing a MPKI improvement as more prefetchers are activated.

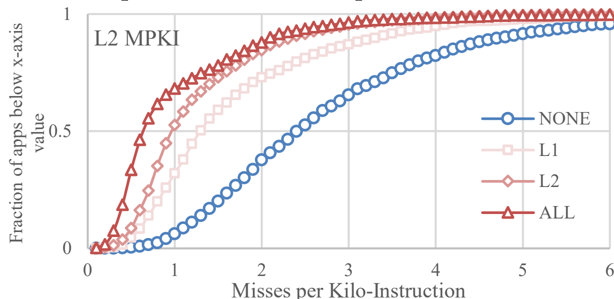


Fig. 13. L2 Cache Misses per Kilo Instructions. Cumulative frequency distribution for different Prefetching configurations.

This result shows that L2 MPKI improvement has a direct impact on performance, a similar tendency being observed for both metrics. Despite seeming very obvious, we highlight this conclusion because, as we will see next, this relationship is not as straightforward for every metric, and in some cases a closer look is necessary.
Next, focusing the attention on LLC metrics, we obtained the MPKI results shown in Figure 14 (above). As can be seen, in this case we obtained an evolution of MPKI inconsistent with performance results. The best MPKI results were obtained when no prefetcher was activated and they degraded progressively as they were activated. This apparently contradictory behavior has a simple explanation: if the raw LLC access numbers are analyzed, it can be immediately observed that the activation of hardware prefetching doubles the number of LLC accesses on average. If we move to alternative events and measure Miss rate in LLC (see Figure 14 (below)), we can observe how, in this case, the results are consistent with performance results and also with the expected prefetching behavior.

5 Related Work

The search for tools and methodologies targeting computer performance evaluation has been constant over time. However, both hardware and (mainly) software heterogeneity have increased the complexity of this task. Consequently, many benchmark suites are currently available, representative of multiple environments where computing systems can be found. Thus, hardware environments such as image processing (GPUs) or High-Performance Computing (HPC) employ specific benchmark suites. Some examples of GPU benchmarking tools are Rodinia [23], Parboil [24] and Lonestar [25]. In contrast, HPC employs suites such as NAS Parallel Benchmark [3] (parallel performance measurement developed by NASA), High-Performance Linpack [26] (CPU's FP performance, employed for setting up the Top-500 rank), High-Performance Conjugate Gradients [27](HPCG) as an alternative to HPL and HPCC suite [28].
From a software perspective, emerging environments such as Big Data, Cloud Computing or Deep Learning have generated the necessity of new benchmarking tools to allow a representative evaluation for these computing fields. Some of the most representative examples of benchmark suites targeting these environments are MLPerf [29], CloudSuite [30], BigDataBench [31], YCSB [4] or HiBench [32] .
General purpose hardware relies on heterogeneous benchmark suites such as PARSEC [7] or SPEC [6], in an attempt to be representative of a computing environment where

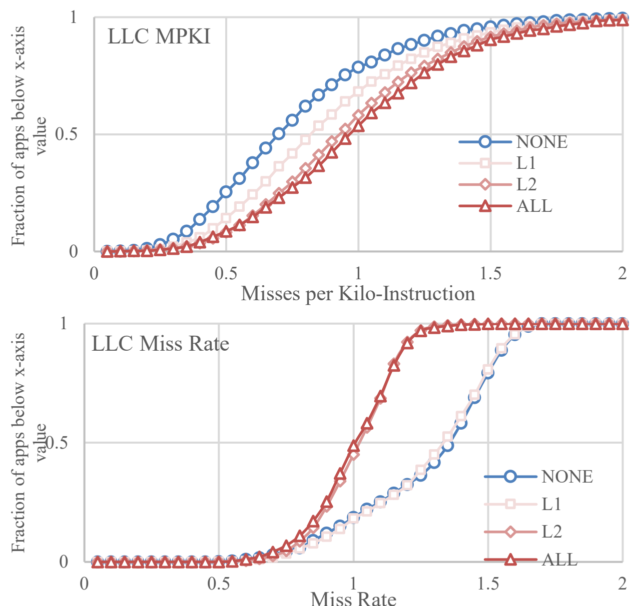


Fig. 14. LLC Cache Misses per Kilo Instruction (up) and Miss Rate (down). Cumulative frequency distribution for different Prefetching configurations.

applications cover a wide spectrum. Focused on the evaluation of commercial CPUs, the initial public release of BenchCast already includes workloads generated from applications from these two suites, as well as from the NAS Parallel Benchmark. However, BenchCast was designed to be a sort of meta-benchmarking framework, like Google's PerfKit Benchmarker [33]. The rules for including new applications are simple and new benchmarks can be easily added.
Some performance evaluation processes are not suitable for the aforementioned benchmarks. This is the case of detailed architectural simulation [34], where the execution time required to run a complete application makes it unaffordable in practice. In those cases, many studies have focused on alternative solutions to reduce the computational cost required for evaluation. One of the most common techniques is known as simulation sampling [15][16][13], where the evaluation process is limited to only a small relevant fraction of each application. SimPoints [35] is a well-known sampling methodology that automatically identifies long, repetitive execution phases in benchmarks, and limits simulations to a few instances of these phases. Similarly, [36] and [37] make use of statistical tools to evaluate the representativeness of a benchmark. This means limiting the execution of an application to a reduced number of instructions, able to maintain representativeness. With a

similar objective, Craeynest and Eeckhout [9] analyze the problem of the limited validity of current practices in multi-core simulation. Velazques et al. [8] carried out a benchmark selection, which was as small as possible, also analyzing alternative sampling methods and Singh and Awasthi [38] evaluated the accuracy of characterizing the SPEC CPU2017 benchmarks using SimPoints methodology. Loop-dominant programs are targeted in [39]. For each loop found, the authors define a signature, creating a signature vector for each application. Through the similarity scores process, they reduced the representative score and created micro-benchmarks for emulating the original one.

In a similar way, our work also looks for a subset of instructions able to resemble a whole application. In contrast, targeting real hardware execution, we have much more flexibility to choose sample size. This enabled the definition of a single sample per application, as well as its precise labeling for synchronization purposes. These two features enabled an easy, uniform and statistically sound evaluation process for multicore architectures running heterogeneous workloads. Runtime sampling makes the synchronization process more difficult, and the large number of samples defined by simulation tools makes it nearly impossible to build BenchCast on top of existing simulation sampling techniques. Alternative application modifications such as iteration or working set reduction were discarded. Reducing iterations is not always possible, as some applications presentconvergent algorithms without a predefined number of iterations. Similarly, reducing the working set to non-realistic inputs could reduce the execution time, but modifies the micro-architectural behavior.

6 Conclusions

In this work we presented a processor evaluation methodology suitable for both performance and microarchitectural analyses. Taking advantage of some basic execution features present in many applications, we identified, labeled and synchronized the execution of their ROIs. This process was standardized to include applications from different benchmarks, starting with the three (SPEC, PARSEC, NPB) already provided in the public release of the tool. The number of combinations (and therefore workloads) was large enough to provide statistically-sound results. Additionally, we demonstrated that a small fraction of the ROI is, in most cases, representative of the whole-program execution, which significantly reduces the computational effort required for evaluation. The experiments that previously required several days can now be finished within hours.

The accuracy of 20-second ROI execution was amply validated, demonstrating its suitability when statistical analysis is required. Hybrid workloads, where different applications run simultaneously on the same system, enabled the exploration of alternative performance metrics such as fairness. Finally, the utilization of hardware events for evaluation enabled the exploration of multiple microarchitectural parameters (as many as were available in the PMU of the system under evaluation).

We defined and presented three simple experiments that demonstrate the flexibility of BenchCast. We carried out a deep performance comparison of two commercial processors, providing more accurate results than existing methodologies and establishing the architectural implications on performance. We also extended the evaluation process to configurable hardware features, such as SMT or prefetching. We encourage readers to adapt the tools to the huge number of possibilities provided. All the code generated for this work is open access, with the intention of facilitating its utilization by the research community.

Acknowledgment

The authors wish to thank Jose Angel Herrero for his valuable assistance with the computing resources within the datacenter 3Mares. This work was supported by the Spanish Government (Agencia Estatal de Investigacion) under grant PID2019-110051GB-I00.

References

[1] F. Faggin, "The Making of the first microprocessor," *IEEE Solid-State Circuits Mag.*, vol. 1, no. 1, pp. 8–21, 2009, doi: 10.1109/MSSC.2008.930938.

[2] J. Dean, D. Patterson, and C. Young, "A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution," *IEEE Micro*, vol. 38, no. 2, pp. 21–29, 2018, doi: 10.1109/MM.2018.112130030.

[3] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," *Natl. Aeronaut. Sp. Adm. (NASA), Tech. Rep. NAS-99-011, Moffett Field, USA*, no. October, 1999, Accessed: Oct. 07, 2011. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.1321&rep=rep1&type=pdf.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, Jun. 2010, p. 143, doi: 10.1145/1807128.1807152.

[5] V. J. Reddi *et al.*, "MLPerf Inference Benchmark," 2020, doi: 10.1109/ISCA45697.2020.00045.

[6] "SPEC CPU 2017," 2017. https://www.spec.org/.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08*, 2008, p. 72, doi: 10.1145/1454115.1454128.

[8] R. A. Velasquez, P. Michaud, and A. Seznec, "Selecting benchmark combinations for the evaluation of multicore throughput," in *ISPASS 2013 - IEEE International Symposium on Performance Analysis of Systems and Software*, 2013, pp. 173–182, doi: 10.1109/ISPASS.2013.6557168.

[9] K. Van Craeynest and L. Eeckhout, "The multi-program performance model: Debunking current practice in multi-core simulation," in *Proceedings - 2011 IEEE International Symposium on Workload Characterization, IISWC - 2011*, 2011, pp. 26–37, doi: 10.1109/IISWC.2011.6114194.

[10] M. Van Biesbrouck, L. Eeckhout, and B. Calder, "Representative multiprogram workloads for multithreaded processor simulation," in *Proceedings of the 2007 IEEE International Symposium on Workload Characterization, IISWC*, 2007, pp. 193–203, doi: 10.1109/IISWC.2007.4362195.

[11] P. Prieto, P. Abad, J. A. Herrero, J. A. Gregorio, and V. Puente, "SPECcast: A Methodology for Fast Performance Evaluation with SPEC CPU 2017 Multiprogrammed Workloads," 2020, doi: 10.1145/3404397.3404424.

[12] A. Yasin, "A Top-Down method for performance analysis and counters architecture," 2014, doi: 10.1109/ISPASS.2014.6844459.

[13] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Tenth international conference on architectural support for programming languages and operating systems - ASPLOS '02*, 2002, p. 45, doi: 10.1145/605397.605403.

[14] T. M. Conte, M. A. Hirsch, and K. N. Menezes, "Reducing state loss for effective trace sampling of superscalar processors," 1996, doi: 10.1109/iccd.1996.563595.

[15] T. Lafage and A. Seznec, "Choosing Representative Slices of Program Execution for Microarchitecture Simulations: A Preliminary Application to the Data Stream," in *Workload Characterization of Emerging Computer Applications*, 2001.

[16] T. Sherwood, E. Perelman, and B. Calder, "Basic Block Distribution Analysis to find periodic behavior and simulation points in applications," 2001, doi: 10.1109/pact.2001.953283.

[17] A. C. de Melo, "The New Linux 'perf' Tools," *17 International Linux System Technology Conference*. Nuremberg, 2010, [Online]. Available: http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf.

[18] B. Gregg, "The flame graph," *Commun. ACM*, 2016, doi: 10.1145/2909476.

[19] F. J. Massey, "The Kolmogorov-Smirnov Test for Goodness of Fit," *J. Am. Stat. Assoc.*, vol. 46, no. 253, pp. 68–78, 1951, doi: 10.1080/01621459.1951.10500769.

[20] S. Mittal, "Survey of recent prefetching techniques for processor caches," *ACM Comput. Surv.*, 2016, doi: 10.1145/2907071.

[21] B. Falsafi and T. F. Wenisch, "A primer on hardware prefetching," *Synth. Lect. Comput. Archit.*, 2014, doi: 10.2200/S00581ED1V01Y201405CAC028.

[22] I. Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide," vol. 3, no. 253665, pp. 1–1386, 2013, [Online]. Available: papers3://publication/uuid/B767D5D8-AF4B-46BB-9893-D8046A5460AB.

[23] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54, doi: 10.1109/IISWC.2009.5306797.

[24] J. A. S. C. Rodrigues *et al.*, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," *IMPACT Tech. Rep.*, 2012.

[25] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, Nov. 2012, pp. 141–151, doi: 10.1109/IISWC.2012.6402918.

[26] J. J. Dongarra, P. Luszczek, and A. Petite, "The LINPACK benchmark: Past, present and future," *Concurr. Comput. Pract. Exp.*, vol. 15, no. 9, pp. 803–820, 2003, doi: 10.1002/cpe.728.

[27] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *Int. J. High Perform. Comput. Appl.*, 2016, doi: 10.1177/1094342015593158.

[28] P. R. Luszczek *et al.*, "The HPC Challenge (HPCC) benchmark suite," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC'06*, 2006, pp. 213-es, doi: 10.1145/1188455.1188677.

[29] P. Mattson *et al.*, "MLPerf Training Benchmark," *arXiv*, Oct. 2019, [Online]. Available: http://arxiv.org/abs/1910.01500.

[30] M. Ferdman and E. Al., "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *ASPLOS'12*, 2012, vol. 40, no. Asplos, pp. 37–48, doi: 10.1145/2189750.2150982.

[31] L. Wang *et al.*, "BigDataBench: A big data benchmark suite from internet services," in *Proceedings - International Symposium on High-Performance Computer Architecture*, 2014, pp. 488–499, doi: 10.1109/HPCA.2014.6835958.

[32] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Lecture Notes in Business Information Processing*, 2011, vol. 74 LNBIP, pp. 209–228, doi: 10.1007/978-3-642-19294-4_9.

[33] google, "PerfKit Benchmarker." https://github.com/GoogleCloudPlatform/PerfKitBenchmarker (accessed May 31, 2021).

[34] J. Lowe-Power *et al.*, "The gem5 Simulator: Version 20.0+∗ A new era for the open-source computer architecture simulator," *arXiv*, 2020, [Online]. Available: https://arxiv.org/abs/2007.03152.

[35] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for accurate and efficient simulation," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 318–319, 2003, doi: 10.1145/885651.781076.

[36] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Conference Proceedings - Annual International Symposium on Computer Architecture, ISCA*, 2003, pp. 84–95, doi: 10.1109/isca.2003.1206991.

[37] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John, "Measuring program similarity: Experiments with SPEC CPU benchmark suites," 2005, doi: 10.1109/ISPASS.2005.1430555.

[38] S. Singh and M. Awasthi, "Efficacy of Statistical Sampling on Contemporary Workloads: The Case of SPEC CPU2017," 2019, doi: 10.1109/IISWC47752.2019.9042114.

[39] E. M. Shaccour and M. M. Mansour, "A Loop-Based Methodology for Reducing Computational Redundancy in Workload Sets," *IEEE Access*, vol. 6, pp. 9570–9584, 2018, doi: 10.1109/ACCESS.2017.2788921.