A Reusable Characterization of the Memory System Behavior of SPEC2017 and SPEC2006

MUHAMMAD HASSAN, CHANG HYUN PARK, and DAVID BLACK-SCHAFER,
Uppsala University, Sweden

The SPEC CPU Benchmarks are used extensively for evaluating and comparing improvements to computer systems. This ubiquity makes characterization critical for researchers to understand the bottlenecks the benchmarks do and do not expose and where new designs should and should not be expected to show impact. However, in characterization there is a tradeoff between accuracy and reusability: The more precisely we characterize a benchmark's performance on a given system, the less usable it is across different micro-architectures and varying memory configurations. For SPEC, most existing characterizations include system-specific effects (e.g., via performance counters) and/or only look at aggregate behavior (e.g., averages over the full application execution). While such approaches simplify characterization, they make it difficult to separate the applications' intrinsic behavior from the system-specific effects and/or lose the diverse phase-based behaviors.

In this work we focus on characterizing the applications' intrinsic memory behaviour by isolating them from micro-architectural configuration specifics. We do this by providing a simplified generic system model that evaluates the applications' memory behavior across multiple cache sizes, with and without prefetching, and over time. The resulting characterization can be reused across a range of systems to understand application behavior and allow us to see how frequently different behaviors occur. We use this approach to compare the SPEC 2006 and 2017 suites, providing insight into their memory system behaviour beyond previous system-specific and/or aggregate results. We demonstrate the ability to use this characterization in different contexts by showing a portion of the SPEC 2017 benchmark suite that could benefit from giga-scale caches, despite aggregate results indicating otherwise.

CCS Concepts: • **Computer systems organization** → **Architectures**;

Additional Key Words and Phrases: Memory systems, cache sensitivity, prefetcher sensitivity, benchmark characterization, workload characterization, memory system characterization

ACM Reference format:

Muhammad Hassan, Chang Hyun Park, and David Black-Schaffer. 2021. A Reusable Characterization of the Memory System Behavior of SPEC2017 and SPEC2006. *ACM Trans. Archit. Code Optim.* 18, 2, Article 24 (March 2021), 20 pages.

<https://doi.org/10.1145/3446200>

New Paper, Not an Extension of a Conference Paper.

This work was supported by the European Research Council through ERC grant no. 715283 and by the Knut and Alice Wallenberg Foundation through the Wallenberg Academy Fellows grant no. 2015.0153.

Authors' address: M. Hassan, C. H. Park, and D. Black-Schaffer, Uppsala University, Department of Information Technology, Box 337, 751 05 Uppsala, Sweden; emails: hassan.muhammad@it.uu.se, {chang.hyun.park, david.black-schaffer}@it.uu.se. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.

1544-3566/2021/03-ART24

<https://doi.org/10.1145/3446200>

1 INTRODUCTION

Benchmark suites enable computer architects to evaluate and compare designs. This makes it particularly important that the benchmark suites themselves are characterized so that researchers understand which behaviors and bottlenecks the benchmarks expose. Characterization is particularly important in the transition to newer versions, such as from SPEC CPU 2006 [27] to SPEC CPU 2017 [28]. Indeed, this transition has spawned a flurry of papers characterizing the changes [13–15, 19, 25, 26, 30].

While there are many challenges in characterizing a benchmark suite, one of the largest is providing a balance between *accurate results* (e.g., correct for a particular system) and *reusable results* (e.g., applicable across a wide range of systems and configurations). One choice on this spectrum can be seen in recent works [13, 15, 19, 26] that provide accuracy by using hardware performance counters to report aggregate statistics (e.g., application misses-per-thousand-instructions (MPKI) averages). While these studies are highly accurate for their particular system, the results present a combination of the system-specific micro-architectural behaviors and the intrinsic application behaviors, making them difficult to reuse in other contexts.

For example, characterizing memory-system-related performance requires evaluating the micro-architectural configuration, including cache sizes, replacement policies, prefetchers, miss status handling registers (MSHRs), Re-order buffer (ROB) sizes, and so on. The resulting complex interaction between the application and these micro-architectural features means that performance results give the combined effects of both the hardware and software, making it hard to extract the application’s intrinsic behaviors.

In this work we use simplified simulation to minimize complex hardware interactions, and thereby making the applications’ *intrinsic* behaviors more visible. Specifically, our *generic system model* provides insight into the applications’ sensitivities to the two most important memory system parameters: cache size and prefetching. We achieve this by using an instrumentation-driven simulator to evaluate MPKI for a generic cache model (single-level, LRU, 8-way associative, 64B line) across a range of sizes from 32 KB to 1 GB, both with and without a generic prefetcher (next-4 strides for loads and stores). Further, we collect these statistics for each 100M instruction window in the applications’ executions, enabling us to examine behavior over time. Our generic cache and prefetcher provide a simplified view of the applications’ interactions with the memory system without the complexity of detailed micro-architectural components. Our use of MPKIs represents a tradeoff to provide first-order insights into the memory system behavior of the applications without tying the results to the micro-architectural specifics needed to determine performance.

This use of a simplified simulation model allows us to see the application’s intrinsic memory access behavior, disentangled from the complexity of a specific micro-architecture. As a result of this simplification, we can identify the applications’ behaviors that will be present across a broad range of systems, and see how the applications’ overall sensitivities to cache size and prefetching change during their execution.

For example, we can see that adding a prefetcher is significantly more effective than doubling the cache size for *lbm* 2017 on a system with a 16-MB cache (MPKI reduction from 48 to 9.8 with prefetching, vs. 30 with the larger cache).

Similarly, our analysis over time shows that while *gcc* 2017 has an average MPKI of 2.4 with a 1GB cache and prefetching, 57% of its total misses occur during only 6% of its execution, with that portion of the application’s execution experiencing an MPKI of greater than 20.¹ These examples show how our generic system model provides insight into what type of optimizations should be

¹As our generic system model does not model performance, percentage of execution refers to the percentage of executed instructions, not cycles or seconds.

Fig. 1. Tradeoff between Accuracy and Re-usability of Results.

applied and how time-varying (non-aggregate) analysis tells us where we should optimize, despite the lack of micro-architectural specific simulation.

To explore our characterization, we first present *aggregate* application characterizations in Section 4. This provides a view of the application’s behavior across its full execution, which allows us to see how sensitive the applications are overall to changes in memory system architecture. In particular, we compare the behavior of SPEC CPU 2017 to 2006 in terms of both cache and prefetcher sensitivity. This analysis shows the larger working set size of the newer benchmarks that others have reported but also identifies places where newer benchmarks are *more* cache and prefetcher friendly.

We then look at the applications’ time-varying behavior in Section 5. We first examine cache sensitivity as a function of both cache size and execution time. While this analysis clearly shows the individual phases of the applications, it is a challenge to interpret due to the complexity of the combined phase and cache size behavior. To address this, we propose *aggregating the data by the amount of execution experiencing a given MPKI* to understand what percentage of each applications’ execution that experiences a certain MPKI, as a function of cache size. This directly reveals short but very memory-intense phases across the applications. Finally, we provide an example of using this data to identify regions where the benchmarks benefit significantly from giga-scale caches, despite their aggregate memory behavior indicating otherwise. This article makes the following contributions:

- The first detailed characterization of the intrinsic application memory behavior of SPEC 2017 across a broad range of cache sizes with and without prefetching, with both aggregate and time-based data.
- The first time-based analysis reporting the portion of the applications that experience significant memory system pressure as a function of cache size.
- A comparison of the SPEC 2017 and SPEC 2006 benchmarks in terms of their per-application and overall benchmark suite cache and prefetcher sensitivity.
- Detailed aggregate and time-based data for all SPEC 2017 and SPEC 2006 applications provided for direct comparison, with all raw and analyzed data available in an online appendix [7, 8].
- An example of using this analysis to show that despite low aggregate MPKIs, there are still significant portions of the applications that benefit significantly from giga-scale caches.

2 CAVEATS OF WORKLOAD CHARACTERIZATION

As performance is a combination of the application, compiler, and hardware, characterizing workload memory behavior involves tradeoffs in accuracy and generality. An overview of this tradeoff is provided in Figure 1. At one extreme are algorithmic analyses that provide high-level insights into the reuse in an algorithm, but only crude insights into behavior on a particular system. At the other extreme, cycle-accurate simulations or hardware performance counters provide full accuracy, but incur very system-specific effects. In this article we strive to choose a middle-ground

with our generic system model to provide sufficient generality to reveal the applications' inherent behaviors, while being realistic enough to provide meaningful insights for future work. To put our tradeoff in context, we first review the range of characterization techniques available.

Hardware Performance Counters: At the most accurate, but least flexible, end of the spectrum, hardware performance counters provide fast and accurate characterization for a particular system. But the specific micro-architectural details, such as the number of MSHRs in the L1, the size of the load queue, the L1 cache latency, and so on, all have significant impact on the performance. Further, there is limited flexibility in evaluating memory configurations other than the one provided by the system. As a consequence, hardware performance counter characterizations have a limited ability to provide insight into the intrinsic behavior of the application and how it might behave on other systems.

Cycle Accurate Simulators: Cycle accurate simulators provide more flexibility than performance counters, as they can be re-configured to provide a range of data points. However, while they are highly accurate for their own performance model, the systems they model do not faithfully represent actual hardware. Unfortunately this flexibility comes at the cost of simulation times that are thousands-to-tens-of-thousands of times slower [21], making them unsuitable for full application analysis and broad parameter exploration.

Statistical Cache Analysis: Statistical cache models typically sample memory accesses and use mathematical models to predict cache behavior [2, 3, 5, 24, 31]. These can provide faster and potentially more flexible analysis of an application's memory behavior, but at the expense of a limited ability to model memory system details. For example, reuse distance analysis can rapidly model different cache capacities, but has a limited ability to handle set-associative caches and replacement policies, and, importantly does not handle prefetching.

Even more abstractly, Weinberg et. al. [29] developed an even more architecturally-independent method to score an application's temporal and spatial locality. While they showed a correlation between their scores and application performance, they were unable to use them to predict performance based on memory behavior, as they could not connect the degree of temporal or spatial locality to specific cache sizes or prefetching.

Algorithmic Analysis: The most extreme architecturally-independent analysis is a first-principle investigation of the algorithm's data reuse characteristics. While this can help identify potential reuse opportunities, it is extremely difficult to apply to system optimization without first making concrete choices about the systems micro-architecture (e.g., cache size, prefetcher, etc.).

This work: In this work our goal is to provide *enough memory system detail* to make our results relevant to modern systems, but *without obscuring the intrinsic behavior of the applications* with micro-architectural specifics. This goal motivates our decision to use a generic system model to capture sensitivities to the key memory system parameters of cache size and prefetching, and to report MPKI, as it can be obtained from the memory system without the need for a detailed micro-architectural model. While our particular cache hierarchy, sizes, associativity, replacement policy, and prefetcher are specific micro-architectural choices, we believe they represent a sufficiently generic baseline to observe the inherent memory behaviour of the benchmarks. These choices provide results that are more widely applicable than micro-architecturally specific performance results while being more accurate than micro-architecturally independent techniques.

Further by using a binary-instrumentation driven approach, we are able to obtain our results with only a 10× to 50× slowdown over native execution, which allows us to model a wide range of cache configurations across the full execution of (nearly) all applications. With this tradeoff, we hope to avoid most of the detailed micro-architectural effects while retaining most of the valuable application memory system behaviors.

Fig. 2. Box/Whiskers: Cache MPKIs across a range of micro-architectural configurations and cache sizes (x -axis) obtained from cycle-accurate simulator ChampSim [1], demonstrating the wide range of results when combining architecture and application behavior. Boxes show the 25-50-75 percentile MPKI values, and the whiskers show the min/max MPKI values. These results include multiple L2, L3 cache sizes, big (B) and little (L) cores, and various prefetchers. Lines: Generic system model MPKIs without prefetching (blue) and with prefetching (dashed orange) showing overall application sensitivity. The generic system model uses a single-level LRU cache (of the cache size denoted in the x -axis) and a stride prefetcher.

2.1 The Perils of Micro-architectural Details

While performance results that take into account as many micro-architectural details as possible are generally considered more accurate, it is important to understand just how large an impact these details have on application memory system behavior. To demonstrate the magnitude of these effects, we plot the MPKI across 32 cache/prefetcher/out-of-order configurations obtained from the ChampSim [1] cycle-accurate simulator in Figure 2. The box plots show the very large range of “cycle-accurate” results that can be obtained for the same applications and cache sizes, by simply varying the micro-architectural details. While each of these values is “accurate” in the “cycle-accurate” sense, it is not at all clear which configuration provides meaningful insight into the applications’ intrinsic memory system behaviors.

In our approach we gave up much of the micro-architectural details to try and capture the overall application behavior. The blue lines in Figure 2 show that the MPKI reported from our simplified, generic cache simulation without prefetching meaningfully reflects the overall application behavior across a range of cache sizes. The dashed orange line includes our prefetching model, and demonstrates the challenge of trading off micro-architectural details: While *mcf*’s performance range is quite well represented, neither *omnetpp* nor *lbm* are. In particular, *omnetpp* (Figure 2(b)) benefits from the next-line prefetcher present in some of the cycle-accurate configurations. If this prefetcher is removed from the cycle-accurate configurations (Figure 2(c)), then we see that our simplified results fit much more closely. Similarly, *lbm* (Figure 2(d)) benefits from the store prefetching present in our prefetcher (dashed orange line), as can be seen when they are disabled in our model (dotted green line).

This analysis makes two points: First, while detailed micro-architectural simulations are precise, their results span a tremendous range depending on the processor and prefetcher configurations. And, second, while our generic system model does not cover all possible configurations (prefetchers, hierarchies, and replacement policies), it does provide a good overall characterization of the *application’s intrinsic memory behavior*, covering a wide range of micro-architectural choices.

Table 1. Length of the SPEC CPU 2017 Floating-point Benchmarks (in Trillions of Instructions) and the Phase Error (Euclidean Distance between Aggregate Phase Vectors) for the First Five Trillion Instructions with Respect to the Full Execution

Benchmark	Total Instructions	Cumulative Phase Error
cactuBSSN	9.6 T	1%
fotonik3d	6.2 T	0.9%
pop2	23 T	0.2%
roms	22 T	10%
wrf	24 T	3.9%
cam4	17 T	0.2%
bwaves (1)	32 T	5%
bwaves (2)	33 T	5.9%
nab	13 T	13%
imagick	66 T	88%

3 METHODOLOGY

We use Intel’s Pin [12] to collect address traces and feed them into our in-house developed cache/prefetcher simulator (which was validated against Dinero[4]) to generate MPKIs for each 100M instruction window, for a range of cache sizes and prefetcher configurations. Benchmarks are compiled with the SPEC recommended flags and optimizations (gcc/g++ v.5.4.0, x86_64, -O3 optimization). For comparing between SPEC 2017 (speed) and 2006, we chose the input pairs that provide the longest execution for applications with multiple input pairs. The numbering of different input pairs in our figures and online raw data follows the order generated by the specinvoke tool. We ran each benchmark with all the possible inputs reported by specinvoke tool as different input lead to different memory behaviour.

Our generic system model implements a cache simulator that evaluates 16 cache sizes (single-level, 32 KB to 1 GB, eight-way set-associative, 64B lines, LRU replacement). Our prefetcher tracks strided misses (both loads and stores) based on the miss Program Counter, fetches the next 4 cache lines in the pattern, and continues on hits to prefetched lines. Prefetched addresses are modeled as installed immediately. As the simulator only looks at data addresses, it models neither performance (Instructions per Cycle) nor instruction cache behavior. As a result, we obtain MPKIs for the net capacity of a multi-level hierarchy from our simpler single-level simulation.

We parallelized the simulation by dividing each benchmark into regions of 500B instructions with 3B instructions of cache warming. For the extremely long floating-point benchmarks, we only simulated the first 5 trillion instructions and compared the phases found in that region to those of the whole application via an online phase detection tool [23].

For all but *roms*, *imagick*, and *nab* the first 5T instructions are highly representative (Table 1). Due to the significant phase error in first 5T instructions, we exclude *imagick* from our evaluation.

4 AGGREGATE CHARACTERIZATION

We begin by analyzing the *aggregate* memory system behavior using our generic system model: e.g., average MPKI for the applications’ full execution across different cache sizes and with/without a prefetcher. This analysis provides insight into the applications’ working set sizes, cache-sensitivity, access behaviour and the effects of prefetching. We study the working set size and prefetching sensitivity of benchmarks by analyzing the variation of their MPKI with cache size and prefetching. As the largest cache size we simulated is 1 GB, we cannot determine working sets larger than 1 GB.

Figure 3 shows the aggregate (averaged over time) cache- and prefetcher-sensitivities, with the benchmarks that are common to both SPEC 2006 and 2017 together at the top. The expected conclusion from this figure is that the working set of many applications has increased from SPEC 2006 to 2017. And, indeed, this effect can be clearly observed as shifts to the right (larger cache sizes) in the MPKI curves for several applications, such as gcc, cactusADM and lbm. However, omnetpp, mcf, and bwaves show *increased cache and prefetcher friendliness* in the 2017 versions compared to the 2006 versions, although their working sets are larger. These changes likely come from new algorithms that are more cache-friendly. In comparison, lbm has an identical cache curve that is pushed out to the right, indicating similar behavior but with a larger dataset. Bwaves 2017 is more cache friendly at small cache sizes than its 2006 counterpart, but still has an MPKI of 1 or higher even with a 1 GB cache. The largest differences between the 2006 and 2017 benchmarks can be found in cactuBSSN and cactusADM, where the difference in MPKI is as high as 100 at 32 KB cache.

For prefetching, with the exception of deepsjeng, most benchmarks from (a) to (m) show similar prefetching sensitivity in both SPEC 2017 and SPEC 2006. For example neither video encoding benchmark (SPEC 2017 x264 and SPEC 2006 h264) show much benefit from prefetching. With regards to general prefetcher-sensitivity, we see that gcc, lbm, mcf, xalancbmk, and bwaves achieve significant benefit from prefetching, while cactuADM-cactuBSSN, omnetpp, leela, x264, and perlbench gain little. Note that we found very similar aggregate cache- and prefetcher-sensitivity across the various input pairs for gcc, x264, perlbench, xz, and bwaves. Both mcf and lbm benefit greatly from prefetching at small cache sizes where the difference can be high as 40 MPKI. mcf 2017 is more cache friendly than mcf 2006, showing a difference of almost 40 MPKI at 32 KB cache with prefetching. Though lbm 2017 benefits greatly from prefetching, it still has >1 MPKI at 1 GB, even with prefetching.

While the aggregate MPKI data in Figure 3 provides extensive details about the applications' intrinsic memory behaviors, it is difficult to use to categorize them at a high-level. To address this, we define two metrics from this data to categorize applications in terms of their sensitivity to cache size and prefetching. For cache sensitivity, we define the *cache working set size* as the cache size at which the application's MPKI drops below 1 with prefetching enabled. This cache size indicates how much capacity is required to capture the majority of the application's working set with the help of a prefetcher. Similarly for prefetcher sensitivity, we define the prefetcher point-of-no-benefit (PoNB) as the cache size above which enabling prefetching reduces the MPKI by less than 1 or there is no benefit of having prefetching enabled. A large PoNB shows that the benchmark benefits from the prefetcher even with larger cache sizes, whereas smaller PoNB point shows that the effect of the prefetcher diminishes as cache capacity is increased. For example, for deepsjeng, the difference in MPKI with and without prefetching is less than 1 at 32-KB cache size, thus deepsjeng is defined to have a low PoNB. While for lbm, the difference is 20 MPKI even at 1-GB cache size, so lbm has a large PoNB.

Figure 4 shows the overall working set size and PoNB for the benchmarks, and compares the changes between SPEC 2006 and SPEC 2017 by connecting those benchmarks with lines. The x -axis shows the working set size of the benchmarks while the y -axis shows the PoNB. The low PoNB (or prefetcher-insensitive) benchmarks are located in the bottom half of the plot.

The benchmarks that have moved toward the upper-right corner of the plot have experienced an increase in cache working set size and need the prefetcher even with larger caches. From this we can see dramatic changes in some of the applications: mcf, gcc, bwaves, and lbm have all seen their cache working set sizes increase to 1 GB (or greater, as we did not simulated more than 1 GB cache), while omnetpp, xz, and wrf see a decreased benefit from prefetching. Conversely, x264 and xalancbmk have reduced cache working set sizes compared to their 2006 counterparts. The changes in prefetcher behavior between 2006 and 2017 are listed in Table 2 with the benchmarks

Fig. 3. Aggregate (average over execution) cache/prefetcher sensitivity for SPEC 2017 and SPEC 2006. The y -axis plots MPKI for the given cache size on the x -axis. Similar applications from the two suites are plotted together.

Fig. 4. Aggregate application cache and prefetcher sensitivity and the change between SPEC 2006 and SPEC 2017. Cache working set size is shown on the x-axis while y-axis shows prefetcher point of no benefit (PoNB).

Table 2. Changes in Prefetcher Effectiveness between 2006 and 2007

2006→2017	Pref. Insensitive	Pref. Sensitive
PoNB Increased		bwaves, gcc, lbm
PoNB Decreased	leela, omnetpp,xz, x264, perlbench, deepsjeng	wrf, xalancbmk
(new in 2017)	exchange2, nab	cam4, fotonik3d, pop2

categorized by their prefetcher sensitivity. We define a benchmark as prefetcher-sensitive if the MPKI difference with and without prefetching is more than 3 at 32 kB. From 2006 to 2017, bwaves, cactusBSSN, gcc, lbm, omnetpp, perlbench, wrf, and xz increased their cache working set sizes, while leela, deepsjeng, x265, mcf, and xalancbmk remained the same or shrank.

5 TIME-BASED CHARACTERIZATION

While the aggregate results presented in Section 4 provide a high-level view of the applications' behaviors, they fail to show how the behaviours change during execution. This is particularly problematic as many benchmarks have phases with very high MPKIs across a wide range of cache sizes, while the whole-execution average remains low. While obscured by aggregate metrics, these short, but very intense, phases can account for a large proportion of the total cache misses, and therefore a large portion of the DRAM energy and memory system performance loss.

Figure 5 demonstrates the importance of time-based characterization. Figure 5(b) shows that the aggregate MPKI for gcc 2017 with a 1 GB cache without prefetching is 4.5 ②. However, over 9% of the application's execution has an MPKI of greater than 20 (Figure 5(a), ③). This represents 73B instructions that are executed during highly memory-bound phases, even with a 1 GB cache. Due to the high MPKI at this time, *this 9% of the execution accounts for 79% of the total misses*, as will be discussed in Section 5.1, (Figure 8(b) ⑤) and (Figure 9(b) ⑦).

Yet without time-based information, the aggregate MPKI of 4.5 would suggest that a 1-GB cache is reasonably effective, while with the time-based information it is clear that there are

Fig. 5. Time-based analysis of gcc 2017 vs. an aggregate analysis. While both show the impact of varying the cache size, the aggregate analysis hides the fact that there are phases of very intense activity (MPKIs over 60) across nearly all cache sizes.

Fig. 6. MPKI heatmap as a function of cache size for mcf 2006 (above) and mcf 2017 (below). The y-axis shows cache size and the x-axis shows number of instructions in billions. The plots are similarly scaled, showing how much longer and more complex the 2017 execution is compared to the 2006 execution.

a non-negligible set of phases for which even a 1-GB cache is vastly inadequate. Our generic system model's time-based and cache-size information allows us to understand the behavior even further: At 650B and 800B instructions into the execution there are two phases that experience very high MPKIs up to cache sizes of 512 and 256 MB, respectively, while the majority of the program is satisfied with caches of 32 MB (50B to 650B) or even 64 KB (810B to 1250B). This insight is important for understanding where we can expect micro-architectural improvements to have an impact, which is essential for knowing how to evaluate their effectiveness.

Time-based analysis is also valuable for comparing the 2006 and 2017 variants of benchmarks. Figure 6 shows a time-based cache heatmap (with prefetching) for both mcf 2006 (top) and mcf 2017 (bottom). The figures have been drawn to scale, and dramatically show the change in execution length between mcf 2006 (300B instructions) and mcf 2017 (1650B instructions), as well as the significantly changed cache behavior of the phases. This time-based perspective is lost in aggregate

Fig. 7. MPKI heatmap as a function of cache size for xalancbmk 2006 (above) and xalancbmk 2017 (below). The y -axis shows cache size and x -axis shows number of instructions in billions.

statistics. Though the aggregate statistics in Figure 3 show clearly that mcf 2017 has a significantly lower MPKI (39) than mcf 2006 (77) (at 32 KB cache with prefetching), it hides the fact that there are still periodic phases with very high (over 100) MPKIs.

Another example is xalancbmk where both 2006 and 2017 show very similar phases of execution in Figure 7. The first phase where the heatmap rises shows a common curve, with slightly higher overall MPKIs in the 2017 version for caches up to 16 MB. In the third phase that plateaus until the end, the newer benchmark has a dramatically higher miss intensity, that continues up to 256 KB caches, compared to a lower overall miss intensity for the 2006 benchmark, but continued MPKIs of about 5 up to caches of 16 MB. The full range of time-based benchmark analysis for both benchmark suites is provided in the online appendix.

5.1 MPKI Bins

Although the heatmap representation of gcc's time-based behavior in Figure 5(a) provides a vivid illustration of its phases and MPKI variation, it is difficult to use to quantify the importance of the phase behaviors. For example, while the heatmap shows that even with a 512MB cache gcc's MPKI is greater than 70 in multiple phases, it does not quantify *what percentage of the application's execution occurs in those phases*. To simplify the analysis of the time-based statistics, we propose aggregating the data into *MPKI bins*. Specifically, each MPKI bin tells what percentage of the program execution² experience a given MPKI for each cache size.

Figure 8 presents the MPKI bins for gcc from the time-based data in Figure 5(a). With the MPKI bins we can immediately see that for a cache of 512 MB without prefetching (Figure 8(b)) 14% of the application's execution time has an MPKI of 20 or higher. This represents 178B instructions, which is roughly as long as the *full SPEC 2006 gcc benchmark*. To put this in perspective, the off-chip energy consumption in these high-MPKI phases from SPEC 2017 would be greater than the entire off-chip energy consumption of the SPEC 2006 application. In fact, the number of misses gcc 2017 incurs with 512 MB cache and no prefetching in regions with >20 MPKI is more than the

²As we are not modeling performance, we use percentage of executed instructions in place of execution time.

Fig. 8. For each cache size (y -axis), each window of execution sorted into its corresponding MPKI bin (x -axis), and the labels and color of each box represents the percentage of the windows that reside in that MPKI bin. This figure shows the results for gcc 2017 with (a) and without (b) prefetching, and the impact (difference) from prefetching (c).

total number of misses in gcc 2006. Without the time-based information we would be unable to see that these accesses are highly concentrated in only a few phases, suggesting that future memory system designs should look particularly closely at those behaviors.

We can further use the MPKI bins to understand the behavior of prefetching for a given application. Figure 8(a) shows the distribution of execution in different MPKI bins with prefetching enabled. Here we can see that for a 2 MB cache the portion of execution that gcc experiences an MPKI of 20 or higher decreases from 60% without prefetching (Figure 8(b)) to 15.2% with prefetching (Figure 8(a)). Conversely, the portion of execution with an MPKI of 10 to 20 increases from 4.1% to 46%, showing us the portion of the benchmark's execution that has been improved by prefetching.

To see the effects of enabling prefetching more explicitly, we plot the change in percentage of execution in each MPKI bin (delta between Figure 8(b) and (a)) when prefetching is enabled in Figure 8(c). Here we directly see how enabling prefetching, as expected, reduces the percentage of execution the application spends in high MPKI regions (red), leading to an increase in execution in lower MPKI regions (blue). This shows the application's sensitivity to prefetching across both cache sizes (vertically) and phase intensity (horizontal) for the first time.

This data helps explain the behavior seen in the aggregate data. For example, for gcc 2017 Figure 5(b) ① shows that the prefetcher caused a drop of 10 MPKI for a cache size of 2 MB. Figure 8(c) goes further and shows that this aggregate MPKI reduction came from 45.3% of the program execution moving from >20 to less than 20 MPKI region, plus an additional 5.3% reduction in the 1–3 and 3–10 regions. This tells us that the prefetching delivered benefits for 51% of the benchmark's execution. Such insight is not available from aggregate data and is difficult to quantify from the time-based heatmaps.

The MPKI bins in Figure 8 show how the application's *execution* (instructions) is spread across different MPKI intensities.

However, it does not show the percentage of misses that each MPKI bin accounts for. To present this data we report the *percentage of total misses* for each MPKI bin for gcc 2017 in Figure 9. Comparing Figures 8 and 9, we can identify which parts of the execution are responsible for the majority of the misses, not just how the misses are spread across regions of different MPKI intensity.

Fig. 9. Percentage of cache misses that occur in each MPKI bin (x-axis) for each cache size (y-axis for gcc 2017, with (a) and without (b) prefetching).

For example, gcc 2017 with a 1 GB cache and without (with) prefetching spends 9% (6%) of its execution with an MPKI > 20, (Figure 8 ④ and ⑤) but almost 79% (57%) of its total misses happen in those high MPKI regions (Figure 9 ⑥ and ⑦).

Another example is mcf 2017, which has an MPKI of 1.2 at 1 GB cache with prefetching and only 2% of the execution has an MPKI > 20. But that 2% of execution causes 53% of the total misses. This emphasizes that time-based characterization is important to reveal small transient phases that can account for a large share of the total misses, which makes it easier to develop techniques to address them.

As the MPKI bin plots provide a compact way of understanding the distributions of cache and prefetcher sensitivities across applications, we now use them to analyze the broader benchmark suites. Figure 10(a) shows the MPKI bins for five of the SPEC 2017 benchmarks and their corresponding SPEC 2006 versions. Figure 10(b) shows the corresponding difference in MPKI with and without prefetching (change in percentage of execution in each MPKI bin).

5.2 Analyzing SPEC2006 vs. 2017 with MPKI Bins

The first row in Figure 10(a) shows the MPKI bins (percentage of execution experiencing different MPKIs) for several SPEC 2006 applications. Both gcc 2006 and lbm 2006 show very simple behavior patterns with essentially all of their execution experiencing MPKIs of less than 1 for caches of sizes 128 MB and 8 MB or larger, respectively. gcc has some execution with MPKIs greater than 3 for smaller caches, but lbm does not. mcf and omnetpp show much greater sensitivity to cache size as seen by the horizontal shift from the lower right (high percentage at high MPKIs at small cache sizes) to the upper left (high percentage of time at low MPKIs at large caches). For omnetpp, beyond 128 MB cache the application essentially never sees MPKIs greater than 1, while mcf has up to 20% of its time with MPKIs as high as 10 even for caches as large as 512 MB.

5.2.1 Cache Behavior Changes. The second row of Figure 10(a) shows the same benchmarks from SPEC 2017:

- bwaves 2017 is more cache- and prefetcher-friendly as more than 80% of its execution has an <1 MPKI (at 32 KB cache with prefetching) while bwaves 2006 has less than 20% of execution with <1 MPKI.

Fig. 10. Percentage of execution (instructions) experiencing different MPKIs by cache size for a selection of SPEC 2006 and 2017 benchmarks with prefetching (top) and the change without prefetching (bottom).

- gcc 2017 has a much greater portion of execution in higher MPKI bins (10–20) at up to 8 MB compared to 2006. We also see that 14% of its execution experiences >3 MPKI for 1 GB caches (Figure 8(a) and (b)), vs. 0.1% for gcc 2006.
- Even with a 1 GB cache, the majority of lbm 2017 experiences an MPKI of 3–10, vs. <1 for 2006.
- mcf 2017 has a smaller cache working set size with majority of its execution at <1 MPKI starting at 32 MB for 2017 vs. 1 GB for 2006.
- omnetpp 2017 shows a reduction in high MPKI bins, most noticeably at 32 KB (97% >20 MPKI for 2006 vs. < 20% for 2017) and 8 MB, but generally follows the same trend. At 32 MB, omnetpp 2017 has 96% of its execution in 1–3 MPKI, vs. 40% for 2006.

5.2.2 Prefetcher Impacts. The applications' prefetcher sensitivities are shown in Figure 10(b). These graphs show the change in percent of execution in each MPKI range when enabling the prefetcher. We expect to see a shift to the left (more execution in lower MPKI bins), which is visible as blue squares (increased execution percentage) to the left of red squares (decreased execution percentage).

- In gcc and lbm the prefetcher is less effective in 2017 vs. 2006. Prefetching in gcc 2006 significantly reduced the >20 MPKI bin down to the <1 MPKI bin; however, gcc 2017 sees a reduction from the >20 bin to the 10–20 bin (e.g., the blue and red are closer together for 2017, showing that the impact of the prefetcher was less for small caches sizes in 2017 compared to 2006. This likely indicates a more complex data access pattern that is less amenable to stride prefetching in the 2017 benchmark).
- bwaves 2017 shows less effect overall from prefetching, which is not surprising given that its overall cache miss intensity is far lower. (Most of the bwaves 2017 execution is in the 0–1 MPKI bin.)
- mcf 2017 shows a large shift in MPKI from the 3–10, 10–20, and >20 bins to the 0–1 bin, while mcf 2006 shows a much more concentrated transfer from just the higher MPKI bins. This is consistent with the overall lower aggregate MPKI of the 2017 version.
- omnetpp has a distinct behavior where the prefetcher is effective at reducing misses at certain sizes of caches: for 256 KB, 8 MB, and 32 MB cache sizes in 2006, and 32 KB, 2 MB, and 32 MB in 2017.

5.2.3 Gigascale Caches. The SPEC 2017 suite introduced benchmarks that experience significant misses even with 1 GB of cache. At the same time, the advent of embedded and stacked DRAM has made such large-scale caches practical. In this section, we explore how some of the benchmarks behave on such gigascale caches. We present the effects of larger caches on MPKI for applications with at least 10% of their execution experiencing MPKI greater than 0.3 with a 1 GB cache in Figure 11.

We found that the MPKI distribution across execution of cactuBSSN, fotonik3d, mcf, and pop is consistent with what one would expect from the aggregate MPKI results: the applications have low average MPKIs at 1 GB cache size, and the time-based results show little time spent in high MPKI phases. However, for the other three benchmarks, the aggregate MPKI results could be misleading. lbm has an aggregate MPKI of 4.1, yet the time-based results show that 70.5% of the execution of lbm experiences MPKI between 5 and 10. Similarly, gcc and bwaves have aggregate MPKIs of 2.42 and 1.98, yet more than 13.4% and 10.7% of their executions experience MPKIs greater than 5 and 10, respectively. These results underscore the importance of looking at the time-based behaviors of the applications to understand the impact of the memory system, as the aggregate results hide the fact that there are phases with very significant memory system challenges.

Fig. 11. MPKI distribution across execution for large caches showing significant use (% of execution with high MPKIs) for large-scale caches despite low overall aggregate MPKIs.

The MPKI bin plots presented here provide the first quantitative view of how the SPEC benchmarks are sensitive to cache size and prefetching in a manner that allows researchers to understand how much of an impact (percent of execution) they can expect from designs that target these sensitivities.

6 RELATED WORK

There has been extensive work to characterize SPEC 2006 and 2017 suites using a wide variety of data for a range of purposes, including identifying common behavior across benchmarks to reduce simulation time, analyzing benchmarks execution to understand how they utilize different parts of the processor, and analyzing the benchmarks' interactions with the memory system, as we have done in this work.

6.1 Micro-architecture-dependent Characterizations

The majority of benchmark characterization studies have used hardware performance counters to measure micro-architecturally-specific behavior due to the ease of collecting and the accuracy for the specific system evaluated: Phansalkar et al. [16, 17] applied PCA and clustering to find redundancy in SPEC 2006 to reduce simulation time, while Limaye and Adegbiya [13] and Panda et al. [15] did so for SPEC 2017. Prakash and Peng [18] reported IPC, execution time, L1/L2 cache miss rates, and branch miss rates for 2006 on an Intel Core 2 Duo, while Kejariwal et al. [11] characterized both 2006 and 2000 for memory and branch misses and found higher cache miss rates for 2006. Hebbar and Milenkovic [20] reported performance and energy efficiency analysis of 2017 on an Intel Core i7-8700K. Limaye and Adegbiya [13] compared 2017 to 2006, and used PCA to find redundancies in the benchmark suite for reduced simulation time. Panda et al. [15] found that three benchmarks from each category of SPEC 2017 can predict around 93% of program performance. They studied differences and similarities among different input sets, rate and speed versions of benchmarks, and compared application domain coverage, power, and performance of 2017 vs. 2006.

Wu et al. [30] studied phase-based behaviour of SPEC 2017 using periodically sampled hardware performance counters, and used this data to determine that up to 12 simpoints are needed for each benchmark. Navarro-Torres et al. [14] characterized SPEC 2017 on an Intel Xeon Skylake processor and studied sensitivity of the benchmarks to the LLC capacity by varying its allocation as well as enabling/disabling hardware prefetching. They reported L1, L2, and L3 cache MPKI for 2006 and 2017. They noted that the 2017 benchmarks had fewer L2 and L3 misses than 2006. This work is the closest to ours but was limited in exploring cache sizes supported by the hardware: 32 KB, 1 MB, and 1.75 MB to 19.25 MB LLC, with smaller LLC partitions suffering from increased conflict misses due to the hardware way-partitioning reducing the associativity. They further studied phase-based behaviour of applications over time with L3 cache MPKI and correlated with simpoints to conclude that 3 simpoints are insufficient to cover all the memory phases.

Table 3. Comparison of Prior Work with Regards to Exploration (Cache, Prefetcher, Time-based vs. Aggregate) and Benchmarks (2006 and 2017)

Prior work	System-independent	Prefetch	Phases	SPEC 06	SPEC 17
[20]					✓
[18], [11]				✓	
[15], [13]				✓	✓
[22]			✓	✓	
[26]			✓	✓	✓
[14]		✓	✓	✓	✓
[25]	✓				✓
[10]	✓		✓	✓	
This work	✓	✓	✓	✓	✓

The use of performance counters in these works provides highly accurate data for the specific platforms evaluated, but the analyses conflate the intrinsic application sensitivities and platform characteristics and are unable to provide insight into how the applications will behave on other systems with different micro-architectures.

6.2 Micro-architecture Independent Analysis

The importance of characterizing memory system behavior for these benchmarks has motivated a range of work that has looked at the application sensitivities in a micro-architecturally-independent manner. Singh and Awasthi [25] characterized SPEC 2017 memory characteristics using binary instrumentation to report aggregate cache sensitivities for cache sizes up to 32 MB, and performance counters to measure memory bandwidth and memory capacity utilization. They varied the associativity with cache size, starting from 32 KB direct-mapped cache up to a 32 MB 1,024 way cache, making a direct comparison problematic. In this work we showed that most applications benefit significantly from far larger caches, and that even evaluating caches up to 1 GB does not fully characterize applications.

Jaleel [10] did an extensive memory characterization of SPEC 2006 and 2000 using a Pin-based cache simulator based on stack distances. That work showed time-based phase behaviour of the benchmarks for three different cache levels. However, the analysis was limited to 128 MB of cache size without the effects of prefetching. Sembrant et al. [22] presented a fast technique for combining hardware sampling with statistical modeling to evaluate cache miss ratios. They presented data for two applications as an example of their technique. Their modeling technique did not address prefetching and only modeled fully associative caches. Henning [9] and Gove [6] measured OS virtual memory footprints and actually used physical memory size for SPEC 2006.

Weinberg et al. [29] scored applications on their temporal and spatial locality, which is purely architecture-independent technique. However, while they were able to make broad comparisons among applications, their techniques were unable to predict memory system performance in the presence of prefetching and set associative caches, or with regards to particular cache sizes.

Table 3 summarizes the prior work. In this work, we covered key aspects for understanding applications' memory system behavior: large and diverse cache sizes, the effect of prefetching, and time-based behavior. By doing so, we have produced memory behavior characterization that we believe to be more reusable as it provides more insight into both the intrinsic sensitivities of the applications and how they vary over time.

7 CONCLUSION

In this work, we have sought to provide insight into the intrinsic memory behavior of the SPEC CPU 2017 and 2006 applications. We have done so by explicitly trading off detailed micro-architectural simulation for a more generic system model that captures the key elements of memory system behavior: sensitivity to cache size and prefetching, both in aggregate and over time. To motivate this tradeoff, we demonstrated how our simplified generic system model covers the broad range of behaviors found in cycle-accurate simulators, while being fast enough to enable detailed exploration of the applications' full execution.

With this approach, we provided an aggregate cache sensitivity analysis of the full SPEC 2006 and 2017 benchmark suites across a broad range of cache sizes, both with and without prefetching. This analysis allowed us to identify the changes in memory system behavior from 2006 to 2017, including cases where the newer benchmarks increased cache pressure (as expected) but also where they reduced it. From this, we categorized both benchmark suite's applications by their cache- and prefetcher-sensitivity.

We then went beyond the aggregate analysis to show the phase-based behavior of the benchmarks. However, in looking at the data it quickly became apparent that it is difficult to draw quantitative conclusions due to the complexity of the phases, particularly when analyzed across a range of cache sizes. To address this, we introduced our MPKI binning technique. This transforms the phase-based data to instead report the aggregate amount of execution that experiences a particular miss rate. We used this approach to analyze the intensities of different miss rates and the impact of prefetchers. This allowed us to understand the relative importance of different phases and how they contribute to the application's overall misses. With this analysis, we were able to find multiple applications that had low aggregate (average) miss rates, but where the majority of the total misses were concentrated in a few phases. By identifying these high-intensity phases that represent the majority of the total misses, this will allow future research to develop more accurately targeted solutions.

In addition to the data and analysis presented in this article, we provide the full analysis and raw data for the SPEC 2006 and 2017 benchmark suite in an online appendix [7, 8] to enable other researchers to identify areas of particular interest.

REFERENCES

- [1] 2020. ChampSim. Retrieved from <https://github.com/ChampSim/ChampSim>.
- [2] Erik Berg and Erik Hagersten. 2004. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, 20–27. DOI: <https://doi.org/10.1109/ISPASS.2004.1291352>
- [3] Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*. Association for Computing Machinery, New York, NY, 245–257. DOI: <https://doi.org/10.1145/781131.781159>
- [4] Jan Edler and Mark D. Hill. 2003. Dinero. Retrieved from <http://pages.cs.wisc.edu/markhill/DineroIV/>.
- [5] David Eklov and Erik Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'10)*. IEEE Computer Society, 55–65. DOI: <https://doi.org/10.1109/ISPASS.2010.5452069>
- [6] Darryl Gove. 2007. CPU2006 working set size. *SIGARCH Comput. Arch. News* 35, 1 (2007), 90–96. DOI: <https://doi.org/10.1145/1241601.1241619>
- [7] Muhammad Hassan, Chang Hyun Park, and David Black-Schaffer. 2020. Architecturally-independent and time-based characterization of SPEC CPU 2017. Retrieved from <http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-417174>.
- [8] Muhammad Hassan, Chang Hyun Park, and David Black-Schaffer. 2020. Raw-data: A reusable characterization of the memory system behavior of SPEC 2017 and SPEC 2006. Retrieved from <http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-428738>.

- [9] John L. Henning. 2007. SPEC CPU2006 memory footprint. *SIGARCH Comput. Arch. News* 35, 1 (2007), 84–89. DOI : <https://doi.org/10.1145/1241601.1241618>
- [10] Aamer Jaleel. 2007. *Memory Characterization of Workloads Using Instrumentation—Driven Simulation*. Technical Report.
- [11] Arun Kejariwal, Gerolf F. Hoflehner, Darshan Desai, Daniel M. Lavery, Alexandru Nicolau, and Alexander V. Veidenbaum. 2008. Comparative architectural characterization of SPEC CPU2000 and CPU2006 benchmarks on the intel Core2 Duo processor. In *Proceedings of the 2008 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. 132–141. DOI : <https://doi.org/10.1109/ICSAMOS.2008.4664856>
- [12] Osnat Levi. 2018. Pin—A dynamic binary instrumentation tool. Retrieved from <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>.
- [13] Ankur Limaye and Tosiron Adegbiya. 2018. A workload characterization of the SPEC CPU2017 benchmark suite. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'18)*. 149–158. DOI : <https://doi.org/10.1109/ISPASS.2018.00028>
- [14] Agustín Navarro-Torres, Jesús Alastruey-Benedé, Pablo Ibáñez-Marín, and Víctor Viñals-Yúfera. 2019. Memory hierarchy characterization of SPEC CPU2006 and SPEC CPU2017 on the Intel Xeon Skylake-SP. *PLoS One* 14, 8 (8 2019), 1–24. DOI : <https://doi.org/10.1371/journal.pone.0220135>
- [15] Reena Panda, Shuang Song, Joseph Dean, and Lizy K. John. 2018. Wait a Decade: Did SPEC CPU 2017 broaden the performance horizon? In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. 271–282. DOI : <https://doi.org/10.1109/HPCA.2018.00032>
- [16] Aashish Phansalkar, Ajay Joshi, and Lizy Kurian John. 2007. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA'07)*. 412–423. DOI : <https://doi.org/10.1145/1250662.1250713>
- [17] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. 2007. Subsetting the SPEC CPU2006 benchmark suite. *SIGARCH Comput. Arch. News* 35, 1 (2007), 69–76. DOI : <https://doi.org/10.1145/1241601.1241616>
- [18] Tribuvan Kumar Prakash and Lu Peng. 2008. Performance characterization of spec cpu2006 benchmarks on intel core 2 duo processor. *ISAST Trans. Comput. Softw. Eng.* 2, 1 (2008), 36–41.
- [19] Ranjan Hebbar S. R. and Aleksandar Milenkovic. 2019. SPEC CPU2017: Performance, event, and energy characterization on the core i7-8700K. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE'19)*. 111–118. DOI : <https://doi.org/10.1145/3297663.3310314>
- [20] Ranjan Hebbar S. R. and Aleksandar Milenković. 2019. SPEC CPU2017: Performance, event, and energy characterization on the core I7-8700K. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE'19)*. ACM, New York, NY, 111–118. DOI : <https://doi.org/10.1145/3297663.3310314>
- [21] Andreas Sandberg, Nikos Nikolieris, Trevor E. Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. 2015. Full speed ahead: Detailed architectural simulation at near-native speed. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'15)*. IEEE Computer Society, 183–192. DOI : <https://doi.org/10.1109/IISWC.2015.29>
- [22] Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. 2012. Phase guided profiling for fast cache modeling. In *Proceedings of the 10th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'12)*. 175–185. DOI : <https://doi.org/10.1145/2259016.2259040>
- [23] Andreas Sembrant, David Eklov, and Erik Hagersten. 2011. Efficient software-based online phase classification. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC'11)*. 104–115. DOI : <https://doi.org/10.1109/IISWC.2011.6114207>
- [24] Xipeng Shen, Yutao Zhong, and Chen Ding. 2004. Locality phase prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. Association for Computing Machinery, New York, NY, 165–176. DOI : <https://doi.org/10.1145/1024393.1024414>
- [25] Sarabjeet Singh and Manu Awasthi. 2019. Memory centric characterization and analysis of SPEC CPU2017 suite. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE'19)*. ACM, New York, NY, 285–292. DOI : <https://doi.org/10.1145/3297663.3310311>
- [26] Shuang Song, Qinzhe Wu, Steven Flolid, Joseph Dean, Reena Panda, Junyong Deng, and Lizy K. John. May 2018. *Experiments with Spec cpu 2017: Similarity, Balance, Phase Behavior and Simpoints*. Technical Report. TR-180515-01, LCA Group, Department of Electrical and Computer Engineering, The University of Texas at Austin.
- [27] Standard Performance Evaluation Corporation. 2018. SPEC CPU2006. Retrieved from <https://www.spec.org/cpu2006>.
- [28] Standard Performance Evaluation Corporation. 2020. SPEC CPU2017. Retrieved from <https://www.spec.org/cpu2017>.

- [29] Jonathan Weinberg, Michael O. McCracken, Erich Strohmaier, and Allan Snively. 2005. Quantifying locality in the memory access patterns of HPC applications. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC'05)*. IEEE Computer Society, USA, 50. DOI : <https://doi.org/10.1109/SC.2005.59>
- [30] Qinzhe Wu, Steven Flolid, Shuang Song, Junyong Deng, and Lizy K. John. 2018. Invited paper for the hot workloads special session hot regions in SPEC CPU2017. In *Proceedings of the 2018 IEEE International Symposium on Workload Characterization (IISWC'18)*. 71–77. DOI : <https://doi.org/10.1109/IISWC.2018.8573479>
- [31] Yutao Zhong, Steven G. Dropsho, and Chen Ding. 2003. Miss rate prediction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*. IEEE Computer Society, 79. DOI : <https://doi.org/10.1109/PACT.2003.1238004>

Received October 2020; revised December 2020; accepted December 2020