CP_{pf}: a prefetch aware LLC partitioning approach

Jun Xiao
University of Amsterdam
Amsterdam, Netherlands
J.Xiao@uva.nl

Andy D. Pimentel
University of Amsterdam
Netherlands
A.D.Pimentel@uva.nl

Xu Liu
College of William and Mary
USA
xl10@cs.wm.edu

ABSTRACT

Hardware cache prefetching is deployed in modern multicore processors to reduce memory latencies, addressing the memory wall problem. However, it tends to increase the Last Level Cache (LLC) contention among applications in multiprogrammed workloads, leading to a performance degradation for the overall system. To study the interaction between hardware prefetching and LLC cache management, we first analyze the variation of application performance when varying the effective LLC space in the presence and absence of hardware prefetching. We observe that hardware prefetching can compensate the application performance loss due to the reduced effective cache space. Motivated by this observation, we classify applications into two categories, prefetching sensitive (*PS*) and non prefetching sensitive (*NPS*) applications, by the degree of performance benefit they experience from hardware prefetchers. To address the cache contention and also to mitigate the potential prefetch-related cache interference, we propose CP_{pf}, a cache partitioning approach for improving the shared cache management in the presence of hardware prefetching. CP_{pf} consists of a method using Precise Event-Based Sampling techniques for the online classification of *PS* and *NPS* applications and a cache partitioning scheme using Cache Allocation technology to distribute the cache space among *PS* and *NPS* applications. We implemented CP_{pf} as a user-level runtime system on Linux. Compared with a non-partitioning approach, CP_{pf} achieves speedups of up to 1.20, 1.08 and 1.06 for workloads with 2, 4 and 8 single-threaded applications, respectively. Moreover, it achieves speedups of up to 1.22 and 1.11 for workloads composed of two applications with 4 threads and 8 threads, respectively.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**.

KEYWORDS

Cache partitioning; Hardware prefetching; Multi-core architectures; Shared cache; Cache allocation technology

ACM Reference Format:
Jun Xiao, Andy D. Pimentel, and Xu Liu. 2019. CP_{pf}: a prefetch aware LLC partitioning approach. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337895

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337895

1 INTRODUCTION

Modern multicore processors implement a large Last Level Cache (LLC) to hide the long memory access latencies. Such a LLC is usually shared by multiple cores to allow high cache utilization. However, cache sharing also causes inter-application cache interference, which occurs when concurrently running applications compete among each other for shared cache space, governed by a cache replacement policy.

Hardware prefetching is another optimization technique that is commonly employed to reduce memory latencies. Although hardware prefetching can improve the applications' performance by fetching useful data in advance, it tends to increase the LLC contention among applications running concurrently on different cores. Taking the hardware prefetching into account, inter-application cache interference becomes more complicated.

Much research has been done to address the problem of inter-application cache interference and the shared cache management [4, 11, 22, 29, 33, 34]. In those works, cache partitioning policies are proposed to improve system throughput, fairness and (or) average slowdown using cache allocation technology, a hardware partitioning approach supported by Intel processors. A significant amount of work has also been devoted to software-based cache partitioning approaches [3, 14, 28, 35] based on a well accepted technique of OS page-coloring. Most of those works have been implemented and evaluated the performance of their cache partitioning policies on real machines. However, those works do not study the impact of hardware prefetching on cache performance nor do they explicitly reveal the interaction between the hardware prefetching and LLC management.

Prior work to this end involves fine-tuned cache insertion and replacement policies [23, 26, 31] to improve the cache management policy in the presence of hardware prefetching. However, the additional hardware components required by those works are not yet available in existing hardware.

Contribution. In a real system, many factors such as cache references by the operating system and hardware prefetching contribute to LLC interference [32]. In this study, we focus on the *LLC* management in the presence of hardware prefetching for multiprogrammed workloads. To study the interaction between hardware prefetching and LLC cache management, we first analyze the variation of application performance when varying the effective LLC space in the presence and absence of hardware prefetching. Here, we show that hardware prefetching can compensate the application performance loss due to the reduced effective cache space. Motivated by this observation, we then classify applications into two

categories, prefetching sensitive (*PS*) and non prefetching sensitive (*NPS*) applications, by the performance benefit they experience from hardware prefetchers. To address the cache contention and to also mitigate the potential prefetch-related cache interference, we propose CP_{pf}, a prefetch aware *LLC* partitioning approach for improving *LLC* management. CP_{pf} consists of a method using Precise Event-Based Sampling (PEBS) techniques for online classification of *PS* and *NPS* applications and a *LLC* partitioning scheme using Cache Allocation technology (CAT) for *PS* and *NPS* applications. Compared with a non-partitioning approach, CP_{pf} achieves performance improvements of up to 1.20, 1.08 and 1.06 for workloads with, respectively, 2, 4, and 8 applications and achieves speedups of up to 1.21 and 1.11 for workloads composed of two applications with 4 threads and 8 threads, respectively.

The rest of the paper is organized as follows. Section 2 presents the motivation of this work. The background of hardware performance monitoring units and cache allocation technology is introduced in Section 3. Section 4 provides the definition of *PS* and *NPS* applications. Section 5 describes CP_{pf}, where we also detail the online classification of *PS* and *NPS* applications and the LLC cache partitioning scheme. Section 6 presents the performance evaluation of CP_{pf}. Section 7 gives an overview of related work, after which Section 8 concludes the paper.

2 MOTIVATION

2.1 The impact of hardware prefetching on cache performance

Hardware prefetching implemented in today's high performance systems significantly influences memory sub-system performance. To understand the effects of hardware prefetching on the LLC performance for a single application, we evaluate the variation of application performance when varying the number of assigned LLC cache-ways in the presence and absence of hardware prefetching.

All the experiments in this work are conducted on a 20-core Intel Xeon commodity processor, of which the specifications are summarized in Table 1. There are five distinct hardware prefetchers on the Xeon platforms. Two prefetchers are associated with the L1-data caches: a Data Cache Unit (DCU) IP prefetcher and a DCU streamer prefetcher per core. Two prefetchers associated with the L2 caches: a Mid-Level cache (MLC) spatial prefetcher and a MLC streaming prefetcher. Finally, there is one LLC prefetcher. We can activate or deactivate these hardware prefetchers by setting the corresponding machine state register (MSR) bits [7].

Table 1: System Configuration

Component	Description
Processor	Intel Xeon Gold 6148 CPU @ 3.50GHz
L1 I-cache	Private, 32KB
L1 D-cache	Private, 32KB
L2 cache	Private, 1MB
L3 cache	Shared, 27.5MB, 11 ways
Memory	376G
OS	CentOS 7, Linux Kernel 4.17

Given the number of assigned LLC cache-ways, we run an application with single thread in isolation and measure its execution time for two cases: (1) hardware prefetchers are disabled, (2) hardware prefetchers are enabled. Figure 1 compares the slowdown for applications in the SPEC CPU2017 [24], NPB [17] and Polybench [20] benchmark suites when varying the number of assigned cache-ways for the two cases. Due to space limitations, we only show the comparison for six representative applications, each application is identified by its index (in SPEC CPU2017) or abbreviation for its name (in NPB and Polybench). In Figure 1(a), the slowdown of an application is calculated by taking the execution time when it utilizes all the cache ways (11, in our experimental platform) and hardware prefetchers are disabled as the baseline, while in Figure 1(b), the baseline is execution time when the application fully utilizes all cache-cache ways and hardware prefetchers are enabled. Note that the baselines in Figure 1(a) and Figure 1(b) are thus different.

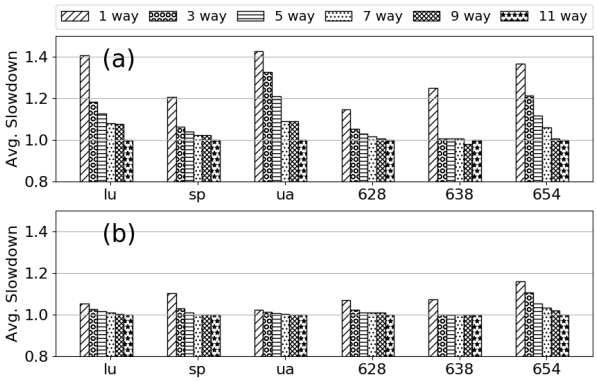


Figure 1: Comparison of application slowdown when varying the number of cache-ways allocated: (a) prefetching is disabled, (b) prefetching is enabled.

As illustrated in Figure 1, some applications, which originally experience significant performance degradation from a smaller LLC space in the absence of prefetching, encounter less performance degradation when the hardware prefetchers are enabled. For example, when hardware prefetching is disabled, the worst slowdowns for applications *lu* and *ua* are 1.41 and 1.42, respectively. However, the worst slowdowns are improved to 1.05 and 1.02 for *lu* and *ua*, if hardware perfetching is enabled. Thus, we make the following observation:

Observation 1. *Hardware prefetching can compensate the application performance loss due to a reduced effective LLC space.*

This can be explained by the fact that a prefetch-enabled LLC cache-controller will prefetch data from main memory before the actual references take place in order to try to avoid memory access latencies. Even though the effective LLC size for an application is decreased, the demanded data can often still be directly and timely serviced by the hardware prefetchers.

2.2 Inter-core prefetch-related cache pollution

The prefetched data for one application are placed in the shared LLC, competing for the available cache resources with its co-runners (i.e.,

other, simultaneously running applications). Therefore, one major drawback of hardware prefetching is the prefetch-related cache pollution which occurs when prefetched blocks of one application evict useful blocks of another application from the LLC. In this work, we assume that hardware prefetching taking place on behalf of an application itself has a more positive than negative influence on its performance. Thus, we neglect cache interference caused by self-prefetching and only consider inter-core prefetch-related LLC interference.

In a multicore system, inter-core prefetch-related cache pollution impacts the performance of applications in a non-uniform fashion. Some applications can be slowed down severely as a large number of its useful blocks can be replaced by prefetched blocks, while others may not. Hardware prefetching can interact poorly with LLC management, which unnecessarily reduces the overall system performance. This leaves a significant opportunity to improve LLC management by means of prefetch-aware cache partitioning.

3 BACKGROUND

3.1 Hardware PMU

To provide realtime micro-architectural information about the processes currently executing on the chip, a rich set of Hardware Performance Monitoring Units (PMUs) is implemented in today's processor micro-architectures. These PMUs offer a programmable way to count hardware events such as cpu cycles, instructions executed, cache statistics, etc. PMUs also support advanced event sampling, a mechanism that collects event samples at a predefined sampling period. The event based sampling is realized by Intel's Precise Event-Based Sampling (PEBS) [6] and AMD's Instruction Based Sampling (IBS) [9].

To use the PEBS mechanism, a counter is configured to overflow after it has counted a preset number of events. After the counter overflows, the processor copies the current state of the general-purpose registers and instruction pointer in the records buffer. The processor then resets the performance counters and restarts the event counter.

Linux' `perf_event` is a standard programming interface to set up performance monitoring through PMUs. More specifically, `perf_event_open` [8] can set the PMUs in sampling mode, and the overflow event can be enabled via `ioctl()` calls. The Linux kernel can deliver a signal to the threads whose PMU event counter overflows. The user code can `mmap` a circular buffer into which the kernel keeps appending the PMU data on each sample. The user can also read those circular buffers.

3.2 Cache Allocation Technology

To address the contention on the LLC from multiple applications running simultaneously on different cores and to enable isolation and prioritization of key applications, recent commodity CPUs have provided hardware support for LLC partitioning [12]. Intel has proposed the so-called cache allocation technology (CAT), which provides software-programmable control over the amount of cache space that can be consumed by a given application.

Machines that support CAT have a predefined number of classes of service (CLOS), for example, 11 in our experimental machine. Each CLOS is associated with a capacity bit mask (CBM) that controls the accessibility of cache resources with cache-way granularity, where each bit in the mask grants write access to one way in the cache. Each application belongs to a CLOS and a particular application can only access the cache-ways defined by the CBM for that CLOS.

In this work, we use `Intel-cat-cmt`, which is a library [5] developed by Intel, to configure CAT.

4 *PS* AND *NPS* APPLICATIONS

In this section, we first classify applications into two categories: prefetching sensitive (PS) and non prefetching sensitive (NPS) applications by the performance benefit they experience from hardware prefetchers. We then study the performance sensitiveness to the available cache space for *PS* and *NPS* applications.

4.1 Definition of *PS* and *NPS* applications

We measure the execution time of an application in the presence and absence of hardware prefetching, respectively. We calculate the speedup of an application i by $SpeedUp_i = \frac{ET_{i,nopf}}{ET_{i,enpf}}$, where $ET_{i,nopf}$ is the execution time of application i when prefetchers are disabled and $ET_{i,enpf}$ is the execution time when hardware prefetchers are enabled.

We define applications whose performance is significantly improved by hardware prefetching as prefetching sensitive (PS) applications. In this work, application i is considered a *PS* application if $SpeedUp_i > 20\%$. An application that is not a *PS* application is considered to be an *NPS* application. By this definition, we classify the applications in the SPEC CPU2017 [24], NPB [17] and Polybench [20] benchmark suites into *PS* and *NPS* applications. The classification is shown in Table 2.

Table 2: Classification of *PS* and *NPS* applications.

Type	Applications
PS applications	619, 654, 628, 638, 603, mg, cg, sp, is, bt, ft, fdtd2d, jacobi2d, heat3d
NPS applications	602, 605, 607, 631, 623, 627, 600, 641, 644, 648, 657, 620, ua, lu, dc, ep, adi

4.2 Cache sensitivity of *PS* and *NPS* applications

In a multiprogramming environment, the shared cache interference caused by co-runners (i.e., simultaneously running applications) reduces the effective number of cache-ways that an application can use. To study the impact of available cache-ways on the performance of *PS* and *NPS* applications, we conduct several experiments in which we use CAT to adjust the number of LLC ways available to the application from 1 to 11 (i.e., the total cache space in our experimental platform). In the experiments, all hardware prefetchers are enabled and each application has one thread. Using this approach, we model the reduction in the available LLC space due to cache interference caused by co-runners.

Figure 2a and Figure 2b show the slowdown for 8 representative *PS* and *NPS* applications, respectively. The slowdown is calculated

by taking the execution time when an application runs in isolation and utilizes all the cache ways as the baseline.

As can be seen, compared with *NPS* applications, the effective LLC size has, on average, a relatively small influence on the performance of *PS* applications. The performance of most *PS* application is sightly degraded if the effective LLC size decreases. The average maximum slowdown (obtained when an application runs with one cache-way) for *PS* applications is 1.05 with a worst-case slowdown of 1.15 for SPEC CPU2017 benchmark 654. For *NPS* applications, however, the average slowdown is 1.18 with a worst case of 1.62, experienced by SPEC CPU2017 benchmark 607. Thus, we make the following observation:

Observation 2. *If hardware prefetchers are enabled, on average, the effective LLC size has a relatively small influence on the performance of PS applications, while the performance of NPS applications can be significantly affected by the effective LLC size.*

The much smaller influence of the LLC size on the performance of *PS* applications can be explained by:

(1) *PS* applications may have a low reuse of data cached in the LLC because of timely prefetched data in the smaller, upper-levels of the cache hierarchy (L1/L2 caches) [23]. Subsequent data requests are directly serviced by the prefetched cache lines inserted into the L1/L2 caches, and rarely reach the LLC.
(2) *PS* applications can more easily cope with higher LLC miss rates caused by the reduction of effective LLC space as a majority of demanded data elements can still be directly and timely serviced by the hardware prefetchers.

5 PREFETCH AWARE LLC PARTITIONING

To exploit Observation 2, this section presents the prefetch aware LLC partitioning approach CP_{pf}. The general idea is to classify *PS* and *NPS* applications at run time and then divide the LLC into two partitions: one for *PS* applications and the other for *NPS* applications. Section 5.1 describes the online classification of *PS* and *NPS* applications, and Section 5.2 presents the LLC partitioning approach.

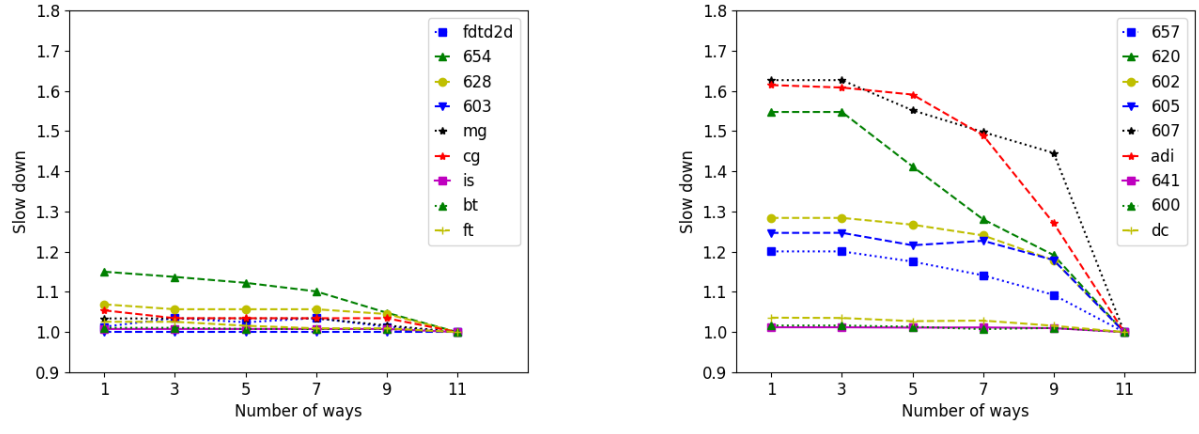
5.1 Online classification of applications

5.1.1 A classification criterion: cache miss distribution. The definition of *PS* and *NPS* application cannot be used directly for the online classification of *PS* and *NPS* applications. Due to the uncontrollable and unclear nature of hardware prefetching mechanisms implemented in modern commodity processors, we developed a non-trivial solution for the online classification of *PS* and *NPS* applications, which is based on the distribution of cache misses over the cache sets. The idea comes from the fact that prefetchers do not prefetch across virtual page boundaries. As indicated in [7], prefetched data will always be within the same 4K bytes memory page as the load instruction that triggered the prefetching.

The first (several) references to a data element in a new virtual page usually cannot be prefetched. Therefore, accesses to those data elements always result in LLC misses. After these first accesses, the hardware prefetchers start to recognize the data access patterns and start to predict and prefetch the data that is expected to be referenced in the near future. As a consequence, later data references inside the same virtual page do not necessarily cause LLC misses, as the hardware prefetchers may have inserted those data elements into the LLC before referencing them.

By using the PMU sampling mechanism, one can obtain the virtual addresses that missed by a process in the LLC. Given a missed virtual address, one can determine the associated LLC set that the virtual address maps to. We will show the method to determine the missed cache set soon. By sampling the LLC misses over a short execution period (for instance, 1 second) for a process, we can obtain the cache miss distribution over the cache sets for the sampled process.

We use histograms to represent the distribution of LLC misses over the LLC sets. Figure 3 illustrates the histogram of missed cache sets when hardware prefetchers are disabled. Due to space limitations, we only show the histograms for four representative



(a) *PS* applications (b) *NPS* applications

Figure 2: Application slowdown when varying the number of available ways with respect to a 11-way cache, if hardware prefethers are enabled.

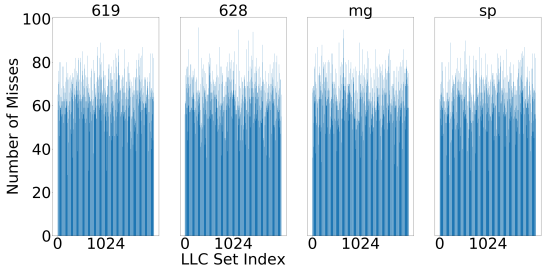


Figure 3: Histogram of missed LLC sets when hardware prefetchers are disabled.

applications. As can be seen, when all hardware prefetchers are disabled, cache misses are mostly uniformly distributed over all the cache sets.

Although we only show cache miss distributions of *PS* applications for later comparison, the uniform distribution is observed also for *NPS* applications. Observation 3 follows:

Observation 3. *When hardware prefetchers are disabled, cache misses are mostly uniformly distributed over all the LLC sets for both* PS *and* NPS *applications.*

Observation 3 verifies the assumption that a program block has a uniform probability of being present in any of the cache sets in the works on analytic cache models [1].

However, if hardware prefetchers are enabled, we obtain different cache miss distributions for *PS* and *NPS* applications, as illustrated in Figure 4. Note that the scale of the y-axes in Figure 4a and Figure 4b are different.

As shown in Figure 4a, the cache miss distributions over cache sets are non-uniform for *PS* applications. It is clear that cache sets associated with spikes exhibit many more (more than 10×) cache misses than other sets. In most cases, the index of those sets is $64p$ with $p = 1, 2, 3, \ldots$, where the beginning of a new virtual page is mapped to. From this, we infer that cache misses at those sets are caused by the first references to the data in a new virtual memory page.

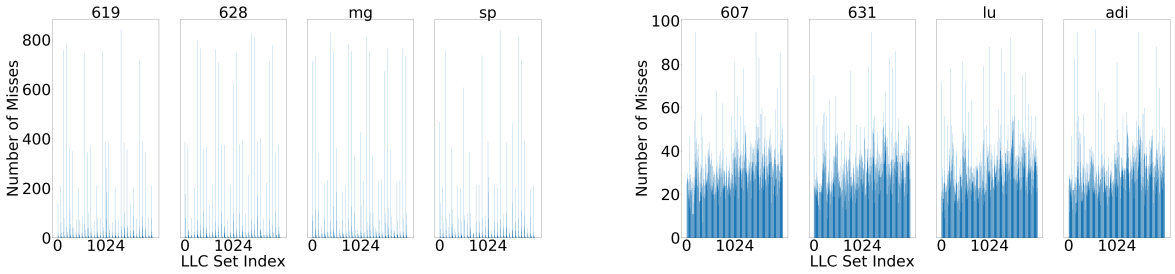
Figure 4b depicts the distributions of missed cache sets for *NPS* applications when hardware prefetchers are enabled. Although there exist a few cache sets with spikes, the gap between the spikes and the average number of misses over a cache set is much smaller. Overall, the cache misses are still uniformly distributed over all the cache sets. Thus, we make the following observation:

Observation 4. *When hardware prefetchers are enabled, cache miss distributions over cache sets are non-uniform for PS applications, while the distributions are mostly uniform for NPS applications.*

Based on the difference in cache miss distributions between *PS* and *NPS* applications when hardware prefetchers are enabled, we propose a ratio between the maximum value and the median value of the frequency of LLC misses exhibited by one cache set to determine whether an application is *PS* or not. To reduce the complexity, the median value is approximately computed as the average of LLC misses exhibited by 30 randomly selected cache sets. We skip selecting the cache sets that exhibit misses more than 70% of the maximum value. We calculated the mean of the ratio for both *PS* and *NPS* applications. The mean of the ratio obtained from the *PS* application is 25.7, while for *NPS*, the mean is 3.23. When the ratio is larger than a threshold (10, in this work), the application is classified as a *PS* application. Otherwise, it is considered to be an *NPS* application.

5.1.2 Obtaining cache miss distribution. As described, the cache miss distribution over the cache sets can be obtained by following these steps: virtual addresses that missed in the LLC can be obtained by using the PMU sampling mechanism, after which each obtained virtual address needs to be translated to the corresponding physical data address to determine the missed LLC cache set. By sampling the LLC misses over a short execution period, one can obtain the cache miss distribution. We describe those steps in details below.

PMU sampling. Intel PEBS supports address sampling, a type of event-based sampling that allows associating sampled performance events with instruction pointers (IP) and effective data addresses. Moreover, PEBS address sampling in recent Intel processors (i.e., Haswell and its successors) allows precisely monitoring cache misses at memory level. In this work, we choose the event `MEM_LOAD_UOPS_RETIRED:L3_MISS` to drive PMU sampling. After experimenting with different sampling periods ranging from 5 (i.e., every 5th miss) to 1000, we decided to use a sampling period of 10, as it incurs a small overhead while still providing enough samples for the later analysis. In this configuration, the PMU therefore samples one per ten data addresses that missed in LLC. Note that the sampled data addresses are virtual addresses.



(a) *PS* applications **(b) *NPS* applications**

Figure 4: Histogram of missed cache sets when hardware prefetchers are enabled.

Virtual-to-physical address translation. As LLCs are physically indexed and physically tagged (PIPT), a virtual address obtained from a PMU sample does not suffice to get the information about the missed LLC set. Therefore, a virtual-to-physical address translation is required. This translation can be done by using `Pagemap`, a set of interfaces in the Linux kernel that allow user space programs to examine the page tables and related information.

Since the default page size of most Linux systems in the virtual address space is 4K bytes, during the virtual-to-physical address translation, bits $0-11$ of the virtual address that encode the page offset are preserved. Bits 12 and above of the virtual address, which encode the page number in the virtual address space, are replaced by the physical page frame number. The mapping from the virtual page to the physical page frame can be found in `/proc/self/pagemap`, a component in `Pagemap`.

LLC addressing. The LLC in a modern multicore processor is usually organized into as many slices as the number of cores with the purpose of reducing the bandwidth bottleneck when more than one core attempts to retrieve data from the LLC at the same time.

Typically, the LLC is set-associative, with a total of k cache sets in each cache slice and m ways. A cache line with a size of c bytes occupies a single way of a cache set. The slice and cache set to which a physical memory address maps is determined by its address bits, as shown in Figure 5.

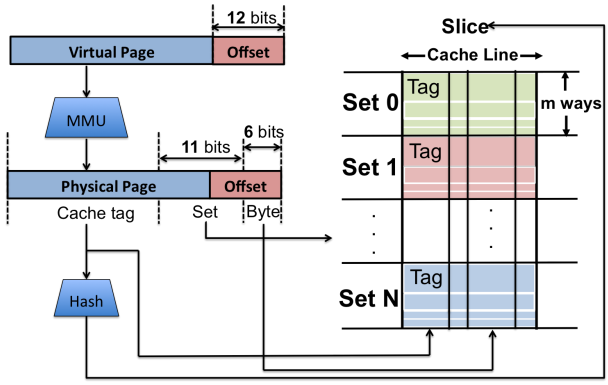


Figure 5: LLC addressing. A virtual data address is translated to a physical data address by the memory management unit (MMU). For a typical caching system with $k = 2048$, $c = 64$, the lowest 6 bits (bits $0-5$) are used to determine the offset within a cache line and bits $6-16$ select the cache set. Higher bits (bits 17 and above) are used as tag and input to a hash function to decide the cache slice.

As indicated in [13], the least significant $\log_2 c$ bits of the physical address are used to address a byte or word within a cache line. The next $\log_2 k$ bits select the set that the cache line belongs to. Bits $\log_2 k + \log_2 c$ and above are utilized as a tag for comparison when looking for data in the cache. The Intel processors use an undocumented hash function of higher bits (bits $\log_2 k + \log_2 c$ and above) of a physical address to decide the cache slice.

In the absence of knowledge about the hash function used for mapping, a given cache line can be present in any of the slices. As cache miss behavior in different cache slices is very similar, in this work, we do not distinguish the cache lines in different cache slices.

Histogram of missed cache sets. The histogram of missed cache sets can be derived by sampling the LLC misses for a short execution period and calculating the missed cache set that corresponds to each sampled miss. We have set the sampling period to 1 second in this work.

The proposed detection approach is accurate and able to detect all the *PS* applications in the benchmarks used in this study, even when they co-run with 10 other applications.

5.2 LLC partitioning for *PS* and *NPS* applications

Most of the *PS* applications are memory-intensive. When *PS* applications run simultaneously with *NPS* applications fully sharing the LLC, we observe that *PS* applications often occupy more LLC space than the *NPS* applications, leading to significant performance degradation of *NPS* applications. We will show such a scenario in the next section.

One of the reasons *PS* applications can occupy more LLC space is that *PS* applications can generate a large number of prefetching requests. As observed in [25], applications gain more benefit from hardware prefetching tend to generate more prefetch requests.

When the hardware prefetchers are enabled, we observed in Section 4 that the effective LLC size has only very limited effect on the performance of *PS* applications. The aim of the cache partitioning in this work is therefore to limit the LLC size occupied by *PS* applications and reserve more LLC space for *NPS* applications. By doing so, the potential prefetch-related cache pollution for *NPS* applications is also mitigated.

Our cache partitioning scheme is simple: it initially allocates one exclusive cache-way to each newly classified *PS* application as it does not benefit greatly from a larger LLC size. It then allocates the remainder of the cache-ways to the *NPS* applications. When a *PS* application finishes its execution, the exclusive cache-way that was previously owned by that application is assigned to the *NPS* applications.

We also observed that the performance of *PS* applications degrades only slightly even when multiple of such applications share a single way of the LLC. If multiple *PS* applications are present, we randomly select two among these applications to share the same way for a short time interval (0.1 second, in this work), We repeat the dynamic adjusting of one shared way for two randomly selected *PS* applications for up to 10 times, each time measuring the IPC of all co-running applications. When the repetition finishes, we keep the best CAT configuration with the maximum sum of IPC of all co-running applications.

Note that, in this work we only focus on LLC partitioning between *PS* and *NPS* applications. Further improvement can be achieved by LLC partitioning among *NPS* applications, as has already been done in [11, 22, 33].

6 EXPERIMENTS

The prototype of CP_{pf} is implemented as a user-level runtime system on Linux. This section evaluates the performance of CP_{pf}. The experiment platform is described in Section 2.1. It has 376GB of

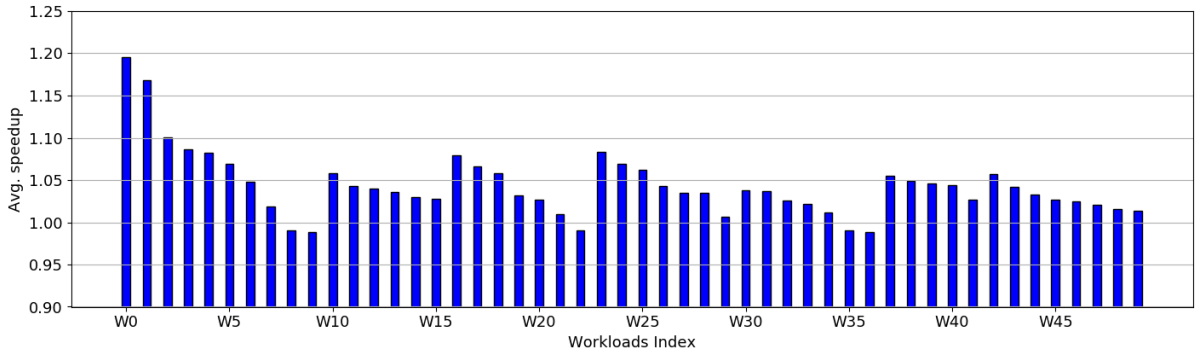


Figure 6: Average speedup achieved by CP$_{pf}$ for each single-threaded workload mix, with respect to the baseline configuration.

main memory and the maximum memory bandwidth is 119.21 GB/s, so the memory contention will be small. Hyperthreading is disabled to avoid intra-core interference. All of the hardware prefetchers are kept enabled during the experiments.

Single-threaded workload mixes: The experiments have been conducted with more than 200 workload mixes from the SPEC 2017 [24], NPB [17] and Polybench [20] benchmark suites. We select three representative sets of 50 multiprogram mixes. The first set contains ten 2-application workloads with index $W0 - W9$, the second set twenty 4-application workloads with index $W10 - W29$ and the third set has twenty 8-application workloads with index $W30 - W49$. Though we would have liked to go beyond 8-application workloads, CAT in our tested platform can only support at most 11 CLOSs.

In each set, the workload mixes were randomly generated by varying the ratio of *PS* applications (25%, 50% and 75%). The proportions of *PS* applications in each workload mix are listed in Table 3. For each workload mix, performance is measured by executing each application until all the applications have completed the same number of instructions they execute when running alone for 20 seconds. The applications are pinned to cores to facilitate the performance monitoring and cache partitioning.

Table 3: Composition of workload mixes.

PS applications (%)	Workloads Index
25%	$W10 - W15$, $W30 - W36$
50%	$W0 - W9$, $W16 - W22$, $W37 - W41$
75%	$W23 - W29$, $W42 - W49$

Metrics: We measure system performance using the average speedup, calculated as follows for the workload with a mix of N applications:

$$AverageSpeedup = \frac{1}{N}\sum_{i=1}^{N}\frac{IPC_{i,CP_{pf}}}{IPC_{i,FullShare}}$$

where $IPC_{i,FullShare}$ is the IPC of program i measured in the baseline configuration, in which the LLC is unpartitioned and is fully shared among all the application; $IPC_{i,CP_{pf}}$ is the IPC of program i obtained when CP$_{pf}$ is applied.

6.1 CP$_{pf}$ performance gain

Figure 6 summarizes the performance gained by CP$_{pf}$ for the workload mixes composed of single-threaded applications. Note that, in Figure 6, workload mixes having the same number of applications and same proportions of *PS* applications are sorted by their speedups. Compared with the baseline performance, CP$_{pf}$ improves the performance for 45 out of 50 workload mixes.

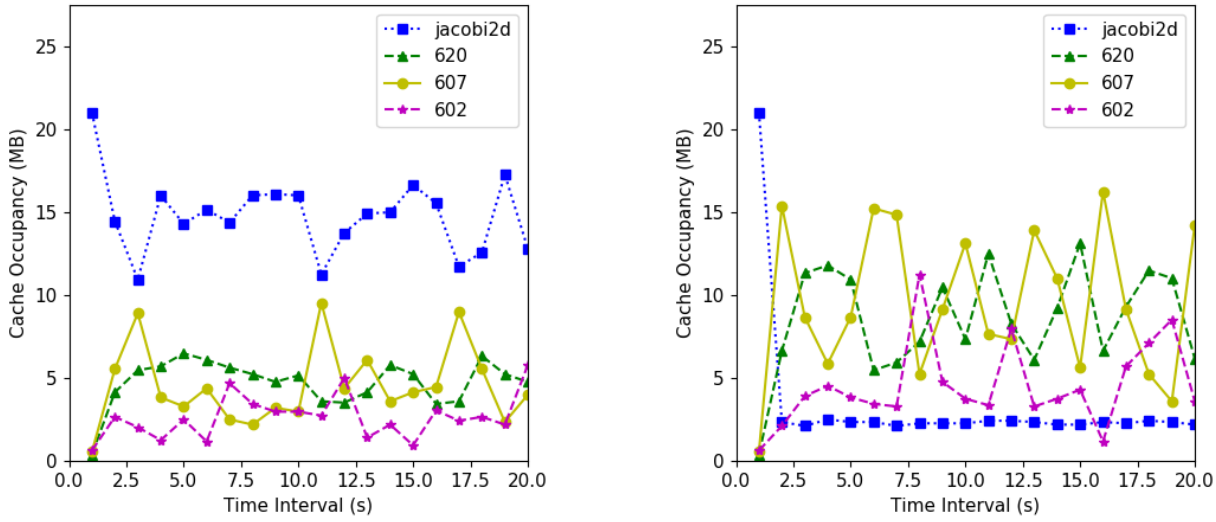
CP$_{pf}$ achieves a speedup of 1.08 on average for workloads with 2 applications, with a best case speedup of 1.20. The average speedup for workloads with 4 applications is 1.04 with a best case of 1.08. Finally, for workloads with 8 applications, the average speedup is 1.03, with a best case of 1.06.

6.2 Cases study of CP$_{pf}$

We now take a closer look at representative workload mix $W11$ consisting of four applications (i.e. *jacobi2d*, 620, 607, 602) to better understand how CP$_{pf}$ can improve the overall system performance.

Figure 7a and Figure 7b illustrate the run-time cache occupancy of the 4 applications in case the LLC is fully shared and CP$_{pf}$ is applied, respectively, during a 20 seconds time interval. When the LLC is fully shared (Figure 7a), the *PS* application *jacobi2d* occupies more than half of the LLC space for most of the time. As a result, *NPS* applications 620, 607, 602 get less LLC space. This situation is improved by CP$_{pf}$. Once CP$_{pf}$ has identified *jacobi2d* as the only *PS* application, it allocates only one way to *jacobi2d*, leaving the rest of the LLC shared by the *NPS* applications 620, 607, 602, as depicted in Figure 7b. In this case, CP$_{pf}$ achieves a 1.10, 1.02 and 1.06 speedup for 620, 607 and 602, respectively, while the speedup of *jacobi* is 0.99. At the cost of a small slowdown of *PS* applications, CP$_{pf}$ yields a higher speedup for *NPS* applications.

We also take a look at one the workload mixes that exhibits a performance degradation under CP$_{pf}$: $W9$. $W9$ is composed of *cg* (the *PS* application) and 641 (the *NPS* application). The performance of 641 cannot be improved enough by getting more cache space, in this case, the speedup of 641 is 1.001. The performance of *cg* is degraded by 0.975 as CP$_{pf}$ allocates a one-way cache space to *cg*. However, no significant performance losses are observed as the lowest speedup (i.e., slowdown) is 0.988.



(a) when LLC is shared

(b) CP$_{pf}$ is applied

Figure 7: Dynamic cache occupancy by applications in workload $W11$.

6.3 CP$_{pf}$ with multithreaded workloads

CP$_{pf}$ also supports multithreaded workloads. For multithreaded workloads, cache miss distributions are obtained per thread, and the LLC is partitioned per thread.

We generate two sets of totally 30 multithreaded workload mixes. Each workload mix consists of two multithreaded applications, one randomly selected from PARSEC [2] or SPLASH [30] as an *NPS* application, and the other from NPB [17] or an OpenMP version of Polybench [20] as a *PS* application (we skip applications from SPEC CPU2017 [24] as it provides multithreaded implementations for a very limited number of applications). The first set contains fifty 4-threaded workloads with index $W50 - W64$ and the second set has fifty 8-threaded workloads with index $W65 - W79$.

Figure 8 presents the average speedups for the multithreaded workload sets. Compared with the baseline performance where caches are fully shared among all the threads, CP$_{pf}$ achieves a speedup of 1.05 on average for workloads with 4 threads, with a best case speedup of 1.22. The average speedup for workloads with 8 threads is 1.04 with a best case of 1.11.

6.4 Sensitivity Analysis

We now analyze CP$_{pf}$'s sensitivity to the characteristics of the workload mix, particularly the ratio between *PS* and *NPS* applications and the workload mix size.

The effect of workload distribution. CP$_{pf}$ achieves average speedups of 1.03 , 1.06 and 1.04 for workloads with 25%, 50% and 75% of *PS* applications.

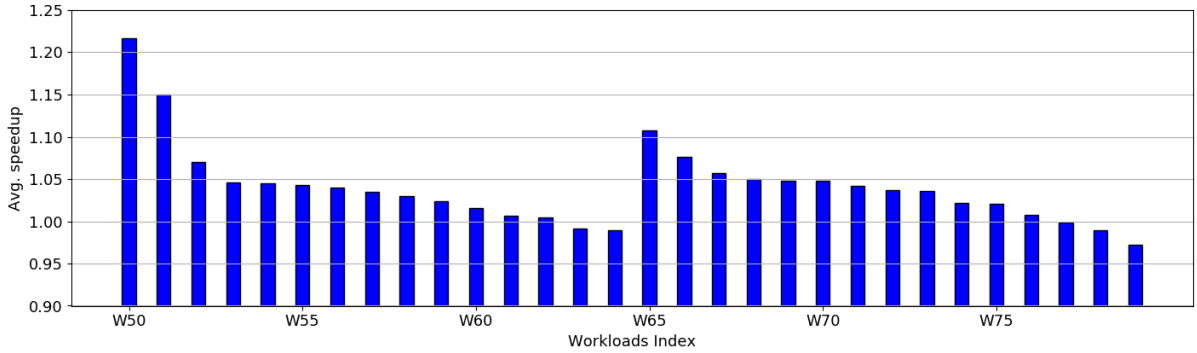


Figure 8: Average speedup achieved by CP$_{pf}$ for each multithreaded workload mix, with respect to the baseline configuration.

When the workload mixes are dominated by *PS* applications, the performance improvement due to an increased LLC space allocated for *NPS* applications by CP$_{pf}$ will be limited by the small number of *NPS* applications in the workload mixes.

When workload mixes are dominated by *NPS* applications, the benefits of CP$_{pf}$ also become more muted. This is because, as indicated in Section 5.2, CP$_{pf}$ does not partition the cache among the *NPS* applications. Even though the cache space occupied by *PS* applications is limited, most of the rest of the cache can be occupied by *NPS* applications whose performance will not be improved greatly by getting more cache space. CP$_{pf}$ cannot guarantee that those *NPS* applications whose performance significantly improves from a larger effective cache size will always occupy more cache space than other applications.

The effect of workload size. We compare the performance gained by CP_{pf} under different sizes of workload mixes (ranging from 2 to 8 applications per workload mix). CP_{pf} gains less performance when the number of co-executing applications increases. This is inevitable because cache contention for both LLC space and cache set associativity is increased as more applications share the LLC.

6.5 Overhead

In order to obtain the actual performance degradation that CP$_{pf}$ results in, we compare the execution times of the applications in SPEC CPU2017, NPB and Polybench benchmarks with two settings, CP$_{pf}$ off and CP$_{pf}$ on. The results show that CP$_{pf}$ causes 0.56% slowdown on average with a worst case of 1.72%.

The overhead of CP$_{pf}$ mainly comes from the online classification of *PS* and *NPS* applications at run time. For an execution phase which typically lasts more than 30 seconds, the PMU samples LLC misses for only 1 second, during which on average 30% of the time is dedicated to PMU sampling and virtual-physical address translation. As PMU can sample up to 200000 data addresses in 1 second, it takes up to 100 milliseconds to obtain the miss distribution over cache sets.

7 RELATED WORK

LLC management. Shared cache management has attracted a lot of research attention in the past decades. UCP [21] and ASM [27] designed additional hardware components to modify the eviction and insertion policies to partition the cache, but these have not been implemented in existing processors.

Heracles [15] and Dirigent [36] control the amount of shared hardware resources, including the LLC, used by latency sensitive applications to improve Quality of Service and utilization. [22] clusters applications using the k-means algorithm and distributes cache ways between the groups to improve system fairness. [19] assigns more cache space to critical applications to improve system turnaround time. [33] proposes a framework that dynamically monitors and predicts a workload's cache demand and reallocates the LLC given a performance target. KPart [11] leverages online profiling to obtain miss ratio curves for clustering applications and assigns each cluster of applications to a cache partition to improve system throughput. [18] proposed a coordinated partitioning of the LLC and memory bandwidth to improve the fairness of workloads on commodity servers.

A significant amount of work has been devoted to software-based cache partitioning approaches [3, 14, 28, 35]. Most of these efforts are based on the classic technique of OS page-coloring, which is used to control where the physical page required by the target application is located in the cache.

All these works have been implemented on existing processors, however, those works do not study the impact of hardware prefetching on cache performance and do not explicitly reveal the interaction between the hardware prefetching and LLC management.

Hardware prefetching. Hardware prefetching is now used in nearly all high-performance commercial processors. [16] presents a survey of prefetching techniques for processor caches.

Some work has also been done to improve the cache management policy in the presence of hardware prefetching. [31] proposed a prefetching-aware cache replacement policy that treats prefetch and demand requests identically. [26] estimates prefetcher accuracy and prefetch-related cache pollution to adjust the aggressiveness of the hardware prefetcher dynamically. In [37, 38], a number of hardware-based prefetch pollution filtering mechanisms is proposed to differentiate good and bad prefetches dynamically to reduce the ineffective prefetches. [23] proposed a self-tuning prefetch accuracy predictor to predict if a prefetch is accurate or inaccurate to mitigate prefetch-related cache pollution. [10] proposed mechanisms that manage the shared resources on a multicore chip to obtain high performance and fairness. However, those approaches require additional hardware components that are not available in existing processors.

The prefetch aware cache partitioning approach presented in this work is a software-only solution by using hardware features like PEBS and CAT, which are readily available in existing multicore processors.

8 CONCLUSION

Hardware prefetching can interact poorly with LLC management, leading to performance degradation. To study the interaction between hardware prefetching and LLC cache management, we analyzed the variation of application performance when varying the effective LLC space in the presence and absence of of hardware prefetching. We observed that hardware prefetching can compensate the application performance loss due to the reduced effective cache space. Motivated by this observation, we classified applications into two categories, prefetching sensitive (*PS*) and non prefetching sensitive (*NPS*) applications, by the degree of performance benefit they experience from hardware prefetchers. To address the cache contention and to also mitigate the potential prefetch-related cache interference, we proposed CP$_{pf}$, a prefetch aware cache partitioning approach for improving the LLC management in the presence of hardware prefetching. CP$_{pf}$ consists of a method using PEBS techniques for the online classification of *PS* and *NPS* applications and a LLC partitioning scheme via CAT to distribute the cache space among *PS* and *NPS* applications. We have implemented CP$_{pf}$ as a user-level runtime system on Linux. Compared with a non-partitioning approach, CP$_{pf}$ achieves speedups of up to 1.20 (1.08 on average), 1.08 (1.04 on average) and 1.06 (1.03 on

average) for workloads with 2, 4 and 8 single-threaded applications, respectively. Moreover, it achieves speedups of up to 1.22 (1.05 on average) and 1.11 (1.04 on average) for workloads mixes composed of two applications with 4 threads and 8 threads, respectively.

REFERENCES

[1] A. Agarwal, J. Hennessy, and M. Horowitz. 1989. An Analytical Cache Model. *ACM Trans. Comput. Syst.* 7, 2 (May 1989), 184–215. https://doi.org/10.1145/63404.63407

[2] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[3] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. 2015. Optimal Cache Partition-Sharing. In *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP) (ICPP '15)*. IEEE Computer Society, Washington, DC, USA, 749–758. https://doi.org/10.1109/ICPP.2015.84

[4] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. 2013. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-efficiency While Preserving Responsiveness, In In Proceedings of the 40th International Symposium on Computer Architecture. *SIGARCH Comput. Archit. News* 41, 3, 308–319. https://doi.org/10.1145/2508148.2485949

[5] Intel Corporation. [n. d.]. User space software for Intel(R) Resource Director Technology. *Available: https://github.com/intel/intel-cmt-cat* ([n. d.]).

[6] Intel Corporation. 2010. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2.*

[7] Intel Corporation. 2018. *Intel 64 and IA-32 Architectures Optimization Reference Manual.*

[8] Perf developers. [n. d.]. *perf_event_open - Linux man page.* https://linux.die.net/man/2/perf_event_open.

[9] Advanced Micro Devices. February, 2015. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 30h-3Fh Processors.*

[10] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. 2011. Prefetch-aware shared-resource management for multi-core systems. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 141–152. https://doi.org/10.1145/2000064.2000081

[11] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sánchez. 2018. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. 104–117.

[12] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. 2016. Cache QoS: From concept to reality in the IntelÂő XeonÂő processor E5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 657–668.

[13] G. Irazoqui, T. Eisenbarth, and B. Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *2015 Euromicro Conference on Digital System Design (DSD)*, Vol. 00. 629–636. https://doi.org/10.1109/DSD.2015.56

[14] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 367–378. https://doi.org/10.1109/HPCA.2008.4658653

[15] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. 450–462.

[16] Sparsh Mittal. 2016. A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Comput. Surv.* 49, 2, Article 35 (Aug. 2016), 35 pages. https://doi.org/10.1145/2907071

[17] NASA. [n. d.]. *NAS Parallel Benchmarks 3.3.* https://www.nas.nasa.gov/assets/npb/.

[18] Jinsu Park, Seongbeom Park, and Woongki Baek. 2019. CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, USA, Article 10, 16 pages. https://doi.org/10.1145/3302424.3303963

[19] Lucia Pons, Vicent Selfa, Julio Sahuquillo, Salvador Petit, and Julio Pons. 2018. Improving System Turnaround Time with Intel CAT by Identifying LLC Critical Applications. In *Euro-Par 2018: Parallel Processing*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). Springer International Publishing, Cham, 603–615.

[20] L.-N. Pouchet and T. Yuki. [n. d.]. *Polybench 4.1.* https://sourceforge.net/projects/polybench/.

[21] M. K. Qureshi and Y. N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 423–432. https://doi.org/10.1109/MICRO.2006.49

[22] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. GÃşmez. 2017. Application Clustering Policies to Address System Fairness with IntelâĂŹs Cache Allocation Technology. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 194–205. https://doi.org/10.1109/PACT.2017.19

[23] Vivek Seshadri, Samihan Yedkar, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2015. Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks. *ACM Trans. Archit. Code Optim.* 11, 4, Article 51 (Jan. 2015), 22 pages. https://doi.org/10.1145/2677956

[24] SPEC. [n. d.]. *SPEC CPU Benchmarks.* https://www.spec.org/cpu2017/.

[25] Aswinkumar Sridharan, Biswabandan Panda, and Andre Seznec. 2017. Band-Pass Prefetching: An Effective Prefetch Management Mechanism Using Prefetch-Fraction Metric in Multi-Core Systems. *ACM Trans. Archit. Code Optim.* 14, 2, Article 19 (June 2017), 27 pages. https://doi.org/10.1145/3090635

[26] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*. 63–74.

[27] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. 2015. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 62–75. https://doi.org/10.1145/2830772.2830803

[28] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. 2007. Managing Shared L2 Caches on Multicore Systems in Software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*.

[29] X. Wang, S. Chen, J. Setter, and J. F. MartÃŋnez. 2017. SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 121–132.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. 1995. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings 22nd Annual International Symposium on Computer Architecture*. 24–36. https://doi.org/10.1109/ISCA.1995.524546

[31] C. Wu, A. Jaleel, M. Martonosi, S. C. Steely, and J. Emer. 2011. PACMan: Prefetch-Aware Cache Management for high performance caching. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 442–453.

[32] Carole-Jean Wu and Margaret Martonosi. 2011. Characterization and Dynamic Mitigation of Intra-application Cache Interference. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '11)*. 2–11.

[33] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. 2018. DCAPS: Dynamic Cache Allocation with Partial Sharing. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Article 13, 15 pages.

[34] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. 2018. dCat: Dynamic Cache Management for Efficient, Performance-sensitive Infrastructure-as-a-service. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Article 14, 13 pages.

[35] Y. Ye, R. West, Z. Cheng, and Y. Li. 2014. COLORIS: A dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 381–392. https://doi.org/10.1145/2628071.2628104

[36] Haishan Zhu and Mattan Erez. 2016. Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. 33–47.

[37] X. Zhuang and H. . S. Lee. 2003. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *2003 International Conference on Parallel Processing, 2003. Proceedings.* 286–293. https://doi.org/10.1109/ICPP.2003.1240591

[38] X. Zhuang and H. S. Lee. 2007. Reducing Cache Pollution via Dynamic Data Prefetch Filtering. *IEEE Trans. Comput.* 56, 1 (Jan 2007), 18–31. https://doi.org/10.1109/TC.2007.250620