Combining Prefetch Control and Cache Partitioning to Improve Multicore Performance

Conference Paper · May 2019

DOI: 10.1109/IPDPS.2019.00103

3 authors, including:

Junjie Shen
University of California, Irvine
24 PUBLICATIONS **117** CITATIONS

SEE PROFILE

Combining Prefetch Control and Cache Partitioning to Improve Multicore Performance

Gongjin Sun Junjie Shen Alex Veidenbaum
University of California, Irvine
{gongjins, junjies1}@uci.edu, alexv@ics.uci.edu

***Abstract*—Modern commercial multi-core processors are equipped with multiple hardware prefetchers on each core. The prefetchers can significantly improve application performance. However, shared resources, such as last-level cache (LLC) and off-chip memory bandwidth and controller, can lead to prefetch interference. Multiple techniques have been proposed to reduce such interference and improve the performance isolation across cores, such as coordinated control among prefetchers and cache partitioning (CP). Each of them has its advantages and disadvantages. This paper proposes combing these two techniques in a coordinated way. Prefetchers and the LLC are treated as separate resources and a multi-resource management mechanism is proposed to control prefetching and cache partitioning. This control mechanism is implemented as a Linux kernel module and can be applied to a wide variety of prefetch architectures. An implementation on Intel Xeon E5 v4 processor shows that combining LLC partitioning and prefetch throttling provides a significant improvement in performance and fairness.**

I. Introduction

Sharing of the LLC and of the off-chip memory bandwidth can lead to destructive inter-core interference as more cores are deployed in modern commercial multicore processors. This interference may be further exacerbated by prefetch traffic from multiple hardware prefetchers deployed in each core. For example, a modern Intel Xeon server processor [1] has four hardware data prefetchers per core, two at the L1 cache and two at the L2 cache. These prefetchers may significantly improve a cores performance by hiding the access latency. At the same time, they may increase the LLC and memory traffic and contention. This, in turn, may reduce throughput and fairness, as when a process consumes more than its fair share of the LLC space or memory bandwidth because of prefetching. Performance isolation and fairness across processes/cores are still important problems and this work proposes to address them via a combination of cache partitioning and prefetch throttling. It targets Intel Server processors, but can be applied to any processor with similar capabilities.

Extensive prior work [2]–[11] proposed reducing prefetcher-caused inter-core interference by controlling prefetch aggressiveness. These included hardware- and software-based schemes. The former are mostly microarchitectural techniques requiring additional hardware support. Key metrics used in these techniques, e.g. *prefetch accuracy*, cannot be collected for software use on today's processors. The latter work on existing processors and require no hardware modifications, e.g. [6] designed for IBM POWER7 processor [12].

Cache Partitioning (CP) has also been researched intensively. Early work [13], [14] proposed theory and michroarchitectural techniques. Recently, Intel server processors incorporated software controllable, way-based cache partitioning of the LLC called Cache Allocation Technology (CAT) [15]. Recent work [16]–[18] utilized CAT for performance isolation. However, it did not consider prefetching directly, which left potential for further improvement.

Both prefetch throttling and partitioning can be used to improve performance isolation on multicore processors. This raises the following research questions: 1) What are their respective pros and cons? 2) Is there a performance difference between them? 3) Is it possible to combine them for additional improvement?

This paper aims to answer the above questions. It develops CMM, a multi-resource management framework for combining prefetch throttling and cache partitioning. CMM is a low-overhead, software-based framework running on Intel multicore processors. Various prefetch throttling and cache partitioning mechanisms can be implemented and combined using the framework. It is used to compare the two techniques and to study their interaction and to explore the design space. The framework is implemented as a Linux kernel module and monitors system behavior via hardware Performance Monitoring Unit (PMU). A integrated throttling/partitioning mechanism is proposed based on this implementation.It targets multi-programmed workloads and is transparent to application software. It is evaluated on an Intel processor, but is easily portable to any processor with similar capabilities.

II. Background and Motivation

Intel Prefetchers. Today's Intel processor has four data prefetchers per physical core. The L1 data cache has the IP (stride) and the next-line prefetchers and the private L2 cache has the stream and the adjacent line prefetchers [15]. By default all prefetchers in a processor are turned on to improve application performance. Intel provides a mechanism to turn on/off each of the four prefetchers independently by programming a special MSR in software. Additionally, Intel exposes several hardware performance counters (PMU events) that are related to prefetch, e.g. L2 prefetch misses that count how many prefetch requests arrive at LLC. These event counts can be used for dynamic prefetch control (Pref_Ctrl).

Demand requests of a core may trigger L1 prefetchers to issue requests. Both demand and prefetch requests first check the local L1 cache and only on a miss continue to the next cache level (L2). Requests arriving at L2 will trigger L2's prefetchers to issue new prefetch requests, which are checked in L2 before issuing to the LLC.

Cache Partitioning. Intel CAT allows users to create **shared** and **overlapping** way-based partitions on a commercial processor. Such partitioning is much more flexible and can be used on larger multi-core systems with a low associativity LLC [19]. Prior work [17] used CAT to improve fairness of multi-programmed workloads.

A. Prefetch-related Terms

Many similar terms have been used to describe prefetching in prior work, such as "prefetch sensitive/effective/aggressive/useful/friendly." This section defines the terms used in this paper to avoid confusion.

A **demand intensive**[1] program has a large working set and generates many demand requests. A demand intensive program *may* or *may not* generate a large number of prefetch requests. This depends on whether its access patterns will trigger a prefetcher. A **prefetch aggressive** program has a high ratio of prefetch to demand requests. A prefetch aggressive program brings many prefetched lines into the cache hierarchy and may interfere with other programs or itself. A program has a high *prefetch accuracy* if most of prefetched data are used for a given prefetcher[2] and is called **prefetch useful**. As in prior work [20], **prefetch accuracy** is defined as percentage of prefetched data used.

Memory bandwidth required by a program is an indication of demand intensity. Fig. 1 shows the measured average memory bandwidth (BW) of SPEC CPU2006 benchmarks (on an Intel E5-2620 v4 processor and for high BW benchmarks) with and without prefetching. The increase from prefetching is shown on top part of each bar. Benchmarks, such as 437.leslie3d, 459.GemsFDTD and 410.bwaves, use approximately 4GB/sec **demand** BW (the blue part in the bar). Thus they are demand intensive compared with other ones. Their BW use increases by more than 80% with prefetching. Therefore, they are also prefetch aggressive.

Prefetch aggressive AND useful programs usually benefit significantly from prefetching, and are called **prefetch friendly**. A demand intensive but not prefetch aggressive (i.e. generating few prefetch requests) program can also be prefetch useful. However, performance improvement brought by its prefetching is very limited. Finally, a non demand intensive program usually will not cause prefetch-caused interference to others. [1] [2]

Today's processors have no PMU counters to directly measure prefetch accuracy.This work uses its approximation to identify prefetch friendly applications (See Sec. III-B1).

[1]The term "memory intensive" is not used because it may not reflect the real working set of a program in the presence of prefetching

[2]The same program may show a low prefetch accuracy for a different prefetcher

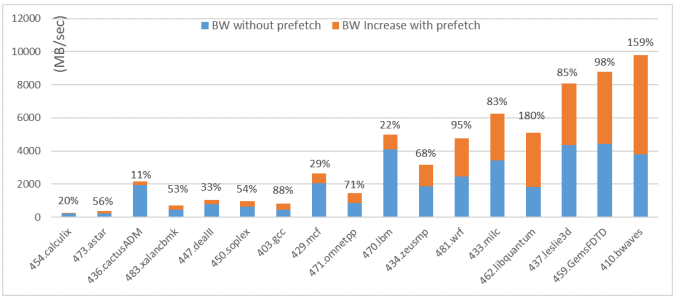

Fig. 1: Memory Bandwidth Consumption of Part of SPEC CPU2006 benchmarks

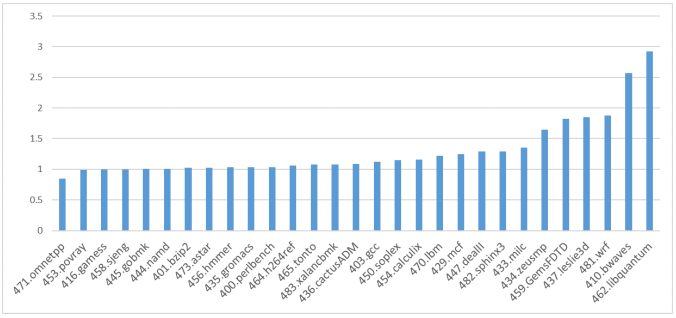

Fig. 2: IPC Speedup from prefetching for SPEC CPU2006 benchmarks

B. Pros and Cons of Prefetch Throttling and Cache Partitioning

Prefetch throttling may improve system performance if applied periodically to select a different prefetcher configuration on each core to reduce interference. However, this may lead to performance loss for applications on throttled cores. These are usually prefetch friendly and benefit significantly from prefetching. Thus they can yield the LLC space and memory bandwidth to other applications and still have reasonable performance. Thus, there is a trade-off between degrading performance of such applications and reducing their interference.

Exclusive CP creates a physical LLC isolation among cores and hence prevents inter-core interference (regardless of prefetchers). However, when multiple cores share a partition, interference is created within a partition. LLC partitioning does not necessarily reduce prefetch traffic arriving at the LLC and even memory. This is because prefetchers have little feedback from the LLC to indicate throttling may be beneficial, let alone coordinating across cores. Thus, a prefetch friendly but aggressive application can generate a large number of prefetch requests to memory. Memory BW contention may also be sigfnificant if prefetch aggressive programs occupy the same partition.

C. Coordinating Prefetching and Cache Partitioning

Let us make two observations vis a vis performance based on SPEC CPU2006 benchmarks.

Fig. 2 shows the IPC speedup from prefetching for SPEC CPU2006 [21] benchmarks. Several benchmarks, such as 462.libquantum, 410.bwaves, 481.wrf, 459.GemsFDTD,etc

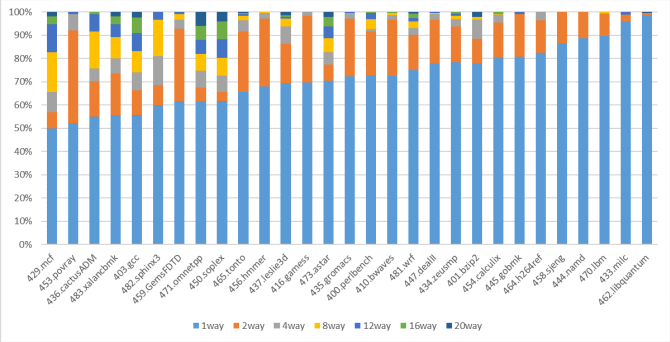

Fig. 3: IPC distribution across LLC ways (all prefetchers on).

have 50+% improvement from prefetchers (are prefetch friendly).

Fig. 3 shows each benchmark's performance as a function of the number of LLC ways with prefetching enabled. Many benchmarks,especially prefetch aggressive and friendly, need no more than 2 ways to achieve 90% of their highest performance. [17] had a similar observation. This observation, that *applications whose high performance is mainly from prefetching may not need many cache ways to achieve its highest performance*, leads us to rethink how to utilize both prefetching and CP in a coordinated way. Prefetchers and LLC can viewed as two different resources. An application's performance can be from either of them or both. Yielding one resource and getting another makes it possible to improve overall system performance and fairness

III. Coordinated Multi-resource Management (CMM)

Our overall goal is to reduce prefetcher-caused inter-core interference. This requires identification of prefetch aggressive (Pref_Agg) applications. One can then throttle their resource usage by applying prefetcher control or CP or both **periodically** to maximize system performance and fairness. Targeting this approach, the CMM framework is designed as a decoupled structure: a Front- and a Back-end. The front-end is responsible for detection and the back-end for resource allocation. This flexible design allows either front- or back-end to be modified independently as needed without affecting the other.

CMM performs resource allocation periodically using sampling. Fig. 4 shows a high-level diagram. The execution of a process is divided into a sequence of "Execution Epochs", each followed by a "Profiling Epoch." The latter consists of multiple "Sampling Intervals". At the end of each execution epoch, the CMM's front-end collects the necessary runtime statistics and identifies the prefetch aggressive applications. Then the back-end profiles an application by trying different resource configurations and chooses a good candidate for next execution epoch (see more detail in Sec. III-B1)

A. Pref_Agg Application Detection

Assume N applications are running on N different cores concurrently. Intel's PMU provides the following relevant events per logical cpu:

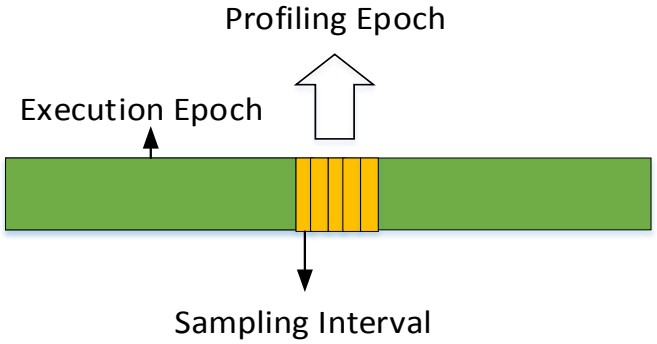

Fig. 4: The Execution and Sampling Mechanism

TABLE I: Multiple Metrics

No.	Metric	Definition	Description
M-1	L2-LLC-traffic	L2_pref_miss + L2_dm_miss	The sum of both demand and prefetch requests between L2 and LLC
M-2	L2_pref_miss_ frac	L2_pref_miss /L2-LLC-traffic	The fraction of the prefetch requests
M-3	L2_pref_miss_ traffic_rate (L2_PTR)	L2_pref_miss per second	L2 prefetch requests arriving at LLC per sec,
M-4	pref_gen_ablity (PGA)	L2_pref_req /L2_dm_req	Measures the ability of an application to generate L2 prefetches
M-5	L2_pref_miss_ rate	L2_pref_miss /L2_pref_req	The fraction of prefetches missing L2.
M-6	L2_PPM	L2_pref_req /L2_dm_miss	Prefetches issued per demand miss
M-7	LLC-Mem-pref-traffic	BW - L3_load _miss*64	An approximation of LLC to memory prefetch traffic

- *L2_pref_req*
  Number of prefetch requests generated by the adjacent prefetcher and streamer prefetcher in an L2 cache.
- *L2_pref_miss*
  Number of L2_prefetch_requests that miss the local L2 cache. Only these missed requests will arrive at LLC.
- *L2_dm_req*
  Number of demand requests that arrive at L2 cache
- *L2_dm_miss*
  Number of L2 demand requests that miss L2 cache.
- *L3_load_miss*
  Number of load micro-ops that miss LLC and issue to memory.

Specific metrics derived from these events are listed in Table I.

These metrics are used for identifying Pref_Agg cores. [8] used M-6 to classify cores into two categories: aggressive and meek, and applied a coarse-grained (5 configurations) group-level prefetch throttling. Their throttling applies to *all* cores and was designed for a totally different prefetcher hierarchy (the per-core LLC prefetcher). Using this metric on the Intel L2 cache side cannot accurately identify the Pref_Agg cores.

This work uses a different detection mechanism for Pref_Agg cores. First, M-4 is used to detect cores whose access patterns cause prefetchers to generate prefetch requests. A core whose M-4 is above the average value across all cores is viewed as potentially prefetch aggressive. Second, M-5 is

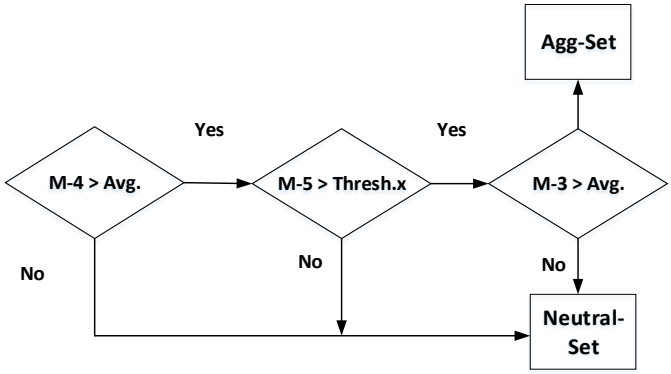

Fig. 5: Detection Mechanism used by CMM

used to filter out cores with high **prefetch locality**, i.e. whose most prefetch requests hit the L2 cache. A threshold is used (say 70%) will be considered. Third, M-3 is used to evaluate the prefetch pressure on the LLC. It measures the bandwidth pressure caused by prefetch (instead of an absolute prefetch number) between L2 and LLC. Figure 5 shows a high level diagram of this mechanism.

One can also use metric M-7 to identify cores that issue a large number of prefetch requests to memory. However, we observed that cores with high L2_pref_miss_traffic_rate usually also have a high M-7. The identified Agg-set by Figure 5 basically stays the same as when using M-7.

B. Back_end Design

Once the agg_set is determined by the front_end, the back_end of CMM will conduct resource allocation across cores to improve performance isolation. Let's us consider the following three approaches.

- **Prefetch Throttling**: Throttle the prefetchers of one or more cores in agg_set to reduce their interference
- **Cache Partitioning**: Place cores in agg_set into a relatively small partition (called agg_partition) and let the cores in neutral_set share the whole cache
- **Coordinated-throttling**: First perform Cache Partitoning, then throttle the prefetchers within agg_partition

*1) **Prefetch Throttling (PT)**:* This approach throttles prefetchers of cores in the agg_set. All four prefetchers per core are either on or off. For example, if agg_set contains two cores, there are four prefetch combinations: {on, on}, {on, off}, ..., {off, off}. PT applies them for a very short sampling interval each and chooses the "best" one. "Best" refers to the combination produces the *lowest* system average normalized turnaround time (ANTT), which is the reciprocal of harmonic speedup (HS) [22]. The calculation of HS requires the knowledge of an application's IPC when running alone.Though prior work [23] tried to estimate the running-alone IPC in multicore processors, it requires additional hardware support.

This work instead uses a proxy metric, the harmonic mean of all cores' IPC (called **hm_ipc**), to estimate the performance and fairness of a prefetch combination. The evaluation (see Sec. V) shows the system fairness and performance are indeed improved over the baseline (no prefetch control) using this metric. This "best" prefetch combination is applied to next execution epoch.

The first sampling interval is **always** {on, on} because some cores' prefetchers could have been turned off in the last execution epoch if their L2_PTR or PGA were 0. So a short sampling period with all cores' prefetchers on is needed to collect the PMU statistics before trying other sampling intervals.

As mentioned above, disabling prefetching for prefetch friendly applications will hurt their performance even though doing so can reduce the interference to others.(Actually, not all cores in agg_set are prefetch friendly. This is discussed in more detail below). The experiments show that some applications (410.bwaves, 462.libquantum) could get a performance degradation of more than 50% if their prefetching is turned off most of the time.

A large agg_set (say 10 cores) makes the search space of all available prefetch settings large. With the four per-core prefetchers viewed as an single entity and only "turned on/off", there are still 2^{10} settings to search. In this case, a practical and **scalable**[1] solution is to use group-level throttling. The cores in agg_set are clustered into a limited number of groups (say 3) and all prefetchers in each group are viewed a single entity. The limited group-level settings ($2^3 = 8$) are then tried. Prior work [8] only uses a 2-group clustering which is coarse grained. This paper uses the **K-Means Algorithm** [24] to cluster the cores in the agg_set into limited groups by their L2_PTRs (M-3), which evaluates the prefetch pressure to LLC brought by each core. The cores with similar L2_PTRs are placed into the same group.

Useful and Useless Prefetching: Cores in agg_set can be further categorized into two classes: prefetch unfriendly and prefetch friendly. The former usually prefetches more useless data because of its inaccurate prediction and thus benefits little from prefetching (or even is hurt) while polluting the cache severely. The latter needs to be treated with care: turning off their prefetching will hurt their performance a lot, but keeping their prefetching on will probably hurt other programs' performance due to interference. Prior work [25] (Paragraph 5 in the introduction) shows that maximum weighted speedup can be achieved when a core with very useful prefetching yields some bandwidth to cores with less useful prefetching. With only prefetch throttling, this might be a good solution. However, as shown in the next section, it is possible to avoid reducing performance of applications with aggressive AND useful prefetching.

Detect prefetch friendliness: This paper uses an indirect method to identify prefetch friendly applications in lieu of prefetch accuracy. Prefetchers of cores in agg_set are turned off in the second sampling interval. A per-core IPC speedup from prefetching is calculated (from data of the first sampling interval with prefetchers on). A core with a predefined speedup over a threshold (say 50%) is considered prefetch friendly.

[1]By "scalable", we mean it can be applied to more cores

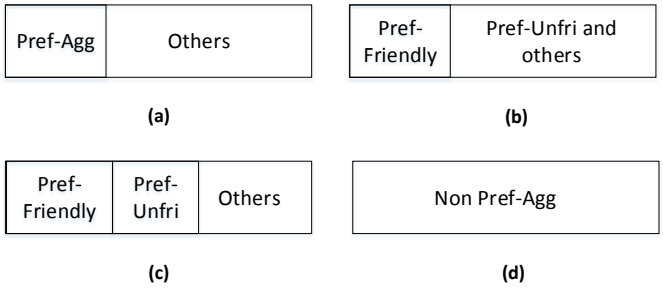

Fig. 6: Partition Options

*2) **Cache Partitioning (CP)**:* There are several proposed CP implementations. [17] groups cores into several clusters and each cluster occupies a separate partition. However, it did not consider the effect of prefetching (see Sec. V-B for more discussion).

This work studies two CP plans: 1) place the agg_set into a small partition, and 2) split agg_set into two subsets: prefetch friendly and unfriendly, then place each subset into a separate partition. They are compared with the best known algorithm, Dunn in [17], in Sec. V-B. Note that CP just needs the first two sampling intervals for the detection of prefetch usefulness.

While CP isolates prefetch interference from cores in the agg_set, the local interference within the partition still exists. Also the number of prefetch requests leaving the LLC may not be reduced and could still cause memory bandwidth contention with other programs. Additionally, if agg_set contains many prefetch-unfriendly cores, it is beneficial to turn off their prefetching.

*3) **Coordinated Throttling**:* A coordinated solution combines prefetch throttling and CP. First, all prefetch-friendly cores are placed into a small shared partition while keeping their prefetchers enabled. They are typically not LLC sensitive and a small partition results in a pretty high performance. Meanwhile, the remaining cores share the whole cache capacity. Second, prefetch-unfriendly cores are identified and a group-level throttling is applied as discussed above. Actually, other implementations are possible. Figure 6 shows the available options.

- (a) puts agg_set into a small partition. Agg_set contains both prefetch-friendly and/or unfriendly cores.
- (b) only puts prefetch-friendly cores into a partition.
- (c) allows prefetch-unfriendly cores to stay in a separate small partition.
- (d) indicates a special scenario: agg_set is empty. This usually happens when there are no prefetch aggressive applications in the multi-programmed workload or current program phase. If so, it will be meaningless to throttle the prefetchers. In this case, CMM will use the "Dunn" algorithm in prior work [17] for cache partitioning.

This paper evaluates the first three options. Note that only the prefetchers of prefetch-unfriendly cores can be throttled. If no such cores are found, only CP will be applied.

Such coordinated throttling trades off the use of two resource types across cores and tries to maximize each core's performance and the whole system performance and fairness as well. It should be noted that prefetch-useless cores could lose either one of them (prefetching) or both depending on the implementation.

***Partition Size**:* Per Fig. 3, most prefetch aggressive applications just need no more than 2 ways to achieve 90% of their highest performance. It was experimentally determined that partition size of 1.5 times the size of the agg_set works well.

IV. Experimental Methodology

A. Processor used

All measurements were performed on an Intel Broadwell-EP Xeon server processor E5-2620 V4 [26]. It contains 8 physical cores and 16 hardware threads and each physical core contains private 32KB L1 I/D-Cache and 256 KB L2 cache. All cores share a 20 MB L3 cache. This processor has a 2.1 GHz base frequency and supports 68.3 GB/s Maximum Memory Bandwidth. The memory is DDR-2400 and OS is Ubuntu 16.04 with the kernel 4.4.24. We disable the "Turbo boost" to keep each core frequency the same. Performance counters used in this work exists on all almost all Intel mainstream processors from the 2rd generation Intel Core to the latest 8th one [15].

Baseline system The baseline used in the evaluation has all four prefetchers in each core enabled and **no prefetch control and no cache partitioning** are applied. Each workload was executed three times and the median value was used in the results.

Implementation details The CMM is implemented as a loadable kernel module. We implement our own PMI (Performance Monitoring Interrupt) and IPI (Inter-Processor Interrupt) handlers to collect necessary PMU statistics and execute the algorithms of prefetch throttling and/or cache partitioning. In order to estimate the module-related **overhead**, we used two counters to collect cycles: PMU and TSC (Timestamp Counter). The former does not count handler-related cycles, and the latter does. Then we compare them and find the overhead is less than 0.1%.

The decoupled design allows the algorithm(s) in either the front- or the back-end to be changed individually. This work provides a flexible and open design framework for exploration. For example, if Intel or other processors expose more PMU events in the future, Table I and related detection algorithm can be easily explored and updated to achieve better results.

B. Mixed Workloads

SPEC CPU2006 benchmarks [21] and an micro-benchmark (see below) are used to create multicore mixed workloads. Each N-core workload contains N benchmarks. Each benchmark was compiled with Intel ICC 17.0.0 or Intel Ifort 17.0.0 with the -O3 option. We classify the benchmarks into different categories: (1) perfetch aggressive (if their demand BW is more than 1500 MB/s AND BW increase brought by prefetching is more than 50% in Fig. 1) or not; (2) prefetch

friendly[2] (if their ipc speedup is more than 30% in Fig. 2) or not; (3) LLC sensitive (if they need at least 8 ways to achieve 80% of its highest performance in Fig. 3) or not. LLC sensitive applications are more easily affected by the interference in LLC.

Since there is no SPEC benchmark whose performance is *significantly* reduced by prefetch (471.omnetpp is reduced only slightly), we manually created a micro-benchmark called "Rand_Access" as a typical prefetch unfriendly application. It is strongly prefetch aggressive and conducts random access in a large memory region. Its performance slowdown with prefetching over no-prefetching is 25% when running alone because its access pattern is irregular.

It is not feasible to evaluate our mechanism with all possible combinations of 8 benchmarks. Instead, typical workload types are created by mixing benchmarks from different classes. The number in parenthesis below indicates how many benchmarks of a given type are in a workload.

- **Pref_Fri**: Prefetch-friendly benchmarks (4) + Non Pref_Agg benchmarks (4)
- **Pref_Agg**: Prefetch-friendly benchmarks (2) + unfriendly ones (2) + Non Pref_Agg ones (4)
- **Pref_Unfri**: Prefetch-unfriendly benchmarks (4) + Non Pref_Agg ones (4)
- **Pref_No_Agg**: Non Pref_Agg benchmarks (8)[3]

Each category contains 10 workloads and the benchmarks are chosen randomly from their respective class (prefetch friendly/unfriendly). Note that four non Pref_Agg benchmarks include at least two LLC-sensitive benchmarks. Each workload runs for 2.5 minutes for both baseline and CMM-based mechanisms. If an application finishes earlier, it restarts from the beginning.

The length of an *execution epoch* and *sampling period* are 5 billion and 100 million core cycles respectively, which are determined to be a good set up experimentally. In fact, [3] shows a 50:1 of "execution epoch vs sampling period" is a reasonable choice. Other lengths (2 billion, 50 million) or (1 billion, 40 million) show similar results. One reason could be that Intel has a different prefetcher design from IBM's POWER.

C. Evaluation Metrics

Prior work [5], [22], [27]–[30] explored the "fairness" and "performance" on multi-core system from various perspectives. Like recent architecture work ([8], [11], [16]–[18]), we use harmonic speedup (HS) and normalized weighted speedup over baseline (WS) [11], [18] to evaluate the system performance and fairness of the above multiprogrammed workloads. They are defined as follows:

$$HS = \frac{N}{\sum_{i=1}^{N} \frac{IPC_i^{alone}}{IPC_i^{together}}}$$

$$WS = \sum_{i=1}^{N} \frac{IPC_i^{Algorithm-x}}{IPC_i^{baseline}}$$

where IPC_i is the IPC of core i and N is the number of cores. IPC_i^{alone} represents $program_i$'s IPC when it runs alone on a single core of a multicore processor. And $IPC_i^{together}$ represents $program_i$'s IPC when it runs together with other programs on different cores in a multicore processor. HS considers both **system performance and fairness**. According to [22], $1/HS$ is equal to the average turnaround time, which is a key performance metric in a multicore system. $IPC_i^{Algorithm-x}$ and $IPC_i^{baseline}$ represent $program_i$'s IPC when Algorithm-x and baseline are applied, respectively. Algorithm-x could be PT, CP or CMM.

V. Evaluation

The proposed mechanisms are evaluated and compared in this section. In all graphs, the first 10 workloads are Pref_Fri, the second 10 are Pref_Agg, followed by 10 Pref_Unfri and 10 Pref_Non_Agg.

A. Prefetch Throttling (PT)

Fig. 7 shows the normalized HS and WS vs baseline for PT. Most workloads gain from PT. Four grey bars show the average value for each workload category. Pref_Unfri and Pref_Agg have the highest and second-highest performance improvement, respectively. Pref_Fri achieve a relatively lower improvement and Pref_No_Agg sees no improvement, as Pref_Unfri and Pref_Agg contain prefetch unfriendly applications. Turning their prefetchers improves both their own and other programs' performance. However, turning off the prefetchers in Pref_Fri reduces the interference to others at the cost of hurting their own performance. As a result, the overall performance improvement is not very significant.

Recall that some prefetch-friendly applications lose performance when their prefetching is disabled. Fig. 8 shows this. The per-application IPC speedup with PT over baseline is calculated. The lowest speedup is called the worst-case speedup in a workload. The figure shows that at least one application's performance is significantly reduced for $\tilde{8}0\%$ of the workloads.

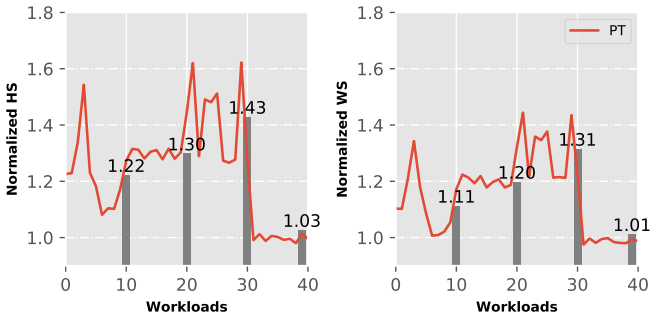

Fig. 7: Normalized HS and WS

[2]In this paper, a "prefetch (un)friendly" application is also prefetch (un)aggressive unless otherwise specified

[3]In some program phases, the agg_set may not be empty

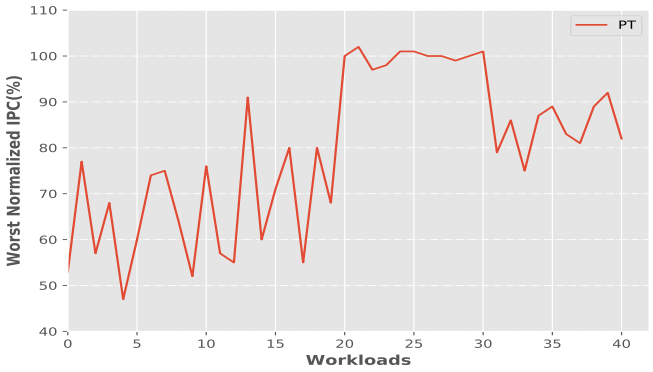

Fig. 8: Lowest Normalized IPC in each workload

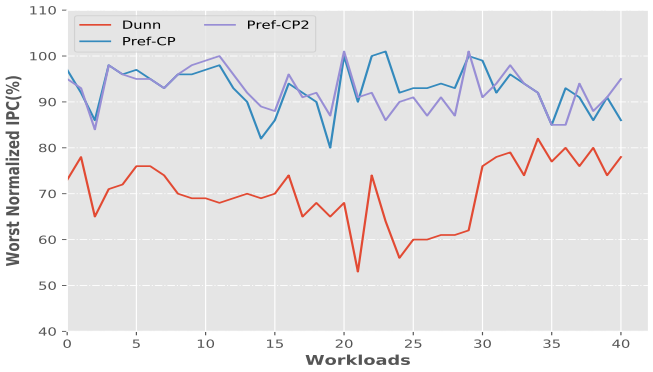

Fig. 10: Lowest Normalized IPC in each workload

B. Cache Partitioning (CP)

Fig. 9 shows the normalized HS and WS to baseline of Dunn, Pref-CP and Pref-CP2. Pref-CP puts agg_set into a small partition and Pref-CP splits agg_set into two subsets (prefetch friendly or not) to assign each a separate partition. It shows Pref-CP(2) significantly improves the system performance over Dunn in [17]. Dunn uses the PMU event "STALLS_L2_PENDING" to cluster cores into several different groups based on similarity of each core's measurement. Each group is then assigned a certain number of LLC ways as its partition. A group with higher average "STALLS_L2_PENDING" gets more ways. The partitions partially overlap with each other, in fact they are nested.

Dunn did not consider the effect of prefetching. For example, A prefetch aggressive program that has similar stalls_l2_pending to other programs will be clustered into the same partition. As a result, the prefetch interference happens within this partition. Fig. 10 shows Pref-CP(2) have a higher worst-case speedup than Dunn. It is worth noting that Pref-CP2 has higher performance than Pref-CP for workload category Pref_Agg and Pref_Unfri. One reason is that some prefetch unfriendly applications (like 471.omnetpp) still need many LLC ways to get more performance as shown in Fig. 3. So Pref-CP2, which puts the unfriendly ones into a small partition, can hurt these applications' performance.

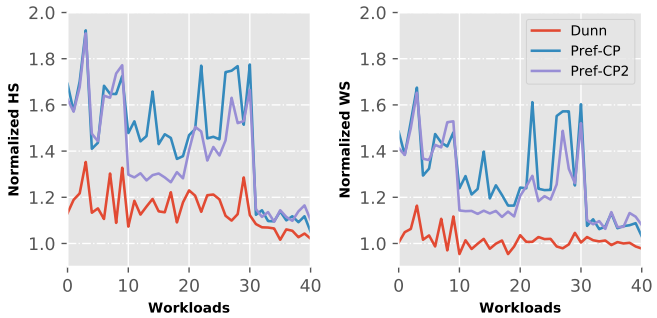

Fig. 9: Normalized HS and WS

C. Coordinated Multi-resource Management (CMM)

Fig. 11 shows the normalized HS and WS to baseline of three different coordinatedly throttling mechanisms: CMM-a, CMM-b and CMM-c, which correspond the (a),(b) and (c) in Fig. 6. CMM-a puts agg_set into a small partition and apply PT for the prefetch unfriendly cores in the set. CMM-b puts only the prefetch friendly cores in agg_set into a small partition and applies PT for the unfriendly ones in the whole cache. CMM-c looks like Pref-CP2, but with PT applied.

Workloads Pref_Fri and Pref_No_Agg show the same performance. The reason is that the three mechanisms degenerate into CP-based throttling for these two workload categories because no prefetch unfriendly cores are throttled. Pref_Agg and Pref_Unfri categories have CMM-a and CMM-c show better performance than CMM-b. In CMM-b, prefetch unfriendly cores still cause demand interference to the cores that share the whole LLC with them even though throttling their prefetching can reduce the prefetch-caused interference.

Additionally, Fig. 12 shows CMM-a/b/c give all workloads a 80%+ worst-case speedup and most of them get 90%+. This indicates no individual application is hurt significantly by the mechanisms.

Finally, putting everything together, Fig. 13 shows the comparison among all 7 throttling mechanisms. Workload category Pref_Agg and Pref_Unfri benefit the most.

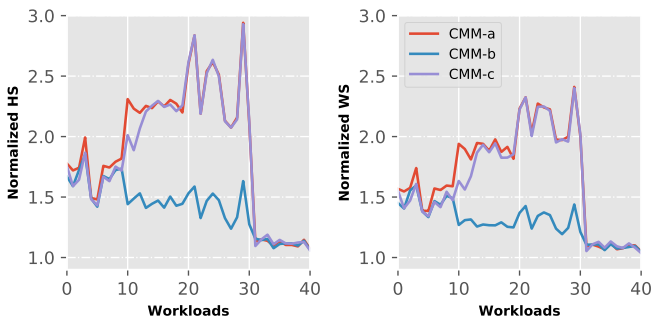

Fig. 11: Normalized HS and WS

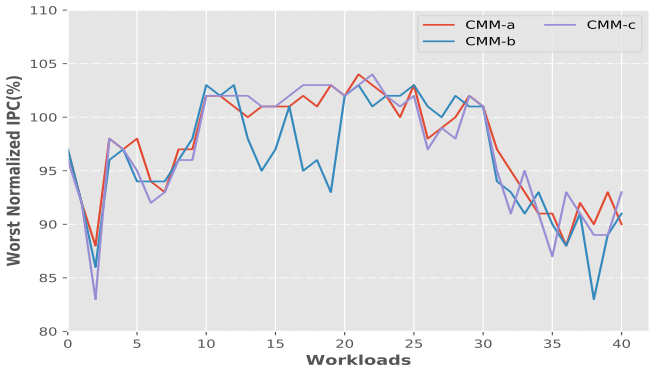

Fig. 12: Lowest Normalized IPC in each workload

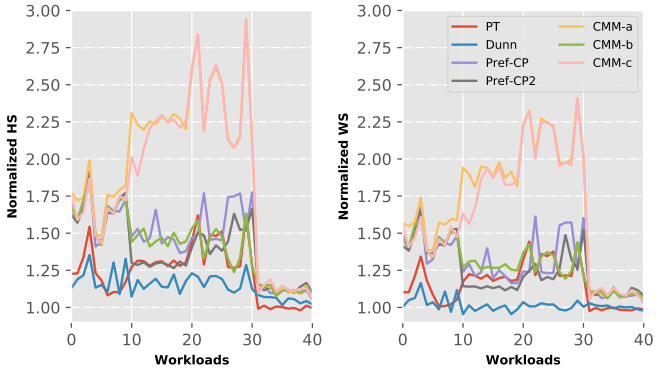

Fig. 13: Normalized HS and WS

D. The Effect of Memory Traffic

Fig. 14 shows the normalized BW to baseline of 7 throttling mechanisms. PT has the lowest bandwidth consumption because it frequently disables some cores' prefetching. By contrast, Dunn, Pref-CP and Pref-CP2 have the highest bandwidth occupancy. This meets our analysis in Sec. II: cache partitioning cannot reduce the prefetch traffic a lot even though they do improve performance isolation. The coordinated throttling mechanisms, CMM-a/b/c, get a good balance between bandwidth consumption and system performance.

E. The Effect of L2 stall cycles

The PMU event "STALLS_L2_PENDING" is used for counting the number of cycles during which the execution of an application is stalled due to L2 cache misses [15]. Prior work [17] shows there is a strong positive correlation between its count and an application's slowdown. This event's count is affected by the interference on shared resource such as LLC, memory bandwidth and controller, on-chip interconnects. As [17] pointed out, as the number of concurrently running applications grow, the stall cycles caused by interference will dominate in total stall cycles experienced by the cores. Therefore, by observing this event, we know how effectively our mechanisms improve the system performance isolation.

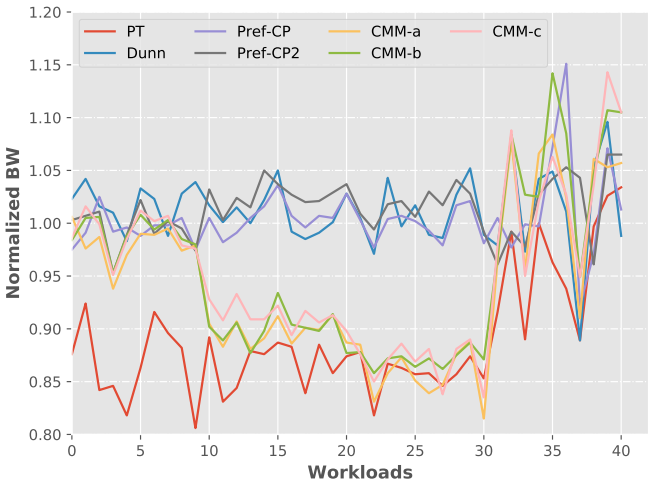

Fig. 14: Normalized Bandwidth Consumption for workloads

Fig. 15 shows the normalized "STALLS_L2_PENDING" for each workload (we sum per-core's number to get a single value for each workload). Again, CMM-a/c has the lowest number for most workloads (the lower the better).

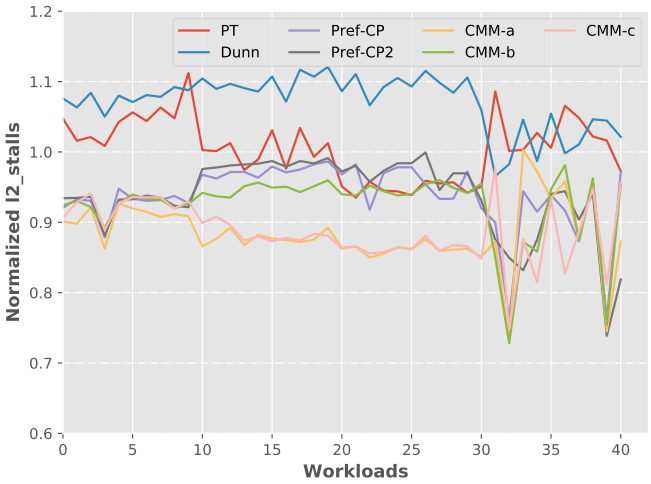

Fig. 15: Normalized L2 stalls per workload

VI. Related Work

A. Local/Global Prefetch Control

Wu et al. [2] proposes a dynamic control of preftching to mitigate **intra-application** prefetch interference on Intel architectures. Liao et al. [4] proposes a machine learning-based model to adjust the prefetch configuration for the individual application in Intel processors. Kang et al. [31] studies the effect of hardware prefetching in virtualized environments. Jimenez et al. [3] proposed an adaptive prefetch control to independently adjust each core's prefetch aggressiveness on IBM POWER7 architecture. Intel prefetch architecture does not provide a fine granularity control like the POWER7 architecture.

Ebrahimi et al. [5] proposes a Hierarchical prefetcher aggressiveness controller (HPAC) to throttle multiple prefetchers on multi-core processors. It dynamically identifies those applications that cause inter-core interference and throttle their prefetchers to reduce the interference. Albericio et al. [32] proposes an adaptive controller (ABS) to manage the aggressive prefetcher of each core on the multicore platform. Panda et al. [8] proposes a synergistic prefetcher aggressiveness controller (SPAC) to improve the system fair-speedup. Jimenez et al. [6] proposes a memory bandwidth based approach to throttle the prefetchers on IBM POWER7 architecture. This approach makes throttling decisions based on each core's bandwidth consumption and prefetch usefulness.

B. Other Prefetch-related research

Unlike the control of multiple prefetchers, the goals of [20], [33]–[38] are to design a single prefetcher. They are orthogonal to the work in this paper, which *does not* propose a new prefetcher architecture. Instead, the global control described in this paper can be applied to a variety of prefetchers. In addition, [25] studies the interaction of prefetching and BW partitioning. [39]–[41] propose various prefetch filters to reduce the prefetch requests that can cause inteference to other applications or be useless. [42]–[46] explores how to manage shared resources on multi-core platforms.

C. Cache Partitioning

Qureshi et al. [14] proposes a utility-based cache partitioning (UCP) mechanism to partition the shared LLC among multiple applications. A way-based partitioning algorithm uses the utility to allocate various amount of cache ways to each application. It should be noted that different partitions cannot overlap with each other. Cook et al. [16] evaluates the effect of cache partitioning on real Intel multi-core processors, including an optimal static LLC partitioning and a dynamic algorithm. They find that measurements on real machines provide different observations than past simulation-based work. Wang et al. [47] proposes a combined cache partition method (SWAP) to take into count both of set and way partitioning. This method can cooperatively manage cache sets and ways and provide many fine-grained partitions. Selfa et al. [17] proposes a clustering-based cache partitioning mechanism to improve the fairness of multi-core processors. The allocations are grouped into clusters according to their l2 cache stalls, and different clusters get different ways.

VII. Conclusions

This paper proposes CMM, a coordinated multi-resource management mechanism to reduce the prefetch caused inter-core interference on mainstream Intel multi-core processors. It does not require additional hardware support and manages the use of hardware prefetchers and LLC in a dynamic and coordinated manner by monitoring the applications' prefetching/cache behavior. Several CMM implementations were evaluated for diverse multiprogrammed workloads. The results show that using shared cache partitions to isolate applications with different prefetching behavior, combined with prefetch throttling, significantly improves system performance and fairness(performance isolation) for multiprogrammed workloads. The results are on Intel architecture but are generally applicable to other processors.

References

[1] Intel_E5, "Intel® xeon® processor e5-2620 v4." https://ark.intel.com/products/series/91287/Intel-Xeon-Processor-E5-v4-Family, 2016.

[2] C.-J. Wu and M. Martonosi, "Characterization and dynamic mitigation of intra-application cache interference," in *Proceedings of the IEEE Intl. Symposium on Performance Analysis of Systems and Software*, 2011.

[3] V. Jimenez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O'Connell, "Making data prefetch smarter: Adaptive prefetching on power7," in *Proceedings of the IEEE Intl. Conference on Parallel Architectures and Compilation Techniques*, 2012.

[4] S.-w. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, "Machine learning-based prefetch optimization for data center applications," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pp. 1–10, IEEE, 2009.

[5] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *in Proc. of the IEEE/ACM Intl. Symposium on Microarchitecture*, 2009.

[6] V. Jimenez, A. Buyuktosunoglu, P. Bose, F. P. O'Connell, F. Cazorla, and M. Valero, "Increasing multicore system efficiency through intelligent bandwidth shifting," in *In Proc. of the IEEE Symposium on High Performance Computer Architecture*, 2015.

[7] B. Panda and S. Balachandran, "Caffeine: A utility-driven prefetcher aggressiveness engine for multicores," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 3, p. 30, 2015.

[8] B. Panda, "Spac: A synergistic prefetcher aggressiveness controller for multi-core systems," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3740–3753, 2016.

[9] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pp. 7–17, IEEE, 2009.

[10] A. Sridharan, B. Panda, and A. Seznec, "Band-pass prefetching: an effective prefetch management mechanism using prefetch-fraction metric in multi-core systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, p. 19, 2017.

[11] M. Khan, M. A. Laurenzanoy, J. Marsy, E. Hagersten, and D. Black-Schaffer, "Arep: Adaptive resource efficient prefetching for maximizing multicore performance," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 367–378, IEEE, 2015.

[12] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, *et al.*, "Ibm power7 multicore server processor," *IBM Journal of Research and Development*, vol. 55, no. 3, pp. 1–1, 2011.

[13] H. Stone, J. Turek, and J. Wolf, "Optimal partitioning of cache memory," *Transactions on Computers*, vol. 41, no. 9, pp. 1054–1068, 1992.

[14] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pp. 423–432, IEEE, 2006.

[15] Intel_3B, *Intel® 64 and IA-32 Architectures Software Developer's Manual Vol. 3B, Order Number: 253669-059US*. June, 2016.

[16] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 308–319, ACM, 2013.

[17] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, "Application clustering policies to address system fairness with intel's cache allocation technology," in *Parallel Architectures and Compilation Techniques (PACT)*, pp. 194–205, IEEE, 2017.

[18] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," in *High Performance Computer Architecture, 2018 IEEE International Symposium on*, pp. 104–117, IEEE, 2018.

[19] Intel, "Intel® xeon® scalable processors." https://ark.intel.com/products/series/125191/Intel-Xeon-Scalable-Processors, 2017.

[20] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *In Proc. of the IEEE Symposium on High Performance Computer Architecture*, 2007.

[21] SPEC, *SPEC CPU 2006 benchmark Suite*. SPEC.org, 2006.

[22] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," in *IEEE Micro*, 2008.

[23] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 62–75, ACM, 2015.

[24] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[25] F. Liu and Y. Solihin, "Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors," in *In Proc. of the ACM Intl. Conference on Measurement and Modeling of Computer Systems*, 2011.

[26] Intel_E5, "Intel® xeon® processor e5-2620 v4." https://ark.intel.com/products/92986/Intel-Xeon-Processor-E5-2620-v4-20M-Cache-2_10-GHz, 2016.

[27] R. Gabor, S. Weiss, and A. Mendelson, "Fairness enforcement in switch-on event multithreading," in *ACM Trans. Architecture and Code Optimization*, 2007.

[28] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in smt processors," in *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, pp. 164–171, IEEE, 2001.

[29] H. Vandierendonck and A. Seznec, "Fairness metrics for multi-threaded processors," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 4–7, 2011.

[30] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-isa heterogeneous multi-cores," in *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pp. 177–187, IEEE, 2013.

[31] H. Kang and J. L. Wong, "To hardware prefetch or not to prefetch? a virtualized environment study and core binding approach," in *In Proc. of the ACM Conference on Architectural Support for Programming Languages and Systems*, 2013.

[32] J. Albericio, R. Gran, P. Ibánez, V. Viñals, and J. M. Llabería, "Abs: A low-cost adaptive controller for prefetching in a banked shared last-level cache," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 19, 2012.

[33] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "Ac/dc: An adaptive data cache prefetcher," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 135–145, IEEE Computer Society, 2004.

[34] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pp. 397–408, IEEE, 2006.

[35] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," in *Proceedings of the 23rd international conference on Supercomputing*, pp. 499–500, ACM, 2009.

[36] T. Kim, D. Zhao, and A. V. Veidenbaum, "Multiple stream tracker: a new hardware stride prefetcher," in *In Proc. of the ACM Intl.Conference on Computing Frontiers*, 2014.

[37] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. fei Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *In Proc. of the IEEE/ACM Symposium on High Performance Computer Architecture*, 2014.

[38] P. Michaud, "Best-offset hardware prefetching," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 469–480, IEEE, 2016.

[39] X. Zhuang and H.-H. S. Lee, "Reducing cache pollution via dynamic data prefetch filtering," in *IEEE Transactions on Computers*, 2007.

[40] J. Yu and P. Liu, "A thread-aware adaptive data prefetcher," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pp. 278–285, IEEE, 2014.

[41] A. Gendler, A. Mendelson, and Y. Birk, "A pab-based multi-prefetcher mechanism," *International Journal of Parallel Programming*, vol. 34, no. 2, pp. 171–188, 2006.

[42] A. Sharifi, S. Srikantaiah, M. Kandemir, and M. J. Irwin, "Courteous cache sharing: Being nice to others in capacity management," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pp. 678–687, IEEE, 2012.

[43] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pp. 318–329, IEEE, 2008.

[44] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared resource management for multi-core systems," in *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 141–152, ACM, 2011.

[45] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "Qos policies and architecture for cache/memory in cmp platforms," in *In Proc. of the ACM Intl. Conference on Measurement and Modeling of Computer Systems*, 2007.

[46] A. Fedorova, S. Blagodurov, and S. Zhuravlev, "Managing contention for shared resources on multicore processors," *Communications of the ACM*, vol. 53, no. 2, pp. 49–57, 2010.

[47] X. Wang, S. Chen, J. Setter, and J. F. Martínez, "Swap: Effective fine-grain management of shared last-level caches with minimum hardware support," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pp. 121–132, IEEE, 2017.