SPAC: A Synergistic Prefetcher Aggressiveness Controller for Multi-core Systems

Biswabandan Panda, *Student Member, IEEE*

Abstract—In multi-core systems, prefetch requests of one core interfere with the demand and prefetch requests of other cores at the shared resources, which causes prefetcher-caused interference. Prefetcher aggressiveness controllers play an important role in minimizing the prefetcher-caused interference. State-of-the-art controllers such as hierarchical prefetcher aggressiveness control (HPAC) select appropriate throttling levels that can lead to improvement in system performance. However, HPAC does not consider the interactions between the throttling decisions of multiple prefetchers, and loses opportunity to improve system performance further. For multi-core systems, state-of-the-art prefetcher aggressiveness controllers controls the aggressiveness based on prefetch metrics such as accuracy, bandwidth consumption and cache pollution. We propose a synergistic prefetcher aggressiveness controller (SPAC), which explores the interactions between the throttling decisions of prefetchers, and throttles the prefetchers based on the improvement in fair-speedup of multi-core systems.

Index Terms—Prefetching, Cache, DRAM, Prefetcher-aggressiveness-Controller, Fairness.

1 INTRODUCTION

Aggressive hardware prefetching improves system performance by fetching data into the cache before processor core demands for the same. However, aggressive prefetching causes inter-core interference at the shared resources, such as last-level-cache (LLC), miss-status holding-registers (MSHRs), DRAM controller, DRAM bus, and DRAM, where prefetch requests of one core interfere with the prefetch and demand requests of the other cores. In multi-core systems, this interference can cause significant degradation in system performance, and system fairness, leading to very low fair-speedup (FS) [1]. Note that FS is a metric that balances both throughput and fairness of a multi-core system. For an N-core system, FS is the harmonic mean of speedups and is defined as

$$FS = N / \sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{together}}, \tag{1}$$

where IPC_i^{alone} is the IPC of an application i running alone on an N-core system and $\text{IPC}_i^{together}$ is the IPC of application i running concurrently with other $N-1$ applications on an N-core system. *We choose FS as our performance metric as it does not overlook large slowdowns of those applications that contribute little to the overall system throughput.*

A prefetcher aggressiveness controller maximizes the system performance and fairness by minimizing the prefetcher-caused inter-core interference at the shared resources. After every fixed-size windows[1], each core uses its prefetcher aggressiveness controller that controls the aggressiveness by selecting an appropriate throttling level.

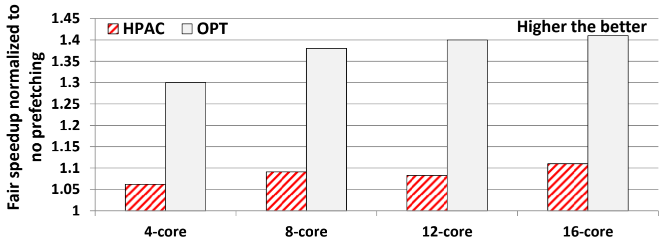


Figure 1: Performance improvement in terms of *fair-speedup (FS)* with HPAC and OPT.

A throttling level is a combination of two important knobs: (i) the number of prefetch requests issued at a given time (*prefetch-degree*), (ii) how far ahead of the demand access stream the prefetch requests are issued (*prefetch-distance*). The lowest/highest throttling level uses a combination of low/high prefetch-degree and low/high prefetch-distance. The job of an aggressiveness controller is to throttle up (↑)/down (↓) the throttling level, where up/down corresponds to the increase/decrease in the aggressiveness by one level.

Impact: Figure 1 shows the improvement in FS, achieved by the state-of-the-art hierarchical prefetcher aggressiveness controller (HPAC) [2], and the optimal prefetcher aggressiveness controller (OPT). OPT is an unrealistic controller, which we model by collecting fair-speedup *traces of all* the throttling decisions, and for each window, we simulate it by selecting the best throttling decision (based on the collected traces) for each core that leads to maximum FS. Compared to a system with no prefetching, for 100 4-core, 50 8-core, 25 12-core, and 10 16-core workloads, OPT improves the average performance by 30%, 38%, 40%, and 41%, respectively. On the other hand, HPAC improves the performance by only 6%, 7.7%, 7.3% and 9%, respectively, which shows a huge gap between HPAC and OPT.

• *Biswabandan Panda is with the INRIA, Rennes, France. E-mail: biswabandan.panda@inria.fr. A major part of this work was done while being at IIT Madras, India.*

1. We define a window in terms of # of demand LLC misses, which we explain in Section 6.

Our focus in this paper is to bridge the gap between HPAC and the OPT, and attempt to answer the following three questions: (i) What are the drawbacks of the existing controllers? (ii) Is it possible to advance the state-of-the-art in terms of performance and fairness? (iii) How to design a simple and effective controller that can control the aggressiveness of prefetchers in a manner that can lead to improvement in FS?

The Problem with the state-of-the-art aggressiveness controllers is that the throttling decision of one core's prefetcher is oblivious to the throttling decisions of other cores' prefetchers. Due to this, a throttling decision that is supposed to perform well from an individual core's perspective becomes less effective when it interacts negatively with other cores' throttling decisions. Furthermore, the throttling decisions are mostly driven by thresholds of prefetch and interference metrics and we find that these metrics do not correlate *strongly* to the FS. Ideally, we would like an aggressiveness controller to explore all possible interactions. However, for an N-core system with N LLC prefetchers, after every window, the controller has to explore 2^N possible interactions [2] to find out the best throttling decisions, which is *feasible* only for small core-count systems. Note that the numbers of interactions increases exponentially with an increase in the core count.

Our approach is based on the observation that hardware prefetcher of a core, at the shared resources, either *interferes significantly* or *interferes marginally*. Based on this observation, for an N-core system with N LLC prefetchers, we create two groups of prefetchers: aggressive (G0) and meek (G1). After creating two groups, we try to understand the *interactions* between the throttling decisions by exploring four possible throttling combinations among G0 and G1 ($\downarrow_{G0}\downarrow_{G1}$, $\downarrow_{G0}\uparrow_{G1}$, $\uparrow_{G0}\downarrow_{G1}$, and $\uparrow_{G0}\uparrow_{G1}$) [3]. Through this grouping, we prune the exploration space. To find out the best throttling combination (a combination that provides maximum FS), which we call as *synergistic combination*, we propose a *proxy* metric called prefetcher-caused fair-speedup (PFS). This metric quantifies the contribution of each throttling combination on FS, and helps in selecting the synergistic combination. Note that all the prefetchers that belong to a particular group use the same throttling decision (either $\uparrow$ or $\downarrow$) at their respective throttling levels. We make the following **contributions**:

- We motivate for the need of a synergistic prefetcher aggressiveness controller and for a proxy metric that can provide the impact of throttling decisions on system performance and fairness (Section 3). We also discuss the challenges in developing a synergistic controller and proposing a proxy metric (Section 4).
- We propose SPAC, a simple and effective mechanism for controlling the aggressiveness of multiple prefetchers in multi-core systems (Section 5).
- We provide the implementation details (Section 6) and evaluate SPAC on a wide variety of workloads. On an average, compared to no prefetching, for 4-, 8-, 12-, and 16-core workloads, SPAC provides improvements of 18.2%, 22.5%, 31.6%, and 27.2%, respectively (Section 8).

2. For each prefetcher, we consider two possible throttling decisions: $\downarrow$ and $\uparrow$.

3. We do not consider the decision of staying at the same level. We find that it has marginal effect and we discuss it in Section 8.8.

2 Background

In this section, we describe our baseline multi-core system and brief about the state-of-the-art prefetcher aggressiveness controllers that improve the system performance and fairness.

Baseline multi-core system: Our simulated multi-core system consists of 4, 8, 12, and 16 cores with 1, 2, 3, and 4 DRAM controllers, respectively. Each core has a private L1 and a private L2 cache, and the LLC (L3) is shared among all the cores. Each core has a prefetcher at the LLC, which sends prefetch requests to the DRAM. We use stream prefetcher, which is available in commercial processors such as IBM and Intel [3]. We also use prefetchers such as GHB [4] and AMPM [5] for our study.

Feedback directed prefetching (FDP) [6] is a single core prefetcher aggressiveness control technique that uses thresholds for prefetch-accuracy (ratio of prefetch-hits to the prefetch-issued), prefetch-timeliness, and cache pollution (prefetched block evicting a demand block that is requested by the core in near future) to control the aggressiveness.

Hierarchical prefetcher aggressiveness controller (HPAC) [2] is an extension of feedback directed prefetching (FDP) [6] for multi-core systems. HPAC uses two kinds of controllers: (i) a per-core local controller, which uses FDP to maximize individual core's performance, (ii) a global controller that finds out the *interfering* applications and overrides the decisions of the local FDPs. The main idea of HPAC is to reduce the prefetcher caused inter-core interference at the shared resources such as LLC and the DRAM by throttling down the prefetchers of those applications that interfere heavily. To find out the interfering applications, HPAC uses thresholds for metrics such as ACC (prefetch-accuracy), POL (inter-core cache pollution), BWC (DRAM-bandwidth consumed by a core), BWN (DRAM bandwidth needed by a core), and OBWN (DRAM bandwidth needed by all other cores except a particular core). The principle that drives HPAC is: *an application with high BWC increases the BWN of all other cores; eventually OBWN becomes high, which becomes the source of interference*. HPAC throttles down the prefetchers of applications that are responsible for increasing the OBWN. HPAC uses five throttling levels (level-1 to level-5 with [prefetch-degree, prefetch-distance]): [1,4], [1,8], [2,16], [4,32], and [4,64].

PFST [7] coordinates HPAC, and fairness via source throttling (FST) [8] to improve the fairness of a system in the presence of prefetching. In addition to the thresholds of HPAC and FDP, it uses additional thresholds to find out the most *interfering* application and the *slowest* application (application with maximum slowdown). The global controller of PFST throttles down the prefetcher of the most *interfering* application.

3 Motivating Observations

In this section, we discuss two fundamental observations that motivate us to propose SPAC.

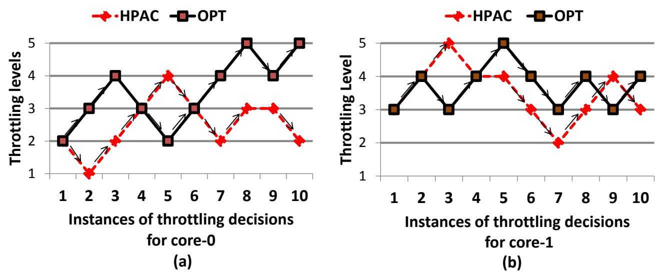


Figure 2: A snapshot illustrating the need for synergistic throttling.

Myopic throttling decisions: State-of-the-art controller, such as HPAC, is a coordinated technique that collects global interference metrics and uses thresholds to track them. However, its *throttling decisions are myopic in nature because the throttling decision assigned to an individual prefetcher is oblivious to the throttling decisions that are assigned to the other cores' prefetchers*. Figure 2 shows the differences in the throttling decisions (in terms of throttling levels) of HPAC and OPT for a 2-core system that runs `leslie3d` on core-0 and `libquantum` on core-1. Note that OPT explores all possible throttling combinations and selects the *synergistic combination*. At the end of the 1st instance, both HPAC and OPT throttle-up core-1's prefetcher (Figure 2 (b)) whereas for core-0's prefetcher (Figure 2 (a)), HPAC contradicts (throttles down) with the OPT (throttles up). The primary reason behind this contradiction is that HPAC does not consider the interactions between the throttling decisions of multiple prefetchers. OPT, on the other hand, explores all the possible interactions and selects the throttling decisions that provide the maximum fair-speedup. This leads to our first observation:

OBSERVATION 1. *There is a need for prefetcher aggressiveness controller whose throttling decisions are synergistic and not myopic.*

Usage of prefetch metrics: One of the metrics that quantifies the usefulness of a hardware prefetcher is *prefetch-accuracy*. In a single core system, we find that prefetch accuracy correlates *strongly* to improvement in the IPC (with a Pearson's correlation coefficient of 0.83 across 25 SPEC CPU 2000/2006 benchmarks [9]). Note that prefetch-accuracy does not capture the effects of memory-level parallelism (MLP). In multi-core systems (e.g., 4-core and above), the Pearson's correlation coefficient drops to 0.61 because of inter-core interference at the LLC and at the DRAM, which leads to variations in the LLC miss latency. Similarly, the per-core interference metrics such as *POL* and *BWN* do not correlate strongly (with Pearson's correlation coefficients of -0.2 and -0.57, respectively [4]) with the FS. However, state-of-the-art aggressiveness controller uses the above mentioned metrics for throttling the prefetchers.

The other issues associated with prefetch-accuracy are as follows: Prefetch accuracy (prefetch-hits/prefetch-issued) does not distinguish between (10000/10000) and (1/1). In both the cases, the accuracy is 1, and changing aggressiveness based on accuracy will not be fruitful as one can interfere while other wont. Similarly, if we go for no prefetching as one of the throttling level, the prefetch accuracy will become infinite (1/0).

Also, we find that applications such as `omnetpp`, `ammp`, and `mcf` *that are not prefetch-friendly*[5], lose performance when prefetcher is always ON and cause significant prefetcher-caused inter-core interference. So it is better to turn off a prefetcher that does not improve the system performance. This leads to our second observation:

OBSERVATION 2. *There is a need for a metric that can correlate strongly to the fair-speedup and can help in making effective throttling decisions. Also, there is a need for an additional throttling-level (level 0) with prefetch-degree 0 (no prefetching).*

For example, `bzip2`, an application with high prefetch accuracy (96%), gets a saving of only 30% of its LLC miss cycles because of hardware prefetching. So the approach of throttling-up a prefetcher that has high prefetch-accuracy is not always correct because prefetch-accuracy exaggerates the benefits of a prefetcher.

In the following sections, we discuss the challenges and explain SPAC, a mechanism that exploits the aforementioned observations to throttle the prefetchers.

4 CHALLENGES

Exponential increase in combinations: To find out the interactions between the throttling decisions, at the end of every fixed size window, an aggressiveness controller should explore all the possible throttling combinations. For an N-core system with N LLC prefetchers, a throttling combination is a combination of N throttling decisions, where a throttling decision can be throttling-up ($\uparrow$) or throttling-down ($\downarrow$). However, the number of possible throttling combinations increases *exponentially* with the increase in the core count, which makes it *infeasible* to explore every possible combination. For example, after every window, for 8 and 16-core systems, the controller has to explore 2^8 and 2^{16} possible throttling combinations before selecting the synergistic combination. A synergistic combination is the throttling combination that provides maximum FS among all the combinations. Also to find out the synergistic combination, for 8-core and 16-core systems, the system has to go through 2^8-1 and 2^{16}-1 non-synergistic combinations, which can lead to sub-optimal FS. *The challenge is to find out the synergistic combination, for each window that can bridge the gap between HPAC and the OPT with minimum number of explorations per window.*

Finding a proxy metric for FS: To find out the synergistic combination, we need to find out the FS of all possible combinations. We revisit Equation 1 to understand why we need a proxy metric. To find out the FS at regular windows, we need IPC$_i^{alone}$ and IPC$_i^{together}$ for each application i running on an N-core system. We can calculate IPC$_i^{together}$ by finding out the IPC of application i for a given window. However, finding out IPC$_i^{alone}$ is not straightforward. Subramanian et al. [10] point out that, without offline profiling, it is difficult to find out IPC$_i^{alone}$ of an application i when it runs concurrently along with the other applications. They

4. Note that these values are negative as *POL* and *BWN* maintain negative relation with IPC.

5. Prefetch-friendly: An application is prefetch-friendly if prefetching improves the performance by more than 10% [2] compared to system with no-prefetching.

propose an approximate solution, which halts the execution of applications to find out the IPC^{alone} of a single application. However, this approach drops the FS as halting the applications causes halting of their prefetchers also. Furthermore, this approach requires frequent sampling.

So if we can find out IPC$_i^{alone}$ and IPC$_i^{together}$ for each application then the ideal way of finding the impact of a throttling combination will be through its corresponding FS. Unfortunately, FS depends on the IPCs of each application and IPC can mislead the prefetcher aggressiveness controller as the improvement in the IPC may be a result of other factors, such as reduction in the inter-core interference (because of intelligent decisions of LLC replacement and DRAM scheduling policies) at the shared-resources, or a change in the behavior of a branch predictor. *So the challenge is to find an alternative of IPC that can provide a one-to-one relationship between the throttling decisions and the system performance. Also we need to find out X$_i^{alone}$, where X is the best possible alternative of the IPC.*

In the next section, we describe our main contributions and provide solutions for the above mentioned challenges.

5 SPAC

SPAC is an aggressiveness controller that throttles the prefetchers after every fixed size windows. The window of the SPAC consists of two phases: *exploration* and *implementation*, where implementation phase follows the exploration phase. In the beginning of every exploration phase, each prefetcher communicates its throttling level to a meta-controller. After receiving the throttling levels, the meta-controller of SPAC explores all possible throttling combinations and finds out the FS provided by each combination. At the end of an exploration phase, it uses the synergistic throttling combination (that provides the maximum FS) in the implementation phase. *The rationale behind this approach is that the best throttling combination of the exploration phase will perform best in the implementation phase also.*

5.1 Search Space Pruning

To reduce the size of the exploration space of throttling combinations, SPAC uses a variant of *hill-climbing* approach, and uses prefetch-issued-per-demand-miss (PPM) at the LLC to create multiple groups of prefetcher. Through this grouping, we prune the search space. Based on our evaluation, we observe that there is an interesting behavior: a hardware prefetcher either *interferes significantly* with other cores' demand and prefetch requests at the shared resources, or it *interferes marginally*. Based on this observation, we create two groups of prefetchers: (i) *aggressive* group (G0) and (ii) *meek* group (G1).

Why PPM? To validate our observation, we correlate the actual prefetcher-caused inter-core interference with the PPM metric. We find that there is a *strong* correlation (a Pearson's correlation coefficient of 0.88) between PPM of a core and the actual inter-core interference caused by a prefetcher of that core. To calculate the actual prefetcher-caused inter-core interference, we count the number of cycles for which a demand or a useful prefetch request is delayed at the shared resources, such as LLC, LLC MSHRs, DRAM request queue, DRAM row-buffers, and DRAM banks, because of prefetch requests of a given core.

Algorithm 1 GROUPING_PROCESS

```
 1: Input: PPM [0:N-1] // $N$-core system.
 2: Output: Group G0 (aggressive) and Group G1 (meek)
 3: total-PPM=0, avg-PPM=0, G0=$\phi$, and G1=$\phi$;
 4: for each $i$ where $i$ is the core-id do
 5:     total-PPM = total-PPM + PPM ($i$);
 6: end for
 7: avg-PPM = $\frac{\text{total-PPM}}{N}$;
 8: for each $i$, where $i$ is the core-id do
 9:     if (PPM ($i$)>avg-PPM) then
10:         G0 ← G0 $\cup$ $i$;
11:     else
12:         G1 ← G1 $\cup$ $i$;
13:     end if
14: end for
15: return G0 [ ] and G1 [ ]
```

Based on this observation, we use a simple approach to divide the prefetchers into these two groups: for an N-core system, if the PPM of a prefetcher is greater than the average-PPM $\left(\frac{\sum PPM}{N}\right)$ then the prefetcher belongs to *aggressive* group else to *meek* group. This simple approach abstracts out the challenges and works well in practice. Also, this approach of grouping is scalable as it is independent of the core-counts and it can explore all kinds of throttling combinations. Algorithm 1 shows the grouping process. Through the grouping process, an N-core system becomes a two-core system (which is two groups) and it becomes feasible to explore four throttling combinations, which are as follows: (i) $\downarrow_{G0}\downarrow_{G1}$, (ii) $\downarrow_{G0}\uparrow_{G1}$, (iii) $\uparrow_{G0}\downarrow_{G1}$, and (iv) $\uparrow_{G0}\uparrow_{G1}$. SPAC also explores another throttling combination, which is continuing with the current combination ($|_{G0}|_{G1}$). SPAC explores these five combinations in five consecutive intervals called *exploration intervals*. For example, in a 4-core system, if the current throttling levels for core-0 to core-3 are 1, 1, 2, and 3, respectively then SPAC groups the 4 prefetchers to 2 groups. Assuming G0 contains core-0 to core-2, and G1 contains core-3, SPAC explores throttling levels of (i) **0, 0, 1** ($\downarrow$), and 2 ($\downarrow$) (ii) **0, 0, 1** ($\downarrow$), and 4 ($\uparrow$), (iii) **2, 2, 3** ($\uparrow$), and 2 ($\downarrow$), (iv) **2, 2, 3** ($\uparrow$), and 4 ($\uparrow$) along with the current throttling combination, which is **1, 1, 2** ($|$), and 3 ($|$). After exploring five combinations (4 possible combinations (2^2 for 2 groups) + 1 current combination), SPAC selects the throttling combination that provides the maximum FS (synergistic-combination). Note that, SPAC has the potential to explore the entire search space in multiple exploration phases as the prefetchers of G0 will switch to G1, and vice versa, based on their PPMs. To exemplify it, in an N-core system with N LLC prefetchers, we can visualize a throttling combination as an n-bit binary string, where each bit represents the throttling decision of one core, with 1 representing throttling-up and 0 representing throttling-down. *Through this grouping process, SPAC explores the possible interactions between N prefetchers.* Please note, we do not compute PPM per committed instructions as we find small variations in the committed instructions during the exploration of each combinations.

5.2 Proxy Metrics

Based on the Observation 2, as mentioned in Section 3, we introduce two metrics: (i) prefetch-benefit and (ii) prefetcher-caused fair-speedup. Prefetch-benefit is the building block for prefetcher-caused fair-speedup. Prefetch-benefit approximates the fraction of processor cycles saved because of prefetching, at the shared resources, and prefetcher-caused fair-speedup finds out fair-speedup provided by the hardware prefetchers. Note that, we compute these metrics at the end of every exploration intervals.

To find out the *prefetch-benefit* of core i's prefetcher, we compute cycles$_{saved}$ (i), which is the fraction of core i's processor cycles that its hardware prefetcher is able to save at the shared resources. For a prefetcher of core i

$$cycles_{saved}(i) = prefetch_{hits}(i) \times miss\text{-}penalty_{avg}(i) \quad (2)$$

$$prefetch_{hits}(i) = prefetch_{hitsLLC+MSHR}(i) \quad (3)$$

prefetch$_{hitsLLC}(i)$ is the number of demand hits to the prefetched cache blocks (cache blocks brought by the prefetcher) of core i and prefetch$_{hitsMSHR}(i)$ is the number of demand hits at the MSHR entries that are occupied by prefetched requests of core i. Note that, through this, we take care of both timely and late prefetch requests. As finding out the cycles saved for each prefetch hit is not straight forward, we assume a prefetch hit saves miss-penalty$_{avg}(i)$, which is the average LLC demand-miss-penalty of core i in terms of processor cycles. Next, we find prefetch-benefit(i) by approximating the fraction of total miss cycles that a prefetcher of core i is able to save ((cycles$_{saved}$ (i)) for core i. We count the total miss cycles by using penalty$_{total}(i)$, which is the sum of LLC miss penalties of core i for a given exploration interval.

$$prefetch\text{-}benefit(i) = \frac{cycles\text{-}saved(i)}{cycles\text{-}saved(i) + penalty_{total}(i)} \quad (4)$$

Prefetch-benefit ranges between 0 to 1, where a prefetcher that saves all the miss-penalties will have prefetch-benefit as 1 and a prefetcher that does not save any cycles will have prefetch-benefit as zero. Note that to ease the implementation complexity, we assume the number of saved cycles is same for both, a prefetch that is injected just before a demand access and a prefetch that is injected long before a demand access, as long as the prefetch is still outstanding at the time of the demand access. We implement a detailed model by modifying the MSHRs, DRAM request queue, and the DRAM bus, and find that there is marginal difference in the prefetch-benefits. So we go for a simple and yet effective design.

Also, note that, *prefetch-benefit* is different from *prefetch-coverage* $\left(\frac{prefetch\text{-}hits}{prefetch\text{-}hits+demand\text{-}misses}\right)$. Prefetch-coverage does not consider various factors (DRAM-row-buffer-conflicts, DRAM-bank-level-interference, and DRAM-bus-level interference) that can affect the off-chip latency. The miss-penalty-total in Equation 4 takes care of all these factors as it sums up the penalties (for some workloads, it is as high as thousand cycles) associated with each demand miss. Also, we do not include the effects of LLC pollution in prefetch-benefit. There are two reasons for this decision: (i) we want to propose a metric, which is simple enough to implement, and (ii) *prefetch-benefit* quantifies the number of processor cycles that are saved but not interfered with other cores, by a particular core's prefetcher.

As we want to measure the impact of prefetching decisions on the system performance, we use a metric that is less sensitive to the unknown system effects. Also, we find that for a single core-system, an increase in the prefetch-benefit *causes an increase in the IPC. Note that reverse is not always true.*

Algorithm 2 BEST_THROTTLING_COMBINATION

```
1: Input: prefetch-benefit$_{G0}$[5] and prefetch-benefit$_{G1}$[5]
2: Output: best-combination
3: max-benefit$_{G0}$=0, max-benefit$_{G1}$=0, max-PFS=0;
4: for each $j$, where $j$ is the combination-id and $0 \le j \le 4$ do
5:    if (prefetch-benefit$_{G0}$[j]>max-benefit$_{G0}$) then
6:       max-benefit$_{G0}$ = prefetch-benefit$_{G0}$[j];
7:    end if
8:    if (prefetch-benefit$_{G1}$[j]>max-benefit$_{G1}$) then
9:       max-benefit$_{G1}$ = prefetch-benefit$_{G1}$[j];
10:   end if
11: end for
12: for each $j$, where $j$ is the combination-id and $0 \le j \le 4$ do
13:    PFS($j$) = $\frac{2}{\frac{max\text{-}benefit_{G0}}{prefetch\text{-}benefit_{G0}[j]} + \frac{max\text{-}benefit_{G1}}{prefetch\text{-}benefit_{G1}[j]}}$;
14:    if (PFS($j$) > max-PFS) then
15:       max-PFS = PFS($j$);
16:       best-combination = $j$;
17:    end if
18: end for
19: return best-combination;
```

Prefetcher-caused fair-speedup (PFS): To find out the *fair-speedup* of a system that is contributed by a particular throttling combination j, we modify FS to PFS by replacing IPC with prefetch-benefit.

$$FS(j) = \frac{N}{\sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{together}}}, PFS(j) = \frac{N}{\sum_{i=0}^{N-1} \frac{prefetch\text{-}benefit_i^{alone}}{prefetch\text{-}benefit_i^{together}}} \quad (5)$$

Note that, in FS, IPC$_i^{alone}$ is the IPC of an application i when it runs alone on an N-core systems. Similarly, prefetch-benefit$_i^{alone}$ corresponds to prefetch-benefit of core i when it runs alone on an N-core system.

Across 185 multi-core workloads, we find that an increase and decrease in PFS results in increase and decrease in FS, which means we have a prefetch metric that correlates to the system performance, and we can use it to make throttling decisions. Note that an increase/decrease in FS does not always come from an increase/decrease in PFS.

As the grouping process of SPAC abstracts out an N-core system to a 2-core system (groups G0 and G1), we replace N by 2 in our modified PFS (refer Equation 6). Also, in the denominator of PFS, i varies from 0 to 1 as we are dealing with only two groups (G0 and G1).

$$PFS(j) = 2/\sum_{i=0}^{1} \frac{prefetch\text{-}benefit_{Gi}^{alone}}{prefetch\text{-}benefit_{Gi}^{together}} \quad (6)$$

Note that to find prefetch-benefits$_{G0}^{together}$, we add the prefetch-benefits of all the prefetchers of G0 when G0 runs with G1 in a multi-core system. As it is difficult to find

Algorithm 3 SPAC

```
1: Input: PPM [0:N-1] // $N$-core system
2: Output: throttling-decision [0:N-1] // contains the throttling decisions
3: GROUPING_PROCESS (PPM [0:N-1]) (Algorithm 1)
4: for each $j$, where $j$ is the combination-id and $0 \leq j \leq 4$ do
5:     Explore combination C[$j$] and store benefit$_{G0}$[$j$] and benefit$_{G1}$[$j$] for next five sub-intervals;
       // C[0]=$\downarrow_{G0}\downarrow_{G1}$, C[1]=$\downarrow_{G0}\uparrow_{G1}$, C[2]=$\uparrow_{G0}\downarrow_{G1}$, C[3]=$\uparrow_{G0}\uparrow_{G1}$, and C[4]=$|_{G0}|_{G1}$
6: end for
7: combination-id = BEST_THROTTLING_COMBINATION (benefit$_{G0}$[0:4], benefit$_{G1}$[0:4]); (Algorithm 2)
8: for each $i$, where $i$ is the core-id do
9:     Throttle core $i$ based on C[combination-id];
10: end for
11: return throttling-decision [0:N-1];
```

prefetch-benefit$_{Gi}^{alone}$ (refer Section 4), we estimate prefetch-benefit$_{Gi}^{alone}$ for G0 and G1 as follows:

$$prefetch\text{-}benefit_{Gi}^{alone} = \max_{0 \leq j \leq 4} (prefetch\text{-}benefit_{Gi}^{together}(j)) \tag{7}$$

where j is the combination-id, which varies from 0 to 4 (five possible throttling combinations). This estimate is a valid estimate as prefetch-benefit$_{Gi}^{alone}$ is always greater than or equal to the prefetch-benefit$_{Gi}^{together}$ provided by all the combinations. Algorithm 2 finds out PFS by using Equation 6. For five throttling combinations, Algorithm 2 stores the prefetch benefits for G0 and G1 in two vectors benefit$_{G0}$[5] and benefit$_{G1}$[5], respectively. At the end, Algorithm 2 returns the throttling combination that provides the maximum PFS.

5.3 Putting it All Together

The meta-controller of SPAC collects the throttling levels from each prefetcher and returns the throttling decisions (up/down) that provides maximum PFS. Algorithm 3 describes the overall mechanism. At the beginning of each exploration phase, all the cores communicate their PPMs to the meta-controller, for next 5 intervals of exploration. Line 3 of Algorithm 3 creates two groups: G0 and G1. Once the grouping process is over, SPAC explores (line 5 of Algorithm 3) five combinations, for next five intervals of an exploration phase. Next, line 7 of Algorithm 3 finds out the best throttling combination and uses it in the implementation phase. The entire process of exploration and implementation repeats at the end of every implementation phase. For example if combination-id 0 to combination-id 4 provide prefetch-benefits of [0.3, 0.3], [0.5, 0.5], [0.9, 0.1], [0.1, 0.1], and [0.1, 0.9] for [G0, G1] then SPAC finds the PFS for each combination, which are 0.3, 0.5, 0.2, 0.1, and 0.2, respectively. In the implementation phase, SPAC will use the *synergistic* combination (combination with the maximum PFS, which is, in this case, combination id 1 with PFS of 0.5).

In summary, the **key insight** that drives SPAC is that it *throttles the prefetcher in a synergistic manner by creating two groups of prefetchers, exploring five throttling combinations, and selecting the synergistic combination.*

Adapting to phase-change behavior: SPAC's throttling decisions may become sub-optimal if one or more applications undergo a phase change. To make SPAC adaptable, we use a phase-change detector that uses the metric called *accesses-per-cycle* (APC) [11]. We calculate per-core APC at the shared LLC. APC finds the ratio of per-core LLC accesses and the cycles consumed by these accesses. To detect the phase change, we use the average-APC of last K intervals and compare it with the APC of the current interval. If the difference between the average-APC and current-APC crosses a threshold (called *apc-threshold*), then the phase-change detector reports SPAC about phase change. After receiving the information from the phase-change detector, SPAC stops its current operation (exploration or implementation) and restarts exploration phase from the scratch. Once the exploration phase is over, SPAC uses the best combination till the end of the implementation phase, or till the next phase-change.

6 IMPLEMENTATION DETAILS

Table 1 shows the throttling levels that we use in SPAC. Note that level-0 corresponds to no prefetching. This additional level helps in further reducing the prefetcher-caused inter-core interference caused by prefetch-unfriendly applications. Before the first exploration phase, all the prefetchers start with the level-4. At the end of each exploration phase, we update the throttling levels based on the outcome of SPAC. We find that the relation, prefetch-distance that is 8 times of prefetch-degree, is the best by sweeping through different combinations of prefetch-degree and prefetch-distance. Also, we do not go beyond the prefetch-degree of 8 as some of the workloads show saturation of DRAM bandwidth (12.8 GB/sec) available per DRAM channel. Note that a prefetcher, which is in level-0/level-4, cannot be throttled down/up further.

Phase lengths: The effectiveness of SPAC depends on the length of each exploration phase, and the ratio of implementation to exploration phase. Based on empirical evaluation, we find that the $\frac{implementation\text{-}phase}{exploration\text{-}phase} = 3$ provides the best fair-speedup. An exploration phase consists of five intervals to explore five throttling combinations. To find out the length of each interval, we sweep through various interval lengths (512 to 32K LLC demand misses) for exploring individual throttling combinations. We find that the interval lengths of 512, 1K, 2K, and 2K LLC misses provide the best performance for 4-, 8-, 12-, and 16-core systems, respectively. For a 4-core system, the exploration phase is of length 2.5K LLC misses (five intervals), and the implementation phase is of length 7.5K LLC demand misses. This leads to a *complete window length of 10K LLC demand misses* for one complete

Throttling Level	Prefetch-degree	Prefetch-distance
0	0	0
1	1	8
2	2	16
3	4	32
4	8	64

Table 1: Throttling levels of SPAC.

Purpose	**Registers**	**Overhead for a 4-core system**
(**A**) For counting interval length	one-complete-window (10K misses) and Sub-interval length (512 misses)	14 + 9 bits=23 bits
(**B**) For counting events (per core): two 20-bit registers and two 9-bit registers	miss-penalty$_{total}$, cycles-saved, demand-misses$_{total}$, prefetch-hits	(2×20 bits + 2×9 bits)×4=232 bits
(**C**) For storing events (per core): three 9-bit registers and one 3-bit register	miss-penalty$_{avg}$, average-PPM, APC, and throttling-level	((3×9) bits+(3 bits))×4=120 bits
(**D**) For storing events (per combination): three 7-bit registers	benefit$_{G0}$, benefit$_{G1}$, PFS	(3×7 bits)/combination×5 combinations=105 bits
(**E**) For storing other events: one per-core 7-bit register, two 7-bit registers and one 3-bit register	prefetch-benefit/core, max-benefit$_{G0}$, max-benefit$_{G1}$, and best-combination	(4×7) + (2×7) + 3 bits=45 bits
Total overhead – A+B+C+D+E		**63B**

Table 2: Hardware overhead of SPAC. In (B), the values of miss-penalty and cycles-saved go up-to 2048 cycles for a single LLC miss. For an exploration interval of 512 LLC misses, miss-penalty$_{total}$ and cycles-saved will need $log_2(2048 \times 512)$ bits = 20 bits. For an 8-, 12-, and 16-core system, the overhead is 114B, 172B, and 230B, respectively.

round of exploration and implementation. Note that, these interval lengths make sure that there is small variations in the number of instructions that are committed across the five explorations intervals. The phase-change detector uses interval lengths of 512 LLC misses, 0.03 (average-apc) as the apc-threshold, and 16 as the value of K. We set these parameters empirically by sweeping through various values. For example, for apc-threshold, we sweep values from 0.01 to 2 in the scale of 0.01. An in-accurate phase-change detector degrades the performance by max. 3.2%.

Additional logic and hardware overhead: To create two groups of prefetchers, SPAC uses per-core adders, multipliers, dividers, and comparators. The meta-controller of SPAC takes ten processor cycles[6] for grouping and for selecting the synergistic combination. Table 2 provides a self-contained description of the hardware overhead of SPAC. For a 4-core system with an 8-MB LLC, SPAC incurs a hardware overhead of **63B** as compared to **34.7KB** of HPAC. Note that, for both SPAC and HPAC, we do not add the hardware overheads that come from cache replacement policy PACMan [12] and the DRAM scheduling policy PADC [13]. Also, to minimize the hardware overhead, we store all the metrics as integers (and not as floating point numbers) by multiplying the floating point metrics with 100. This saves the hardware overhead of each metric from 32 bits to 7 bits. Note that the logic for SPAC is not on the critical path of execution. In terms of implementation complexity, SPAC is much simpler than HPAC as HPAC uses bloom filters for tracking inter-core LLC pollution, and use performance counters per DRAM banks to track the bandwidth consumption.

Changes to the baseline Organization: We place the meta controller of SPAC beside the LLC, with an access latency of 1 cycle. At the beginning of each exploration phase, each prefetcher sends its PPM to the meta controller. After sending the PPMs, the prefetchers reset them to zero. After receiving the PPMs, the meta controller creates two groups, and informs all the prefetchers to explore five different throttling combinations for next five intervals. At the end of an exploration phase, each prefetcher communicates its prefetch-benefits for each combination-id to the meta-controller of the SPAC. The meta-controller calculates the prefetch-benefits of groups G0 and G1 and selects the synergistic throttling combination. Also, it communicates the throttling decisions based on the synergistic combination, to the prefetchers of G0 and G1. On a phase-change, the phase-change-detector informs the meta-controller about the same.

6. We verify it using Synopsys design compiler for 45nm technology.

7 EVALUATION

Evaluation Methodology: We use gem5 [14] simulator to evaluate the effectiveness of SPAC. Table 3 shows the baseline configuration of our simulated system. The baseline system uses HPAC as the prefetcher aggressiveness controller, PACMan [12] as the LLC replacement policy and PADC [13] as the DRAM scheduling policy. Note that the combination of PACMan + PADC + HPAC outperforms LRU + PADC + HPAC. To make PACMan work properly with HPAC, we revisit the thresholds used in HPAC, and we find a small decrease in the LLC pollution and bandwidth consumption thresholds work well with the combination of PACMan and PADC. We collect the statistics for workloads by running each benchmark in a workload for 500M instructions after a fast-forward of 20B instructions and warm-up of 500M instructions, which is similar to the methodology used in [15] and [16]. A workload terminates when the slowest benchmark completes 500M instructions.

Workload selection: Table 4 classifies 25 benchmarks (from SPEC CPU 2000/2006 benchmark suite) into 3 types (`T1`, `T2`, and `T3`), based on their nature of prefetch-benefit curves. A prefetch-benefit curve is a curve that shows the change in the prefetch-benefits with the change in the throttling levels - from lower level to the higher level. A benchmark is memory-intensive if the number of L3 misses per kilo instructions (MPKI) is greater than 1 [2] and a benchmark is prefetch friendly if the performance improvement with prefetching is more than 10%. Table 5 shows 13 different

	Benchmarks and their Types
$\triangle$throttling-level=+ve/-ve, $\triangle$prefetch-benefit=constant	`T1`: gromacs, sjeng, *bzip2*, namd, *hmmer*
$\triangle$throttling-level=+ve/-ve, $\triangle$prefetch-benefit=+ve/-ve	`T2`: calculix, **zeusmp**, ***lbm***, **mesa**, **wupwise**, ***libquantum***, ***soplex***, mgrid, ***bwaves***, ***swim***
Irregular, neither `T1` nor `T2`	`T3`: **art**, **ammp**, **omnetpp**, **galgel**, h264ref, **twolf**, **mcf**, **milc**, ***leslie3d***, ***GemsFDTD***

Table 4: Classification of SPEC CPU 2000/2006 benchmarks. We embolden the memory-intensive ones, and we italicized the benchmarks with accuracy > 0.7.

Processor	4/8/12/16-cores, 3.7 GHz, out of order
L1 D/I, L2	32 KB (4 way), 256KB (8 way)
Shared L3	8/16/32/64 MB for 4/8/12/16 cores with 16/16/32/32 way
MSHRs	16, 16, 64/128/256/512 MSHRs at L1, L2, L3 with 4/8/12/16 cores
Cache line size	64B in L1, L2 and L3
Replacement policy	LRU at L1/L2, PACMan [12] at L3
Per-core L3 prefetcher	Streaming, 32 streams with degree = 4 and distance = 64
Coherence Protocol	MOESI
On-chip interconnect	Crossbar
DRAM controller	1/2/3/4 controllers for 4/8/12/16-cores, Open Row, 64 read/write queues, PADC [13], drain-when-full
DRAM bus	split-transaction, 800 MHz, BL=8
DRAM	DDR3 1600 MHz (11-11-11) 2 Ranks/Channel and 8 Banks/Rank, Max bandwidth/channel – 12.8 GB/sec

Table 3: Parameters of the simulated system.

`M1, M2, M3`	`All-T1, All-T2, All-T3`
`M4, M5, M6, M7, M8, M9`	`0.75T1-0.25T2, 0.75T1-0.25T3, 0.75T2-0.25T3, 0.75T3-0.25T2, 0.75T3-0.25T1, 0.75T2-0.25T1`
`M10, M11, M12`	`0.5T1-0.5T2, 0.5T1-0.5T3, 0.5T2-0.5T3`
`M13`	`Random`

Table 5: 13 categories of multi-programmed WL mixes. `xT1-yT2` corresponds to a mix that contains `x` fraction of `T1` benchmarks and `y` fraction of `T2` benchmarks.

categories of workload (WL) mixes that we create by mixing the benchmarks based on the above 3 types. To evaluate our mechanism, we create 100 4-core, 50 8-core, 25 12-core, and 10 16-core WLs that are distributed almost equally across thirteen categories.

Performance and fairness metrics: Apart from FS, which is our primary performance metric, we also use weighted-speedup (WS) [17] to quantify the effectiveness of SPAC. To measure the bandwidth consumption, we use DRAM bus accesses per kilo instructions (BPKI) [6]. To evaluate the degree of unfairness with SPAC, we use the metric *unfairness* [18]. We calculate *unfairness* (UF) as the ratio of the maximum slowdown to the minimum slowdown of applications that are running simultaneously. We also report the system power consumption by breaking it into CPU and DRAM power consumption. We define these metrics as follows:

$$IS_i = \frac{CPI_i^{together}}{CPI_i^{alone}} \tag{8}$$

$$WS = \sum_{i=0}^{N-1} \frac{IPC_i^{together}}{IPC_i^{alone}} \tag{9}$$

$$UF = \frac{max\{IS_0, ..., IS_{N-1}\}}{min\{IS_0, ..., IS_{N-1}\}} \tag{10}$$

IPC$_i^{together}$ and CPI$_i^{together}$ are the IPC and CPI of core i when it runs along with other N-1 applications on a multi-core system of N cores. IPC$_i^{alone}$ and CPI$_i^{alone}$ are the IPC and CPI of core i when it runs alone on a multi-core system of N cores. IS corresponds to individual slowdown.

8 Results and Analysis

In this section, we evaluate SPAC with HPAC and PFST in terms of FS, WS, unfairness, BPKI, and power consumption for a wide variety of 185 WLs. Due to space limitations, we analyze only the 4-core WLs in detail. Note that, we compare SPAC with PFST for 4-core WLs only as HPAC outperforms PFST for more than 88% of the 8-,12-, and 16-core WLs that we study. For 4-core WLs, HPAC outperforms PFST on 55 out of 100 WLs.

8.1 Overall Results

This section provides the results for 185 WLs that we use to study the effectiveness of SPAC. Table 6 shows the overall results in terms of FS and BPKI for 4-core to 16-core systems. SPAC outperforms HPAC on all the 185 WLs with an average FS improvement of 15.3% and a max/min improvement of 31%/3%. The performance improvement comes with a modest improvement in the BPKI, which is less than 2%. Compared to NOPF, SPAC provides an improvement of 21.1% with a maximum improvement of

4core(100 WLs)	**FS**	**BPKI**
Over NOPF	18.2% (39%)	7% (11%)
Over HPAC	12.3% (30%)	1.9% (4%)
8-core (50 WLs)		
Over NOPF	22.5% (31%)	7.1% (13%)
Over HPAC	14.9% (23%)	1.4% (2.8%)
12-core (25 WLs)		
Over NOPF	31.6% (41%)	8.7% (12.2%)
Over HPAC	18.3% (30.5%)	2.1% (5.1%)
16-core (10 WLs)		
Over NOPF	27.2% (46%)	6.6% (11%)
Over HPAC	17.5% (31%)	1.8% (5.6%)

Table 6: Summary of average (max) improvements with SPAC.

46% (for a 16-core WL that consists of applications of type `T3`). Out of 185 WLs, HPAC performs worse than NOPF for 29 WLs whereas SPAC outperforms NOPF for all the 185 WLs. The 29 WLs where HPAC performs worse, belong to WL mixes of type `M11`, `M8`, `M12`, and `M7`.

Weighted speedup: The primary aim of SPAC is to improve FS of a system but it outperforms HPAC in terms of WS also. Compared to HPAC, SPAC provides an additional average (geomean) improvement of 8.1% with max/min improvement of 13.2%/2.1%. Out of 185 WLs, 98 WLs show a trend where the scale of improvement in FS is similar to the scale of improvement in WS. However, for the rest, SPAC focuses more on fairness, which causes an 8.1% improvement in WS in contrast to a 15.3% improvement in FS. If fairness is not the major concern then a modified SPAC can be used to improve WS. We modify line no. 13 of Algorithm 2 with prefetcher-caused weighted-speedup (PWS) and define PWS for a combination-id j as follows: $\texttt{PWS}\,(j) = \frac{\texttt{max-benefit}_{G0}}{\texttt{benefit}_{G0}\texttt{[j]}} + \frac{\texttt{max-benefit}_{G1}}{\texttt{benefit}_{G1}\texttt{[j]}}$

Unfairness: In terms of unfairness, compared to HPAC, SPAC provides additional reduction of 14.1%. The main reason behind this reduction is that SPAC throttles down the memory-intensive benchmarks of `T2` and `T3` types, which contribute little to the PFS. The maximum slowdown that we observe for an application is 4.3%, which comes from a 4-core WL consisting of `omnetpp`, `libquantum`, `bwaves` and `milc`, where `libquantum` and `bwaves` get individual performance improvements of 20% causing a slowdown of 4.3% for `milc`.

Power consumption: In terms of power consumption, on an average, compared to HPAC, SPAC reduces the DRAM power consumption by 5.6%. The primary reason for this behavior is the usage of throttling-level 0 (no prefetching) for `T3` benchmarks such as `omnetpp` and `mcf`, and also the reduction in the inter-core interference at the LLC. However, SPAC increases the CPU power consumption by 4.4% as the benchmarks that are getting benefitted with hardware prefetching run at throttling-level 4. This results in reduction of the LLC misses, which improves the CPU utilization and also the CPU power consumption.

In summary, SPAC outperforms HPAC in terms FS, WS, fairness, and BPKI. *The primary reason behind this performance is that for 4-, 8-, 12-, and 16-core workloads, SPAC's throttling decisions match with 69%, 64%, 70%, and 60% of the OPT's decisions, respectively, whereas HPAC's decisions match only with 37%, 29%, 22%, and 24% of the OPT's decisions, respectively.*

8.2 Some Interesting Results

HPAC+PFS+level-0: So far, we consider HPAC does not have a level-0. However, for better understanding, we add an additional level (level-0 with no prefetching) to HPAC. Also, we explore a different version of HPAC, where HPAC's decisions are based on the PFS and the thresholds for different metrics. With these changes, HPAC performs a bit better, and it bridges the gap between SPAC and itself by additional 3.7%. The objective of this experiment is to show that *coordination indeed plays an important role*.

Addition of per-core local feedback: We study the effect

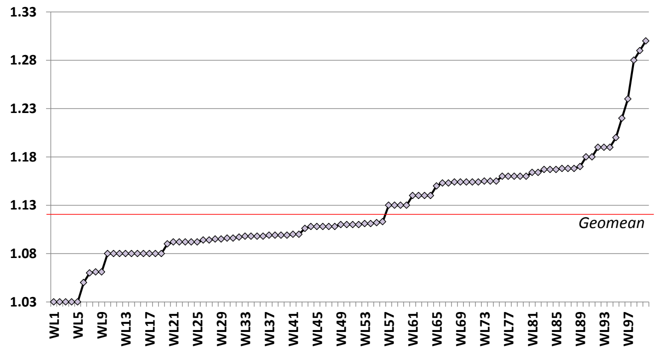


Figure 3: SPAC over HPAC across 100 4-core WLs. The Y-axis shows the performance improvement achieved by SPAC over HPAC.

of adding a local (per-core) feedback controller that can control the aggressiveness on the basis of improvement in per-core prefetch-benefits. We find that an addition of local feedback on top of SPAC improves FS by 1.66% only. This is an important insight as SPAC is able to do the job of both the global and local controller and there is no need of local controller like FDP, which is used as part of HPAC.

Impact on throughput: The main objective of SPAC is to improve the fair-speedup. However, if the objective is to improve the system throughput only ($\sum$IPC) then the current version of SPAC provides an improvement of 11.9% over HPAC. SPAC can be modified for throughput only by modifying the PFS as summation of prefetch-benefits-together. This modification improves throughput by 17.62%. A hybrid version of SPAC can also be used based on the user and system requirements.

8.3 Analysis of 4-core WLs

In this section, we provide a detailed analysis of 4-core WLs. Our observations are as follows:

Figure 3 shows the effectiveness of SPAC over HPAC in terms of FS for 100 4-core WLs. The WLs are sorted in an ascending order based on their performance improvement. Compared to HPAC, SPAC provides an average (geomean) improvement of 12.3% with a max/min improvement of 30%/3%.

1. Out of 100 4-core WLs, 44 WLs show an improvement of more than 12.3% (the average) and only 5 WLs show marginal improvement (less than 5%). Figure 4 shows the performance improvement for different kinds of 4-core WLs over no prefetching. Compared to HPAC, WL mix of types `M2`, `M6`, and `M8` are the major gainers with performance improvement of 19%, 18%, and 16%, respectively. WL mix that consists of all the benchmarks of `T1` types show the minimum improvement of 3.2%. As `T3` contains benchmarks of irregular behavior, the synergistic throttling of SPAC plays an important role. For WL mix of type `M5` (`0.75T1-0.25T3`), SPAC outperforms HPAC marginally (less than 4%) as $3/4^{th}$ of the applications are not affected by their own throttling decisions and the 4% improvement comes from the usage of throttling-level 4 with prefetch-degree 8. There are WL mixes, where the performance difference between SPAC and OPT is high, and we analyze the reasons for this behavior in Section 8.7.

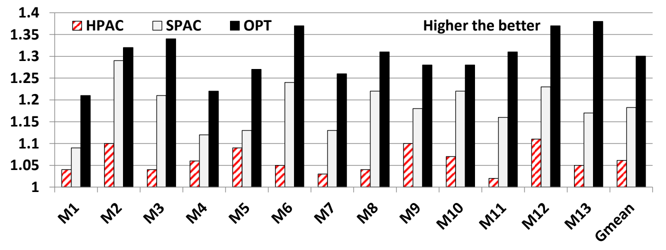


Figure 4: Performance improvement for 13 categories of 4-core WL mixes, normalized to no prefetching.

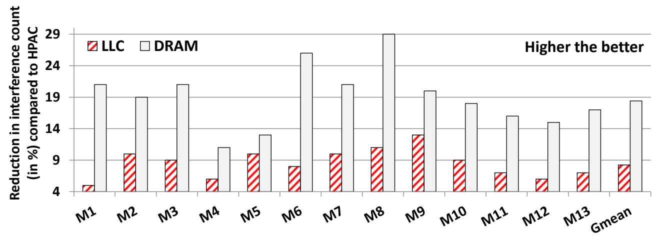


Figure 5: Reduction in LLC and DRAM interference-count (in %) compared to HPAC.

For WL mixes that belong to `M4` (`0.75T1-0.25T2`) and `M5` (`0.75T1-0.25T3`) categories, the performance of PFST approaches to the performance of SPAC if the WL contains only one prefetcher, which belongs to the aggressive group. For WLs with multiple aggressive prefetchers, SPAC outperforms PFST.

2. Figure 5 shows the reduction in the inter-core prefetcher-caused interference count at the LLC and DRAM. For LLC, we count the number of inter-core cache pollution and for DRAM, we count the number of row-buffer-conflicts, bank-level-interferences, and bus level interference because of a prefetcher. On an average, compared to HPAC, SPAC reduces the count of inter-core prefetcher-caused interference by 8.3% at the LLC and 18.5% at the DRAM. This reduction in the interference-count translates into the 12.3% of performance improvement compared to HPAC.

8.4 Comparison with HPAC and PFST

In terms of synergistic combinations, For 4-core WLs, HPAC's throttling combinations matches with OPT for 37% of the time, whereas SPAC's combinations matches with OPT for 69% of the time. We provide some of the key insights that help in understanding the differences between SPAC and HPAC, and PFST.

1. Coarse-grained decisions: Apart from the differences in synergistic combinations, one of the key insights that differentiate SPAC from HPAC is that HPAC throttles down the prefetchers of applications that cross the thresholds of interference metrics that quantify the bandwidth consumption, such as BWC and OBWN, whereas SPAC does not throttle them down as long as the throttling decisions improve the PFS of system. Similarly, there are scenarios where HPAC throttles up prefetchers of applications with high prefetch accuracy and low bandwidth consumption but SPAC throttles them down if they provide low PFS. Out of 100 4-core WLs, we observe these kinds of scenarios in 96 WLs.

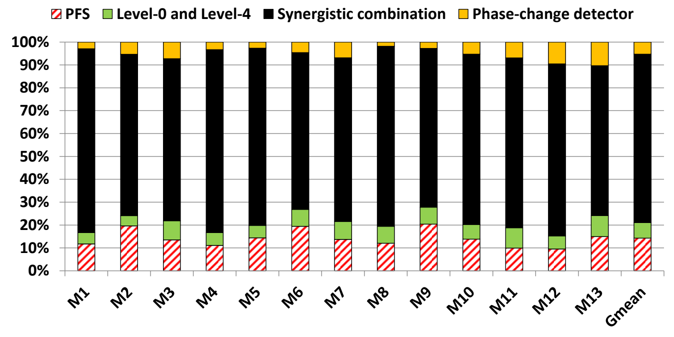


Figure 6: Contribution of each component in the performance of SPAC over HPAC.

2. Adapting to phase-changes: Out of 25 SPEC CPU benchmarks that we study, 11 benchmarks undergo phase-changes during their concurrent execution with other benchmarks. On a phase-change, on an average, HPAC takes around six implementation phases to recover[7] from a phase change, whereas SPAC recovers from a phase change in the next two implementation phases. Also, the addition of throttling level 0 helps SPAC during the phase changes. For example, in case of `milc`, there are phase changes that cause zero prefetch-benefits for a period of 50M LLC misses. In this case, SPAC throttles down `milc` to the level 0 (no prefetching), whereas HPAC continues prefetching with prefetch degree of 1.

3. Engineering cost and effort: FDP, the local controller of HPAC requires tuning of six per-core thresholds [6], HPAC [2] requires tuning of additional four per-core thresholds, and PFST requires tuning of additional seven thresholds for FST [8], which leads to tuning of 17 thresholds. When we change the system parameters (such as LLC size, MSHR entries, and DRAM row size), re-tuning procedure becomes non-trivial and demands significant engineering cost and effort. On the other hand, SPAC uses only two thresholds: apc-threshold and the value of K that is used with the phase-change detector. Note that we do not add the effort required to set the window length, which is common in both HPAC and SPAC.

8.5 Resilient SPAC

Apart from stream prefetcher, we apply SPAC on GHB-PC/DC and AMPM prefetching techniques. Compared to HPAC, SPAC provides an average improvement of 11%, and 9.8% for GHB-based and AMPM prefetchers, respectively. Please note that, we chose AMPM as AMPM outperforms one of the recent techniques, called sandbox prefetching [19] on 70.8% (131/185) of the workloads. However, Sandbox is better for single-core workloads. Similarly, we study the effect of LLC replacement policies such as LRU and SHiP [20], and DRAM scheduling policies such as TCM [21] and BLISS [22]. The effectiveness of SPAC remains unaffected, and it provides at-least 9% performance improvement over HPAC.

7. Number of phases needed to match with the OPT's combinations.

8.6 Contribution of individual components

Figure 6 shows the components that are responsible for better performance improvement with SPAC when compared with HPAC. On an average, the usage of synergistic combination provides a contribution of 72.4% on the overall performance improvement. Phase-change detector contributes marginally (5%). For this experiment, we run SPAC by removing one selected component, and find the difference in performance when it runs with all five components. For example, to find out the contribution of synergistic combination, we run SPAC by selecting a random combination as the best combination, and find the difference in performance when we use synergistic combination.

8.7 Comparison with OPT

In this section, we explain the reasons behind the performance gap between SPAC and OPT. On an average, across 185WLs, SPAC's synergistic combinations matches with the OPT's synergistic combinations for 63% of the time. There are WLs, where SPAC matches with the OPT's synergistic combinations for 94% of the time but there are also WLs for which SPAC matches to the OPT's combinations for 49% of the time. This difference results in the fair-speedup gap between SPAC and OPT. There are two primary reasons for this: (i) phase-change behavior, and (ii) foothills and plateaus.

1. Phase-change behavior: Though SPAC uses a phase-change detector, it still suffers from the phase changes in the applications. Figure 7a shows a 4-core WL, where a phase-change of `leslie3d` (17th implementation phase of Figure 7a) causes a gap between the performance of SPAC and OPT. In the worst case, a sudden phase change in the applications' behavior causes SPAC to take four to five implementation phases to recover. The primary reason for this behavior is that, the average accuracy of the phase-change detector is 88.57%. On an average, the detector detects the phase change once in 61 exploration-phases and an inaccurate detector causes a maximum 3.2% reduction in the SPAC's performance.

2. Foothills and plateaus: We find, the prefetch-benefit curves of most of the applications when run along with other applications, are *non-convex* [8]. A prefetch-benefit curve is a curve that shows the change in the prefetch-benefits with the change in the throttling levels - from lower level to the higher level. As the WLs exhibit non-convexity, there are multiple foothills and plateaus in the exploration space. *Foothills* are the local maximas at which all the throttling combinations explored by SPAC provide less prefetch-benefits than the current throttling combination but there are other combinations that provide more prefetch benefits, which SPAC does not explore. Similarly *plateaus* are the local maximas at which all the throttling combinations of a given exploration phase provide same prefetch benefits. Figure 7b shows one of the instances of foothills and plateaus and how it affects SPAC's performance. Note that a phase-change detector won't be helpful in this case as all the applications would be running in their respective stable phases. So if a WL is stuck in the local maxima then the *recovery time* (time taken in terms of exploration phase to come out of the local maxima) is the key. In the worst case, SPAC gets stuck in the local maxima till the next phase-change of an application or till a different grouping of applications is created (which is 7 exploration phases, on an average).

To minimize the number of occurrences in which SPAC gets stuck at the local maximas, we try to provide the notion of history for each combination. For a given combination, we calculate the prefetch-benefit by finding the average of the prefetch-benefits of current exploration phase and the previous exploration phase. This modification does help some of the WLs but overall provides a marginal improvement of 0.85% over the proposed SPAC.

The good ones: There are WLs of type M2, M4, and M10, where SPAC bridges the gap between itself and OPT significantly. These WLs contain mostly the applications that show *convexity* and have only one or two local maximas in the entire execution of the WL.

8.8 Sensitivity and Scalability Analysis

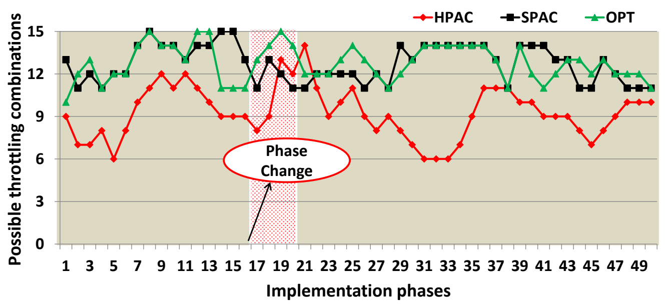
Why 2-groups works well? Figure 8a shows the fraction of synergistic combinations selected by HPAC and SPAC normalized to OPT. Figure 8b shows the corresponding benefits in terms of fair-speedup normalized to no prefetching. From Figure 8a, we can see that HPAC is not able to select 40% of the OPT's combinations, whereas SPAC is able to select more than 60% of the OPT's combinations. To understand the utility of the grouping process, we create 3 groups and 4 groups, and compare it with the 2 groups. To create 3 and 4 groups, we divide PPM into 3 and 4 zones, respectively.

From Figure 8a and Figure 8b, we can see that 4-groups is able to explore more synergistic combinations for 4-core, 8-core, and 12-core systems. However, the selection of more synergistic combinations does not translate into the improvement in fair-speedup. The primary reason for this is the *exploration time* (time spent in the exploration phase). Note that this is a shortcoming of the implementation and a better implementation of the proposed idea could result in better performance gains.

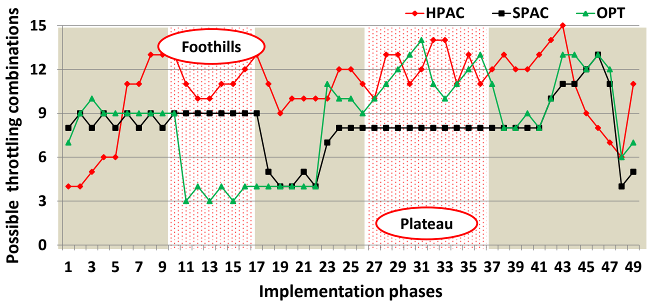
In case of 2-groups, at the beginning of an exploration phase, SPAC selects the synergistic combination in 5 exploration intervals. That is the system goes through four non-synergistic combinations (as one out of five is the synergistic combination) to find one synergistic combination. Similarly, in case of 4-groups, the system goes through 17 exploration intervals, which forces the system to be in the less-synergistic combination for 16 exploration intervals (40K LLC misses), which is huge.

Sensitivity to the system parameters: For sensitivity studies, we analyze the effect of change in LLC size from 8MB to 16MB and 32MB, and at the DRAM, the effect of change in the DRAM row size and number of DRAM banks. We also vary the number of DRAM controllers. Compared to HPAC, SPAC provides a performance improvement of 14% and 15.8% for 16MB and 32MB LLC, respectively. We observe a similar behavior for changes in DRAM parameter with an improvement of more than 12.7% across different configurations. The primary reason behind this improvement is the efficient usage of the DRAM

8. A prefetch-benefit curve is convex if it lies on one side of each of its tangent lines [23].

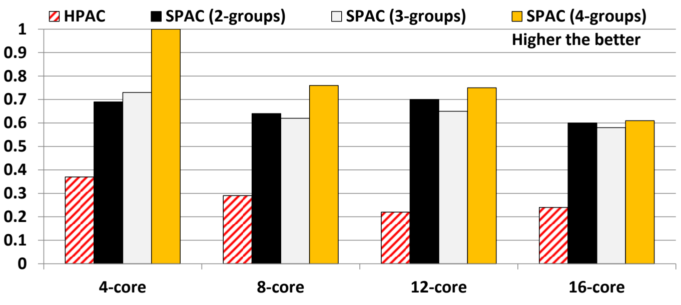


(a) Effect of a phase-change on SPAC in a WL that contains `lesli3d-gemsFDTD-bzip2-lbm`.

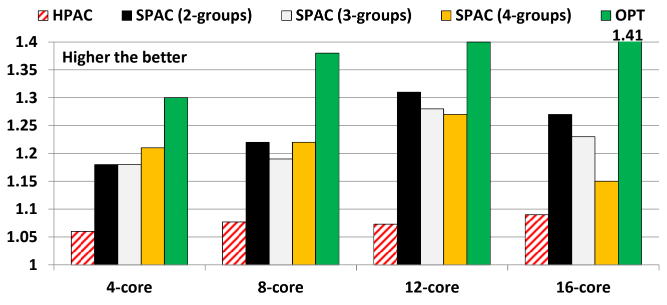


(b) Effect of foothills and plateaus on SPAC in a WL that contains `omnetpp-mcf-hmmer-gromacs`.

Figure 7: Throttling combinations over multiple implementation phases. Y-axis: 16 possible throttling combinations for a 4-core WL, which we visualize as a 4-bit binary string (values ranging from 0 to 15), with 0 corresponds to throttling down the prefetchers of all the cores (↓↓↓↓), and 15 corresponds to throttling up the prefetchers of all the cores (↑↑↑↑). Local maximas and phase-changes cause a performance difference of ≥3% between SPAC and OPT. In all other cases, the difference is < 3%.



(a)



(b)

Figure 8: (a) Fraction of optimal exploration space covered by HPAC and SPAC, normalized to OPT. (b) Comparison of HPAC, SPAC, and OPT in terms of fair-speedup, normalized to no prefetching.

bandwidth.

Effect of one more throttling decision: Apart from ↑ and ↓, we also explore one more throttling decision, which is *"to stay at the same level"* (|). With this addition, SPAC explores nine throttling combinations (2^3 + 1 combination communicated by individual prefetchers). 2^3 combinations come from 3 possible decisions for groups G0 and G1. We find that the usage of five combinations outperform nine combinations in 172 WLs.

Inaccurate prefetch requests: SPAC handles inaccurate prefetch requests through PADC that drops and de-prioritizes the inaccurate prefetch requests at the DRAM controller. For a system without PADC, SPAC can use prefetch-benefit as a replacement of accuracy for de-prioritization.

Importance of implementation-exploration ratio: In our simulations, we use 3 as the implementation-exploration ratio. We perform a study on the effect of this ratio on system performance and we find that only 13 out of 100 WLs provide better performance improvement when we change this ratio to 4 and only 6 WLs show better performance improvement when we use 5 as the ratio. We also try 1 and 2 as our ratios and find the ratio of 3 outperforms them.

Accuracy of our Rationale: SPAC relies on the rationale that the best combination of an exploration phase remains best in the implementation phase also. This rationale is accurate for 82%, 79%, 73%, and 77% of the time for 4-, 8-, 12-, and 16-core systems, respectively. This rationale helps SPAC in bridging the gap between HPAC and the OPT.

Accuracy of our estimation: The estimation of prefetch-benefit^{alone} for G1 and G2 is accurate for 78% of the time with the max/min error in estimation is 17%/0.8%. We find the maximum error in estimation for WL mix of `M12` category that contains applications such as `ammp` and `omnetpp`.

We also quantify the sensitivity of the prefetch-benefit^{alone} by varing the LLC size, number of cores, and the length of the exploration phase. Prefetch-benefit^{alone} is less sensitive to the LLC size. We sweep through LLC sizes of 512KB/core, 1MB/core, 2MB/core and 4MB/core and find that there is marginal differences in the accuracy of the estimation. However, increase in the core count from 4-core to 16-core increases the estimation error by 3.1%. Also, exploration phases that are too small (less than 512 LLC misses) and too high (greater than 10K LLC misses) increase the estimation error. We find similar conclusions for the length of implementation phase.

Effect on homogeneous and parallel workloads: We test the effectiveness of SPAC for homogeneous workloads by running multiple copies of the same applications with two variations: (i) same input sets and (ii) different input sets. The effectiveness of SPAC remains the same. However, in certain WLs, it degrades the performance marginally (<1.5%), when compared to the heterogeneous workloads. The primary reason for this behavior is that a major portion of SPAC's grouping process creates only one group (either G0 or G1). To mitigate this, we use the one-versus-all approach, where we assign one prefetcher to G0 and rest to G1. Also, there are WLs where all the throttling combinations provide same PFS. In those cases, we choose the throttling combination with the lowest-levels of throttling as it provides the same prefetch-benefits with less interference (because of low throttling levels). SPAC can be applied directly to the parallel workloads. However, for data parallel workloads, it has to use one-versus-all approach. One of the promising extensions of SPAC for parallel workloads will be to explore the notion of critical threads as mentioned in TCPT [24], and create groups based on the thread criticality.

Scalability: We study the scalability of SPAC by running 5 32-core WLs, 3 64-core WLs, and a single 128-core WL, where prefetcher-caused inter-core interference is high. SPAC provides an average improvement of 21.2% with a maximum of 48%. For 32-core, 64-core, and 128-core WLs, we simulate a system that consist of 2, 4, and 8 clusters (16 cores/cluster sharing an LLC) and we apply SPAC on a cluster. Also the hardware overhead that comes from SPAC is negligible to the LLC size.

9 Related Work

This section describes the most closely related prior works excluding HPAC. Albericio et al.'s ABS [25] is a prefetcher aggressiveness controller for multi-ported multi-banked LLC, where each LLC bank has a hardware prefetcher. It controls the aggressiveness of each bank. We compare ABS with SPAC by finding out the prefetch-benefits of each bank and then exploring the 5 combinations to decide the throttling levels for each bank. On an average, SPAC outperforms ABS by 9.7% for 4-core WLs. Jiménez et al. [26] propose a technique that uses a per-core configuration status register (CSR) to inform the OS about the different prefetch configuration settings. The OS explores all the possible configuration settings and chooses the best setting for each individual core. This work is a software based technique which is orthogonal to our work. Similarly, Bandwidth-shifting [27] is a software-based technique that is orthogonal to SPAC. Prior proposals such as Zhuang et al.'s filtering mechanism [28] uses cache pollution as the metric to control the aggressiveness. Wu and Martonosi [29] characterize cache pollution in the real system and propose a prefetch manager that controls the aggressiveness at run time. Liu and Solihin [30] propose an analytical model to study the interaction of hardware prefetching and bandwidth partitioning on multi-cores.

Recently, prefetching techniques such as sandbox [19] and best-offset [31] have been proposed that aim for different aspects of prefetching. However, their net effects have some overlapping with SPAC. In the results section, we show that the proposed technique is effective even for the better tuned prefetchers such as sandbox. Note that all these techniques do not consider fair-speedup as their performance metric, which is an important metric for the future many-core system. The best-offset prefetcher prefetches with degree 1 only, and in a multi-core system, SPAC can be used to turn-ON and turn-OFF the prefetchers based on the PFS metric.

One of the important components of SPAC is the proxy metric (PFS) that estimates fair-speedup in the presence of hardware prefetching. As fairness is related to the slowdown of an application, techniques that estimate the slowdown of an application in the presence of hardware prefetching can be applied to SPAC. A recent work called ASM [32] estimates the slowdown of an application at the shared resources by estimating the corresponding cache access rate (CAR). We believe, a slowdown-aware SPAC based on the CAR metric is an interesting problem to explore.

10 Conclusions

We presented, synergistic prefetcher aggressiveness controller (SPAC), a new prefetcher aggressiveness controller that outperforms the state-of-the-art controllers in terms of fair speedup. SPAC achieves this by grouping prefetchers into two groups (aggressive and meek), and exploring only five throttling combinations in an exploration phase. We also presented a metric called PFS that helps to find out the synergistic throttling combination.

Experimental evaluations showed that, With negligible hardware overhead, SPAC results in an average improvement of 15.3% across 185 multi-core workloads and outperforms the state-of-the-art technique for all the workloads in terms of fair-speedup, weighted-speedup, system fairness, and bandwidth consumption.

We conclude that SPAC is an effective, low-cost, and resilient prefetcher aggressiveness controller that optimize fair speedup in multi-core systems.

Acknowledgements

We would like to thank the anonymous reviewers for their useful comments. We would like to thank Prof. Andre Seznec, Dr. Rupesh Nasre, and the members of PACE and RISE lab of IIT Madras for their detailed comments that helped us significantly in improving the quality of the paper.

References

[1] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in SMT processors," in *2001 IEEE International Symposium on Performance Analysis of Systems and Software, November 4 - 6,2001, Tucson, Arizona, USA, Proceedings*, pp. 164–171, 2001.

[2] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 316–326, ACM, 2009.

[3] J. Doweck, "Inside intel core microarchitecture and smart memory access, intel technical white paper," 2006.

[4] 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*, PACT ’04, (Washington, DC, USA), pp. 135–145, IEEE Computer Society, 2004.

[5] Y. Ishii, M. Inaba, and K. Hiraki, “Access map pattern matching for high performance data cache prefetch,” *J. Instruction-Level Parallelism*, vol. 13, 2011.

[6] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, “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, (Washington, DC, USA), pp. 63–74, IEEE Computer Society, 2007.

[7] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, “Prefetch-aware shared resource management for multi-core systems,” in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA ’11, (New York, NY, USA), pp. 141–152, ACM, 2011.

[8] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, “Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems,” in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 335–346, ACM, 2010.

[9] “Standard performance evaluation corporation. SPEC CPU2006, SPEC CPU 2000,”

[10] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, “MISE: providing performance predictability and improving fairness in shared main memory systems,” in *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013*, pp. 639–650, 2013.

[11] D. Wang and X.-H. Sun, “APC: a novel memory metric and measurement methodology for modern memory systems,” *IEEE Trans. Comput.*, vol. 63, pp. 1626–1639, July 2014.

[12] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer, “PACMan: prefetch-aware cache management for high performance caching,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 442–453, ACM, 2011.

[13] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, “Prefetch-aware dram controllers,” in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, (Washington, DC, USA), pp. 200–209, IEEE Computer Society, 2008.

[14] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, Aug. 2011.

[15] R. Manikantan, K. Rajan, and R. Govindarajan, “Probabilistic shared cache management (PriSM),” in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA ’12, (Washington, DC, USA), pp. 428–439, IEEE Computer Society, 2012.

[16] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez, “B-fetch: Branch prediction directed prefetching for chip-multiprocessors,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (Washington, DC, USA), pp. 623–634, IEEE Computer Society, 2014.

[17] A. Snavely and D. M. Tullsen, “Symbiotic job scheduling for a simultaneous multithreaded processor,” in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, (New York, NY, USA), pp. 234–244, ACM, 2000.

[18] O. Mutlu and T. Moscibroda, “Stall-time fair memory access scheduling for chip multiprocessors,” in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, (Washington, DC, USA), pp. 146–160, IEEE Computer Society, 2007.

[19] P. et al., “Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers,” in *Proceedings of the 20th International Conference on High Performance Computer Architecture*, HPCA ’14, (Washington, DC, USA), IEEE Computer Society, 2014.

[20] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer, “Ship: Signature-based hit predictor for high performance caching,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 430–441, ACM, 2011.

[21] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, “Thread cluster memory scheduling: Exploiting differences in memory access behavior,” in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’43, (Washington, DC, USA), pp. 65–76, IEEE Computer Society, 2010.

[22] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, “The blacklisting memory scheduler: Achieving high performance and fairness at low cost,” in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pp. 8–15, Oct 2014.

[23] A. Gray, *Modern Differential Geometry of Curves and Surfaces with Mathematica*. Boca Raton, FL, USA: CRC Press, Inc., 1st ed., 1996.

[24] B. Panda and S. Balachandran, “TCPT: thread criticality-driven prefetcher throttling,” in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT ’13, (Piscataway, NJ, USA), pp. 399–400, IEEE Press, 2013.

[25] J. Albericio, R. Gran, P. Ibáñez, V. Viñals, and J. M. Llabería, “ABS: A low-cost adaptive controller for prefetching in a banked shared last-level cache,” *ACM Trans. Archit. Code Optim.*, vol. 8, pp. 19:1–19:20, Jan. 2012.

[26] V. Jiménez, 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 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT ’12, (New York, NY, USA), pp. 137–146, ACM, 2012.

[27] V. Jiménez, A. Buyuktosunoglu, P. Bose, F. P. O’Connell, F. J. Cazorla, and M. Valero, “Increasing multicore system efficiency through intelligent bandwidth shifting,” in *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pp. 39–50, 2015.

[28] X. Zhuang and H.-H. Lee, “A hardware-based cache pollution filtering mechanism for aggressive prefetches,” in *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pp. 286–293, Oct 2003.

[29] C. Wu and M. Martonosi, “Characterization and dynamic mitigation of intra-application cache interference,” in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2011, 10-12 April, 2011, Austin, TX, USA*, pp. 2–11, 2011.

[30] F. Liu and Y. Solihin, “Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors,” in *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS ’11, (New York, NY, USA), pp. 37–48, ACM, 2011.

[31] P. Michaud, “Best-Offset Hardware Prefetching,” in *International Symposium on High-Performance Computer Architecture*, (Barcelona, Spain), Mar. 2016.

[32] 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*, MICRO-48, (New York, NY, USA), pp. 62–75, ACM, 2015.

Biswabandan Panda received Ph.D. degree in computer science and engineering from the Indian Institute of Technology Madras, in 2015. He is currently a post-doc researcher at INRIA, Rennes. His current research interests are hardware prefetching techniques, memory compression techniques, and shared resource management techniques for CMPs.