CAFFEINE: A Utility-Driven Prefetcher Aggressiveness Engine for Multicores

BISWABANDAN PANDA and SHANKAR BALACHANDRAN,
Indian Institute of Technology Madras

Aggressive prefetching improves system performance by hiding and tolerating off-chip memory latency. However, on a multicore system, prefetchers of different cores contend for shared resources and aggressive prefetching can degrade the overall system performance. The role of a prefetcher aggressiveness engine is to select appropriate aggressiveness levels for each prefetcher such that shared resource contention caused by prefetchers is reduced, thereby improving system performance. State-of-the-art prefetcher aggressiveness engines monitor metrics such as prefetch accuracy, bandwidth consumption, and last-level cache pollution. They use carefully tuned thresholds for these metrics, and when the thresholds are crossed, they trigger aggressiveness control measures. These engines have three major shortcomings: (1) thresholds are dependent on the system configuration (cache size, DRAM scheduling policy, and cache replacement policy) and have to be tuned appropriately, (2) there is no single threshold that works well across all the workloads, and (3) thresholds are oblivious to the phase change of applications.

To overcome these shortcomings, we propose CAFFEINE, a model-based approach that analyzes the effectiveness of a prefetcher and uses a metric called net utility to control the aggressiveness. Our metric provides net processor cycles saved because of prefetching by approximating the cycles saved across the memory subsystem, from last-level cache to DRAM.

We evaluate CAFFEINE across a wide range of workloads and compare it with the state-of-the-art prefetcher aggressiveness engine. Experimental results demonstrate that, on average (geomean), CAFFEINE achieves 9.5% (as much as 38.29%) and 11% (as much as 20.7%) better performance than the best-performing aggressiveness engine for four-core and eight-core systems, respectively.

CCS Concepts: • **Computer systems organization → Parallel Architectures;** *Processors and memory architectures;*

Additional Key Words and Phrases: Prefetching, multicore, memory systems, intercore interference, cache pollution

ACM Reference Format:
Biswabandan Panda and Shankar Balachandran. 2015. CAFFEINE: A utility-driven prefetcher aggressiveness engine for multicores. ACM Trans. Archit. Code Optim. 12, 3, Article 30 (August 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/2806891

1. INTRODUCTION

Hardware prefetching plays an important role in improving system performance. A prefetcher prefetches data into a cache before the processor demands the data and hence improves system performance. But a prefetcher can also degrade system performance because of interference at the shared resources, such as last-level cache

Biswabandan Panda is supported by a TCS Ph.D. fellowship.
Authors' addresses: B. Panda, Department of Computer Science and Engineering, IIT Madras, Chennai, India, 600036; email: biswa@cse.iitm.ac.in; S. Balachandran, BSB 349, Department of Computer Science and Engineering, IIT Madras, Chennai, India, 600036; email: shankar@cse.iitm.ac.in.

DOI: http://dx.doi.org/10.1145/2806891

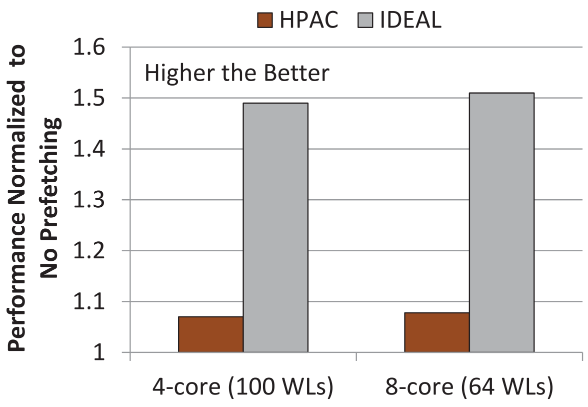


Fig. 1. Performance improvement (geometric mean of harmonic speedups) for four-core and eight-core system.

(LLC), DRAM controller, and DRAM. The interference can happen between the prefetch requests of one core and the demand requests of the other cores (*prefetch–demand* interference) and between the prefetch requests of one core and the prefetch requests of the other cores (*prefetch–prefetch* interference).

Prefetcher aggressiveness engines, on multicore systems, try to improve the system performance by controlling the aggressiveness of each prefetcher. Aggressiveness of a prefetcher is described using two factors: how far ahead of the demand access stream[1] the prefetch requests are issued (*prefetch distance*) and the number of prefetch requests issued in one instant (*prefetch degree*). A prefetcher aggressiveness engine controls the aggressiveness of a prefetcher by selecting an appropriate aggressiveness level. Each level consists of two important prefetching knobs: the prefetch degree and the prefetch distance. The aggressiveness engine's job is to switch between the aggressiveness levels of the prefetchers by *throttling* (*throttling up/down refers to the increase/decrease in the aggressiveness by one level*).

Opportunity: To illustrate the importance of an aggressiveness engine, Figure 1 shows the performance improvement that can be achieved with the *ideal but unrealizable* multicore system. The ideal system would not have prefetcher-caused intercore interference at the shared resources. To model the *ideal* system, we eliminate the prefetcher-caused intercore interference at the LLC, DRAM request queue, DRAM bus, DRAM banks, and DRAM row-buffers by making the prefetcher-caused intercore interference latency zero cycle. Compared to no prefetching, on average (geomean), the ideal system improves the performance by 49% and 51% for four- and eight-core systems, respectively. In contrast, the state-of-the-art prefetcher aggressiveness engine called the hierarchical prefetcher aggressiveness controller (HPAC) [Ebrahimi et al. 2009] improves the performance by 7% and 7.8%, respectively. This experiment shows that there is opportunity to bridge the gap between the state-of-the-art technique and the ideal multicore system.

The Problem: State-of-the-art aggressiveness engines such as HPAC [Ebrahimi et al. 2009] use thresholds for various prefetching metrics such as accuracy, cache pollution, and off-chip bandwidth consumption. The effectiveness of threshold-driven engines depends on the system configuration (such as LLC replacement policy, DRAM scheduling policy) and has to be tuned appropriately. Also, there is no fixed set of thresholds that work well across all the workloads, and thresholds are oblivious to

[1]Sequence of cache block aligned memory addresses.

the phase-change behavior of applications. In effect, these techniques fail to identify certain scenarios where the prefetcher-caused intercore interference is significant and loses opportunities for performance improvement.

Our Goal: *Our goal is to design a prefetcher aggressiveness engine, which can improve the system performance by (1) tolerating prefetcher-caused intercore interference as long as performance is improved and (2) minimizing the prefetcher-caused intercore interference that affects performance.*

Our Approach: In this work, we propose CAFFEINE,[2] a utility-driven prefetcher aggressiveness engine that is based on the *buffet principle* [Mahajan et al. 2008], to "continue using more resources as long as the marginal cost can be driven lower than the marginal benefit." CAFFEINE advocates the application of the *buffet principle* in controlling the aggressiveness of prefetchers on a multicore system. CAFFEINE continues increasing the aggressiveness if such a decision is likely to improve overall system performance. We propose a metric called net utility (utility$_{net}$), which quantifies the net processor cycles saved by a prefetcher. We use this metric to divide the prefetchers into two groups: *affecting* and *affected*. Our technique throttles down the *affecting* prefetchers and throttles up the *affected* prefetchers if they are likely to improve the system performance. We make these throttling decisions without using any threshold. CAFFEINE uses CAFFEINATION when the prefetcher-caused interference is *tolerable* (we define in Section 3.1) and it uses DE-CAFFEINATION when the prefetcher-caused interference is *intolerable*. Both these techniques use utility$_{net}$ to make throttling decisions.

Key Idea: Our idea is based on this observation: "different levels of prefetcher aggressiveness provide different net utilities." At a given instant of time, CAFFEINE tries to maximize the utility$_{net}$ of an entire prefetching unit[3] of a multicore system.
We make the following contributions:

1. We propose a metric called utility$_{net}$ to measure the utility of hardware prefetchers. utility$_{net}$ indicates the net processor cycles saved because of prefetching at a given aggressiveness level (Section 3.1).

2. We design a model-based utility-driven prefetcher aggressiveness engine for multicore systems, called CAFFEINE, which uses utility$_{net}$ (Section 3.2).

3. We evaluate CAFFEINE on four- and eight-core systems. We show the effectiveness of CAFFEINE by comparing it with the HPAC [Ebrahimi et al. 2009]. For four- and eight-core systems, compared to HPAC, CAFFEINE improves performance (geomean of harmonic speedups) by 9.5% and 11% across 100 and 64 workloads, respectively (Section 6).

2. BACKGROUND

2.1. Baseline Multicore Design

Figure 2 shows our baseline multicore system, which consists of N cores and K memory controllers. Each core has a private L1 cache and a private L2 cache. The LLC (L3) is shared among all the cores. Each core has a stream prefetcher [Tendler et al. 2002] at the L3 cache, which sends prefetch requests to the DRAM. All the per-core prefetchers generate prefetch requests, which are buffered in a prefetch queue (PFQ). Prefetch requests enter into the PFQ after probing into the cache. The prefetcher does not insert a prefetch request if it gets a hit at the cache. PFQ is a FIFO queue where the

[2]CAFFEINE is a stimulant; when taken in a moderate amount, it helps in boosting the alertness of the nervous system, but high consumption of CAFFEINE causes restlessness. We envision CAFFEINE is analogous to prefetcher aggressiveness—too much aggressiveness is sometimes harmful and too little does not "stimulate."

[3]A prefetching unit consists of all the prefetchers of a multicore system.

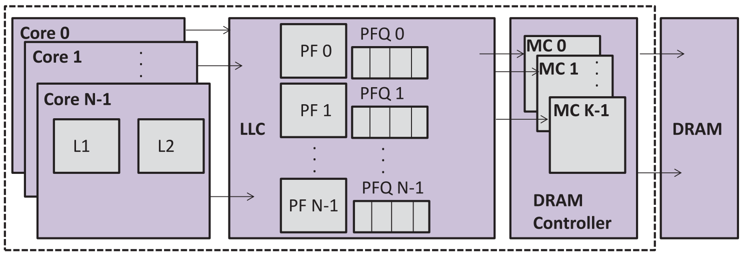


Fig. 2. Baseline multicore organization.

oldest prefetch request is sent to the miss status holding register (MSHR) allocators. The prefetch requests are looked up in the cache before inserting them into the MSHRs. The PFQ is similar to the prefetch buffer of the Intel Core processor [Doweck 2006]. On a demand access, the prefetcher issues prefetch requests by dequeuing the predicted addresses from the PFQ, and these requests enter the MSHRs (not shown in Figure 2). The baseline MSHR prioritizes demand requests over prefetch requests and LLC fill buffers are fused with the MSHRs.

2.2. Hardware Prefetchers

In this work, we use a stream prefetcher, which is similar to the prefetchers of IBM's POWER 4/5 [Tendler et al. 2002; Srinath et al. 2007; Ebrahimi et al. 2009]. It is a low-cost hardware prefetcher that works well across a large number of applications. A stream prefetcher keeps track of multiple access streams. Once trained, the stream prefetcher issues K requests at a time where K is the prefetch degree. We also evaluate our technique with a GHB-based prefetcher [Nesbit et al. 2004].

2.3. Hierarchical-Prefetcher-Aggressiveness-Controller

To the best of our knowledge, for multicore systems (with N cores), HPAC is the only technique that tries to minimize the prefetcher-caused intercore interference by controlling the prefetcher aggressiveness. HPAC is an extension of feedback-directed prefetching (FDP) [Srinath et al. 2007] and it works in two layers: local throttling and global throttling. The per-core local throttling mechanism uses FDP [Srinath et al. 2007], which tries to improve the single-core performance by throttling the prefetcher, based on the prefetch metrics such as accuracy ($\frac{\#\text{pfhits}}{\#\text{pfissued}}$), cache pollution,[4] and lateness ($\frac{\#\text{lateprefetches}}{\#\text{pfhits}}$). The global throttling mechanism collects the feedback on the bandwidth consumption of core i (BWC_i) , the bandwidth needed by all the cores except core i ($BWNO_i$), and the core i's prefetcher-caused intercore LLC pollution (POL_i). It also uses the prefetch accuracy of core i (ACC_i) in its decision process. For each of these four metrics, finely tuned thresholds place the value as high and low, giving rise to 16 (2^4) possible cases, and assigns a throttling decision for each case. The global throttling mechanism uses a global controller and it overrides (if needed) the decisions of the local controller to minimize the prefetcher-caused intercore interference. HPAC uses five different levels of aggressiveness ($<1, 4>$, $<1, 8>$, $<2, 16>$, $<4, 32>$, and $<4, 64>$), where $<x,y>$ corresponds to $<$prefetch degree, prefetch distance$>$. It uses the following thresholds: for high ACC, 0.60; high POL, 90; high BWC, 50k; and high BWN, 75k [Ebrahimi et al. 2009].

[4]A prefetched block evicting a cache block that would otherwise be reused in the near future.

2.4. SOURCES of PREFETCHER-CAUSED INTERFERENCE

This section describes the sources of prefetcher-caused intercore interference at the shared resources and how to quantify the same. First, we describe the interference at the LLC that causes cache pollution. Second, we explain the interference at the DRAM banks and row buffers. Finally, we look into the interference that happens at the DRAM bus. In all the cases described next, $0 \leq j \leq N-1$ and $j \neq i$, where N is the total number of cores in the system.

LLC POLLUTION (POLL): LLC pollution can happen when an LLC block of core j is evicted by a prefetched block of core i and the evicted block is requested by core j in the near future. We quantify it using $\boldsymbol{Poll_i}$. $\boldsymbol{Poll_i}$ captures the number of additional LLC misses of other cores that happen because of evictions of their cache blocks by prefetched blocks of core i. For core j, an additional demand miss at the LLC contributes an additional latency of α cycles (LLC miss penalty of core j).

BANK-LEVEL INTERFERENCE (BLI): This can happen when demand requests and useful prefetch requests (which are likely to be hit at the LLC) of core j wait, because a DRAM bank is occupied by a prefetch request P of core i. This incurs a latency. We quantify this interference that happens because of a prefetcher of core i by counting it using the metric $\boldsymbol{Bli}_i$. This interference incurs an approximate latency of $\beta = \frac{latency(P)}{BankWaitingParallelism}$ on core j, where latency (P) is the DRAM latency of prefetch request P and *BankWaitingParallelism* corresponds to core j's requests that are waiting to get service from K DRAM banks. If *BankWaitingParallelism* is K, then core j is waiting to get service from K number of DRAM banks. We divide $latency(P)$ by *BankWaitingParallelism* as $latency(P)$ will be amortized across concurrent requests of core j that are waiting to get serviced.

ROW BUFFER CONFLICT (RBC): Row buffer conflicts can happen when a prefetch request of core i causes *row conflicts* for core j's demand and useful prefetch requests. We use $\boldsymbol{Rbc}_i$ to count the same. This results in an approximate latency of $\gamma = \frac{t_{RP}+t_{RCD}}{BankAccessParallelism}$, where t_{RP} corresponds to latency for row precharge and t_{RCD} is the latency that comes from row to column delay of DRAM. *BankAccessParallelism* is the number of core j's requests that are currently getting serviced by the DRAM banks. We divide $t_{RP} + t_{RCD}$ by *BankAccessParallelism* as $t_{RP} + t_{RCD}$ will be amortized across concurrent requests of core j [Glew 1998].

DRAM BUS INTERFERENCE (DBI): This interference happens when a demand and useful prefetch request of core j has to wait because the DRAM bus is occupied by a prefetch request of core i. We use $\mathbf{Dbi}_i$ to count the same. This interference incurs a latency of ζ cycles, which is approximately $\frac{BL}{2}$ cycles (for read, write, and prefetch request), where BL is the burst length.

So a prefetcher of one core interferes with the other cores at the LLC, DRAM banks, DRAM row-buffers, and DRAM bus, resulting in approximate penalties of α, β, γ, and ζ cycles, respectively. Our definitions of BLI, RBC, and DBI are similar to STFM [Mutlu and Moscibroda 2007], and our definition of POLL is similar to HPAC [Ebrahimi et al. 2009] and FST [Ebrahimi et al. 2010]. In the next section, we describe how we use these measures of prefetcher-caused interference in our proposed metric called utility$_{net}$.

3. CAFFEINE FRAMEWORK

3.1. DEFINITIONS AND METRICS

This section describes various metrics that we use in our CAFFEINE framework.

Utility$_{positive}$ of core i's prefetcher: It corresponds to the number of processor cycles saved by core i's prefetcher for core i and we define it as follows:

$$\text{utility}_{positive}(i) = \frac{\text{prefetch}_{hits}(i) * (\text{LLC}_{penalty}(i))}{\text{prefetch}_{issued}(i)}, \tag{1}$$

where prefetch$_{hits}$ is the number of LLC hits to the prefetched blocks and prefetch$_{issued}$ is the number of prefetch requests issued. LLC$_{penalty}(i)$ is the average LLC miss penalty of core i. Note that utility$_{positive}$(i) does consider the effects of memory-level parallelism (MLP) and bank-level parallelism (BLP) indirectly. A higher MLP or BLP leads to lower LLC$_{penalty}$(i).

Utility$_{negative}$ of core i's prefetcher: It corresponds to the number of additional processor cycles that other cores have to wait at the shared resources because of core i's prefetcher. We define it as

$$\text{utility}_{negative}(i) = \frac{(Poll_i * \alpha_i) + (Bli_i * \beta_i) + (Rbc_i * \gamma_i) + (Dbi_i * \zeta_i)}{\text{prefetch}_{issued}(i)}. \tag{2}$$

Utility$_{net}$ of core i: It corresponds to the contribution of a prefetcher (in terms of processor cycles saved) in the overall system performance. For a core i, the utility$_{net}$ of its prefetcher is utility$_{net}(i)$ = utility$_{positive}(i)$ − utility$_{negative}(i)$.

Cycles$_{affecting}$ and Cycles$_{affected}$ of core i: Cycles$_{affecting}$ corresponds to the number of cycles for which core i affects core j, where $0 \leq j \leq N-1$ and $j \neq i$. These metrics help in finding out the prefetchers that cause high interference at the shared resources and the cores that are affected by prefetcher-caused intercore interference.

Cycles$_{affected}$ corresponds to the number of cycles for which core i is affected (has to wait because of interference at the LLC pollution, RBC, and BLI) by core j's prefetcher, where $0 \leq j \leq N-1$ and $j \neq i$.

$$\text{cycles}_{affecting}(i) = (Poll_i * \alpha_i) + (Bli_i * \beta_i) + (Rbc_i * \gamma_i) + (Dbi_i * \zeta_i) \tag{3}$$

To calculate cycles$_{affected}(i)$, we use a register, which gets incremented by either $Poll_j{*}\alpha$ (because of POLL), $Bli_j{*}\beta$ (because of BLI), $Rbc_j{*}\gamma$ (because of RBC), or $Dbi_j{*}\zeta$ (because of DBI) when a demand request or a useful prefetch request of core i waits because of interference caused by the prefetcher of core j.

utility$_{totalnegative}$ and utility$_{totalpositive}$: For a multicore system with N processor cores, utility$_{totalnegative}$ = $\sum_{i=0}^{N-1}$utility$_{negative}(i)$ and utility$_{totalpositive}$ = $\sum_{i=0}^{N-1}$utility$_{positive}(i)$. utility$_{totalnegative}$ corresponds to the number of processor cycles the entire multicore system waits because of prefetching and utility$_{totalpositive}$ corresponds to the number of processor cycles saved for the entire multicore system because of prefetching. *If utility$_{totalnegative}$ ≥ utility$_{totalpositive}$, then the prefetcher-caused intercore interference is intolerable.*

△ utility$_{positive}$ and △utility$_{net}$ of core i's prefetcher: △utility$_{positive}$ corresponds to the effect of change in the aggr. (aggressiveness) levels on utility$_{positive}$. For a prefetcher of core i, at an interval t, we define it as follows:

$$\triangle\text{utility}_{positive}(i)_t = \frac{\text{utility}_{positive}(i)_t - \text{utility}_{positive}(i)_{t-1}}{\text{aggr. level}(i)_{t-1} - \text{aggr. level}(i)_{t-2}}. \tag{4}$$

If the denominator is zero, then

$$\triangle\text{utility}_{positive}(i)_t = \text{utility}_{positive}(i)_t - \text{utility}_{positive}(i)_{t-1}. \tag{5}$$

At an interval t, a prefetcher of core i benefits core i if utility$_{positive}(i)_t$ is positive and △utility$_{negative}(i)_t$ is nonnegative. Note, the denominator finds the difference in the aggressiveness levels at t– 1 and t– 2, but the numerator finds the difference in the utility$_{positive}$ at t and t– 1. The reason for this is the cause–effect relationship between an aggressiveness level and the corresponding utility$_{positive}$. A change in the aggressiveness level at t– 1 contributes to the utility$_{positive}$ at t. We define △utility$_{net}(i)_t$ similar

ALGORITHM 1: CAFFEINE

```
1: compute $\text{utility}_{totalpositive}$ and $\text{utility}_{totalnegative}$
2: if ($\text{utility}_{totalpositive} - \text{utility}_{totalnegative}) > 0$ then
3:    // Interference is tolerable.
4:    for all $i$, where $i$ is the the core-id do
5:       Throttling Direction[$i$] = CAFFEINATION (i) (Algorithm 2)
6:    end for
7: else
8:    // Intolerable interference.
9:    DE-CAFFEINATION (Algorithm 3)
10: end if
11: return Throttling Direction[0: $N - 1$]
```

to $\triangle\text{utility}_{positive}(i)_t$ as follows:

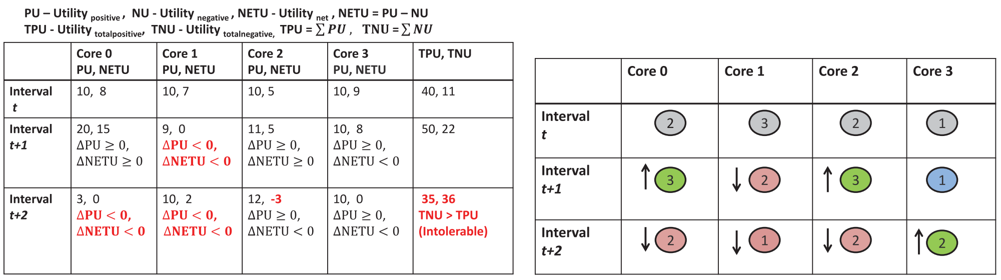
$$\triangle\text{utility}_{net}(i)_t = \frac{\text{utility}_{net}(i)_t - \text{utility}_{net}(i)_{t-1}}{\text{aggr. level}(i)_{t-1} - \text{aggr. level}(i)_{t-2}}. \tag{6}$$

Finally, a prefetcher of core i likely contributes to the overall system performance improvement if its $\text{utility}_{net}(i)_t > 0$.

3.2. CAFFEINE

CAFFEINE is an interval-based technique, which consists of two subtechniques: CAFFEINATION and DE-CAFFEINATION. Similar to HPAC [Ebrahimi et al. 2009], it uses five different levels of aggressiveness (level 1 to level 5) with <prefetch-degree, prefetch-distance> as <1, 4>, <1, 8>, <2, 16>, <4, 32>, and <4, 64>. A prefetcher that is at level 5/level 1 cannot be throttled up/throttled down. After a predetermined fixed interval, CAFFEINE computes $\text{utility}_{positive}$, $\text{utility}_{negative}$, utility_{net}, $\triangle\text{utility}_{positive}$, $\triangle\text{utility}_{net}$, $\text{utility}_{totalnegative}$, and $\text{utility}_{totalpositive}$ by collecting the interference count from the LLC controller and DRAM controllers. Based on these utility values, CAFFEINE makes throttling decisions.

Algorithm 1 describes CAFFEINE for an N-core system (core-id 0 to $N - 1$). Line 1 of Algorithm 1 computes the $\text{utility}_{totalpositive}$ and the $\text{utility}_{totalnegative}$. Line 2 of Algorithm 1 shows the condition at which the prefetcher-caused intercore interference is tolerable. If the condition at line 2 succeeds, then CAFFEINE uses CAFFEINATION or else DE-CAFFEINATION. **CAFFEINATION:** In CAFFEINATION, each prefetcher controls its aggressiveness based on Algorithm 2. CAFFEINATION takes a core-id of the prefetcher as the input and returns the throttling direction. The throttling directions are ↑, ↓, and ↔ and these directions correspond to throttling up, throttling down, and no throttling. CAFFEINATION throttles up (↑) a prefetcher if $\triangle\text{utility}_{positive} \geq 0$ and $\triangle\text{utility}_{net} \geq 0$. If a prefetcher shows $\triangle\text{utility}_{positive} < 0$, CAFFEINATION throttles it down (↓). *The rationale behind these decisions is that a prefetcher, which is able to save processor cycles for its own core and for the system, should get more opportunity to improve the performance further*. Note that we first check $\text{utility}_{positive}$ in line 2 and not utility_{net}. Putting utility_{net} in line 2 will lead to aggressive throttling in the CAFFEINATION phase, where interference is tolerable. If the conditions in line 2 fail, line 9 would throttle down a prefetcher always. The current algorithm conservatively allows a prefetcher to stay in the same aggressiveness level (line 6). The rationale is that if line 2 evaluates TRUE, then there is a chance that line 3 would evaluate to TRUE in subsequent windows. Line 6 of Algorithm 2 shows a special case, in which CAFFEINATION chooses no throttling (↔) for a prefetcher with $\triangle\text{utility}_{net} < 0$ and $\triangle\text{utility}_{positive} \geq 0$. The rationale behind this decision is that the prefetcher is able to save processor



PU – Utility $_{positive}$, NU - Utility $_{negative}$, NETU - Utility $_{net}$, NETU = PU – NU
TPU - Utility $_{totalpositive}$, TNU - Utility $_{totalnegative}$, TPU = $\sum PU$, TNU = $\sum NU$

	Core 0 PU, NETU	Core 1 PU, NETU	Core 2 PU, NETU	Core 3 PU, NETU	TPU, TNU
Interval *t*	10, 8	10, 7	10, 5	10, 9	40, 11
Interval *t+1*	20, 15 ΔPU ≥ 0, ΔNETU ≥ 0	9, 0 **ΔPU < 0,** **ΔNETU < 0**	11, 5 ΔPU ≥ 0, ΔNETU ≥ 0	10, 8 ΔPU ≥ 0, ΔNETU < 0	50, 22
Interval *t+2*	3, 0 **ΔPU < 0,** **ΔNETU < 0**	10, 2 **ΔPU < 0,** **ΔNETU < 0**	12, **-3** ΔPU ≥ 0, ΔNETU < 0	10, 0 ΔPU ≥ 0, ΔNETU < 0	**35, 36** **TNU > TPU** **(Intolerable)**

	Core 0	Core 1	Core 2	Core 3
Interval *t*	2	3	2	1
Interval *t+1*	↑ 3	↓ 2	↑ 3	1
Interval *t+2*	↓ 2	↓ 1	↓ 2	↑ 2

Fig. 3. An example illustrating CAFFEINE on a four-core system for interval t to $t+2$. For the definitions of $\triangle$ utility$_{positive}$ and $\triangle$ utility$_{net}$, refer to Section 3.1.

cycles for its own core but it is also interfering with the other cores. *CAFFEINE decides no throttling as the interference is tolerable. But if the interference is intolerable, DE-CAFFEINATION (Algorithm 3) overrides the decisions of CAFFEINATION*.

DE-CAFFEINATION: DE-CAFFEINATION kicks in when the system shows signs of performance degradation. DE-CAFFEINATION finds out the prefetchers that are responsible for intercore interference, prefetchers with high values of cycles$_{affecting}$, and calls them *affecting* prefetchers (we quantify in Section 4). Similarly, prefetchers with high values of cycles$_{affected}$ are called *affected* prefetchers. There are prefetchers that neither *affect* others nor are *affected* by others. Line 1 of Algorithm 3 finds out the *affecting* prefetchers and line 2 throttles down (↓). For all the prefetchers that do not belong to the *affecting* group, DE-CAFFEINATION throttles them using CAFFEINATION (Algorithm 2). Line 7 shows a special case in which DE-CAFFEINATION throttles up (↑) an *affected* prefetcher if the CAFFEINATION returns no throttling(↔).

3.3. CAFFEINE: AN EXAMPLE

This section provides an example that illustrates CAFFEINE on a four-core system. Figure 3 shows the utility$_{positive}$ and utility$_{net}$ of each prefetcher along with the utility$_{totalpositive}$ and utility$_{totalnegative}$ of the entire system. In Figure 3, we show the throttling directions for each prefetcher based on CAFFEINE. For ease of understanding, we assume the aggressiveness levels of core-0 to core-3 at interval $t-1$ (not shown in Figure 3) are 2, 3, 2, and 1, respectively. Also, we assume there is no change in the aggressiveness levels at interval t; that is, the aggressiveness levels of core-0 to core-3 remain at 2, 3, 2, and 1, respectively.

At interval $t+1$ (refer to the left panel of Figure 3), the system is *tolerable* to the prefetcher-caused intercore interference as the utility$_{totalpositive}$ (TPU) is 50 and the

ALGORITHM 2: CAFFEINATION (core-id $= i$)

```
1:  Input: $i$, Output: Throttling Direction($i$)
2:  if ((utility$_{positive}(i)_t > 0$) and ($\triangle$utility$_{positive}(i)_t \geq 0$)) then
3:      if ((utility$_{net}(i)_t > 0$) and ($\triangle$utility$_{net}(i)_t \geq 0$)) then
4:          Throttling Direction[$i$] = ↑
5:      else
6:          Throttling Direction[$i$] = ↔
7:      end if
8:  else
9:      Throttling Direction[$i$] = ↓
10: end if
11: return Throttling Direction[$i$]
```

Interval t+2	Core 0	Core 1	Core 2	Core 3
$Affecting_{cycles}$	3	8	**15**	10
$Affected_{cycles}$	7	5	5	**16**

Fig. 4. Snapshot of the interference at interval $t + 2$.

ALGORITHM 3: DE-CAFFEINATION

```
1:  Find the core-ids of *affecting* and *affected* prefetchers
2:  **for all** $i$, where $i$ is the the core-id **do**
3:    **if** $i$'s prefetcher is *affecting* **then**
4:      Throttling Direction[$i$] = $\downarrow$
5:    **else**
6:      Throttling Direction[$i$] = CAFFEINATION(i)
7:      **if** Throttling Direction[$i$] == ($\leftrightarrow$) and $i$ is *affected* **then**
8:        Throttling Direction[$i$] = $\uparrow$
9:      **end if**
10:   **end if**
11: **end for**
12: **return** Throttling Direction[0: $N - 1$]
```

$\text{utility}_{totalnegative}$ (TNU) is 22 and (50–22) $>$ 0. So CAFFEINE uses CAFFEINATION for all the prefetchers. For core-0 and core-2, $\triangle\text{utility}_{positive}$ and $\triangle\text{utility}_{net}$ are $\geq$ 0. CAFFEINE throttles up ($\uparrow$) both core-0 and core-2. For core-3, $\triangle\text{utility}_{positive} \geq 0$ but $\triangle\text{utility}_{net} < 0$. CAFFEINE does not change ($\leftrightarrow$) the aggressiveness level for core-3. For core-1, $\triangle\text{utility}_{positive} < 0$ and CAFFEINE throttles down ($\downarrow$) its prefetcher.

At interval $t + 2$, the system is *intolerable* to the prefetcher-caused intercore interference as the $\text{utility}_{totalnegative}$ (TNU) $>$ $\text{utility}_{totalpositive}$ (TPU) and thus CAFFEINE uses DE-CAFFEINATION for the system. The right panel of Figure 4 provides a snapshot of the prefetcher-caused intercore interference at the interval $t + 2$, which shows that the prefetcher of core-2 is an *affecting* prefetcher and core-3 is an *affected* prefetcher. CAFFEINE throttles down ($\downarrow$) the prefetcher of core-2. For core-0 and core-1, both $\triangle\text{utility}_{positive}$ and $\triangle\text{utility}_{net}$ are less than zero, so CAFFEINE throttles down ($\downarrow$) the prefetcher of core-0 and core-1. For core-3, $\triangle\text{utility}_{positive} \geq 0$ but $\triangle\text{utility}_{net} < 0$, and based on line 7 of Algorithm 3, DE-CAFFEINATION throttles up ($\uparrow$) the prefetcher of core-3.

4. IMPLEMENTATION DETAILS

This section provides the hardware implementation of CAFFEINE. We place the digital circuit of CAFFEINE beside the LLC with an access latency the same as that of LLC.

Interval Length: We set an interval of 8K LLC misses for CAFFEINE. We sweep through interval lengths of 2K, 4K, 8K, and 16K LLC misses and find that an interval length of 8K provides a good tradeoff. At the end of every 8K misses, the CAFFEINE circuit is connected to DRAM controllers, an LLC controller, and a prefetch controller and collects the relevant metrics to compute various utilities and resets these metrics to zero after making the throttling decisions. All these calculations are not in the critical path as the applications continue to run and their execution is oblivious to the time spent by CAFFEINE. Also, CAFFEINE makes a decision in a timely manner, which takes 0.5% of the interval length.

CAFFEINE Hardware Support: CAFFEINE needs combinational logic circuits for finding out the metrics of interest and comparing them with zero. For each core, CAFFEINE uses three multipliers, two dividers, two comparators, and three adders. We assume four cycles for multiplication, one cycle for addition, and eight cycles for division. CAFFEINE uses an *extreme outlier detection* [Outliers] algorithm to detect the *affected* and *affecting* prefetchers. It uses a deterministic circuit, which sorts the core-ids' $\text{cycles}_{\textit{affected}}$ and $\text{cycles}_{\textit{affecting}}$. Then it finds out the median, upper-quartile (UQ), lower-quartile (LQ), and inter-quartile range (IQR). $\text{IQR} = k \times (UQ - LQ)$, where k is 3, which we verify and adopt from Outliers [Outliers]. The circuit provides a range [LQ-IQR, UQ+IQR] and the core-ids whose affecting/affected values are greater than UQ+IQR are called *affecting/affected* prefetchers. All these calculations take 22 processor cycles.[5]

$\text{utility}_{\textit{positive}}$: To calculate $\text{utility}_{\textit{positive}}$, we augment each cache block at the LLC with a bit called *pfbit*. We use counters to count the number of prefetches issued (*pfissued*) and the number of prefetch hits (*pfhits*). To calculate α, we use two registers per core to store the number of misses and their corresponding miss penalties. We calculate the average miss penalty by dividing the total miss penalty by the total number of misses and assign it to α.

POLL: To calculate $Poll_i$ at the LLC, we use a per-core bloom filter [Bloom 1970], which is an XOR-based filter, consisting of 1,024 entries [Ebrahimi et al. 2009], and each entry contains a *pol* bit along with the core-id. *pol* is set when a core j's LLC block is evicted by a prefetched block of core i. In future, if a demand access from core j gets a hit at the bloom filter, we increment $Poll_i$. In future, we reset the *pol* bit of core i's filter once the DRAM finishes responding core j's request. Our Bloom filter implementation is the same as that of HPAC's Bloom filter.

RBC: To calculate Rbc_i, for each DRAM bank, we maintain a register called *last-row-access*, which stores the last row accessed by core j before a prefetch request of core i closes it. In the future, if a request from core j gets a hit at the *last-row-access* register of core i, we increment the counter for Rbc_i.

BLI: We calculate Bli_i by keeping an additional bit per bank that we set when a bank is busy serving a prefetch request of core i. If a demand or useful prefetch request of core j waits to get a response from the same DRAM bank, we increment Bli_i.

DBI: To calculate Dbi_i, whenever a prefetch request keeps the DRAM data bus busy, we use a register that stores the core-id of the prefetch request and we count such events using a per-core DBI counter.

A prefetch request of core j is useful if the prefetch accuracy of core j is ≥ 0.85 [Lee et al. 2008]. Our implementations of *Bli, Rbc*, and *Dbi* are similar to STFM [Mutlu and Moscibroda 2007], and our implementation of *Poll* is the same as HPAC [Ebrahimi et al. 2009] and FST [Ebrahimi et al. 2010].

Implementation of α, β, γ, and ζ: We measure α, β, and γ dynamically and the process is completely online. As ζ incurs a fixed latency, we measure it offline. To measure α, we use a per-core register, which stores the average LLC penalty. We calculate the average LLC penalty by finding out the total miss penalty of an application and dividing it with the total number of demand misses. Modern systems provide performance counters to find out miss penalty and the number of demand misses. We use a similar approach to count the same for each each application. To measure β, we divide the LLC latency of a prefetch request with the *BankWaitingParallelism*. We measure the LLC latency of a prefetch request with the help of a performance counter and store

[5]We get this value by synthesizing the circuit with a Synopsys design compiler for 45nm technology.

Table I. Hardware Overhead of CAFFEINE

Register/Counter	Size	Overhead for Four-Core
Core-id per LLC tag	131,072 blocks $\times$ 2 bits/block	32 KB
Bloom filter for POLL	1,024 entries $\times$ 4 $\times$ (1 pol bit + 2 bits for Core-id)	2KB
Pref bit per LLC tag	131,072 blocks $\times$ 1 bit	16KB
Interval length	13 bits	13 bits
Pfissued, Pfhits	12 bits per core	$12\times2\times4 = 96$ bits
BankAccessParallelism, *BankWaitingParallelism*	$\log_2$(#banks) (3 bits)	6 bits
last-row-access	$\log_2$(#rows/bank) (13 bits)	$13\times8\times2 = 208$ bits
Positive, negative, and net utilities	for 2 intervals, 8 bits/core	$3\times8\times2\times4 = 192$ bits
$\text{Cycles}_{affected}$	24 bits per core	$24\times4 = 96$ bits
$\text{Cycles}_{affecting}$	24 bits per core	$24\times4 = 96$ bits
POLL, BLI, RBC, DBI	13 bits per core	$13\times4\times4 = 208$ bits
α, β and γ	22 bits per core	$22\times4 = 88$ bits
Miss count	13 bits per core	$13\times4 = 52$ bits
Miss penalty	12 bits per core	$12\times4 = 48$ bits
Total		$\approx$ **50.1KB**

it in a register, and similarly we have a register, which stores the *BankWaitingParallelism*. The register is updated after every DRAM cycle. The measurement process for γ is similar to β. We measure γ by dividing the summation of t_{RP} and t_{RCD} with *BankAccessParallelism*. Note that t_{RP} and t_{RCD} are constants. We divide this constant with a register that contains *BankAccessParallelism*. *BankAccessParallelism* for a given request and for a given core is incremented whenever a DRAM command is scheduled and decremented when the command is serviced. To measure ζ, we have a register that stores the constant $\frac{BL}{2}$.

Significance of α, β, γ, and ζ: α, β, γ, and ζ play an important role in approximating the number of processor cycles for which other cores wait because of a prefetcher-caused intercore interference at the LLC, DRAM row-buffer, DRAM banks, and DRAM bus, respectively. The accuracies of BLI, RBC, DBI, γ, and ζ are 100%. On average, α and β are correct for more than 95% of time and the average false-positive rate of the Bloom filter is 3.25%.

Hardware Overhead: Table I shows the hardware overhead of CAFFEINE. For a four-core system with 8MB of LLC, CAFFEINE incurs an additional hardware of approximately 50.1KB, whereas HPAC incurs an overhead of 52.56KB. This hardware overhead is negligible (0.6% of 8MB of LLC). Note, we do not add the hardware overhead of techniques such as PACMAN [Wu et al. 2011] and PADC [Lee et al. 2008] as both are part of our baseline configuration.

5. EVALUATION METHODOLOGY

Simulation Framework: We use the gem5 [Binkert et al. 2011] simulator for our evaluation. Table II shows the baseline configuration of cores and the shared resources. The baseline system uses HPAC as the aggressiveness engine, PACMAN [Wu et al. 2011] as the LLC replacement policy, and PADC [Lee et al. 2008] as the DRAM scheduling policy. We find that the combination of HPAC, PACMAN, and PADC provides significant performance improvement compared to the combination of HPAC, LRU, and FR-FCFS. We use SPEC 2000 and SPEC 2006 benchmarks for our evaluation. We collect the statistics for workloads by running each benchmark in a workload for

Table II. Parameters of the Simulated System

Processor	4/8/16-core system, 3.7GHz, Out of Order
Fetch/decode/commit width	8
ROB/LQ/SQ/issue queue	192/96/64/64 entries
L1 D/I cache, L2 cache	32KB (4 way), 256KB (8 way)
L3 shared unified cache	8/16/32MB for 4/8/16 cores with 16/32/32 way
MSHRs	16, 32, 64/128/256 MSHRs at L1, L2, L3 with 4/8/16 cores Targets per MSHR = 4
Cache line size	64B in L1, L2 and L3
Replacement policy	LRU at L1/L2, PACMAN [Wu et al. 2011] at L3
Per-core prefetchers at L3	Streaming, 32 streams with degree = 4 and distance = 64
Prefetcher aggressiveness engine	HPAC [Ebrahimi et al. 2009] at LLC
Coherence protocol	MOESI
On-chip interconnect	Crossbar Transfer latency - 4 clock cycles, Arbitration latency - 5 clock cycles, Width of transmission channel - 64B
DRAM Controller	Open row, 64 read/write queues, PADC [Lee et al. 2008], drain-when-full
DRAM Bus	Split-transaction, 800MHz, BL = 8
DRAM	DDR3 1600 MHz (11-11-11) 1/2 channels for 4/8-core system, 2 Ranks/channel and 8 banks/rank, Max bandwidth/channel - 12.8GB/sec

500 million instructions after fast-forwarding 10 billion instructions and warming up 1 billion instructions. A workload terminates when the slowest benchmark completes 500 million instructions.

Performance and Fairness Metrics: We use the *harmonic mean of speedups (HS)* [Luo et al. 2001] and *weighted speedup (WS)* [Snavely and Tullsen 2000] as our performance metrics. *HS* is the reciprocal of the average normalized turnaround time and WS is equivalent to system throughput [Eyerman and Eeckhout 2008]. We measure the additional traffic because of prefetching using DRAM bus accesses per kilo instructions (BPKI). The bus accesses correspond to the DRAM bus transactions (reads, writes, and prefetch requests). We also evaluate the DRAM bandwidth consumption in GB/sec. We evaluate fairness of CAFFEINE by calculating the metric called *unfairness* [Mutlu and Moscibroda 2007], which is the ratio of the largest slowdown to the smallest slowdown of applications that are running simultaneously. We define these metrics as follows:

$$IS_i = \frac{CPI_i^{together}}{CPI_i^{alone}}, \qquad WS = \sum_{i=0}^{N-1} \frac{IPC_i^{together}}{IPC_i^{alone}} \tag{7}$$

$$HS = \frac{N}{\sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{together}}}, \qquad Unfairness = \frac{MAX\{IS_0, IS_1, \ldots, IS_{N-1}\}}{MIN\{IS_0, IS_1, \ldots, IS_{N-1}\}}. \tag{8}$$

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

Table III. Benchmark Characteristics When Run Alone on a Single-Core System with Stream Prefetcher ON

Benchmark	Pf ACC (%)	MPKI	Class
435.gromacs	60	0.05	$\overline{MI}\,\overline{PF}$ (0)
458.sjeng	1	0.57	$\overline{MI}\,\overline{PF}$ (0)
401.bzip2	84	0.17	$\overline{MI}\,\overline{PF}$ (0)
410.bwaves	99	0.37	$\overline{MI}PF$ (1)
456.hmmer	96	0.05	$\overline{MI}PF$ (1)
177.mesa	95	0.29	$\overline{MI}PF$ (1)
168.wupwise	40	0.25	$\overline{MI}PF$ (1)
429.mcf	31	30.12	$MI\overline{PF}$ (2)
433.milc	21	24.00	$MI\overline{PF}$ (2)
173.applu	6	2.35	$MI\overline{PF}$ (2)
188.ammp	3	1.98	$MI\overline{PF}$ (2)
471.omnetpp	10	10.23	$MI\overline{PF}$ (2)
255.vortex	32	1.25	$MI\overline{PF}$ (2)
434.zeusmp	55	2.50	$MIPF$ (3)
437.leslie3d	89	3.10	$MIPF$ (3)
450.soplex	80	4.12	$MIPF$ (3)
459.GemsFDTD	90	2.00	$MIPF$ (3)
462.libquantum	99	3.12	$MIPF$ (3)
470.lbm	91	3.00	$MIPF$ (3)
171.swim	95	9.11	$MIPF$ (3)

MPKI - Misses Per Kilo Instructions, ACC - Accuracy, COV - Coverage, MI - Memory intensive, PF - Prefetch Friendly.

Workload Selection: Table III classifies the benchmarks into four classes `class-0` ($\overline{MI}\,\overline{PF}$), `class-1` ($\overline{MI}PF$), `class-2` ($MI\overline{PF}$), and `class-3` ($MIPF$). MI corresponds to memory intensive and PF corresponds to prefetch friendly. A benchmark is memory intensive if the LLC misses per kilo instructions (MPKI) is greater than 1 [Ebrahimi et al. 2009]. We refer to a benchmark as prefetch friendly if the performance improvement with hardware prefetching is greater than 10% [Ebrahimi et al. 2009]. Table III shows the characteristics of SPEC 2006 and SPEC 2000 benchmarks with a stream prefetcher ON. To evaluate our mechanism on multicore systems, we create 100 four-core and 64 eight-core workload mixes by mixing benchmarks based on the previous four classes. Due to space limitations, we choose a set of representative workloads to analyze the effectiveness of CAFFEINE. For ease of analysis, we divide the workload mixes based on the intensity of interference at the LLC and at the DRAM into high (H) and low (L). At the LLC, a workload with an interference count of more than 16 (average interference count of all the workload mixes) per million instructions is termed as high (H); otherwise, the workload is termed as low (L). Similarly, at the DRAM, a workload with an interference count of more than 512 (average interference count of all the workload mixes) per million instructions is termed as high; otherwise, it is termed as low. We divide 100 four-core workloads into four cases such as HH, HL, LH, and LL, where each case consists of 25 workloads. Table IV and Table V show the representative workload mixes along with their class mixes for four-core and eight-core systems, respectively.

6. PERFORMANCE EVALUATION

We show the effectiveness of CAFFEINE on four-core and eight-core systems by comparing it with the state-of-the-art HPAC and NOPF (system with no prefetching). HPAC is an efficient technique, which improves the system performance across a wide set of workloads. We do not compare CAFFEINE with a system with no throttling as

Table IV. Representative Quad-Core Workloads (WLs); Class Mix Corresponds to the Class Numbers (as per Table III) of Applications

WL.NO.	Quad-Core Workloads	Class Mix	Case
Q0	libquantum-swim-GemsFDTD-bzip2	3-3-3-0	HH
Q1	mesa-gromacs-lbm-leslie3d	1-0-3-3	HH
Q2	soplex-leslie3d-lbm-hmmer	3-3-3-1	HH
Q3	GemsFDTD-hmmer-bwaves-mcf	3-1-1-2	HH
Q4	bwaves-hmmer-bzip2-leslie3d	1-1-0-3	HL
Q5	bzip2-milc-omnetpp-mcf	0-2-2-2	HL
Q6	milc-soplex-omnetpp-mcf	2-3-2-2	HL
Q7	mesa-gromacs-wupwise-hmmer	1-0-1-1	HL
Q8	bwaves-lbm-libquantum-milc	1-3-3-2	LH
Q9	ammp-lbm-milc-swim	2-3-2-3	LH
Q10	leslie3d-omnetpp-libquantum-milc	3-2-3-2	LH
Q11	GemsFDTD-libquantum-lbm-leslie3d	3-3-3-3	LH
Q12	gromacs-hmmer-bzip2-soplex	0-1-0-3	LL
Q13	hmmer-swim-ammp-bzip2	1-3-2-0	LL
Q14	sjeng-applu-hmmer-bzip2	0-2-1-0	LL
Q15	soplex-zeusmp-ammp-hmmer	3-3-2-1	LL

Table V. Representative Eight-Core Workloads (WLs)

WL.NO.	Eight-Core Workloads	Class Mix	Case
E0	wupwise-vortex-sjeng-leslie3d-sjeng-applu-hmmer-bzip2	1-2-0-3-0-2-1-0	LL
E1	soplex-zeusmp-ammp-hmmer-bwaves-omnetpp-milc-swim	3-3-2-1-1-2-2-3	HL
E2	libquantum-swim-GemsFDTD-leslie3d-bzip2-milc-lbm-sjeng	3-3-3-3-0-2-3-0	HH
E3	bzip2-leslie3d-soplex-swim-ammp-lbm-GemsFDTD-hmmer	0-3-3-3-2-3-3-1	LH

on average, HPAC performs better than no throttling. Also, the coordination of two techniques, FST [Ebrahimi et al. 2010] and HPAC [Ebrahimi et al. 2009], in prefetch-aware shared resource management [Ebrahimi et al. 2011] addresses a problem similar to CAFFEINE. On top of HPAC's seven thresholds, FST uses six additional thresholds. The coordination becomes less effective for most of the four-core workloads and performs worse than HPAC for 8 and 16-core systems because it finds the slowest and most interfering application only. For these reasons, we do not compare CAFFEINE with the coordination of FST and HPAC.

First, we present the aggregate results for both four-core and eight-core systems and then provide a detailed analysis for four-core workloads with four different case studies.

6.1. Aggregate Results for Four-Core and Eight-Core System

Table VI shows the overall results of CAFFEINE on four-core and eight-core systems for 100 and 64 workloads, respectively. CAFFEINE outperforms HPAC in terms of HS, WS, BPKI, and DRAM bandwidth consumption.

Four-Core Results: Figure 5 shows the system performance improvement with HPAC and CAFFEINE, in terms of HS. We report the performance improvement based on the four cases of prefetcher-caused interference as mentioned in Section 5. CAFFEINE outperforms NOPF in all the 100 workloads in terms of HS with a

Table VI. Summary of Average Results with CAFFEINE

Four-core (100 WLs)	HS	WS	BPKI
Over NOPF	17% (38.98%)	8.82% (15.13%)	6.25% (11%)
Over HPAC	9.5% (38.29%)	3.61% (17.12%)	−3.11% (−9.09%)
Eight-core (64 WLs)			
Over NOPF	19% (30.6%)	10.91% (12.34%)	7.1% (12.63%)
Over HPAC	11% (20.7%)	6.71% (9.13%)	−2.79% (−6.12%)

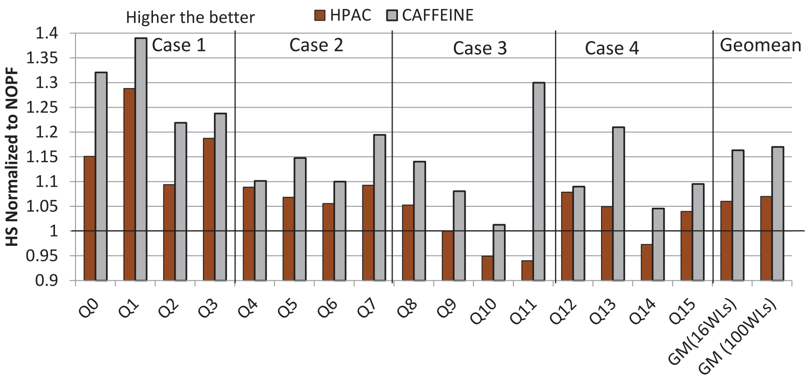


Fig. 5. Harmonic speedup for 100 four-core WLs normalized to NOPF.

maximum (minimum) performance improvement of 38.98% (1.01%) in workload mix Q1 (Q10). Our observations are as follows:

—On average (geomean), across 100 four-core workloads, compared to HPAC, CAFFEINE improves the performance by 9.5% and 3.61% in terms of HS and WS and an off-chip bandwidth reduction of 10%. Compared to NOPF, CAFFEINE achieves an improvement in terms of HS and WS of 17% and 8.82%, with a 6.25% increase in the BPKI.

—Compared to NOPF, HPAC improves the system performance by 7% and 5.3% in terms of HS and WS, with a 7.13% increase in the BPKI.

Why does CAFFEINE perform better than HPAC? We analyze the results of all four-core and eight-core workloads and find that CAFFEINE is more effective than HPAC across all the workloads. The primary reasons for this effectiveness are as follows:

(1) CAFFEINE's tracking of prefetcher-caused interference is *fine-grained* (saved cycles), whereas HPAC is *coarse grained* (dependent on seven thresholds: four for HPAC and three for FDP). As HPAC's thresholds are dependent on architectural parameters, they correlate to the actual interference (interference that is intolerable) 57.3% of the time. *For the other 42.7% of the time, the throttling decisions of HPAC are conservative and it throttles down* (↓) *the prefetchers even though they are not interfering with others.* Also, there are four cases out of 16 cases (refer Section 2.3) in which ACC is high (>0.60) and POL is low (<90). *In these four cases, the global controller of HPAC does not override the local controller even if the BWC (50K) and BWNO (75K) are high. This case happens in 21 four-core workloads out of 100 four-core workloads.* In contrast to HPAC, CAFFEINE's utility$_{negative}$ correlates strongly with the actual prefetcher-caused intercore interference with a correlation coefficient of 0.89 and CAFFEINE's utility$_{positive}$ correlates strongly with the IPC with a correlation coefficient of 0.81.

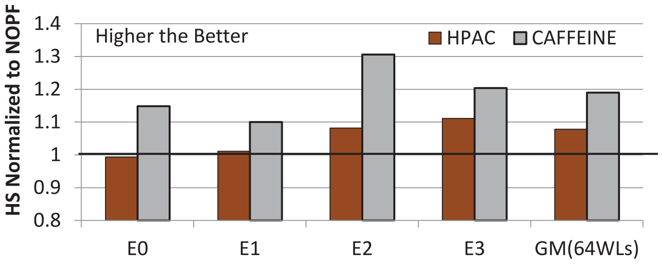


Fig. 6. Harmonic speedup for 64 eight-core WLs normalized to NOPF.

(2) CAFFEINE's usage of marginal utilities models the contribution of each prefetcher in improving the system performance and the scenarios where interference is intolerable are the key in making better decisions. Also, identification of *affected* and *affecting* prefetchers help in throttling down the prefetchers, which are responsible for interference. Compared to HPAC's mechanism for detecting the *affecting* prefetchers, CAFFEINE's mechanism is 31.7% more accurate.
(3) The aforementioned differences between HPAC and CAFFEINE cause a reduction of the LLC interference (based on the count of intercore LLC pollution) and the DRAM interference (based on the count of bli, rbc, and Dbi) by an additional 7.89% and 13.69%, respectively.

—In terms of unfairness, compared to HPAC, CAFFEINE reduces the unfairness by 9.52% as it throttles down (↓) prefetch-friendly and memory-intensive applications that interfere heavily with memory-nonintensive applications.

Eight-Core Results: Figure 6 shows the performance improvement (in terms of HS) with HPAC and CAFFEINE. In eight-core systems also, CAFFEINE outperforms NOPF in all the 64 workloads with the maximum performance improvement of 30.6% in workload mix E2 and the minimum of 3.45%.

—Compared to HPAC, CAFFEINE improves the performance in terms of HS and WS by 11% and 6.7%, with an off-chip bandwidth reduction of 11.65%, and compared to NOPF, CAFFEINE achieves a 19% improvement in HS and a 10.91% improvement in WS, with a 7.1% increase in the BPKI.
—HPAC improves the system performance in terms of HS and WS by 7.8% and 4.15%, with an 8% increase in the BPKI when compared to NOPF.
—In terms of unfairness, compared to HPAC, CAFFEINE reduces the unfairness by 10%. This shows that the effectiveness of CAFFEINE increases with the increase in the core count.

6.2. Individual Workload Analysis

We analyze the effectiveness of CAFFEINE with the use of four case studies of LLC and DRAM interference. We select one workload mix from each of these four cases (HH, HL, LH, and LL). These case studies provide insights for rest of the workloads.

Case I: High at LLC, High at DRAM—HH (WL Q0): This mix contains three prefetch-friendly and memory-intensive applications (`libquantum`, `swim`, `and` `GemsFDTD`) and one application that is prefetch unfriendly and memory nonintensive (`bzip2`). The observations are as follows:

—CAFFEINE improves the performance of `bzip2`, `swim`, and `libquantum`, and it incurs a slight degradation in the performance of `GemsFDTD`. DE-CAFFEINATION finds out *affecting* prefetchers (prefetchers of `GemsFDTD` and `swim`) and the *affected* prefetcher (prefetcher of `bzip2`). Figure 7 shows the difference in the distribution of the

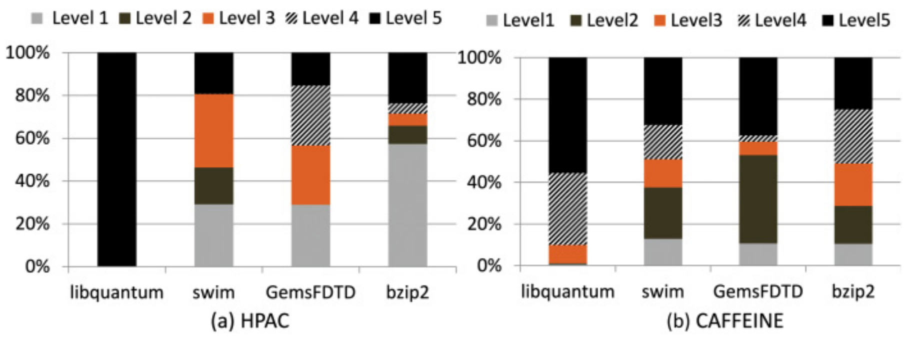


Fig. 7. Case Study I: Distribution of aggressiveness levels.

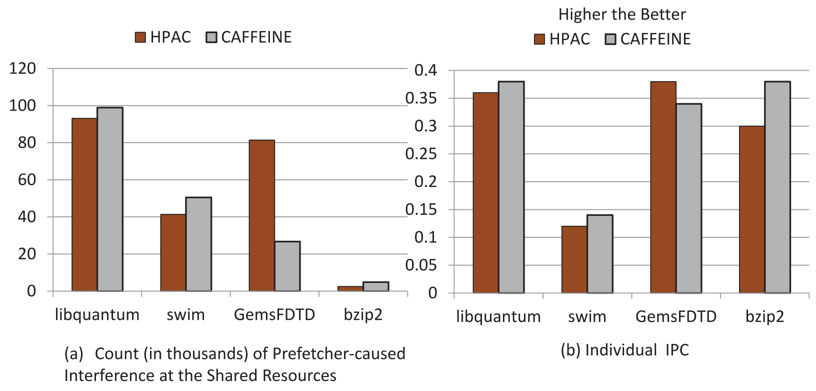


Fig. 8. Case Study I: Individual behavior.

aggressiveness levels of `swim`, `GemsFDTD`, and `bzip2`. With HPAC, `GemsFDTD` spends more than 60% of the execution time at level 3 and above. In contrast, CAFFEINE throttles down (↓) `GemsFDTD` and because of this it spends more than 60% of its time at level 3 and below. On the other hand, `bzip2`, which spends 60% of the time at level 1 in HPAC, spends only 10% of the time at level 1 with CAFFEINE.

—The throttling decisions of CAFFEINE reduce the prefetcher-caused interference coming from `GemsFDTD`. For the rest, it increases the interference slightly. Figure 8(a) shows the count of prefetcher-caused interference at the LLC and at the DRAM. Compared to HPAC, CAFFEINE reduces the interference count by three times, which comes from `GemsFDTD`, and this results in improvement in IPCs for the rest (refer to Figure 8(b)). The increase in the intercore interference count with CAFFEINE for applications such as `libquantum`, `swim`, and `bzip2` does not affect the increase in the IPCs, which shows that the increase in the interference count from these three applications is tolerable.

—Compared to HPAC, CAFFEINE improves the performance in terms of HS and WS by 14.75% and 7.79%, with a 6.04% reduction in the unfairness. Compared to NOPF, CAFFEINE improves the performance in terms of HS and WS by 32% and 13.3%, with a 6.7% increase in the BPKI.

—Figure 9 shows the changes in the aggressiveness levels that vary over 20 windows. Compared to HPAC, CAFFEINE throttles down `GemsFDTD` for a large fraction of time, which helps other applications such as `bzip2` and `swim`.

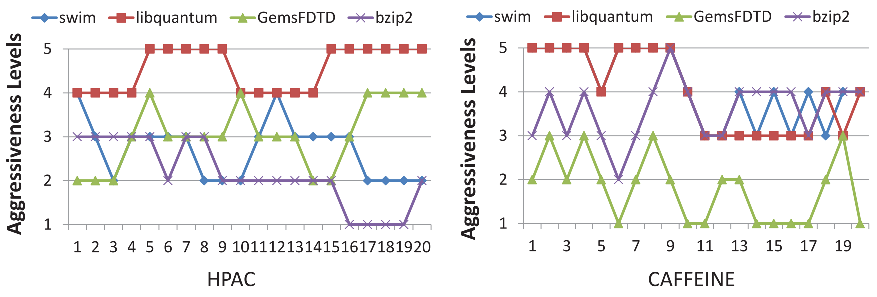


Fig. 9. Case Study I: Variations in the aggressiveness levels over 20 time windows.

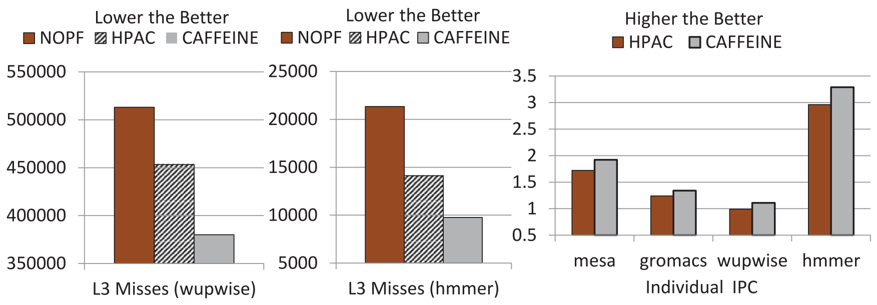


Fig. 10. Case Study II: Individual behavior.

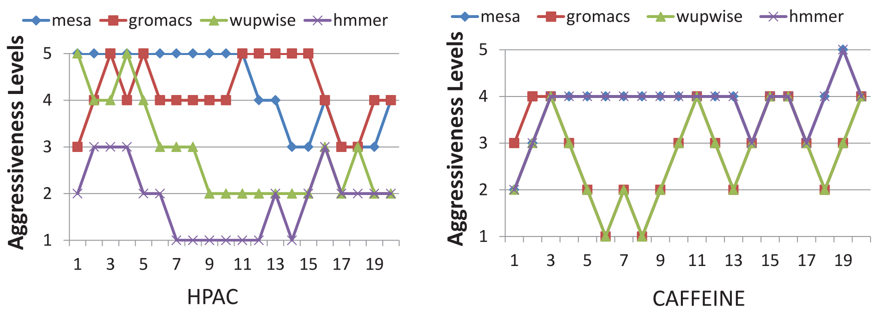


Fig. 11. Case Study II: Changes in the aggressiveness levels over an interval of 20 windows.

Case II: High at LLC, Low at DRAM—HL (WL Q7): All the applications in this mix are memory nonintensive. `hmmer`, `wupwise`, and `mesa` are prefetch friendly and `gromacs` is prefetch unfriendly.

—CAFFEINE finds out `mesa` as the *affecting* prefetcher, which affects `wupwise` and `hmmer`. Out of `wupwise` and `hmmer`, `wupwise` is affected the most. CAFFEINE throttles up (↑) both `wupwise` and `hmmer` and throttles down (↓) `mesa`. Compared to HPAC, CAFFEINE reduces the LLC miss count of `hmmer` and `wupwise` by 44.81% and 19.2%, respectively (Figure 10). CAFFEINE outperforms HPAC with performance improvement in terms of HS and WS by 9.32% and 5.58%, respectively. Figure 10 also shows significant IPC improvement for `mesa`. This improvement comes from CAFFEINATION. Compared to the HPAC, with CAFFEINATION, `mesa` spends 17.58% more time at level 4 and level 5 (it happens in the intervals in which `mesa` is not an *affecting* prefetcher). Figure 11 shows the variations in the aggressiveness levels for an interval of 20 windows where CAFFEINE throttles up `hmmer` for more time as

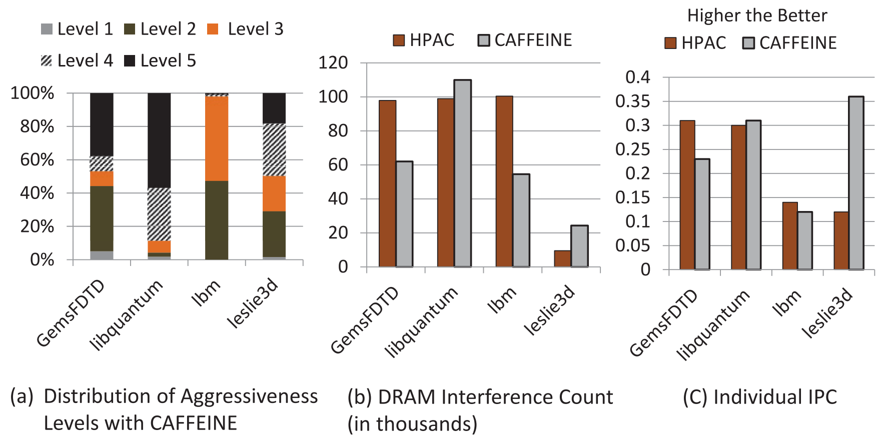


Fig. 12. Case Study III: Individual behavior. y-axis in (b) shows the prefetcher-caused interference count.

compared to HPAC. Similarly, compared to HPAC, CAFFEINE throttles down `mesa` for more time.

—Compared to NOPF, HPAC improves the performance by 9.25% and 6.11% in terms of HS and WS, respectively. Although this mix shows high interference at the LLC, in a given interval length, none of the applications cross the threshold set for the LLC POL (90) and stay at level 3 and level 4.

—Compared to NOPF, CAFFEINE improves the performance by 19.4% and 8.66% in terms of HS and WS, with an increase in the BPKI of 11%. CAFFEINE reduces the unfairness by 2.7%.

Case III: Low at LLC, High at DRAM—LH (WL Q11): This mix contains applications that are both prefetch friendly and memory intensive. The observations are as follows:

—Figure 12(a) shows the distribution of aggressiveness levels with CAFFEINE. With HPAC, `leslie3d` stays at level 1 for 88.2% of the time and the rest spend more than 96% of the time at level 5. On the other hand, CAFFEINE throttles down (↓) the prefetchers of `GemsFDTD` and `lbm`, and both of these applications spend their time at level 3 and above. These decisions translate into the reduction in the prefetcher-caused intercore interference count (because of `GemsFDTD` and `lbm`) at the DRAM (refer to Figure 12(b)). CAFFEINE reduces the interference count of `lbm` and `GemsFDTD` by 43% and 38%, respectively. Figure 12(c) shows the individual IPC of each application with CAFFEINE and HPAC. The reduction in interference at the DRAM results in IPC improvement for `leslie3d` and `libquantum`.

—Compared to HPAC, CAFFEINE improves the performance by 35.12% and 17.12% in terms of HS and WS, with a 2.87% reduction in the unfairness. Compared to NOPF, CAFFEINE improves the performance in terms of HS and WS by 30% and 15.13%, with an 11.4% increase in the BPKI.

—HPAC performs worse (−7%) than NOPF. HPAC improves the individual performance of `GemsFDTD`, `libquantum`, and `lbm` but fails to deliver the same for `leslie3d`. The reason for this behavior is that `leslie3d` undergoes a phase change. Figure 13 shows the throttling decisions of HPAC and CAFFEINE over a window of an initial thirty-five 8K misses intervals. `leslie3d` undergoes a phase change from the 14th window to the 28th window. In these windows, `leslie3d` issues useless prefetch requests (requests that cause low prefetch accuracy). In between window 7 and

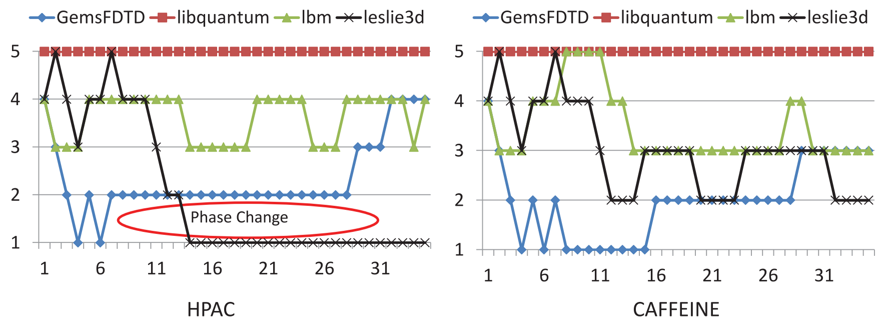


Fig. 13. Case study III: Phase analysis. y-axis shows the aggressiveness level and x-axis shows the interval.

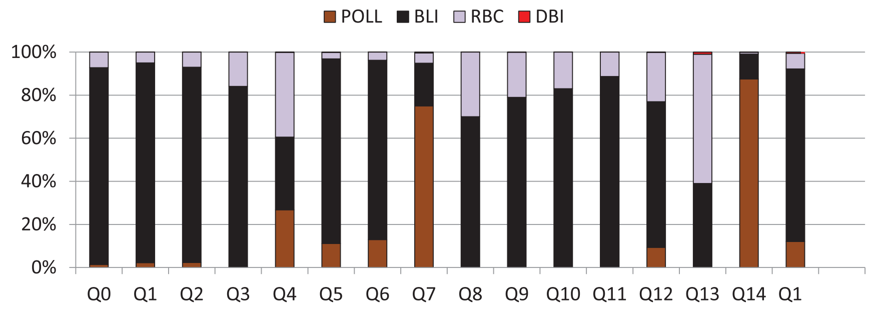


Fig. 14. Contribution stack: Contribution, in terms of number of processor cycles saved by *Poll, Bli, Rbc*, and *DBI*.

window 12, the global controller of HPAC throttles down (↓) `leslie3d` (from level 4 to level 2) because it affects others. From the 14th to the 28th window, the accuracy of `leslie3d` goes down below the predefined accuracy threshold and the local controller of HPAC throttles it further down (↓) to level 1. On the other hand, with CAFFEINE, `leslie3d` spends its phase change time mostly at level 3. CAFFEINE throttles up (↑) `leslie3d` even if it has low prefetch accuracy because $\triangle \text{utility}_{positive} \geq 0$. This shows how CAFFEINE adapts to the phase changes in applications.

Case IV: Low at LLC, Low at DRAM—LL (WL Q12): In this mix, `soplex` is the only application that is memory intensive. `soplex` and `hmmer` are prefetch friendly applications and the rest are not. `soplex` is the application that affects the rest but the interference is not significant. Compared to NOPF, HPAC improves the performance by 7.8% and CAFFEINE improves it by 8.85%. This slight improvement comes from the decisions of CAFFEINATION.

6.3. Analysis of CAFFEINE

CAFFEINE on 16-Core System: We now quantify the effectiveness of CAFFEINE on 16-core system. We create two 16-core workload mixes by mixing (E0 and E1) and (E2 and E3). In both workloads, in terms of HS, CAFFEINE outperforms HPAC by more than 11.28%. We believe the effectiveness of CAFFEINE will increase with the increase in the core count.

Contributions of Interference Metrics: To understand the individual contribution of *Poll, Bli, Rbc*, and *DBI* on the overall performance improvement, we show the number of processor cycles saved by each one of them in Figure 14, for 16 representative workloads. *Bli* dominates in all the workloads except in Q4, Q7, and Q14. *Dbi* provides

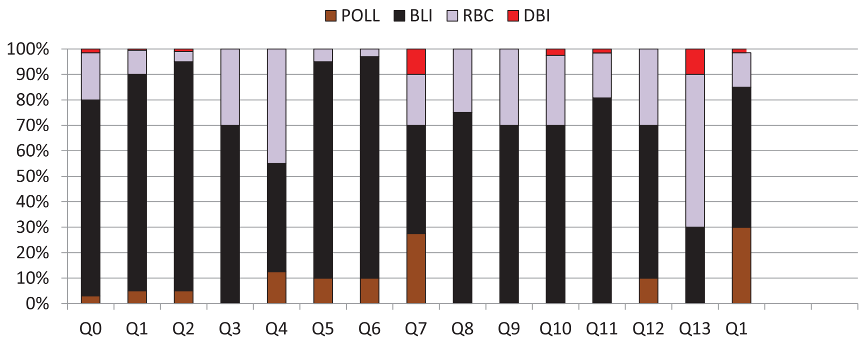


Fig. 15. Performance stack: Contribution of *Poll, Bli, Rbc*, and *Dbi* in improvement in system performance (in terms of HS).

Table VII. Sensitivity Studies: Average (max) Changes in HS and BPKI Compared to HPAC

LLC Size	HS	BPKI
16MB	11.28% (42.11%)	−3.68% (−7.25%)
32MB	10.69% (36.59%)	−3.16% (−5.12%)
#DRAM Banks		
16	9.69% (29.12%)	−2.76% (−3.96%)
32	10.33% (36.24%)	−1.66% (−3.11%)

marginal benefit (less than 0.4%) across all the workloads. We conclude that *Poll, Bli,* and *Rbc* play an important role for CAFFEINE. Figure 15 shows the contribution of *Poll, Bli, and Rbc* in the improvement in the harmonic speedup. We find a similar trend for the rest of the 84 WLs. We can conclude that the prefetcher-caused intercore interference plays an important role at the DRAM system and not at the LLC.

Sensitivity Studies: We perform sensitivity studies by changing the LLC size and the number of DRAM banks on four-core systems. Table VII shows the effects of different LLC sizes and different numbers of DRAM banks on HS and BPKI over HPAC. As the thresholds used in HPAC are oblivious to the LLC size and the number of DRAM banks, there is a slight degradation in the performance of HPAC. On the other hand, the effectiveness of CAFFEINE remains the same and the performance of CAFFEINE improves with the increase in LLC size and number of DRAM banks.

Effect of Intracore Interference: Till now, we emphasized the prefetcher-caused intercore interference. We also study the effect of prefetcher-caused intracore interference on the performance. We add the intracore interference in our utility$_{negative}$ and run CAFFEINE. We find that the change in the HS compared to CAFFEINE is marginal (less than 1%), with one exception. Workloads consisting of applications that belong to `class-2` ($MI\overline{PF}$) show an average improvement of 1.78% over CAFFEINE. We conclude that the effect of prefetcher-caused intracore interference is marginal.

Effect of LLC Replacement Policy and DRAM Scheduling: Till now, we showed the effectiveness of CAFFEINE with PACMAN [Wu et al. 2011] and PADC [Lee et al. 2008] as the LLC replacement policy and DRAM scheduling policy. We also study the behavior of CAFFEINE with TA-DRRIP [Jaleel et al. 2010] and FR-FCFS as the replacement policy and DRAM scheduling policy, respectively. Compared to HPAC, on four-core systems, CAFFEINE improves the performance (HS) by 13.19%. We conclude that the effectiveness of CAFFEINE is dependent neither on the LLC replacement policy nor on the DRAM scheduling policy. Also, CAFFEINE can help in making

Table VIII. Comparison of Performance Improvement and Hardware Overhead of HPAC, ABS, and CAFFEINE

Technique	Cost	Benefit
HPAC	52.5KB (20.5KB)	6.1%
ABS	17KB	10.5%
CAFFEINE	50.1KB (18.1KB)	19%

Compared to a system with no prefetching. For HPAC and CAFFEINE, we report two types of hardware costs in the form x (y). x corresponds to the actual hardware cost and y corresponds to the hardware cost excluding the hardware overhead that comes from using a core-id bit per tag.

better DRAM scheduling decisions by helping techniques such as PARBS [Mutlu and Moscibroda 2008] and TCM [Kim et al. 2010] in creating batches and clusters.

CAFFEINE Versus ABS: ABS [2012] is a prefetcher aggressiveness controller for multiported, multibanked distributed LLC, where each LLC bank has a hardware prefetcher. It uses a hill-climbing approach to control the aggressiveness of each bank. For a fair comparison with CAFFEINE, we employ ABS in our baseline organization with a centralized LLC that uses the state-of-the-art AMPM prefetcher [Ishii et al. 2011]. The rest of the implementation of the ABS remains the same for our comparison. We use five different levels of aggressiveness with prefetch degree (0, 1, 4, 8, and 16) as per ABS [Albericio et al. 2012] and compare it with CAFFEINE.

Compared to ABS, on an average, across 100 four-core WLs, CAFFEINE improves the system performance in terms of HS by 7.8% with a reduction of 5.21% in unfairness. Out of 100 4-core WLs, CAFFEINE outperforms ABS in 87 WLs and performs slightly worse (−1.29%) for 13 WLs. The primary reason behind this behavior is that there are WLs that contain applications such as `bzip2`, `hmmer`, and `gromacs` where CAFFEINE throttles them up, whereas ABS does not change their aggressiveness levels. These applications are not sensitive to the changes in the prefetcher aggressiveness level, but CAFFEINE throttles them up assuming these applications will improve the overall utility_{net}. Please note these applications are not memory intensive in nature and they do not affect others, and because of this, CAFFEINE's decisions are biased toward them.

ABS outperforms HPAC on 45 out of 100 WLs but unable to outperform CAFFEINE on 87 WLs. The primary reason behind this behavior of ABS is the throttling decisions of ABS, which are more biased toward utility_{net}. Please note that ABS does not use utility_{net} and uses reduction in LLC bank misses as an indicator of the utility_{net}. In WLs where utility_{net} is positive but the prefetcher of the corresponding core is an *affecting* one, CAFFEINE's decisions performs better than ABS. ABS does not consider $\text{utility}_{negative}$ and does not throttles down the *affecting* prefetchers (continuously throttle up as long as the bank miss ratio is improving) and similarly throttles up the *affected* prefetcher. This kind of scenario happens in 89 out of the 100 WLs.

Cost-Benefit Analysis Between HPAC, ABS, and CAFFEINE: Table VIII shows the comparison of performance improvement and hardware overhead of HPAC, ABS, and CAFFEINE for a four-core multicore system with the state-of-the-art hardware prefetcher called AMPM [Ishii et al. 2011]. To control the aggressiveness of AMPM, we use five levels of aggressiveness with prefetch degrees of 0, 1, 4, 8, 16. Please note that the hardware cost mentioned in ABS [Albericio et al. 2012] does not consider the overhead that comes from a pref-bit as it uses a sequential tagged prefetcher. On an average, CAFFEINE outperforms both HPAC and ABS. If we ignore the hardware cost associated with the core-id bit per LLC tag (which is already available in the multicore systems), then CAFFEINE provides an improvement of 12.5% over HPAC with a 2.4KB reduction in the hardware overhead. Compared to ABS, CAFFEINE provides an additional 7.8% improvement with an additional hardware overhead of 1.1KB, which is modest. So CAFFEINE provides a fair tradeoff between performance improvement and the hardware overhead.

CAFFEINE on GHB Prefetcher: We run CAFFEINE on GHB [Nesbit et al. 2004] and evaluate it on the 16 four-core representative workloads. For a GHB prefetcher, the aggressiveness is controlled by its prefetch degree. We use five different levels of aggressiveness with prefetch degrees (2, 4, 8, 12, 16) as per Ebrahimi et al. [2009] and compare CAFFEINE with HPAC. CAFFEINE outperforms HPAC in all the 16 workloads with a speedup (HS) of 6.19%.

CAFFEINE for Parallel Applications: For parallel applications where all threads perform similar computations and show similar behavior at the memory system, CAFFEINE will throttle down or throttle up all the prefetchers, which is a correct decision as all the prefetchers will belong to either the *affected* or *affecting* group. One way to eliminate this decision process, which can lead to a live-lock situation (all prefetchers use same throttling decisions), is through randomization. Prefetchers can be assigned to the *affected* group in a random way and the rest can be assigned to the *affecting* group.

Intolerability: In this work, we call the prefetcher-caused intercore interference intolerable if $\text{utility}_{totalnegative} \geq \text{utility}_{totalpositive}$. We also study the effect of allowing some level of intolerance on the system performance by using a parameter w (weight of intolerance). We change the condition for intolerable interference as follows: $(\text{utility}_{totalpositive} - w \times \text{utility}_{totalnegative}) <$ zero. We sweep through different levels of w (from 0.1 to 1) and find that $w = 1$ provides the best performance.

7. RELATED WORK

Sandbox Prefetching (SBP): Pugsley et al. [2014] propose a simple yet effective mechanism to control the aggressiveness of hardware prefetchers. It uses a Bloom filter [Bloom 1970] to test whether a prefetch address (without issuing actual prefetch requests) is part of any strided stream. After testing, it issues prefetch requests based on a score that is driven by prefetch accuracy. SBP does not consider the effect of prefetcher-caused intercore interference as it is a technique proposed mainly for single-core systems that also performs well for some of the multicore workloads. As CAFFEINE is orthogonal to the underlying prefetching technique, it can be used along with SBP to further increase the effectiveness of SBP.

Adaptive Prefetching on IBM POWER 7: Jiménez et al. [2012] propose a technique that uses a per-core configuration status register (CSR) to inform the operating system about the different prefetch configuration settings The operating system 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.

TCPT: Panda and Balachandran [2013] propose an aggressiveness controller for multithreaded applications called TCPT. TCPT uses a metric called criticality along with the prefetch accuracy to throttle the prefetchers. Our model can be used along with TCPT to further improve the execution time of a multithreaded application. As threads do cooperate and share data in multithreaded applications, intercore cooperative sharing at the shared resources can be added to CAFFEINE.

Prior proposals such as the filtering mechanism of Zhuang and Lee [2003] use cache pollution as the metric to control the aggressiveness. Wu and Martonosi [2011] characterize cache pollution in the real system and propose a prefetch manager that controls the aggressiveness at runtime. Liu and Solihin [2011] propose an analytical model to study the interaction of hardware prefetching and bandwidth partitioning on a multicore system.

8. CONCLUDING REMARKS

This article proposed CAFFEINE, a low-cost and robust aggressiveness engine for multicore systems. CAFFEINE uses a utility-driven model to find out the prefetchers that cause intercore interference. The main idea behind CAFFEINE is it finds out the number of processor cycles saved in the entire system because of prefetching and is free from setting and tuning of thresholds. With negligible hardware overhead, CAFFEINE results in 9.5% and 11% performance improvement on four-core and eight-core systems, respectively. We conclude that CAFFEINE is an effective prefetcher aggressiveness engine.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Rupesh Nasre, Dr. Madhu Mutyam, and Prof. R. Govindarajan for their valuable comments.

REFERENCES

Jorge Albericio, Rubén Gran, Pablo Ibáñez, Víctor Viñals, and Jose María Llabería. 2012. ABS: A low-cost adaptive controller for prefetching in a banked shared last-level cache. *ACM Trans. Archit. Code Optim.* 8, 4, Article 19 (Jan. 2012), 20 pages. DOI:http://dx.doi.org/10.1145/2086696.2086698

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011).

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (July 1970), 422–426. DOI:http://dx.doi.org/10.1145/362686.362692

J. Doweck. 2006. Inside Intel Core Microarchitecture and Smart Memory Access. Intel technical white paper.

Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2010. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, 335–346. DOI:http://dx.doi.org/10.1145/1736020.1736058

Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2011. Prefetch-aware shared resource management for multi-core systems. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 141–152. DOI:http://dx.doi.org/10.1145/2000064.2000081

Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. 2009. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, 316–326. DOI:http://dx.doi.org/10.1145/1669112.1669154

Stijn Eyerman and Lieven Eeckhout. 2008. System-level performance metrics for multiprogram workloads. *IEEE Micro* 28, 3 (May 2008), 42–53. DOI:http://dx.doi.org/10.1109/MM.2008.44

A. Glew. 1998. MLP yes! ILP no! wild and crazy idea session. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems.*

Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2011. Access map pattern matching for high performance data cache prefetch. *J. Instruction-Level Parallelism* 13 (2011).

Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 60–71. DOI:http://dx.doi.org/10.1145/1815961.1815971

Victor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. 2012. Making data prefetch smarter: Adaptive prefetching on power7. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, NY, 137–146. DOI:http://dx.doi.org/10.1145/2370816.2370837

Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10).* 65–76. DOI:http://dx.doi.org/10.1109/MICRO.2010.51

Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. 2008. Prefetch-aware DRAM controllers. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, 200–209. DOI:http://dx.doi.org/10.1109/MICRO.2008.4771791

Fang Liu and Yan Solihin. 2011. 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). ACM, New York, NY, 37–48. DOI:http://dx.doi.org/10.1145/1993744.1993749

Kun Luo, Jayanth Gummaraju, and Manoj Franklin. 2001. Balancing thoughput and fairness in SMT processors. In *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*. 164–171. DOI:http://dx.doi.org/10.1109/ISPASS.2001.990695

Ratul Mahajan, Jitu Padhye, Ramya Raghavendra, and Brian Zill. 2008. Eat all you can in an all-you-can-eat buffet: A case for aggressive resource usage. In *Proceedings of the 7th ACM Workshop on Hot Topics in Networks (HotNets-VII)*. 43–48. Available at http://conferences.sigcomm.org/hotnets/2008/papers/8.pdf.

Onur Mutlu and Thomas Moscibroda. 2007. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society, 146–160. DOI:http://dx.doi.org/10.1109/MICRO.2007.40

Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA'08)*. 63–74. DOI:http://dx.doi.org/10.1109/ISCA.2008.7

Kyle J. Nesbit, Ashutosh S. Dhodapkar, and James E. Smith. 2004. AC/DC: An adaptive data cache prefetcher. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*. IEEE Computer Society, 135–145. DOI:http://dx.doi.org/10.1109/PACT.2004.4

Calculating Outliers. *NIST/SEMATECH e-Handbook of Statistical Methods*. http://www.itl.nist.gov/div898/handbook/, http://www.itl.nist.gov/div898/handbook/prc/section1/prc16.htm.

Biswabandan Panda and Shankar Balachandran. 2013. TCPT: Thread criticality-driven prefetcher throttling. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. IEEE Press, 399–400.

Seth H. Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L. Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. 2014. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *20th IEEE International Symposium on High Performance Computer Architecture (HPCA'14), February 15-19, 2014, Orlando, FL*. 626–637. DOI:http://dx.doi.org/10.1109/HPCA.2014.6835971

Allan Snavely and Dean M. Tullsen. 2000. Symbiotic job scheduling for a simultaneous multithreaded processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM, New York, NY, 234–244. DOI:http://dx.doi.org/10.1145/378993.379244

Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. IEEE Computer Society, 63–74. DOI:http://dx.doi.org/10.1109/HPCA.2007.346185

Standard Performance Evaluation Corporation. SPEC CPU2006, SPEC CPU 2000. Available at http://www.spec.org.

J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. 2002. POWER4 system microarchitecture. *IBM J. Res. Dev* 46, 1 (Jan. 2002), 5–25.

Carole-Jean Wu and Margaret Martonosi. 2011. Characterization and dynamic mitigation of intra-application cache interference. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'11), April 10-12, 2011, Austin, TX*. 2–11. DOI:http://dx.doi.org/10.1109/ISPASS.2011.5762710

Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely Jr., and Joel Emer. 2011. PACMan: Prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, 442–453. DOI:http://dx.doi.org/10.1145/2155620.2155672

X. Zhuang and H.-H. S. Lee. 2003. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *Proceedings of the 2003 International Conference on Parallel Processing*. 286–293. DOI:http://dx.doi.org/10.1109/ICPP.2003.1240591

Received June 2015; revised July 2015; accepted July 2015