Efficient Selective Multicore Prefetching under Limited Memory Bandwidth

Vicent Selfa, Julio Sahuquillo, María E. Gómez, Crispín Gómez

Dept. of Computer Engineering, Universitat Politècnica de València, Spain

Abstract

Current multicore systems implement multiple hardware prefetchers to tolerate long main memory latencies. However, memory bandwidth is a scarce shared resource which becomes critical with the increasing core count. To deal with this fact, recent works have focused on adaptive prefetchers, which control the prefetcher aggressiveness to regulate the main memory bandwidth consumption. Nevertheless, in limited bandwidth machines or under memory-hungry workloads, keeping active the prefetcher can damage the system performance and increase energy consumption. This paper introduces *selective prefetching*, where individual prefetchers are activated or deactivated to improve both main memory energy and performance, and proposes ADP, a prefetcher that deactivates local prefetchers in some cores when they present low performance and co-runners need additional bandwidth. Based on heuristics, an individual prefetcher is reactivated when performance enhancements are foreseen. Compared to a state-of-the-art adaptive prefetcher, ADP provides both performance and energy enhancements in limited memory bandwidth.

Keywords: Multicore prefetching, adaptive prefetching, deactivation policies, global feedback

1. Introduction

Addressing memory latencies is a major design concern in modern multicores. In this regard, hardware prefetching plays a key role in modern high-performance processors. Because of this reason, modern microprocessors [1, 2, 3] implement

Email address: viselol@disca.upv.es (Vicent Selfa)

multiple prefetchers, which work along the different caches of the memory hierarchy.

In current processors, prefetch requests from multiple cores (i.e. applications) compete with regular memory requests for off-chip DRAM bandwidth. Therefore, since prefetching is a speculative technique, it increases the total number of accesses to main memory [4, 5, 6]. This fact can turn into significant performance losses in some individual applications, which running in isolation benefit from prefetching, in case of limited memory bandwidth. Unfortunately, this case predominates in most current multicores due to two main reasons: i) contention appear in the access to DRAM modules, ii) systems are configured with a limited number of modules because of they are costly.

A straightforward solution to increase bandwidth availability would be to turn off all the per-core individual prefetchers and removing speculative prefetches. However, this is not an acceptable solution from a performance perspective, since hardware prefetching can bring worth performance enhancements in some applications. A solution, recently proposed in recent approaches [7], is to control the memory bandwidth by implementing throttling up/down mechanisms to control the prefetcher aggressiveness. An individual prefetcher is only throttled down when no performance benefits for that application are expected. However, due to limited bandwidth, keeping active the prefetchers –as we will show in this paper– if no benefits are expected, even with low aggressiveness, could damage the performance of some applications due to inter-application interference.

To provide further insights on the mentioned behavior, this paper characterizes the relation between main memory (prefetches and regular accesses) activity and performance on SPEC CPU2006 while multiple applications are running together in **multicore execution**. The study shows that most benchmarks use to exhibit execution phases (e.g. *memory intensive prefetch friendly*) that can be highly benefited by prefetching, and execution phases (e.g. *memory intensive prefetch unfriendly*) that are negligibly benefited or even negatively affected. This study suggests that, in multicore execution, properly handling the memory bandwidth of those applications in *memory intensive prefetch unfriendly* phases with adaptive prefetchers can help improve performance of applications in *memory intensive prefetch friendly* phases.

In this paper we propose the Activation/Deactivation Prefetcher (ADP), which in addition to throttle up/down the aggressiveness, activates and deactivates individual per-application prefetchers considering both local and global (inter-application interference) information. Deactivation policies turn off the prefetcher in specific cores, thereby increasing the available bandwidth for those prefetchers that re-

quire it to improve their cores' performance. Activation policies rely on activation conditions that estimate when an individual prefetcher could improve the performance. The key challenge of activation conditions is that they must be applied to reactivate prefetches when the prefetcher engine is **turned off** and there is no information about prefetcher activity (e.g. accuracy or coverage).

ADP has been compared with an aggressive prefetcher and the state-of-art Hierarchical Prefetcher Aggressiveness Control (HPAC) [8]. Experimental results, when running 4 applications concurrently, show that ADP increases performance up to 33.8% (12.3% on average) compared to no prefetching. In contrast, HPAC and aggressive prefetching *only* increase performance by 7.8% and 3.2% on average. Regarding energy, HPAC and the aggressive prefetcher increase main memory energy consumption by 12% and 20%, while ADP performance improvements are reached with minimal main memory energy consumption increase (only by 3%) over no prefetching.

This work makes two main contributions:

- Our characterization study shows that benchmarks exhibit execution phases that can be highly benefited by prefetching, and phases that can be adversely affected. Based on this fact, we show that i) properly handling the memory bandwidth with adaptive prefetchers can help improve performance over aggressive prefetching, and ii) further performance and energy gains can be achieved by selectively deactivating/activating individual prefetchers.

- The proposed ADP prefetcher improves both performance and energy consumption with respect to the state-of-art Hierarchical Prefetcher Aggressiveness Control (HPAC) scheme [8] in both memory-intensive workloads and in workloads combining memory and CPU intensive applications.

We would also like to remark that the proposed activation/deactivation policies are orthogonal to the underlying prefetcher, so they can be applicable to stream-based prefetchers, global-history-buffer delta correlation prefetchers, PC-based stride prefetchers, and temporal prefetchers [9, 10]. Regarding prefetchers implemented in commercial machines, the IBM POWER8 [1, 11] prefetches up to three sequential instruction lines in single thread mode. Additionally, it has a stream based data prefetcher that detects strides in load requests and optionally store requests, issuing prefetches in all the three levels of the cache hierarchy. In Intel processors [12], there are four prefetchers per core. It has two L1-data cache prefetchers, namely an One Block Lookahead prefetcher (OBL), that fetches the

next cache line, and a prefetcher that detects strides in the load history by indexing the loads with the program counter. It also has two L2 cache prefetchers, an L2 adjacent cache line prefetcher, and another one that fetches additional cache lines.

The remainder of this paper is organized as follows. Section 2 summarizes the related work. Section 3 describes the baseline system. Section 4 presents the characterization study. Section 5 introduces the proposal. Section 6 presents the evaluation methodology. Section 7 evaluates the proposal. Finally, Section 8 presents some concluding remarks.

2. Related Work

This section describes previous work focusing on the prefetcher aggressiveness, the reduction in the number of memory requests, and other proposals addressing multicores.

In [13] the AC/DC adaptive method for prefetching data from main memory to the L2 cache is proposed. Like the mechanism devised in this work, AC/DC uses concentration zones (also called *CZones*) [14] that divide memory into fixed size zones. The mechanism is enhanced to make use of delta correlations to find access patterns. They propose an adaptive algorithm that dynamically adjusts the prefetch degree in a range from 2 to 16. The mechanism provides the opportunity to turn off the prefetcher but only in those cases where prefetching hurts the system performance, and no policy is devised to turn on the prefetcher again. More recently, PATer [5] has been proposed. It uses a prediction model based on machine learning with the aim of dynamically tuning the prefetch configuration in the IBM POWER8 with more than 2^{25} configurations. This prediction is based on the value of performance monitoring counters. Opposite to our proposal, that one is specific to POWER8 processor.

The FDP adaptive approach, presented in [7], dynamically selects among five different levels of aggressiveness, ranging from very conservative to very aggressive. The baseline prefetcher is a stream prefetcher like the one used in this work. Similarly to our work, the prefetcher selects at the end of each sampling interval the aggressiveness for the next interval. For this purpose, accuracy, lateness, and pollution metrics are used to throttle up or down the aggressiveness level. This mechanism was extended in another proposal, the Hierarchical Prefetcher Aggressiveness Control (HPAC) [8]. HPAC is proposed for multicore processors, where each core implements a FDP prefetcher, but local decisions to change the aggressiveness can be overridden by the memory controller, which collects global

information about the memory requirements of each application. Unlike our work, these proposals always keep enabled the prefetcher.

Other previous proposals use prefetch filtering techniques. A recent work [4] proposes a weighted majority filter to predict the usefulness of the prefetch addresses. Other works, like [15, 16] estimate *a priori* if a given prefetch will be useful or not, discarding it in the latter case. While their goal is the same as ours, the metrics used are not. For example, our proposal considers the memory contention for decision taking, and not just if a prefetch is useful for the core that has issued it. Additionally, the way the goal is achieved is also different. Our approach throttles and disables the prefetcher, but filtering techniques dynamically decide whether to issue or not a prefetch, based on prefetch history. However, both techniques are orthogonal to each other, so using them together could further reduce the amount of wasted bandwidth.

Regarding main memory bandwidth, [17] proposes FST, a throttling mechanism that limits the number of memory requests that each application is allowed to launch to the main memory. Unlike the previous schemes, both regular and prefetch memory requests are taken into account. This mechanism works on global information, such as the interference that an application causes to its corunners, or the memory bandwidth each application consumes.

A prefetcher that classifies prefetches according to their impact on performance is proposed in [18]. This impact is estimated with a history table indexed by the program counter that collects the stall cycles caused by each load. The mechanism prioritizes the prefetches associated to loads that have caused more ReOrder Buffer (ROB) stalls. That is, the prefetcher is mainly guided by core performance instead of prefetcher performance.

In [19], Sandbox Prefetching, a mechanism to determine at runtime the appropriate prefetcher is proposed. Rather than actually fetching data into the cache to evaluate the prefetcher accuracy, the tags for the blocks that the evaluated prefetcher would fetch are stored in a Bloom Filter. On each memory access, the Bloom Filter is checked to estimate the expected accuracy of the prefetcher. The candidate prefetchers are evaluated one at a time, in a multiplexed fashion, with the sandbox being reset in between evaluations. The main weakness of this mechanism, unlike ours, is that prefetch decisions are taken local to the cores without considering global conditions. More recently, Best–offset Prefetching [20] has been proposed as a Sandbox improvement by considering prefetch timeliness to calculate the prefetch offset.

Prefetching performance can be also improved by enhancing the policies managing memory requests at the shared resources, that is, the arbiter at the NoC or

the scheduling policy at the memory controller. Regarding the NoC, some interesting approaches [21, 22] implement virtual channels and dynamically adjust the priority between regular and prefetch requests coming from multiple cores. With respect to memory controller policies, recent proposals [23, 24, 25] have also focused on multicores. These policies take into account the prefetcher performance to dynamically select the priority of both regular and prefetch requests. NoC and memory controller works are orthogonal to our proposal and can be applied together to achieve further performance improvements.

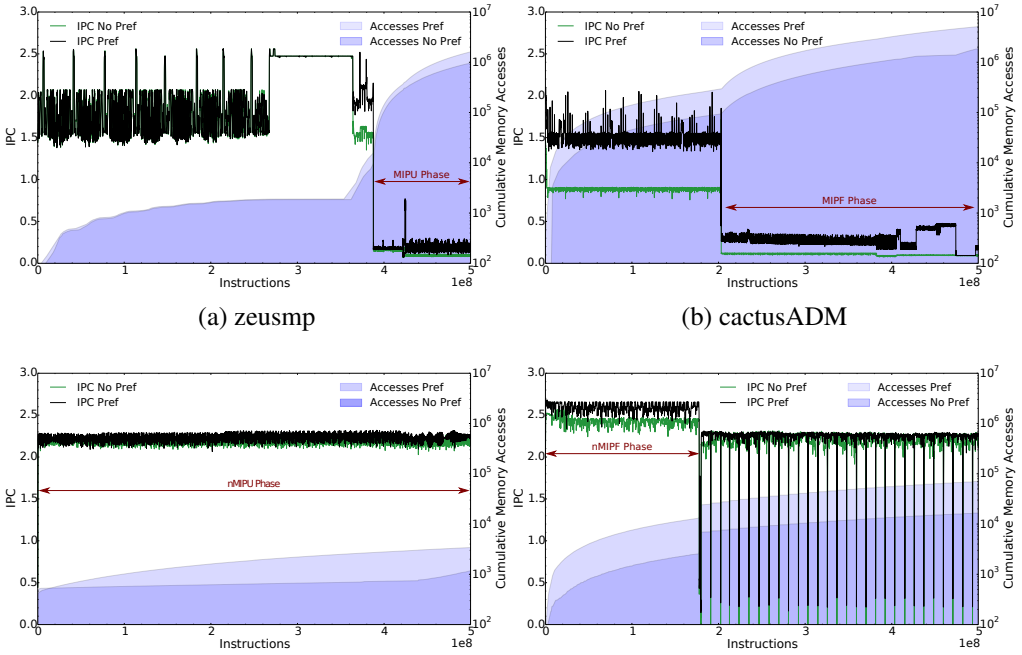
A preliminary version of this work, with a simpler prefetching mechanism, is presented in [26]. Our current proposal achieves better performance, uses a more detailed system model, includes a characterization study and provides more experimental results.

3. Baseline System

The three main components of the modeled baseline system are the cores, the network on chip and the memory controller.

Each tiled core consists of a processing unit, its private caches and the prefetching engine. The prefetcher engine used as baseline is the stride-based prefetcher described in [14], which brings blocks from the main memory to the second level cache (L2). The basic idea is to dynamically partition the physical address space in different zones, referred to as CZones, and to detect strided references within each of these zones [13, 14]. Two memory requests belong to the same partition if they have the same tag (higher order) bits. The processor sets the size of the tag by storing a mask in memory-mapped references. A history buffer [27] is used to store the tags of the currently active partitions. Additionally, a stride field is added to keep the prefetch stride. Strided references within each partition are dynamically detected by using a finite state machine (FSM), which checks whether the last three accesses are offsetted by a fixed stride. If so, a pattern has been detected and the engine starts a new prefetch. Prefetches based on history buffers indexed by CZones have the desirable property of not needing the program counter value of the memory instruction to predict the following accesses, which is important for L2 and lower level caches.

The prefetched data are kept in an auxiliary buffer, called stream buffer [28]. A stream buffer is a small memory close to the cache that only stores prefetched data blocks waiting to be accessed by the core, that is, data blocks that have been brought from main memory but have not yet been requested by the processor and are still in a speculative state. Stream buffers are used to avoid cache pollution



(a) zeusmp (b) cactusADM

(c) povray (d) namd

Figure 1: Characterization study. Examples of benchmarks in the different categories. No prefetching and aggressive prefetch are used.

because speculative data allocated in the cache can replace cache blocks that the core could reference later. Each stream buffer tracks a different stream and consists of a tag, a valid bit, and some data lines of that stream depending on the prefetcher aggressiveness.

In addition, the NoC and the memory controller have been also modeled in detail for the sake of accuracy. The reason is that as the core count increases, the NoC is becoming a critical component affecting the overall system performance, energy consumption, and reliability of emerging multicore systems [29, 30, 31]. The NoC connects all the on-chip components. In this work, a typical mesh topology has been implemented, where each node consists of the tiled core and a router. Congestion and contention are realistically modeled because they are key contributors of the network latency. After traversing the NoC, memory requests reach the memory controller to access to the main memory. Both main memory organization and the memory controller are key contributors to the memory latency

perceived by the processor [32]. Therefore, the memory subsystem should be accurately modeled to obtain representative performance and energy results.

4. Characterization Study

The aim of this section is to study the relation between memory activity, prefetching, and performance (i.e. IPC) in order to provide insights in the design of selective prefetchers. For this purpose, first, all the benchmarks are analyzed in isolation with the aim of identifying those execution phases where prefetching can bring benefits, paying special attention to phases with high memory activity since in such cases, a selective prefetcher can potentially provide extra bandwidth for the co-runners depending on the benefits provided by prefetching. After that, the potential in terms of performance and main memory energy savings of a selective prefetcher in multicore execution is presented.

4.1. Characterizing Benchmark Phases

To analyze individual behavior, all the benchmarks have been run in isolation in a system with and without prefetching[1] showing the obtained IPC and the number of memory accesses.

Execution phases of the benchmarks have been classified in four main categories based on the combined behavior of two main metrics: i) the impact of prefetching on performance (i.e. IPC), and ii) the memory activity of the application. For illustrative purposes, Figure 1 shows examples of benchmarks showing execution phases belonging to the four different categories. Each graph shows the IPC evolution (left Y axis) across the execution time in 500K-instruction intervals with prefetching and without prefetching. To analyze the relationship with the memory behavior, the cumulative amount of memory accesses (right Y axis) is also shown in the same plot using a logarithmic scale.

Below, the four categories are presented highlighting the main characteristics of each of them:

- *Memory Intensive, Prefetch Unfriendly (MIPU)*. This category refers to memory intensive phases where prefetching does not improve the performance over no prefetching. This kind of phase can be observed at the end of the execution of `zeusmp` (see Figure 1a).

[1]Results with prefetching have been obtained with the prefetching mechanism described in Section 3.

- *Memory Intensive, Prefetch Friendly (MIPF)*. This category includes memory intensive phases where prefetching brings important performance benefits. This kind of behavior dominates through the entire execution of some benchmarks, like `cactusADM`, as shown in Figure 1b.

- *non Memory Intensive, Prefetch Unfriendly (nMIPU)*. These phases refer to those excerpts of the execution where the memory activity is rather low and prefetching brings scarce or null benefits. Examples of phases in this category can be observed in `povray` across all its execution (see Figure 1c) and at the beginning of the execution of `zeusmp`.

- *non Memory Intensive, Prefetch Friendly (nMIPF)*. This category refers to non memory intensive phases in which prefetching can boost the performance. This behavior can be observed in Figure 1d during the first part of the execution of `namd`.

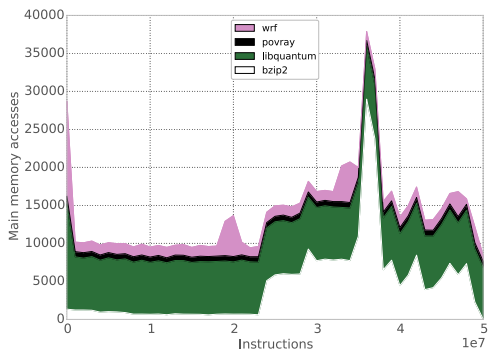
We define an execution phase as a fragment of the execution of an application where IPC behavior is homogeneous or follows the same pattern. Therefore, we assume that a new phase in the execution starts when the IPC changes its trend. So they do not have a predetermined length but depend on the application behavior. We found that these changes are usually related to variations in memory activity.

During prefetch unfriendly phases (categories MIPU and nMIPU), the prefetcher could be turned off or throttled down with minimal impact on performance. This claim can be observed at the end of the execution in Figure 1a, where a significant amount (notice the log scale) of prefetches brings minor performance benefits. Therefore, deactivating the prefetcher could result in important main memory energy savings, especially in MIPU phases where a high number of accesses can be reduced.

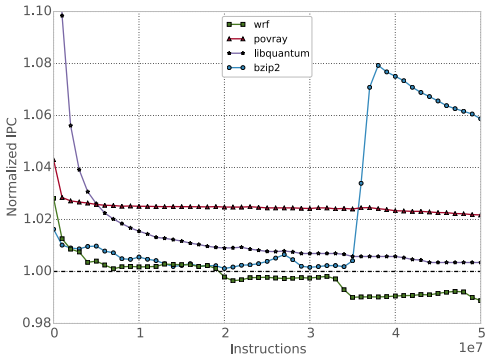
In prefetch friendly phases (categories MIPF and nMIPF), the prefetcher should be enabled to enhance the performance; however, its aggressiveness can be adjusted. This would reduce wasted energy and it is particularly useful in multicore execution in order to leave more bandwidth to the co-runners. Especial attention should be paid to nMIPF phases since the potential energy savings might not compensate the possible performance losses.

4.2. Analysis in Multicore Execution

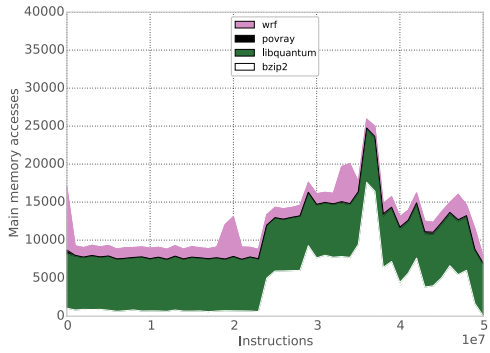
As mentioned above, prefetching brings scarce or null benefits in *MIPU* applications/phases in spite of having high memory activity. This observation is especially relevant to multicore execution, where individual prefetching requests



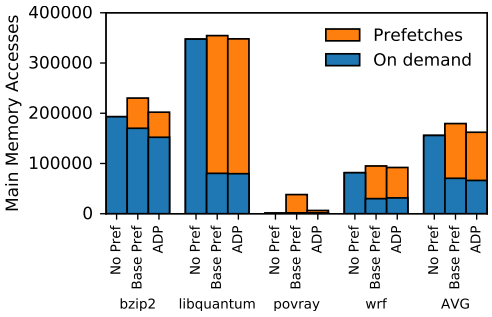
(a) Memory Accesses with the Baseline Prefetcher



(b) Normalized IPC with respect to the Baseline Prefetcher



(c) Memory Accesses with ADP



(d) Aggregated Memory Accesses

Figure 2: Selective ADP prefetcher vs. baseline prefetcher.

compete among them for main memory resources. Therefore, if prefetching could be selectively disabled in specific cores (and enabled when required), then, an extra amount of bandwidth would become available for the co-running applications that really benefit from it. Moreover, important savings in main memory energy could be achieved.

This claim can be observed in Figure 2, which compares the memory activity and performance of a selective prefetcher (ADP, the prefetcher presented in Section 5) with respect to an aggressive prefetcher in multicore execution with four applications. The IPC of ADP is normalized over the baseline aggressive prefetcher. The value for each point is computed from the start of the execution; therefore, the last value represents the overall performance enhancement. It can be

appreciated that the selective prefetcher reduces the number of memory accesses (compare Figures 2a and 2c) by detecting phases in the individual benchmarks where the prefetchers can be disabled without harming the performance. This fact can be clearly observed across the execution of `povray`, a nMIPU application (see Figure 1c), where prefetches are drastically reduced in the selective approach.

Not only does the reduction in prefetchers not decrease the performance; ADP actually increases it. There are two points in Figure 2b of the execution of the applications that are interesting to analyze. The first one, at the beginning of the execution, is a significant increase in performance in comparison with the baseline prefetcher. It appears because the selective prefetcher reduces the amount of prefetches issued by `libquantum` and `wrf`. This reduction unclogs the memory access, which translates in IPC improvements across all the benchmarks. The other point of interest occurs when approximately $3.5e7$ instructions have been executed. At this point, the main memory suffers an important congestion that bottlenecks the system performance. Consequently, the reduction of prefetches alleviates the congestion, reducing the perceived memory latency, which turns into performance enhancements. This claim can be observed in the IPC rise of `bzip2` at the same point of the execution in Figure 2b. This application is the most affected one since it is the most memory intensive at that point. Finally, the minor performance loses exhibited by `wrf` (around 1%) are because the selective prefetcher prioritizes unclogging the main memory access and reduces the aggressiveness of `wrf`'s local prefetcher. However, this slight reduction in the performance of this application is clearly compensated by the increase in performance that the co-runners experience. The reduction in main memory accesses can be also appreciated in Figure 2d, which presents, for each approach (not prefetching, baseline prefetcher and ADP), the total amount of main memory reads, classified in two main groups: prefetch requests and on demand accesses. Two important observations can be drawn: i) the baseline prefetcher significantly increases the total amount of main memory accesses in some applications over not prefetching, while ADP does not suffer this drawback; and ii) an important fraction of on demand accesses are replaced by prefetches, which indicates that the prefetches are useful and their timeliness is adequate, since the block is already in cache when requested, and thus, the main memory access is not performed.

In summary, this analysis has shown that selective prefetching can provide a good tradeoff between main memory accesses reduction (and therefore, energy savings) and performance. Moreover, a good design could enhance both of them. The key challenge that designers must face is to decide when individual prefetch-

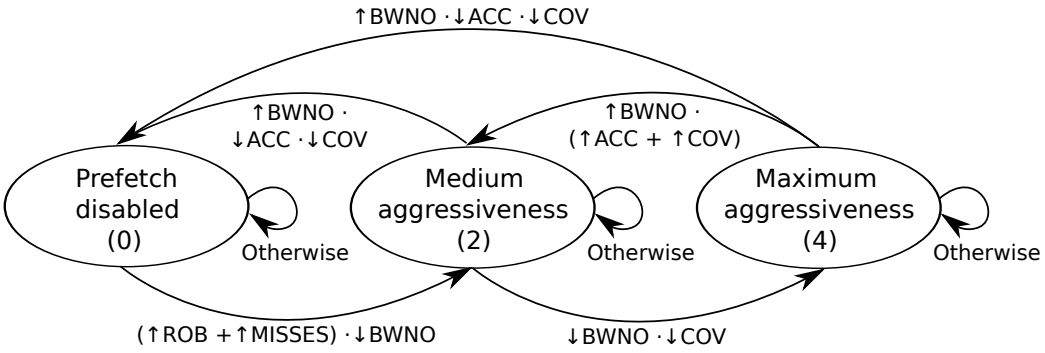


Figure 3: ADP aggressiveness states and transition rules.

ers should be either activated or deactivated.

5. ADP prefetcher

The proposed ADP approach, apart from throttling up or down core prefetchers, also selectively activates and deactivates individual core prefetchers considering both local and global information. This information is used by the devised mechanism to better distribute the available memory bandwidth among the competing cores, leading to a more effective prefetching scheme.

This section introduces the FSM that governs the behavior of the selective prefetcher and discusses the throttling/deactivation and activation policies.

5.1. Finite State Machine of ADP Prefetcher

Existing prefetchers generally use two metrics, accuracy (ACC) and coverage (COV), to quantify the prefetcher performance. Based on these performance metrics, the aggressiveness is throttled accordingly. A recent metric that is being considered in multicore execution is the memory bandwidth needed by others (BWNO). It is computed as follows. First, we define the Bandwidth Consumed by Core *i* (BWC_i) on a given cycle as the number of DRAM banks servicing requests from core *i*. Therefore, for a system using DDR3 memory modules and only one main memory rank, this value is between 0 and 8. Additionally, we define the Bandwidth Needed by Core *i* as the number of banks that are busy serving a request from a core different than *i*, and that have requests pending from core *i*. Based on these, the Bandwidth Needed by Others (BWNO) from the perspective of core *i* is computed as the sum of the bandwidth needed by the cores other than *i*. While BWNO values are computed every cycle, they are averaged for 50K-cycle intervals.

This work uses the three mentioned metrics not only to throttle down the prefetcher aggressiveness but also to completely deactivate the mechanism when it is estimated that the prefetcher is not properly working due to not obtaining benefits. In addition, an extra set of performance metrics has been explored to reactivate the prefetcher. The final design uses two metrics to activate prefetcher (see Section 5.3): the percentage of time the Reorder Buffer (ROB) is stalled due to a long latency memory access (referred to as ROB condition) and the increase in the number of L2 misses (MISSES).

Notice that simple hardware is required to implement the ADP prefetcher and much of it is based on performance counters already available in current processors [33]. Both the *per process* number of cache misses and stall cycles can be gathered on most current commercial processors with the available performance counters. ACC can be calculated as the ratio of two hardware counters that keep track of the number of useful prefetches and the total number of prefetches in each core. This can be done by adding a single bit to each cache entry to indicate that the block has been prefetched [34, 35]. The first counter is updated when there is a hit in a prefetched block and the second one is increased each time a prefetch is issued. COV is computed as the ratio between the counter keeping track of the number of cache misses and the performance counter tracking the number of useful prefetches. With respect to BWNO, it requires three simple counters in the memory controller which are updated every cycle as explained in [8].

Figure 3 depicts the FSM (Finite State Machine) of the ADP prefetcher. The number between brackets within each node represents a prefetcher aggressiveness level and the arcs represent transitions between states. Transitions are labeled with the condition driving the corresponding state change. Upward and downward arrows in the labels mean high or low values (e.g. high or low accuracy), respectively, compared to a threshold (see Section 7).

The devised policies adjust the prefetcher depending on the values of the mentioned performance metrics, which are gathered during fixed-length intervals of 50K processor cycles. At the end of each interval, the hardware logic determines the machine state for the following interval. Below, we detail the deactivation/throttling and activation policies for the devised selective prefetcher.

5.2. Deactivation/throttling policy

This policy is applied at the end of all the intervals when the prefetcher is activated to decide if a state change is required for the next interval. During the interval the metrics used by this policy have been tracked and at the end of the interval they are evaluated to make the decision on a state change. Three main

Algorithm 1 Deactivation/throttling algorithm.

```
1: if co-runners need more bandwidth (BWNO) then
2:     if low accuracy and low coverage then
3:         disable prefetch;                          (2 → 0 || 4 → 0)
4:     else
5:         reduce local aggressiveness;               (2 → 2 || 4 → 2)
6: else
7:     if low coverage then
8:         increase local aggressiveness;             (2 → 4 || 4 → 4)
```

Algorithm 2 Activation algorithm.

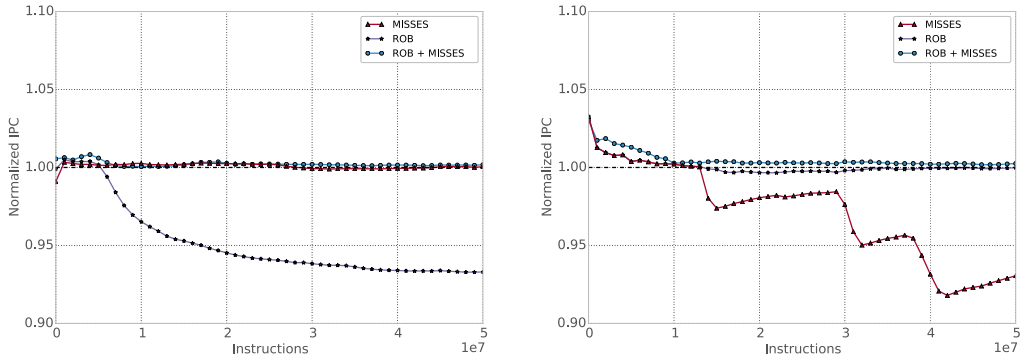
```
1: if sudden rise in misses (MISSES) or high ROB stalls due to memory instructions (ROB) then
2:     if co-runners do not need more bandwidth (BWNO) then activate prefetcher;   (0 → 2)
```

changes can be selected: throttle up the aggressiveness, throttle down the aggressiveness, or turn off the prefetcher. Algorithm 1 depicts the conditions that must be satisfied to carry out such actions. On the right side of each action, the associated transitions in the FSM (Figure 3) are presented.

When some co-runners need more bandwidth, the option to reduce or even deactivate the local prefetcher is checked. In case the local prefetcher is performing poorly (low accuracy and coverage), then the prefetcher is completely disabled. Otherwise, the aggressiveness is set to its midlevel (remember that ADP aggressiveness levels have been set to 0 –disabled–, 2, and 4) to increase memory bandwidth availability for the co-runners. On the other hand, if BWNO is not a constraint and the local prefetcher is not saving enough cache misses, then the mechanism speculatively increases the aggressiveness (aggressiveness level is set to 4, the maximum value) to improve its performance. Upon misspeculation, the algorithm will return to the previous aggressiveness in the subsequent interval.

5.3. Activation policy

This policy is applied at the end of each interval in the cores when the prefetcher is disabled to decide whether it should be reactivated for the next interval. When a local prefetcher is disabled, all the prefetcher related structures like stride and pattern detection are disabled, thus no information about the prefetcher activity is available to make the decision.



(a) Performance of mcf in execution with namd, omnetpp and soplex

(b) Performance of cactusADM in execution with gcc, hmmer and libquantum

Figure 4: Normalized IPC of ADP over the baseline prefetcher using different activation conditions.

The proposed activation policy estimates if noticeable performance losses have appeared with respect to the last interval the prefetcher was enabled, and based on this estimate it determines if the prefetcher should be reactivated the next interval. We compare against the last time the prefetcher was enabled since execution phases tend to exhibit the same behavior for a relatively long time as studied in Section 4.1. The challenge is which performance metrics should be used for this purpose. We need metrics that can provide hints about if enabling the prefetcher could improve performance.

Algorithm 2 summarizes the devised activation mechanism, that relies on two conditions (ROB stalls and cache MISSES). If any of both metrics suffers a sudden rise, that is any of the two conditions (ROB or MISSES) is fulfilled, then the local prefetcher is activated provided that the co-runners do not need more bandwidth. The reason for this restriction is that reactivating the prefetcher when bandwidth is scarce could rise the congestion and damage the global performance.

Activation Conditions Analysis

The goal of the activation conditions is to decide when the prefetcher should be activated in order to keep the benefits of aggressive prefetching on performance. That is, the IPC of prefetch friendly benchmarks should not *significantly* suffer with respect to using an aggressive prefetcher. In other words, main memory bandwidth savings should not be achieved at the cost of performance.

Among the studied metrics we explored the IPC, the percentage of ROB stalls,

and MISSES in the LLC. With respect to IPC and MISSES, we evaluated the difference (in percentage) between the value in the current and the last interval the prefetcher was enabled. Experimental results proved that IPC and the percentage of time ROB is stalled are inversely correlated since when the ROB is blocked the core cannot follow decoding instructions, and consequently, the IPC drops. In addition, the ROB is mainly blocked because of long latency memory instructions, therefore a ROB based metric can be used to provide insights about when the prefetcher can improve performance. On the other hand, a primary goal of prefetching is saving cache misses. Therefore, a sudden rise in misses since the last time the prefetcher was enabled could be used as a hint to reactivate the prefetcher.

The final design activates the prefetcher when ROB stalls or MISSES conditions are fulfilled. Of course, using only one of them would be more restrictive and would provide additional bandwidth savings but at the cost of performance. Experimental results show that the use of any of both conditions alone yields to significant performance losses in some applications of the studied workloads. This can be appreciated in Figure 4. Each graph shows the IPC of an individual benchmark in multicore execution –normalized over the IPC obtained by an aggressive prefetcher– for the proposed ADP approach using both activation conditions jointly (ROB + MISSES) and individually. As observed, at the end of the execution, using the ROB condition alone drops the performance by 6% in `mcf` (Figure 4a) and using only the MISSES condition penalizes performance around 7% in `cactusADM` (Figure 4b).

6. Evaluation Methodology

The proposal has been evaluated with the Multi2sim [36] simulation framework. Multi2sim performs detailed simulation of out-of-order execution cores. We have widely extended this simulator to model the prefetchers and the NoC of the proposed system. To obtain accurate DRAM performance and energy results, we linked the DRAMSim2 [37] simulator to Multi2sim. Table 1 shows the configuration parameters of the core, the interconnection network, and the main memory. Main memory parameters have been set according to a recent commercial MICRON DDR3 memory device [38].

Experiments have been performed with multiprogram mixes composed of applications from the SPEC2006 benchmark suite. Each application runs until it commits 300M instructions after fastforwarding 500M instructions. To avoid performance differences caused by early finalization of the execution of some bench-

marks, all the applications are kept running until the slowest benchmark commits the targeted number of instructions. This implies that some benchmarks will execute more instructions than the targeted number. For comparison purposes, performance is measured in these benchmarks for the first 300M committed instructions.

Table 1: System configuration.

Processing core	
# Cores	4 cores at 3GHz
Issuing policy	Out of order
Issue/Commit width	4 instructions/cycle
ROB size	256 entries
Load/Store queue	64/48 entries
Cache hierarchy	
L1 Icache (private)	32KB, 8ways, 64B-line, 2cc
L1 Dcache (private)	32KB, 8ways, 64B-line, 2cc
L2 (private)	512KB, 16ways, 64B-line, 11cc, 16 MSHR
Prefetching logic	
Stream prefetcher	32 16-entry streams in L2
Aggressiveness	4 blocks
Interconnection network	
Topology	Mesh
Routing	X-Y
Input/ouput buffer size	128B
Link bandwidth	64B/cycle
Main memory & memory controller	
DRAM bus freq.	1066MHz
DRAM device	DDR3 (2133 Mtransfers/cycle)
Latency	t_{RP}, t_{RCD}, t_{CL} 13.09ns each
DRAM banks	8
Page size	8KB
Burst length (BL)	8
Scheduling policy	FCFS

6.1. Mix Design

The characterization study presented in Section 4, classified application phases in four different categories. However, although benchmarks tend to exhibit phases in different categories, in most benchmarks a given category clearly dominates over the others. Thus, we can take advantage of this to classify applications according to the category that dominates across its execution time. Following this approach, the applications in SPEC2006 benchmark suite have been classified depending upon the predominant phase in the part of the application that is executed in the experiments (300M instructions after skipping the first 500M instructions). After this classification, a set of mixes has been designed to study the effects of prefetching on performance and energy in two main scenarios: under normal conditions and in extreme conditions stressing the main memory.

Table 2: Mix composition. Numbers 1, 2, 3 and 4 between brackets correspond to categories MIPU, MIPF, nMIPU, and nMIPF, respectively, defined in Section 4.

Mix type	Mix	Benchmarks (categories)			
Combined	m0	tonto (4)	h264ref (3)	hmmer (2)	omnetpp (1)
	m1	bwaves (2)	gamess (3)	GemsFDTD (3)	sjeng (3)
	m2	astar (1)	bzip2 (1)	gcc (2)	GemsFDTD (3)
	m3	gamess (3)	GemsFDTD (3)	leslie3d (2)	wrf (2)
Memory intensive	m4	xalancbmk (1)	gcc (2)	gobmk (2)	dealII (1)
	m5	leslie3d (2)	dealII (1)	soplex (2)	gromacs (2)
	m6	mcf (2)	soplex (2)	perlbench (2)	xalancbmk (1)
	m7	dealII (1)	soplex (2)	xalancbmk (1)	gobmk (2)

To evaluate the first scenario, mixes were designed with benchmarks randomly chosen from the identified categories. To evaluate the second scenario, the designed mixes only include memory intensive applications from MIPU and MIPF categories. We refer to the first type of mixes as *combined* and to the second type as *memory-intensive*. Table 2 shows the composition of the mixes. Mixes from m0 to m3 are combined and mixes from m4 to m7 are memory-intensive.

6.2. Performance Indexes

To evaluate multicore performance, for each mix, we use the harmonic mean of IPC and the *Harmonic Mean of Individual Speedup*, which in addition to quantify performance also provide a notion of fairness [39].

Harmonic Mean of Individual Speedup is calculated according to Equation 1.

$$WS_{hm} = \frac{n}{\sum_{i=1}^{n} \frac{1}{IS_i}} \tag{1}$$

Where IS_i represents the *Individual Speedup* of application i that is obtained according to Equation 2, where *n* refers to the number of benchmarks in the mix.

$$IS_i = \frac{IPC_{i,multi}}{IPC_{i,alone}} \quad \forall i \in \{1, n\} \tag{2}$$

Individual Speedup estimates how much the performance of an individual benchmark is affected by its co-runners, by comparing the IPC of the application running in the CMP with co-runners and running alone. Experimental results consider $IPC_{i,alone}$ as the IPC of application i in isolated execution with the prefetcher enabled.

Table 3: Thresholds used in ADP.

Thresholds				
Low Accuracy	Low Coverage	Rise in misses	High % ROB stall	High BWNO
$< 40\%$	$< 30\%$	$> 15\%$	$> 60\%$	> 2.75 banks

7. Experimental Evaluation

To evaluate the proposed prefetcher, ADP is compared to the same system without prefetching and two other prefetchers: HPAC [8] and an aggressive prefetcher. HPAC is an adaptive prefetcher that implements throttling up and down policies to control the aggressiveness. The aggressive prefetcher is always working at the maximum aggressiveness level (4 blocks). The adaptive prefetchers (HPAC and ADP) use 2-block and 4-block as middle and high aggressiveness levels, respectively. The lowest aggressiveness is set to 1-block for HPAC[2], while ADP completely deactivates the prefetcher[3]. The ADP threshold values used in the experiments for Algorithm 1 and Algorithm 2 are shown in Table 3. Parameters were empirically determined using a limited number of simulation runs and optimized to reduce the number of memory accesses. Therefore, further performance improvements could be achieved but at the cost of increasing the number of main memory accesses.

[2]HPAC does not deactivate prefetchers.

[3]Notice that this allows reducing even more memory accesses for those applications that do not take advantage of prefetching while providing more bandwidth for the others.

7.1. Performance Analysis

Before studying the performance on multicore execution, we explore the potential of the compared prefetching approaches in the system in absence of interference due to memory requests from other applications.

Figure 5 shows the IPC when each application runs alone in the multicore. Labels *No pref* and *Pref* refer to no prefetching and the aggressive prefetcher, respectively, while HPAC and ADP are the adaptive prefetchers. A sample of memory intensive and non memory intensive benchmarks has been taken for illustrative purposes. As observed, aggressive prefetching (*Pref*) brings important performance benefits in most of the applications. Performance improves on average by 16% over no prefetching and up to 34% in the `wrf` benchmark. Notice that the adaptive prefetchers improve the performance of the aggressive prefetcher in memory intensive applications like `bwaves`. This means that aggressive prefetching suffers from limited memory bandwidth even in standalone execution in some benchmarks, a problem that exacerbates when multiple applications are run alongside in the multicore. Therefore, adaptive prefetchers are required in multicores to sustain the performance.

After considering the standalone execution of benchmarks, now we analyze the multicore execution performance of the designed mixes for the different prefetchers. Figure 6 shows the IPC (harmonic mean) of each mix for the studied approaches. The *HM0–3*, *HM4–7* and *HM* columns represent the average values for combined (m0 to m3), memory intensive (m4 to m7), and all the mixes, respec-

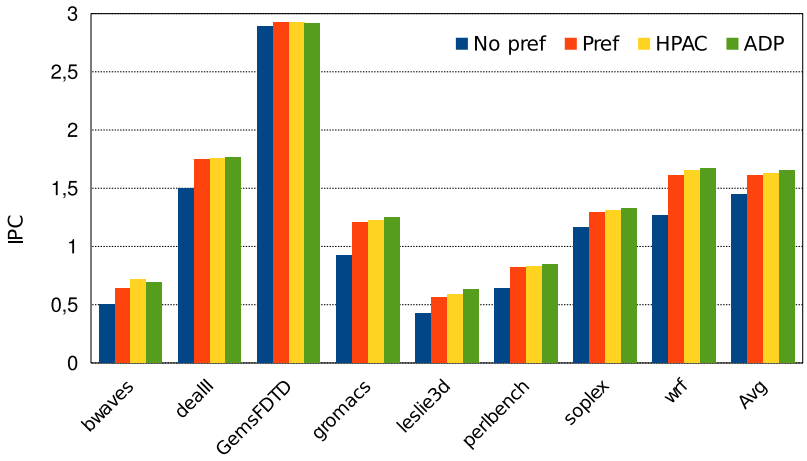


Figure 5: Performance of prefetchers running benchmarks in isolation.

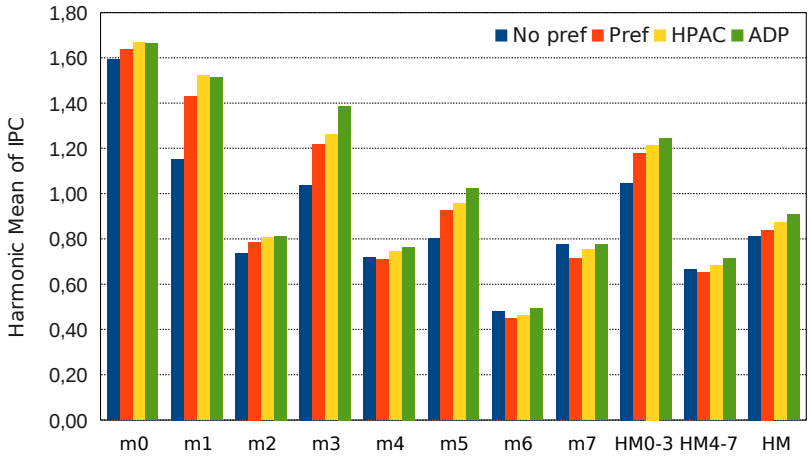


Figure 6: Harmonic Mean of IPC.

tively. As can be observed, the aggressive prefetcher improves the performance compared to no prefetching for combined mixes, but decreases the performance in most of the memory intensive mixes. This performance drop is due to timeliness. That is, the prefetched data come too late from main memory because of the longer latencies experienced in memory intensive mixes, so data are not ready in cache when they are needed [40]. In addition, the extra memory accesses due prefetching delay other requests, penalizing the performance. On the other hand, the adaptive approaches, which stress less the memory hierarchy, perform significantly better than the aggressive prefetcher in both combined and memory intensive mixes.

Compared to HPAC, the proposed approach achieves, on average, better performance regardless of the type of mix. ADP increases IPC by 12.3% with respect to no prefetching considering both types of mixes while HPAC only improves performance by 7.4%. An important observation is that in memory intensive mixes, ADP is the only approach whose performance is on par or even higher than no prefetching. However, in mixes m0 and m1 HPAC is the best performing approach by a slim margin. The reason is that in these cases ADP is too conservative, keeping the prefetcher disabled for too much time. Despite that, it performs better than the baseline prefetcher, and the difference with HPAC is smaller than 1%.

Performance gains of the proposal are also analyzed taking into account Individual Speedup (harmonic mean). Figure 7 shows the results normalized to no prefetching. In combined mixes, all the prefetchers improve the performance achieved by the non-prefetching approach. In contrast, in memory intensive mixes,

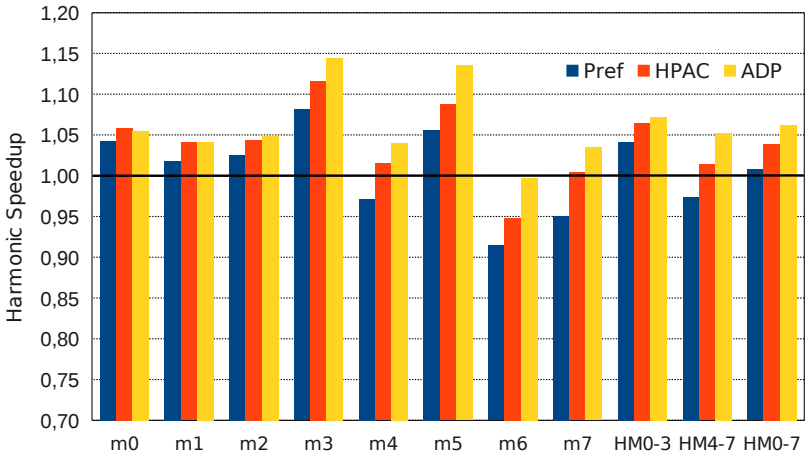


Figure 7: Harmonic Mean of Individual Speedup normalized w.r.t. no prefetching.

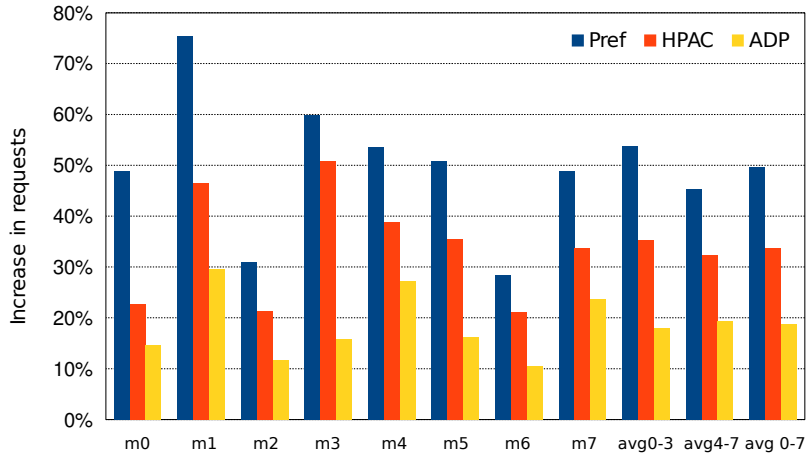


Figure 8: Memory requests increase of the studied prefetchers over no prefetching.

differences among prefetching schemes become wider. ADP improves the performance of no prefetching by 5% while HPAC only improves it by 1%. These results show that ADP, even with scarce memory bandwidth, makes the prefetcher capable of achieving important performance gains.

An interesting observation is that ADP presents a consistent behavior between memory intensive and combined mixes, while HPAC performs worse for intensive ones than for combined mixes.

7.2. Prefetch Activity Reduction Analysis

This section shows how ADP saves memory traffic by reducing the amount of prefetches with respect to HPAC and aggressive prefetching.

Figure 8 shows the increase in the number of memory requests of the studied prefetchers compared to no prefetching. Keeping always the maximum aggressiveness generates more prefetches so more memory bandwidth is consumed, which can strangle the performance in memory intensive mixes. The aggressive prefetcher increases the amount of memory requests, on average, over no prefetching by 50%. In contrast, HPAC and ADP reduce this amount by around one third (requests increase by 34%) and two thirds (requests increase by 19%), respectively.

In short, we conclude that ADP improves performance by significantly reducing the amount of useless prefetches, saving energy and bandwidth.

7.3. Main Memory Energy Analysis

This section compares the main memory energy consumption of the studied schemes. Figure 9 presents the energy results of the DDR3 module provided by the linked DRAMSim2 simulator. This simulator provides accurate performance and energy results since it models memory devices at a very low level. Unlike IPC and memory requests, which are gathered when a benchmark commits 300M instructions (see Section 6), energy consumption at the main memory is gathered at the end of the execution of the mix for simplification purposes. Therefore, energy provided by DRAMSim2 also considers those memory accesses issued after a benchmark executes 300M instructions (where IPC and memory requests metrics are collected) until the slowest benchmark of the mix finishes its execution. This is the reason why bar differences in Figure 9 are not so wide as in Figure 8. Thus, the presented energy results are conservative.

Energy results are broken down in four components depending on the memory activity that consumes the energy: i) activation and precharge, ii) background energy, iii) data bursts, and iv) refresh. The first component accounts for the energy consumed activating rows for reads and writes, plus the energy consumed due to precharging the bitlines. The second component refers to the energy consumed in background to keep memory devices powered on. Burst energy is consumed when data are transferred by the memory bus in write and read operations. Finally, refresh energy is required to avoid capacitors loose the stored value.

The studied prefetchers differ in the number of memory accesses they perform, so this section focuses on energy consumed by DRAM memory modules. Nevertheless, the important reduction in the number of prefetches is also expected

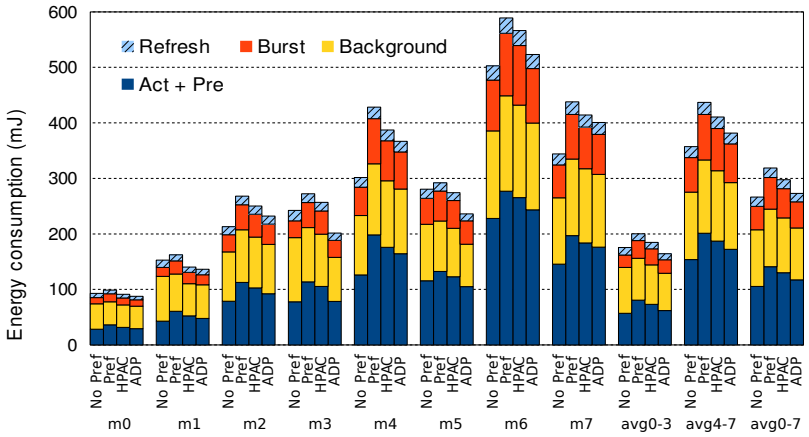


Figure 9: Energy consumption of the prefetching mechanisms.

to save dynamic energy in Last Level Cache structures (where prefetches are triggered) and in the NoC.

As expected, the prefetching schemes consume more activation and precharge energy as well as burst energy than no prefetching since more memory requests are served as observed in Figure 8. On the other hand, employing prefetching helps reduce both background and refresh energy, especially in combined mixes because these mixes significantly reduce their execution time.

The aggressive prefetcher increases total energy consumed by the DDR3 module on average by 20% over no prefetching. This expense in energy may be unacceptable, especially taking into account that aggressive prefetching can damage the performance in memory intensive mixes. An interesting observation is that ADP achieves the performance gains presented in Section 7.1 with a minimal impact (a 3% increase) on the total memory energy, which is a much lower impact than the one obtained with adaptive HPAC prefetcher (a 12% increase).

7.4. Benefits of Deactivating the Prefetcher

To quantify which part of the benefits come from completely deactivating the prefetcher, we increased the minimum aggressiveness of ADP from 0 –disabled– to 1, keeping unchanged the remaining state machine (see Figure 3). The only difference is that this approach transits to a minimum aggressiveness level instead of turning off the prefetcher.

Figure 10 shows how the number of prefetches increases (in percentage) when the minimum aggressiveness is set to 1 instead of completely turning off the

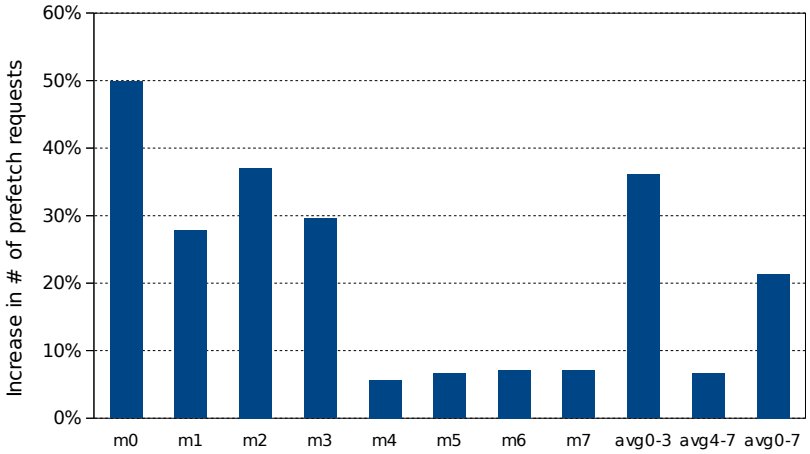


Figure 10: Effect of changing the minimum aggressiveness from 0 (completely disabled) to 1 in the number of prefetch requests for ADP.

prefetcher. As observed, keeping activated the prefetcher even with minimum aggressiveness increases the number of prefetches on average by 36% in combined mixes and by 7% in memory intensive mixes. Moreover, this reduction is achieved with minor performance differences (less than 1% on average). If the total number of memory requests (prefetches and on demand accesses) are taken into account, the overall amount of requests increases by 6% and by 2% for combined mixes and memory intensive mixes, respectively when the prefetcher is not deactivated. These results show i) the important impact of deactivating the prefetcher in the reduction of memory accesses and ii) that the devised activation/deactivation policies work properly, since performance remains almost the same.

7.5. Analysis with a different prefetching mechanism

While the results presented in the previous section have all been obtained using an specific prefetching mechanism, both ADP and HPAC, and in general other techniques that smartly adapt the prefetching aggressiveness are orthogonal to the underlying prefetcher. To show this, in this section we present some results obtained with a different prefetcher, inspired in one of the used by current Intel processors [12], that detects patterns in load streams, classifying them using the Program Counter.

As figures 11 and 12 show, similar results are achieved when a different underlying prefetcher is used. Although the IPC increases due selective prefetching, shown in Figure 11, are not as high as with the prefetcher evaluated in the previous

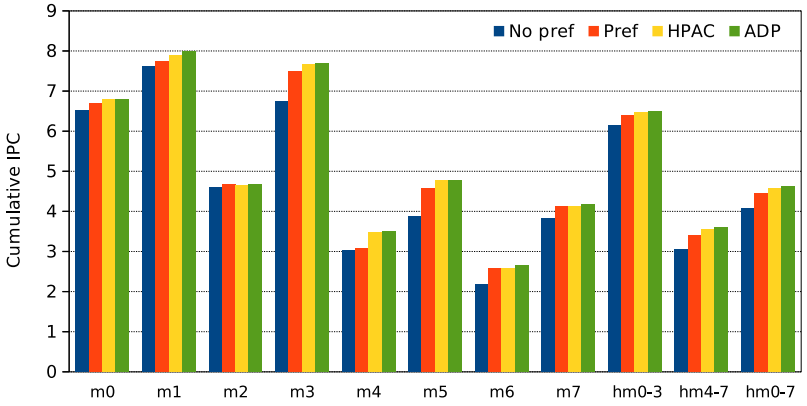


Figure 11: Cumulative IPC for the studied mixes.

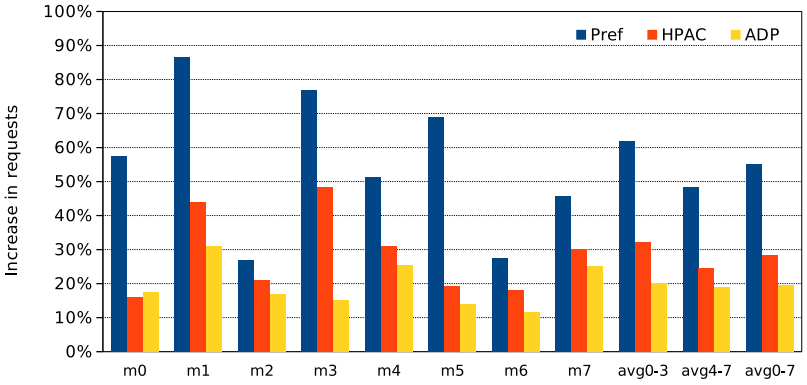


Figure 12: Relative increase in memory accesses, compared with no prefetching.

sections, they are still important. This is because the thresholds used to transition to the different states of the activation/deactivation mechanism need further refinement, out of the scope of this work. Regarding the reduction in memory requests issued, the results are similar to the ones already discussed. Note that in both cases ADP still performs better than HPAC.

8. Conclusions

In this work we have characterized the behavior of prefetching in multicores. This study has shown that some applications exhibit execution phases where local prefetching could be disabled in order to allow higher bandwidth available to their co-runners, with minimal impact on the local performance. This way would allow

to improve their co-runners' performance and achieve energy savings, especially in main memory modules.

This paper has presented the ADP selective prefetcher that dynamically deactivates or activates individual core prefetchers. A core prefetcher is deactivated when the co-runners need more bandwidth, provided that the local prefetcher presents low accuracy and coverage regardless of the actual prefetcher aggressiveness. ADP smartly activates the prefetcher based on activation conditions that estimate if prefetches will improve the system performance, but only if the co-runners do not need more memory bandwidth.

ADP increases performance on average by 12.3% over no prefetching considering both memory intensive and combined mixes, while HPAC improves this metric by 7.4%. Compared to no prefetching, the aggressive prefetcher and HPAC increase the amount of memory requests by 50% and 34%, respectively, while ADP reduces this amount as much as 19%. This reduction, jointly with speeding up the execution time, results in important main memory energy savings. On average, energy consumption increases by 2% and 20% in HPAC and the aggressive prefetcher, respectively, over no prefetching. In contrast, in ADP, energy just increases by 3%. Additionally, we have evaluated ADP with a different underlying prefetcher, and the results looked promising, proving the orthogonality of the approach.

Finally, we can conclude that selective prefetching may be a valuable approach for future multicores bringing the best of two worlds: performance improvements with respect to state-of-the-art prefetchers under limited available memory bandwidth or memory-hungry applications, with minimal increase of energy consumption over no prefetching.

[1] B. Sinharoy, J. A. V. Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, K. M. Fernsler, Ibm power8 processor core microarchitecture, IBM J. of Res. and Dev. 59 (1) (2015) 2:1–2:21.

[2] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, Y. C. Liu, Knights landing: Second-generation intel xeon phi product, IEEE Micro 36 (2) (2016) 34–46.

[3] J. Owen, M. Steinman, Northbridge architecture of amd's griffin microprocessor family, IEEE Micro 28 (2) (2008) 10–18. doi:10.1109/MM.2008.29.

[4] S. B. B. Panda, Expert prefetch prediction: An expert predicting the usefulness of hardware prefetchers, IEEE Computer Architecture Letters 15 (1).

[5] M. Li, G. Chen, Q. Wang, Y. Lin, P. Hofstee, P. Stenstrom, D. Zhou, Pater: A hardware prefetching automatic tuner on ibm power8 processor, IEEE Computer Architecture Letters 15 (1).

[6] A. Flores, J. L. Aragón, M. E. Acacio, Energy-efficient hardware prefetching for cmps using heterogeneous interconnects, in: 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, 2010, pp. 147–154. doi:10.1109/PDP.2010.12.

[7] S. Srinath, O. Mutlu, H. Kim, Y. Patt, Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers, in: HPCA., 2007, pp. 63–74. doi:10.1109/HPCA.2007.346185.

[8] E. Ebrahimi, O. Mutlu, C. J. Lee, Y. N. Patt, Coordinated control of multiple prefetchers in multi-core systems, in: MICRO, 2009, pp. 316–326. doi:10.1145/1669112.1669154.

[9] A. Jain, C. Lin, Linearizing irregular memory accesses for improved correlated prefetching, in: MICRO, 2013, pp. 247–259. doi:10.1145/2540708.2540730.

[10] C. Kaynak, B. Grot, B. Falsafi, SHIFT: Shared History Instruction Fetch for Lean-core Server Processors, in: MICRO, 2013, pp. 272–283. doi:10.1145/2540708.2540732.

[11] P. Bergner, Performance Optimization and Tuning Techniques for IBM Power Systems Processors Including IBM POWER8, IBM redbooks, 2015. URL https://books.google.es/books?id=bdP_jgEACAAJ

[12] V. V. (Intel), Disclosure of H/W prefetcher control on some Intel processors, https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.

[13] K. Nesbit, A. Dhodapkar, J. Smith, Ac/dc: an adaptive data cache prefetcher, in: PACT, 2004, pp. 135–145. doi:10.1109/PACT.2004.1342548.

[14] S. Palacharla, R. E. Kessler, Evaluating stream buffers as a secondary cache replacement, in: ISCA, 1994, pp. 24–33. doi:10.1145/191995.192014.

[15] X. Zhuang, H. h. S. Lee, Reducing cache pollution via dynamic data prefetch filtering, IEEE Transactions on Computers 56 (1) (2007) 18–31. doi:10.1109/TC.2007.250620.

[16] X. Dang, X. Wang, D. Tong, Z. Xie, L. Li, K. Wang, An adaptive filtering mechanism for energy efficient data prefetching, in: 2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC), 2013, pp. 332–337. doi:10.1109/ASPDAC.2013.6509617.

[17] E. Ebrahimi, C. Joo, L. Onur, M. Yale, N. Patt, Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems, in: ASPLOS, 2010, pp. 335–346.

[18] R. Manikantan, R. Govindarajan, Performance oriented prefetching enhancements using commit stalls, J. Instruction-Level Parallelism 13.

[19] S. Pugsley, Z. Chishti, C. Wilkerson, T. Chuang, R. Scott, A. Jaleel, S.-L. Lu, K. Chow, R. Balasubramonian, Sandbox prefetching: Safe, run-time evaluation of aggressive prefetchers, in: HPCA, 2014.

[20] P. Michaud, Best-offset hardware prefetching, in: 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016, 2016, pp. 469–480. doi:10.1109/HPCA.2016.7446087.
URL http://dx.doi.org/10.1109/HPCA.2016.7446087

[21] N. Chidambaram Nachiappan, A. K. Mishra, M. Kademir, A. Sivasubramaniam, O. Mutlu, C. R. Das, Application-aware prefetch prioritization in on-chip networks, in: PACT, 2012, pp. 441–442. doi:10.1145/2370816.2370886.

[22] J. Lee, M. Shin, H. Kim, J. Kim, J. Huh, Exploiting mutual awareness between prefetchers and on-chip networks in multi-cores, in: PACT, 2011, pp. 177–178. doi:10.1109/PACT.2011.27.

[23] E. Ebrahimi, C. J. Lee, O. Mutlu, Y. N. Patt, Prefetch-aware shared resource management for multi-core systems, in: ISCA, 2011, pp. 141–152. doi:10.1145/2000064.2000081.

[24] F. Liu, Y. Solihin, Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors, in: ICMMCS, 2011, pp. 37–48. doi:10.1145/1993744.1993749.

[25] F. Liu, X. Jiang, Y. Solihin, Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance, in: HPCA, 2010, pp. 1–12. doi:10.1109/HPCA.2010.5416655.

[26] V. Selfa, J. Sahuquillo, M. E. Gómez, C. Gómez, A Simple Activation/Deactivation Prefetching Scheme for Chip Multiprocessors, in: PDP, 2016, pp. 143–150.

[27] K. J. Nesbit, J. E. Smith, Data cache prefetching using a global history buffer, in: HPCA, 2004, pp. 96–. doi:10.1109/HPCA.2004.10030.

[28] N. Jouppi, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, in: ISCA, 1990, pp. 364–373. doi:10.1109/ISCA.1990.134547.

[29] A. Sharifi, E. Kultursay, M. T. Kandemir, C. R. Das, Addressing end-to-end memory access latency in NoC-based multicores, in: MICRO, 2012, pp. 294–304.

[30] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, C. R. Das, Exploring fault-tolerant network-on-chip architectures, in: DSN, 2006, pp. 93–104.

[31] Y. Hoskote, S. R. Vangal, A. Singh, N. Borkar, S. Borkar, A 5-ghz mesh interconnect for a teraflops processor, IEEE Micro 27 (5) (2007) 51–61.

[32] B. Jacob, D. Wang, Principles and practices of interconnection networks, Morgan Kaufmann, 2007.

[33] P. Irelan, S. Kuo, Performance monitoring unit sharing guide - white paper.

[34] A. J. Smith, Cache memories, ACM Comput. Surv. 14 (3) (1982) 473–530. doi:10.1145/356887.356892.
URL http://doi.acm.org/10.1145/356887.356892

[35] J. D. Gindele, Buffer block prefetching method, IBM Technical Disclosure Bulletin, 20(2):696-697, 1977.

[36] R. Ubal, J. Sahuquillo, S. Petit, P. Lopez, Multi2sim: A simulation framework to evaluate multicore-multithreaded processors, in: SBAC-PAD, 2007, pp. 62–68. doi:10.1109/SBAC-PAD.2007.17.

[37] P. Rosenfeld, E. Cooper-Balis, B. Jacob, Dramsim2: A cycle accurate memory system simulator, IEEE Comput. Archit. Lett. 10 (1) (2011) 16–19. doi:10.1109/L-CA.2011.4.

[38] Micron, Data sheet: 4Gb DDR3 SDRAM MT41J512M8-64Meg x 8 x 8 banks.

[39] A. Snavely, D. M. Tullsen, Symbiotic Jobscheduling for a Simultaneous Multithreading Processor, in: ASPLOS, 2000, pp. 234–244. doi:10.1145/356989.357011.

[40] W. A. Wong, J.-L. Baer, The Impact of Timeliness for Hardware-based Prefetching from Main Memory, Technical Report.