Bandwidth-Aware Dynamic Prefetch Configuration for IBM POWER8

Carlos Navarro , Josué Feliu , Salvador Petit , Maria E. Gómez , Julio Sahuquillo

Abstract—Advanced hardware prefetch engines are being integrated in current high-performance processors. Prefetching can boost the performance of most applications, however, the induced bandwidth consumption can lead the system to a high contention for main memory bandwidth, which is a scarce resource in current multicores. In such a case, the system performance can be severely damaged. This work characterizes the applications' behavior in an IBM POWER8 machine, which presents many prefetch settings, varying the bandwidth contention. The study reveals that the best prefetch setting for each application depends on the main memory bandwidth availability, that is, it depends on the co-running applications. Based on this study, we propose Bandwidth-Aware Prefetch Configuration (BAPC) a scalable adaptive prefetching algorithm that improves the performance of multi-program workloads. BAPC increases the performance of the applications in a 12%, 15%, and 16% of 6-, 8-, and 10-application workloads over the IBM POWER8 default configuration. In addition, BAPC reduces bandwidth consumption in 39%, 42%, and 45%, respectively.

Index Terms—Prefetch engine, prefetch settings, performance measures.

1 Introduction

Hardware data prefetching is an effective technique to hide the long memory latency and can greatly improve the performance of many applications. Much research [1], [2], [3], [4], [5] focusing on monolithic processors has shown the high performance enhancements prefetch approaches can bring. Because of this fact, many works [6], [7], [8], [9], [10], [11] have focused on data prefetching for multicores over the last decade. Research has concentrated on either parallel workloads or multi-program workloads consisting of a set of single-threaded applications, each one running on a different core, and exhibiting a different memory access pattern. To deal with this issue, some approaches, like *sandbox prefetching* [6], propose to implement a set of prefetchers and select the best prefetcher for each core, depending on the running application.

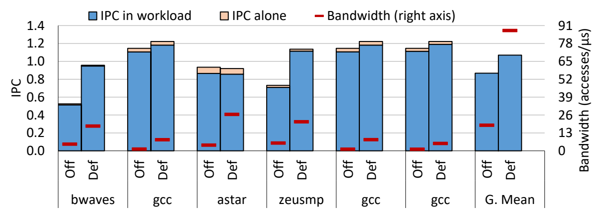
The major shortcoming of prefetching is the introduced interference due to it is a speculative technique. In other words, prefetch requests compete with regular requests for memory resources, which can adversely impact on performance. This issue has been addressed in some approaches [7], [8], [9], [10], [11]. Despite the interference, the important performance gains shown by prefetching in multicores over the last decade have led processors manufacturers to integrate complex prefetch engines in modern processors. These prefetchers, especially in high-performance servers, are deployed with a wide set of prefetch settings aimed at capturing the different access patterns that applications exhibit. Thanks to this hardware support, recent research has focused on commercial machines.

LibPRISM [12] focuses on dynamically selecting at runtime the best prefetcher of an IBM POWER8 to enhance the system performance when running a single parallel workload. In [13] an *adaptive* prefetcher is proposed for the IBM POWER7 processor, where the prefetch *aggressiveness* of each application can be throttled down and up. The study presents results when running just two benchmarks. This approach allows the prefetching to achieve good performance, however, it does not scale with the number of cores. Probably because of this fact, in a posterior work [14] by the same authors, a more scalable approach is devised but at the cost of limiting the prefetching flexibility. Instead of an adaptive prefetching, authors rely on a less efficient but much more simple *on/off* approach.

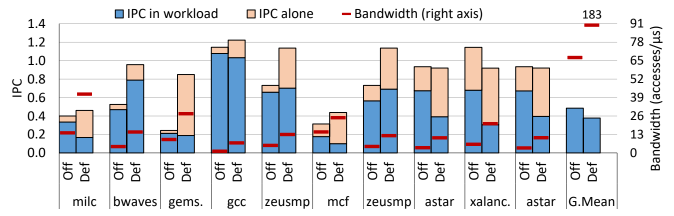
In order to make a prefetch approach scalable and improve performance when a large number of applications is running on the system, the approach must trade off performance and memory bandwidth consumption. To illustrate this claim, Figure 1 presents two cases studied on a 10-core IBM POWER8 machine with the prefetching disabled (Off) and the *default prefetch setting* or simply default prefetcher (Def). Figure 1a and Figure 1b present the IPC of each application running in the mix and in isolation of two workloads consisting of 6 and 10 applications, respectively. The figures also plot the bandwidth (red mark and right y-axis) of the applications when running in the workload with the two studied prefetch configurations. The top-right data in the figure reports the average (geo. mean) IPC of the applications that form the workload and their aggregated bandwidth. As observed, with 6 applications the default prefetch setting improves the performance in almost all the applications, so improving the overall IPC by 23%. In this case, it can be appreciated a relatively low bandwidth consumption, which overall amounts to 18.6 and 87.4 accesses/μs, with the prefetcher disabled and with the default prefetch setting, respectively.

In contrast, when running 10 applications, the default prefetch setting achieves on average worse performance (by 15% drop) than no prefetching. Note that most of the performance benefits that prefetching could provide as observed in the individual execution of the applications disappear due to bandwidth contention when applications run in the

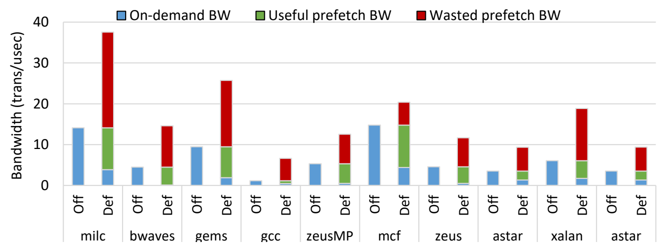
Departamento de Informática de Sistemas y Computadores, Universitat Politècnica de València, e-mail: `carnase1@inf.upv.es {jfeliu, spetit, megomez, jsahuqui}@disca.upv.es.`



(a) 6-application mix: low contention



(b) 10-application mix: high contention



(c) Bandwidth break-down of the 10-application mix

Fig. 1: Example of mixes exhibiting (a) low and (b) high main memory bandwidth contention. The last column reports the average (geo. mean) IPC for the applications of the workload and their aggregated main memory bandwidth. (c) breaks down the bandwidth of the 10-application mix.

mix. In this case the default prefetch setting consumes on average more than twice the bandwidth than no prefetching, aggregating 182.8 accesses/μs versus 67.2 accesses/μs with no prefetching. This high amount of aggregated bandwidth consumption of the default prefetch setting increases the inter-application interference making the memory bus a major performance bottleneck.

Figure 1c breaks down the bandwidth consumed by the applications of the 10-application workload into on-demand bandwidth, useful prefetch bandwidth (i.e. saves memory access latency), and wasted prefetch bandwidth (i.e. due to data not accessed by the applications). This breakdown shows that, despite prefetches have a positive effect on the performance of most applications (on-demand bandwidth is reduced), for many applications more than half of the bandwidth consumed by prefetches is wasted. The wasted bandwidth negatively impacts on the performance of the applications and their co-runners. As the figure shows, in general, the applications with highest bandwidth consumption (e.g., *milc*, *gemsFDTD*) waste more bandwidth and suffer higher performance degradation due to contention. Moreover, contention impacts on the performance of the co-running applications that access main memory, whose IPC when running in the workload is noticeably lower than when running alone with the same prefetch setting.

In this work we propose BAPC (Bandwidth-Aware Prefetch Configuration), an adaptive and scalable bandwidth-aware prefetch configuration approach for the IBM POWER8. BAPC focuses on multi-program workloads composed of single-threaded applications and seeks to dynamically select the best prefetch setting for each application, considering the tradeoff between performance and bandwidth consumption each prefetch configuration presents for each particular application.

The memory bandwidth consumed by an application mainly depends on the performance of the prefetch algorithm for a given application (e.g. coverage, accuracy and timeliness), and on the inter-application interference. LLC interference refers to memory requests (regular and prefetches) competing among them for cache space, thus inaccurate prefetches –specially in aggressive prefetch configurations– can pollute the cache causing the eviction of live cache blocks being used by other applications [15]. To provide a sound understanding of how different levels of interference may affect individual per application performance, this work characterizes, varying the interference level, the impact of distinct prefetch settings on both per-application performance and consumed bandwidth. We found that the *best prefetch setting* for a given application does not depend only on the application itself, but also on the inter-application interference. This claim, illustrated in this work through a rigorous characterization study, differs from the conclusions drawn in [14].

This paper makes three main contributions:

- We characterize applications varying the bandwidth requirements of the competing applications, and we show that the prefetch setting that mostly impacts on performance mainly depends on the bandwidth availability.
- We propose a scalable adaptive prefetch configuration approach that is able to improve the performance of multi-program workloads consisting on a high number of applications.
- The proposed scheme improves performance regardless of the number of running applications and amount of bandwidth the applications request.

The remainder of this paper is organized as follows. Section 2 describes the related work. Section 3 introduces the hardware platform. Section 4 discusses the characterization study. Section 5 presents the BAPC proposal, and Section 6 evaluates its results. Finally, Section 7 presents some concluding remarks.

2 Related Work

Recent research work has focused on prefetching on commercial machines. In [16], authors propose a methodology based on machine learning to select which of the 4 prefetchers available in an Intel Core 2 Quad CPU must be turned on or off. Thus, they select one configuration out of the 16 possible settings. One important difference with respect to our work is that these processors do not allow graduating the prefetchers, they only allow switching them on and off.

Other works focus on recent IBM POWER processors. Some works [12], [17] have focused on dynamically selecting at run-time the best prefetch setting to enhance the performance of parallel workloads when running a single application on an IBM POWER8. PATer [17] proposes a methodology that dynamically adjusts the prefetch setting in the IBM POWER8. This approach uses performance counters as inputs for machine learning strategies to improve the accuracy of the prefetching models. A background daemon is employed to make use of the prediction models to decide the best prefetch setting for each application running on the processor. Unlike our work, this proposal focuses exclusively on parallel workloads. Finally, LibPRISM [12], also focusing on parallel workloads, manages the SMT level and the prefetch settings in the IBM POWER8.

Other works [13] focus on *adaptive* prefetching to select the best prefetch setting for single-threaded multi-program workloads, similarly to our work. In [13], authors present an adaptive prefetcher capable of boosting the performance by leveraging the prefetch configuration with respect to the default prefetch configuration of the IBM POWER7. However, it is mainly driven by performance so that the prefetch settings are mainly chosen depending on the performance each prefetch setting is able to achieve for the target application. Although this strategy allows the prefetcher to achieve reasonable performance, it fails in that it does not scale with the number of cores, in particular it is evaluated just with two running applications. Mainly due to this fact, in a posterior work [14], these authors face a more scalable approach to configure the prefetch engine in the IBM POWER7 processor with the aim of allocating bandwidth dynamically to the co-running applications. This approach simply activates or deactivates the individual per-core prefetchers of the different threads. Authors claim that they explored other prefetching configurations with an intermediate aggressiveness but did not observe any benefit. Nevertheless, we show that significant performance benefits can be observed, at least in the IMB POWER8. Based on this observation, unlike [14], our approach selects the best prefetch setting instead of only switching the prefetch engine on or off.

Previous or contemporary with research on commercial machines, many research work has been carried out on simulation frameworks focusing on regulating the prefetcher aggressiveness. A representative subset of these works is [18], [19], [8], [20], [21], [22], [23], and [24]. In a similar way, *sandbox prefetching* [6], propose to implement a set of prefetchers and, at run-time, compare the prefetch patterns generated by each prefetcher with the memory accesses of the application to select the prefetcher that better fits the application memory access pattern. In essence, the problem might look similar to the one we address but a comparison cannot be done mainly due to implementation issues. The sandbox prefetching is implemented in a simulation environment. The meachanism implements a set of independent prefetchers and, for each interval, it gathers the memory access patterns and estimates the performance of each prefetcher included in the sandbox in order to choose the best one. This simulation approach considers extra hardware not deployed on current commercial processors. The prefetcher of the IBM POWER8, however, consists of 9 independent *but complementary* fields which amount 25 bits. This gives a huge amount of configurations which makes the sandbox approach impractical. Nevertheless, the key problem from a comparison perspective is that it is not possible to simulate or measure how different prefetch configurations behave in a real processor in the same interval since only one can be active at the same time.

3 Experimental Platform and Prefetcher

The results presented in this work have been carried out on an IBM Power System S812L with a 10-core IBM POWER8 processor working at 3.69GHz. Each core implements a 64KB L1 data cache and a 512KB L2 cache. The processor has a 80MB Last Level Cache (LLC), shared among all the cores, and is equipped with a 32GB off-chip DRAM module. The OS installed in the system is Ubuntu Linux 14.04 with kernel version 4.0.2.

The processor allows the user to configure the prefetcher through the *Data Streams Control Register* (DSCR) [25]. There is one register to define the prefetch setting of each running thread. Each register uses 25 bits to configure the prefetcher along 9 fields.

Because exploring all possible prefetching configurations (2^{25}) could become a tedious and though task, in this work we focus on the parameters with higher impact on the system performance. To this end, we studied each DSCR field in isolation and found that the *load stream disable* (LSD), the *default prefetch depth* (DPDF), and the *prefetch urgency* (URG) are the features that most affect the performance and bandwidth consumption of the studied applications. Other researchers reached similar conclusions working on parallel workloads [12], [17].

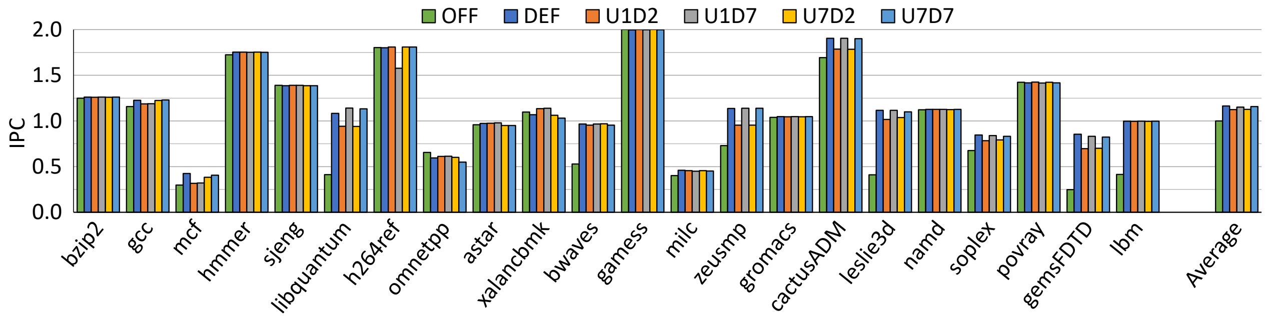
The LSD feature allows detecting bursts of loads to prefetch them, and can only be enabled or disabled. We always keep it enabled, since disabling LSD is equivalent to disabling the prefetcher. DPFD and URG are configured with a 3-bit DSCR field each. The DPFD field selects the prefetch depth, which ranges from 2 (shallowest) to 7 (deepest). Setting DPFD to 1 indicates zero depth (i.e. prefetch disabled, regardless of any other DSCR field), while setting it to 0 selects the default depth (4). The URG field indicates how quickly the prefetch depth can be reached. Prefetch urgency can vary from 1 (not urgent) to 7 (most urgent). Analogously to the DPFD field, setting URG to 0 selects the default urgency (4). By default, all the DCSR bits are 0, which enables the LSD characteristic and sets the prefetch urgency and depth to their default values, that is, URG=4 and DPFD=4.

4 Characterization Analysis

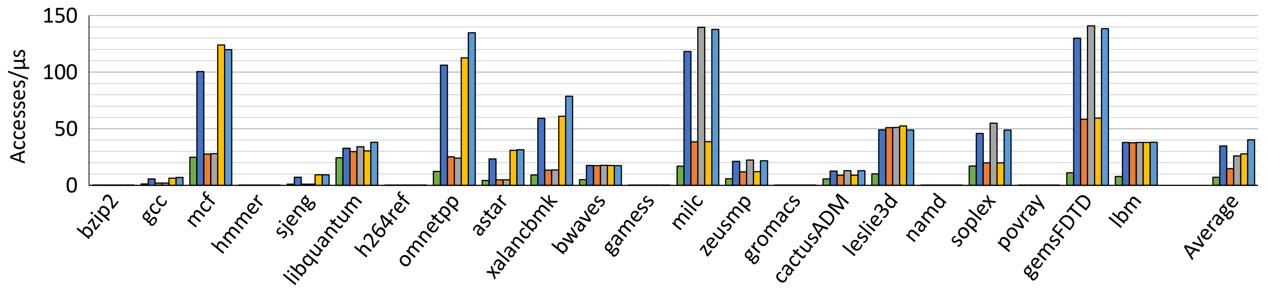
This section characterizes the applications when running in isolation in terms of performance and bandwidth utilization with different prefetch settings. We study the extreme (i.e. the lowest and the highest) values for both urgency and depth. That is, U1D2, U1D7, U7D2 and U7D7, where the UxDy settings means URG $= x$ and DPFD $= y$. In addition, we also evaluate the default prefetch setting (DEF) and a setting where prefetech is disabled (OFF).

4.1 Performance and Memory Bandwidth Utilization with no Inter-Application Bandwidth Contention

Figure 2a presents the IPC achieved by each application when running alone in the system across the studied prefetch settings. Since, in these experiments, each application runs in isolation, there is no inter-application interference, hence the contention for memory bandwidth is the lowest that applications can experience.



(a) Performance



(b) Bandwidth

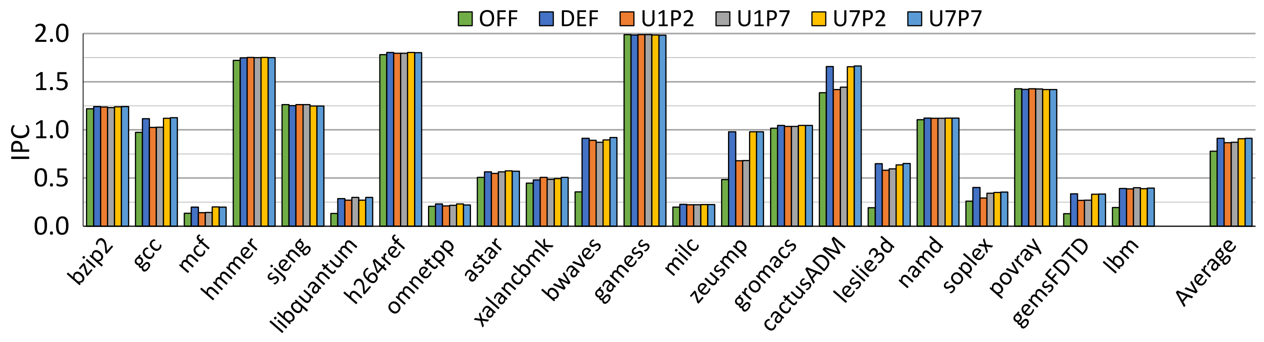
Fig. 2: Impact of the prefetch setting on performance and bandwidth of applications in isolated execution.

From these results, three main observations can be made. First, although the performance of some applications, such as `gamess` or `povray`, is not affected by prefetching, the performance of many others, such as `libquantum`, `zeusmp`, `cactusADM`, `leslie3d` and `gemsFDTD`, highly improves when prefetching is enabled. The former are typically referred as *prefetch friendly* application and the latter as *prefetch unfriendly*. These results show that it is crucial to keep prefetch enabled for *prefetch friendly* applications. Second, in general, the *prefetch depth* parameter has a higher impact on performance than urgency (e.g. in `libquantum` and `zeusmp`) when the application runs alone in the system. A deeper prefetch depth prefetches more cache lines from the main memory and, when running in isolation, usually improves performance even when the prefetch accuracy reduces since there is enough LLC space and main memory bandwidth to deal with the inaccurate prefetches. Nevertheless, there are applications, such as `gcc` and `mcf`, that are more affected by the urgency parameter. These applications benefit more from speeding up prefetches so that data can move quicker to the upper levels of the cache hierarchy than from issuing more prefetches. Third, the performance achieved by the best performing prefetch configuration hardly outperforms the default configuration, with just a few exceptions in some applications like `libquantum` and `xalancbmk` where the default prefetch setting performs slightly worse.

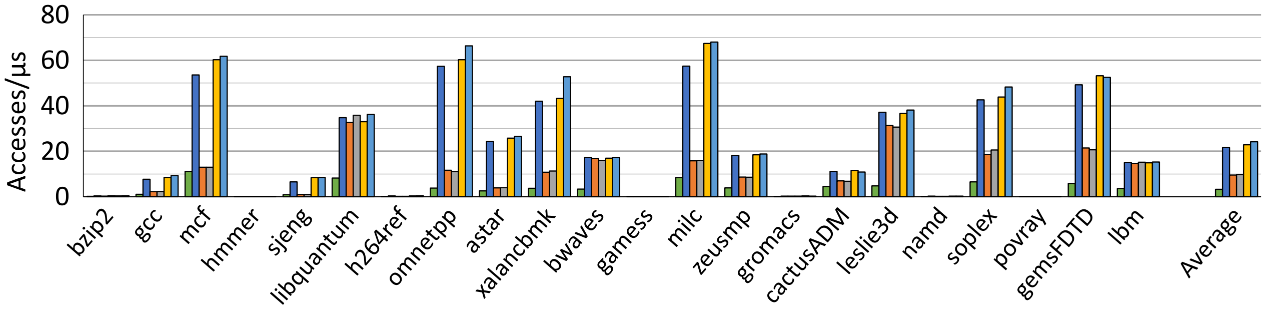
As shown in the example of Section 1, the main memory bandwidth is a critical resource that must be efficiently handled for performance in current multicores. To address this issue, this section nalyzes the relationship between the prefetch settings and the consumed bandwidth.Figure 2b shows the bandwidth each application consumes when running in isolation, quantified in main memory accesses per μs, varying the prefetch setting. Three main observations can be made. Firstly, bandwidth strongly varies depending on the prefetch setting, even for those applications whose performance is not significantly affected (e.g. `milc`). As observed, depth is the parameter that impacts the most on the consumed bandwidth. Secondly, some prefetch settings achieve IPC improvements at the cost of increasing the bandwidth consumption (e.g. DEF, U1D7, and U7D7 in `gemsFDTD`), which in many cases does not justify the performance gains. Thirdly, there are situations where a higher bandwidth consumption translates into performance drops. For instance, in `xalancbmk`, the settings with highest bandwidth consumption (U7D2 and U7D7) offer lower performance than the remaining settings. This is because in these settings prefetches have a high urgency but also very low accuracy, so bringing from memory a high number of non-useful prefetches. It can be appreciated that the default prefetch setting is clearly affected by the discussed shortcomings. For instance, in `milc`, the DEF's bandwidth consumption triples those of U1D2 and U7D2 while achieving similar performance; and in `xalancbmk`, DEF achieves lower performance than U1D2 and U1D7 in spite of its bandwidth consumption is more than 4 times higher.

This section has characterized the behavior of the applications of the SPEC CPU2006 benchmark suite when running in isolation. The study has also been carried out for the SPEC CPU20017 benchmarks, observing neither additional behaviors nor conclusions from the purposes of this work. Thus SPEC CPU2017 characterization results are not shown neither in isolation nor in other studied bandwidth contention scenarios, which are discussed below. Nevertheless, for the sake of completeness, the experimental results (see Section 6) include applications randomly chosen from both SPEC CPU2006 and CPU2017 suites.

To sum up, this study illustrates that, for some applications and prefetch configurations, an important in-



(a) Performance



(b) Bandwidth

Fig. 3: Impact of the prefetch setting on performance and bandwidth of applications in the highest bandwidth contention.

crease in the bandwidth consumption due to more triggered prefetches only improves performance marginally. In some cases, such as *xalancbmk*, the most aggressive prefetch configurations even degrade performance. Consequently, since the main memory bandwidth is an scarce resource and can quickly become a performance bottleneck, it is important to configure the prefetch aggressiveness based on the application's benefit of prefetch to limit bandwidth consumption.

4.2 Performance and Memory Bandwidth Utilization with Highest Bandwidth Contention

This section evaluates the impact of prefetching on the *highest* main memory bandwidth contention scenario. To this end, each application is executed with 3 instances of the microbenchmark designed in [26] (see Section 4.3.1) that, when running all three together, consume the entire memory bandwidth available in the system. This scenario presents the highest bandwidth contention studied in this paper. It would be possible to create worse scenarios for some applications, varying the writing or reading rates to main memory as well as the stride of the target page. Nevertheless, the analysis of such situations is out of the scope of this paper.

Figure 3 presents the IPC and memory bandwidth consumption of the applications for the analyzed prefetch settings. As in the previous study, disabling the prefetching can still have a significant adverse impact on the performance of many applications such as `libquantum`, `zeusmp`, `cactusADM` or `leslie3d`. The performance gains in these applications are also related to a higher bandwidth utilization. However, unlike it was previously observed in absence of inter-application interference (Section 4.1), under high bandwidth contention, the prefetch urgency parameter affects the performance and bandwidth consumption more than the prefetch depth. The reason is that when multiple applications co-run and share the main memory bandwidth, increasing the prefetch depth to trigger more prefetches might not always improve performance since it increases bandwidth contention and, in addition, these prefetches compete for LLC storage capacity. In this scenario, the prefetch urgency becomes more important since the higher bandwidth contention translates into a longer memory latency of the prefetches, thus triggering them earlier improves their chances of being timely prefetched.

Taking into account performance and bandwidth consumption, prefetch friendly applications can be classified in two main groups:

- *Prefetch-setting sensitive* applications such as `gcc`, `mcf`, `zeusmp`, or `cactusADM`. The performance of these applications can significantly vary with the prefetch setting. For instance, `zeusmp` improves its IPC from 0.7 to 1 when the prefetch urgency is increased.
- *Prefetch-setting insensitive* applications such as `bwaves` or `leslie3d`. The performance of these applications slightly differs by varying the prefetch setting. However, the associated bandwidth consumption may greatly vary, as observed in `leslie3d`.

In summary, the discussed results show that a prefetch configuration policy should analyze, for each application, both performance and bandwidth metrics with the different prefetch settings. This analysis should guide the prefetching strategy. For instance, the policy should i) disable the prefetch engine for prefetch unfriendly applications, ii) choose the configuration with the lowest bandwidth consumption for prefetch-setting insensitive applications, and iii) find a trade-off between performance gains and bandwidth consumption for prefetch-setting sensitive applications.

4.3 IPC and Bandwidth Consumption Sensitivity to Bandwidth Contention

So far we have studied the impact of prefetching on individual application performance in two extreme bandwidth scenarios: all the memory bandwidth is available for the studied application, and the application competes for bandwidth with the microbenchmark (i.e. three instances[1]), described below, at its maximum main memory access rate. This section studies the impact of intermediate levels of bandwidth contention on performance and bandwidth consumption of each application varying the prefetch setting. To this end, the microbenchmark is tuned to precisely consume a specified amount of memory bandwidth when it is executed in isolation. This way, when executed together with the studied application, the microbenchmark is used to induce the desired level of inter-application bandwidth contention.

4.3.1 *Memory Bandwidth Microbenchmark*

Algorithm 1 Memory bandwidth microbenchmark

```
1: int A[ARRAY_SIZE]
2: while true do
3:     for (i = 0; i < $ARRAY\_SIZE$; i=i+$stride$) do
4:         A[i]++
5:     end for
6:     for (i = 0; i < $\#nops$; i++) do
7:         asm("nop")
8:     end for
9: end while
```

Algorithm 1 presents the pseucodode of the microbenchmark. It consists of a main loop (lines 2–9), where a burst of memory requests (lines 3–5) is issued. This burst is followed by a customizable number of NOP operations, which allow us to set the bandwidth consumption of the microbenchmark and study different levels of bandwidth contention. Memory requests access a memory array defined in line 1. The array size should, at least, double the size of the LLC of the processor to ensure that data is evicted from the cache before it is re-accessed, and the variable *stride* should also be configured, depending on the memory architecture, to make sure that two consecutive accesses to the array access different cache lines. In this way, we avoid LLC hits, maximizing the memory bandwidth the microbenchmark consumes.

We found empirically that the main memory bandwidth saturates when at least three independent instances of the microbenchmark are executed together each one on a different core. Further details on this empirical experiment can be found in Section 1.1 of the Appendix.

This scenario, as mentioned above, is referred to as the highest bandwidth contention scenario. With this setup we have empirically estimated that the maximum bandwidth an application can consume ranges from 180 to 190 accesses per μs. For analysis purposes we consider 190 accesses/μs as the maximum bandwidth utilization.

To model mid range scenarios providing different amounts of available bandwidth, we have estimated the number of #nops the microbenchmark instances must execute to consume from 90% of its maximum bandwidth consumption (i.e. 190 accesses/μs) down to 10% in 10% steps. From now on we use BCI_XX (Bandwidth Consumed in Isolation) to refer to the configuration in which the microbenchmark instances consume XX% of the overall bandwidth when running in isolation. Notice that this percentage will be effectively reduced when the available bandwidth saturates. In this case, the effective bandwidth consumed by the microbenchmarks will be the difference between the bandwidth consumed by the application and the maximum bandwidth consumption.

4.3.2 *Performance and Bandwidth Sensitivity*

This section analyzes the behavior of the studied applications in eleven memory contention scenarios. Six prefetch settings have been studied in this work but, in addition to OFF and default, results are only shown for U1D2 and U7D2 prefetch settings because they present similar values to U1D7 and U7D7, respectively; that is, those having the same urgency but the highest depth are presented. The configurations with lowest depth have been chosen because in some cases the consumed bandwidth is lower than when the depth parameter is equal to 7.

Figure 4 depicts the IPC (y-axis) and bandwidth consumption (x-axis) of the studied applications in four plots, one for each prefetch configuration. Each application is represented with a line with 11 points (or marks) which indicate the level of main memory bandwidth contention introduced by the microbenchmark. The rightmost point represents the values achieved by the application running alone under no contention (i.e. BCI_00), the leftmost represents the performance and bandwidth achieved by the application under maximum contention (i.e. running with BCI_100), and the 9 intermediate points (from right to left) represent the values achieved by the application in mid range bandwidth contention scenarios; that is, from right to left, running with BCI_90 to BCI_10. For instance, the second point (from right to left) of `astar` in Figure 4a indicates that when this application is under the BCI_10 scenario it achieves an IPC by 0.24 and performs by 22.5 accesses/μs. Section 1.2 of the Appendix complements these figures and shows the bandwidth consumption of each application and the co-running microbenchmarks on each point for a subset of benchmarks with the default prefetch configuration.

In each plot, regardless of the prefetch setting, three main application behaviors can be observed. Below these behaviors are studied for the default prefetch setting depicted in Figure 4c. Firstly, applications located at the leftmost side (e.g. close to the Y axis) in a single point are insensitive to the bandwidth contention. The reason is that their bandwidth consumption is really low and thus their performance is not affected by main memory bandwidth contention. Secondly, we can distinguish applications extremely sensitive to the bandwidth contention. In this plot, these applications present from 5 to 35 accesses/μs. Their performance experience a sharp rise with a small increase in the bandwidth consumption. This trend slightly decreases as we move on to the right. And thirdly, as we move on by about 50 accesses/μs, increasing the bandwidth consumption still allows improving the performance but

1. The microbenchmark is always launched in three instances

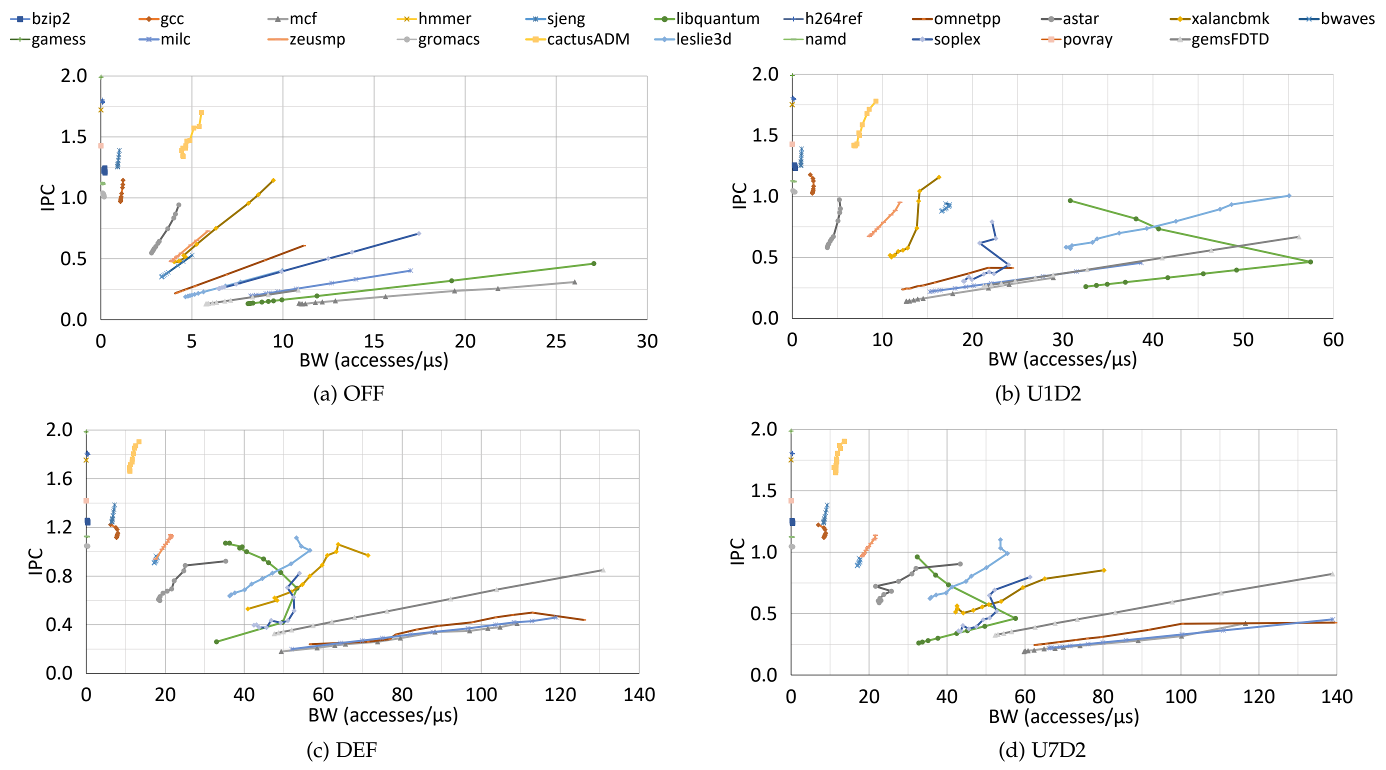


Fig. 4: Performance and bandwidth consumed by the applications for OFF, U1D2, DEF and U7D2 prefetch settings varying the bandwidth consumption of the microbenchmark.

to a minor extent. That is, bandwidth consumption must significantly increase to turn into performance gains. For instance, increasing bandwidth consumption from 60 to 130 accesses/μs in `omnetpp` just improves IPC in 0.2. Notice, however, that such a large amount of bandwidth represents almost one third of the total system bandwidth.

Another interesting observation is that the curve of some applications does not always experience a positive slope in terms of bandwidth as bandwidth contention increases, but it first grows and then reduces as the bandwidth consumed by the microbenchmark increases. For instance, *libquantum* shows an IPC by 1.1 in isolation with the default prefetch setting. Then, as the microbenchmarks start consuming bandwidth, *libquantum*'s bandwidth consumption, contrary to expected, increases. The increase in the bandwidth utilization is a result of the cache pollution [27] introduced by the microbenchmarks. This trend is kept until *libquantum* runs in the BCI_30 scenario. From that point on, the bandwidth contention induced by the microbenchmarks causes *libquantum*'s bandwidth utilization to decrease. In summary, this behaviour is mainly caused by the interference between the applications and the microbenchmarks both in the main memory bandwidth and LLC space. Both of them grow with the prefetch aggressiveness. In fact, the plot corresponding to the OFF setting is the only one where all the application's curves present a positive slope.

Comparing the four plots among them, it can be observed that, as expected, the bandwidth consumption of a given application increases as the applied prefetch settings are more aggressive (bottom plots). However, an interesting observation is the magnitude of these values, which widely differ among them depending on the prefetch setting. For instance, the maximum bandwidth an application consumes is by 30, 60, 130 and 140 accesses/μs with the prefetcher disabled (OFF), U1D2, DEF, and U7D2 settings, respectively. This huge difference allows some interesting cross comparisons, for instance, the bandwidth consumption of some applications (e.g. `omnetpp`) under maximum contention with the default prefetch setting is much higher than running the application alone with the prefetcher disabled.

5 BAPC Prefetching Approach

This section presents the prefetch approach proposed in this paper, Bandwidth-Aware Prefetch Configuration (BAPC), which pursues to dynamically adapt the prefetch configuration for each application taking into account the tradeoff between the performance and bandwidth consumption of each prefetch configuration. The results of the characterization study claim that a per-application custom prefetch configuration is needed to regulate the use of prefetching among the co-running applications. This per-application configuration should take into account both performance and bandwidth consumption, since in some cases the performance of a given application can be improved by a given prefetch setting but at expense of a high bandwidth increase. Otherwise, bandwidth consumption can significantly grow, causing important bandwidth contention and leading the system to significant performance losses.

With this aim, in order to quantify the trade-off between performance and bandwidth to select the proper prefetch setting, we define the *P2B ratio* ratio calculated with Equation 1. This metric considers the improvement rate in IPC of a given prefetch setting over the prefetch disabled, and divides this value by the bandwidth wasting rate introduced by that prefetch setting over no prefetch.

$$P2B\ ratio_{config} = \frac{IPC_{config}/IPC_{OFF}}{BW_{config}/BW_{OFF}}. \quad (1)$$

For instance, a speedup by 1.1 (10% performance increase) and bandwidth consumption rate by 2.2 (120% bandwidth increase) gives a P2B ratio equal to 0.5. In general, the IPC increase is usually much smaller than the bandwidth increase, as experimental results will show. Once the P2B ratio is quantified, it is compared to a threshold to discern if a given prefetch configuration is eligible for that application or not. We experimentally found that a good trade-off is achieved when $P2B\ ratio \geq 0.25$, which means that a 10% performance increase is *worth enough* if it comes with a bandwidth consumption increase lower than 340%. Notice that applications consuming a significant amount of bandwidth present and almost flat curve in Figure 4.

5.1 Static BAPC

With the aim of ascertain the potential of our approach, we have first devised a static version of the mechanism to be applied in high bandwidth contention scenarios. The static approach selects a *customized* prefetch setting for each application of the workload that is used during the whole execution of the application. For each application, the selected setting is the one that better fits the characteristics of the application trading off performance and bandwidth. For this purpose the approach works as follows.

First, for each considered prefetch configuration, the IPC and bandwidth consumption of each application are obtained when running in the highest contention scenario (BCI_100). Second, a list of eligible prefetch settings is obtained. To be included in the list, a configuration must fulfill two conditions: i) its corresponding P2B ratio value must be higher or equal than the P2B threshold (i.e. prefech configurations with P2B ratio below the threshold are discarded), and ii) its IPC must be better than the one obtained with no prefetching. If the list is empty, then there is no prefetch configuration that matches these conditions and, consequently, the prefetching is turned off for the application. On the contrary, if several configurations are in the list, they are arranged in descending IPC order, and the prefetch configuration at the head of the ordered list is chosen.

Table 1 presents the selected configuration obtained with this method for each SPEC CPU2006 and SPEC CPU2017 application. For example, the configuration OFF is chosen for `sjeng` because, as observed in Figure 3, neither the performance improves due to prefetching (second condition) nor the P2B ratio is higher or equal than 0.25 (first condition). Another example is `zeusmp`, whose selected prefetch configuration is U7D2 because its performance gain is reasonable for the bandwidth consumption increase that the application experiments with this configuration.

By applying a custom prefetch configuration to each application, we ensure a profitable utilization of the available system bandwidth. In contrast, if the default configuration is used in every application, the system will present a much higher contention, which leads to a negative impact in performance. Experimental results obtained when statically assigning the selected prefetch settings to applications are presented in Section 6.

TABLE 1: Selected configurations for each application in SPEC CPU2006 and SPEC CPU2017.

Benchmark 2006	**Conf.**	**Benchmark** 2017	**Conf.**
bzip2	U7D2	perlbench_r	U1D2
gcc	U1D2	gcc_r	U7D2
mcf	U7D2	mcf_r	U1D2
hmmer	U1D2	parest_r	U7D2
sjeng	OFF	deepsjeng_r	U1D2
libquantum	DEF	lbm_r	U1D2
h264ref	OFF	x264_r	U7D2
omnetpp	U1D2	omnetpp_r	U1D2
astar	U1D2	cam4_r	U7D2
xalancbmk	OFF	imagick_r	DEF
bwaves	U7D2	bwaves_r	U7D2
gamess	OFF	wrf_r	U7D2
zeusmp	U7D2	leela_r	U1D2
gromacs	DEF	nab_r	U7D2
cactusADM	U7D2	cactuBSSN_r	U7D2
leslie3d	U7D2	roms_r	U7D2
namd	OFF	namd_r	U7D2
soplex	U1D2	xz_r	U1D2
provray	OFF	povray_r	OFF
gemsFDTD	DEF	exchange2_r	U7D2

5.2 Dynamic BAPC

Static BAPC relies on profiled execution information of applications obtained offline. This information is used to select the prefetch setting for each application that is applied during the whole execution. In contrast, dynamic BAPC acts without previous knowledge of applications' behavior, and dynamically selects the target prefetch setting at run-time, so different settings can be chosen for a given application. This way allows BAPC to adapt to the different execution phases each application experiences.

Algorithm 2 presents the pseudo-code of dynamic BAPC. This algorithm can be divided into three main phases: sampling, configuration and execution. These phases are executed sequentially in a loop until the workload execution completes

In the sampling phase, metrics related to performance and bandwidth consumption are estimated. To do so, the prefetcher is globally disabled to gather IPC_{OFF} and BW_{OFF} for all the applications. Then, the prefetch settings are applied, one by one, to each application while the remaining applications run with the prefetcher turned off. In this way, IPC_{config}, BW_{config}, and $P2B\ ratio_{config}$ can be independently estimated for each application and prefetch configuration.

To reduce the overhead of sampling, the sampling quantum duration is set to 50ms, and the checked prefetch settings are limited to DEF, U1D2 and U7D2, because prefetch depth in general, as discussed in Section 4 does not affect the performance and bandwidth consumption under important bandwidth contention.

The metrics estimated in the sampling phase are employed in the configuration phase to obtain the best prefetch setting for each application. In this phase, a list of candidate settings is created for each application. A setting is eligible if the measured IPC of the application with that setting is

Algorithm 2 Dynamic BAPC

```
1: while there are running applications do
2:   ——— SAMPLING PHASE ———
3:   for all $app$ do Prefetch($app$) $= OFF$
4:   Execute 1 quantum and update $IPC_{OFF,app}$ and $BW_{OFF,app}$ for all apps
5:   for all $app$ do
6:     for all $config$ in {DEF, U1D2, U7D2} do
7:       Prefetch($app$) $= config$
8:       Execute 1 quantum and update $IPC_{config,app}$, $BW_{config,app}$, and $P2B\ ratio_{config,app}$
9:       Prefetch($app$) $= OFF$
10:     end for
11:   end for
12:   ——— CONFIGURATION PHASE ———
13:   for all $app$ do
14:     $config\_list_{app}$ = {}
15:     for all $config$ in {DEF, U1D2, U7D2} do
16:       if $IPC_{config,app} \geq (IPC_{factor} \times IPC_{OFF,app})$ then
17:         Add $config$ to $config\_list_{app}$
18:       end if
19:     end for
20:     Sort $config\_list_{app}$ in descending IPC order
21:     for all $config$ in $config\_list_{app}$ do
22:       if $P2B\ ratio_{config,app} \geq P2B\ ratio_{threshold}$ then
23:         Prefetch($app$) $= config$
24:       end if
25:     end for
26:   end for
27:   ——— EXECUTION PHASE ———
28:   Set quantum duration (e.g. 400ms)
29:   for 1 to $N$ quantums do
30:     Execute 1 quantum
31:     Update $BW_{System}$
32:     if $BW_{System} \geq BW_{threshold}$ then
33:       $min\_app$ = $app$ with lowest $P2B\ ratio_{config}$ value
34:       Prefetch($min\_app$) $= OFF$
35:     end if
36:   end for
37: end while
```

significantly higher than the IPC with prefetcher disabled. This is regulated with the IPC_{factor} parameter, which avoids selecting wrong settings due to estimation errors in the sampling phase. This factor is not necessary in static BAPC since it relies in profiling information obtained offline.

Once the list of prefetch settings is made for a given application, the prefetch settings in the list are arranged in descending IPC order. Then, the list is looked up to find the configuration with the maximum IPC among those whose P2B ratio value is greater or equal than the threshold. In case of there is not any configuration that fulfills the condition, the prefetch engine is disabled.

The prefetch settings chosen in the configuration phase are applied to the execution phase, which keeps the selected settings for a given number of quanta while monitoring the total accumulated main memory bandwidth consumption. If the bandwidth consumption is over the $BW_{threshold}$, BAPC disables prefetching for the application whose active prefetch setting presents the lowest P2B ratio value. The rationale behind this design choice is that this setting is the least efficient in terms of the performance/bandwidth ratio.

Note that the time taken in the execution phase must be limited in order to capture dynamic phase changes in the application behaviors that affect both performance and bandwidth. In this regard, previous works like [28] have shown that, on average, execution phases last a few seconds without significant changes in the IPC of the application.

6 Performance Evaluation

6.1 Methodology

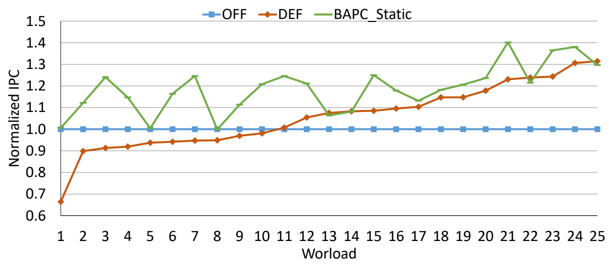
To study the performance and bandwidth utilization under BAPC, we have implemented both the static and dynamic versions in a user-level scheduler. In the dynamic version, this scheduler orchestrates the sampling, configuration and execution phases, collects performance counters to obtain the IPC and main memory bandwidth utilization of the applications, and updates the prefetch configuration for each application based on Algorithm 2. To analyze how BAPC performs compared to other prefetch configurations, we have implemented two static policies in the user-level scheduler: one keeps prefetching disabled, the other keeps it to the default prefetch setting. Note that the later is the default case in a Linux system. In addition, we have also implemented the *Intelligent Bandwidth Shifting* (IBS) mechanism proposed by Jiménez et al. [14].

To evaluate the proposal we have designed 3 sets of 25 workload mixes each composed of 6, 8, and 10 applications randomly selected from SPEC CPU2006 and SPEC CPU2017 benchmark suites. In order to give the same weight to all the applications in a mix, we have measured the number of instructions that each application executes in isolation during 120 seconds with the prefetcher turned off. This value is used as the *target* number of instructions of each application. During the execution of a workload mix, when an application completes its target number of instructions, its IPC is obtained and the application relaunched again to keep a constant load during the whole experiment, which finishes when all the applications of the mix complete their target number of instructions. We measure performance by means of the weighted speedup metric [29], which is calculated as the sum of the performance of each application of the workload normalized to its performance when running in isolation with prefetching disabled.

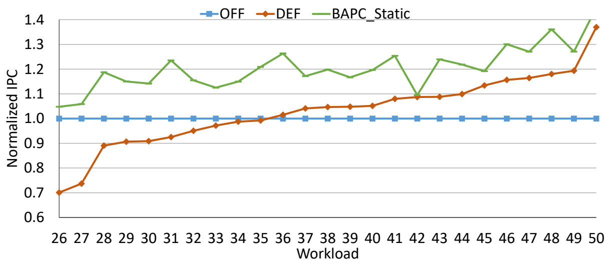
The thresholds and parameters used by the dynamic BAPC approach have been obtained through a wide set of experiments. The presented experimental results have been obtained with $P2B\ ratio_{threshold} = 0.3$, $BW_{threshold} = 185$, $IPC_{factor} = 1.1$, while the execution phase length in quanta (N) and the quantum length are set to 50 and 400ms, respectively. Sensitivity studies of all these parameters have been carried out. For illustrative purposes, Section 2 of the Appendix presents two sensitivity studies varying the P2B threshold and sampling quantum length.

6.2 Static BAPC

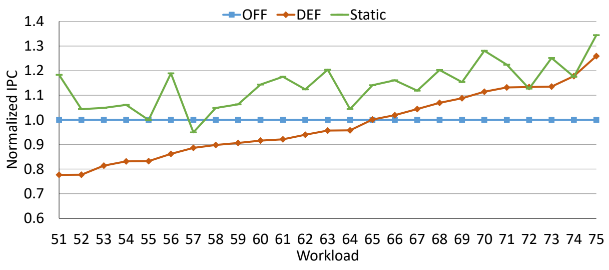
Figure 5 presents the performance achieved by the default (DEF) and proposed static (BAPC_static) prefetch configuration algorithms normalized to the performance of prefetching disabled (OFF) across 6-, 8-, and 10-application workloads. The 25 workloads of each scenario are presented in increasing speedup order of DEF over no prefetching. As observed, the default prefetch setting does not always achieve higher performance than no prefetching. Moreover, performance losses in some mixes can be as high as 30%. These cases appear due to prefetch inaccuracy and the high memory contention that prefetching causes. Because bandwidth contention increases with the number of applications, the number of workloads where DEF is outperformed by



(a) 6-application workloads

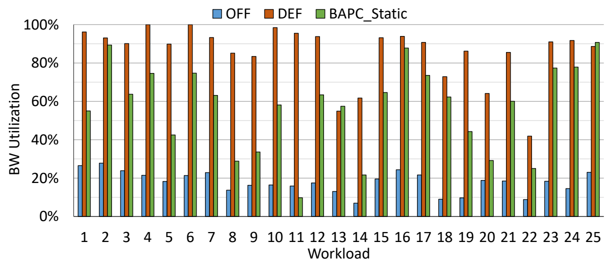


(b) 8-application workloads

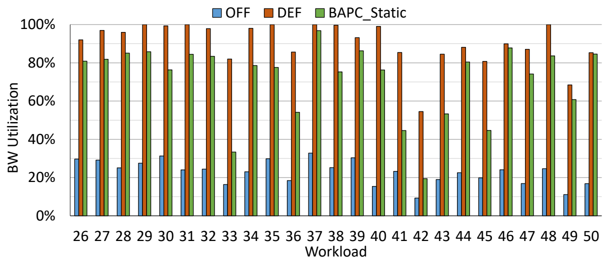


(c) 10-application workloads

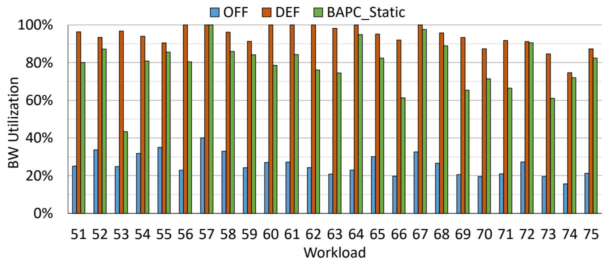
Fig. 5: Performance of DEF and BAPC_Static prefetch configurations normalized to prefetching disabled.



(a) 6-application workloads



(b) 8-application workloads



(c) 10-application workloads

Fig. 6: Bandwidth consumed by BAPC_Static, DEF, and OFF with respect to the maximum available in the system.

OFF tends to grow with the mix size. There are 10, 9, and 14 workloads in the 6-, 8-, and 10-application scenarios respectively, where the OFF setting outperforms DEF. However, despite these cases, DEF achieves better performance than OFF in most workloads, reaching speedups close to or above 40% regardless of the number of applications in the mix.

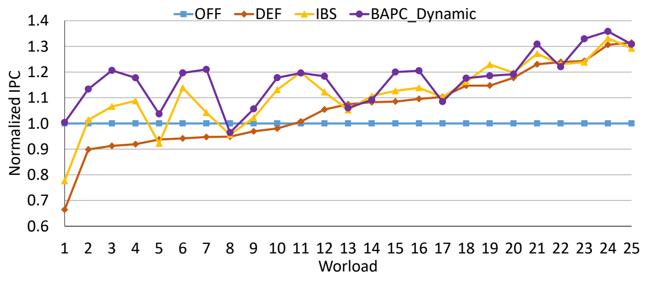
Figure 6 presents the bandwidth utilization (190 accesses/μs is assumed as the 100%) consumed by each workload mix in the studied approaches. The reported bandwidth consumption includes both on demand and prefetch accesses. As observed, the bandwidth utilization greatly varies across the studied approaches. While it approximately ranges from 10% to 40% when prefetching is disabled, it grows over 90% in many workloads with the default configuration. This means that DEF saturates the system bandwidth with prefetches, which translates into system performance degradation when prefetch accuracy and timeliness are low and do not compensate the induced bandwidth contention.

In contrast, the BAPC_static approach reaches an intermediate bandwidth consumption, usually ranging from 40% to 90% of the system bandwidth. This is because BAPC_static disables prefetching for prefetch unfriendly applications and selects the best prefetch setting, considering performance and bandwidth, for the remaining, prefetch friendly, applications. This way allows BAPC_static to reduce the bandwidth consumption of those applications that do not benefit from the additional prefetch requests, thus reducing the bandwidth contention and increasing the bandwidth availability for those applications that really need it. These key actions turn into performance gains.

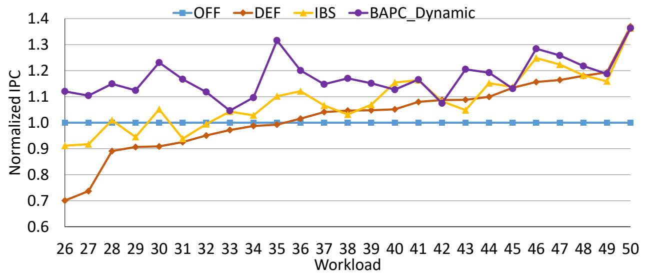
To better figure out how bandwidth contention impacts on performance, notice that BAPC_static greatly outperforms the default prefetch configuration when the bandwidth utilization exceeds 90% (e.g. workloads 7, 28, and 51).

6.3 Dynamic BAPC

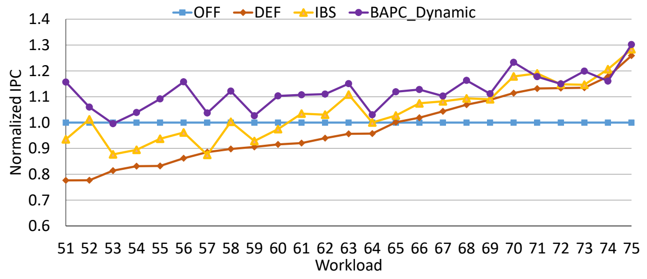
This section evaluates the dynamic version of the proposed mechanism, which, as mentioned above, selects the prefetch settings at run-time for each application depending on the execution phase and the system bandwidth utilization. We compare its performance and bandwidth consumption with no prefetching, the default prefetch configuration, and the IBS mechanism [14]. As our approach, IBS also seeks to save bandwidth and only enables prefetching for those applications that IBS estimates that can benefit from it. However, it only considers two prefetch configurations: disabled or enabled at its maximum aggressiveness. Consequently, it does not take advantage of less aggressive prefetch configurations which, as shown in Section 4.2, can achieve most of the performance benefits of the most aggressive configuration while significantly consuming less bandwidth thus reducing the bandwidth contention. This section evaluates the studied schemes under multiprogram workloads.



(a) 6 applications workloads

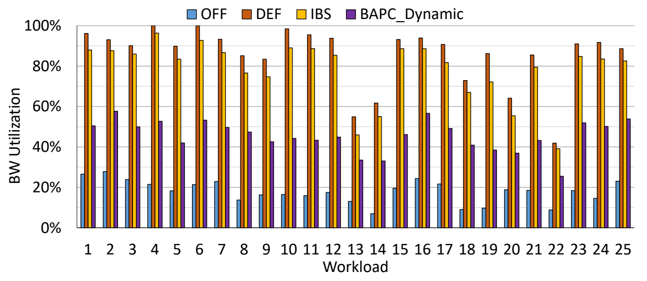


(b) 8 applications workloads

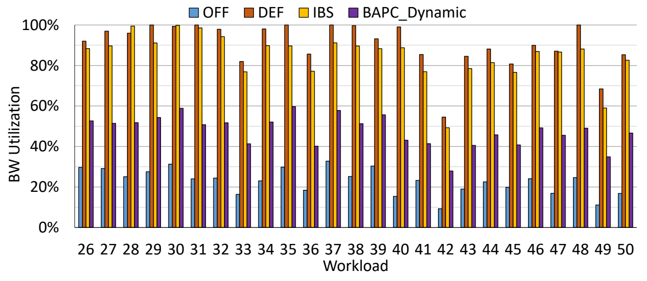


(c) 10 applications workloads

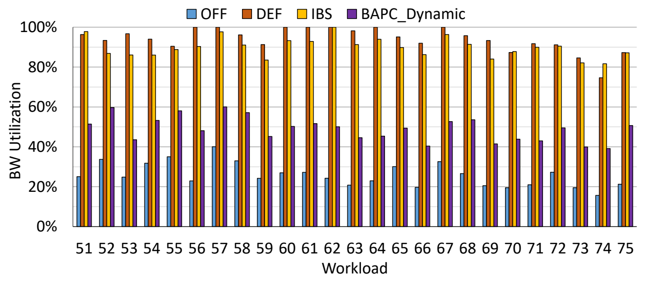
Fig. 7: Performance of DEF, IBS, and BAPC_Dynamic configurations normalized to prefetching disabled.



(a) 6-application workloads



(b) 8-application workloads



(c) 10-application workloads

Fig. 8: Bandwidth consumption of the OFF, DEF, and BAPC_Dynamic prefetch configurations with respect to the maximum bandwidth available in the system.

Sections 3 and 4 of the Appendix discuss the accuracy of the performance and bandwidth estimated in the sampling phase, and the per-application performance achieved when the workloads run under dynamic BAPC compared to the default prefetch configuration, respectively.

Figure 7 shows the normalized IPC of the dynamic approach (BAPC_dynamic), the default configuration, and IBS with respect to no prefetching. Comparing Figure 7 to Figure 5 we can observe that the dynamic approach achieves similar results to the static version. BAPC_dynamic achieves a noticeable IPC improvement over OFF, DEF and IBS, with the only exception of just a few workloads where they present barely the same behavior. The performance improvement is on average by 12%, 15%, and 16% over DEF for 6-, 8- and 10-application workload mixes, respectively. This means that BAPC_dynamic is able to detect which is the proper configuration in the distinct execution intervals, making an excellent use of the system bandwidth. IBS outperforms the default prefetch configuration by turning off the prefetching for the applications that do not benefit from it, which reduces bandwidth contention. However, BAPC_dynamic has a finer control of the prefetch aggressiveness and thus it achieves a better bandwidth utilization, which results in significantly higher performance for most of the workloads.

Figure 8 shows the bandwidth utilization results for the studied prefetch configurations and mechanisms. We can drawn the same conclusions as in the static BAPC approach. In this case, the dynamic approach achieves, on average, 39%, 42%, and 45% bandwidth savings compared to the default configuration for 6-, 8-, and 10-applications.

Finally, notice that the bandwidth utilization of the dynamic approach is significantly lower than that of the static approach. Thus, there is still room to further increase the performance by properly relaxing the thresholds of the algorithm. However, we choose relatively strict thresholds since they provide both high performance and low bandwidth, which makes the approach to stand out in scalability.

6.4 Case study

In order to better understand the reason why our proposal is able to improve performance and bandwidth, this section analyzes the behavior of each individual application of a given workload. For illustrative purposes we study mix 58. Figure 9a, Figure 9b, and Figure 9c depict the IPC, bandwidth utilization, and prefetch configuration for each application of the workload and the studied prefetch configurations and BAPC mechanisms.

First of all, looking at Figure 9a, we can observe how the configuration with prefetching disabled outperforms the default prefetch configuration in most benchmarks, which

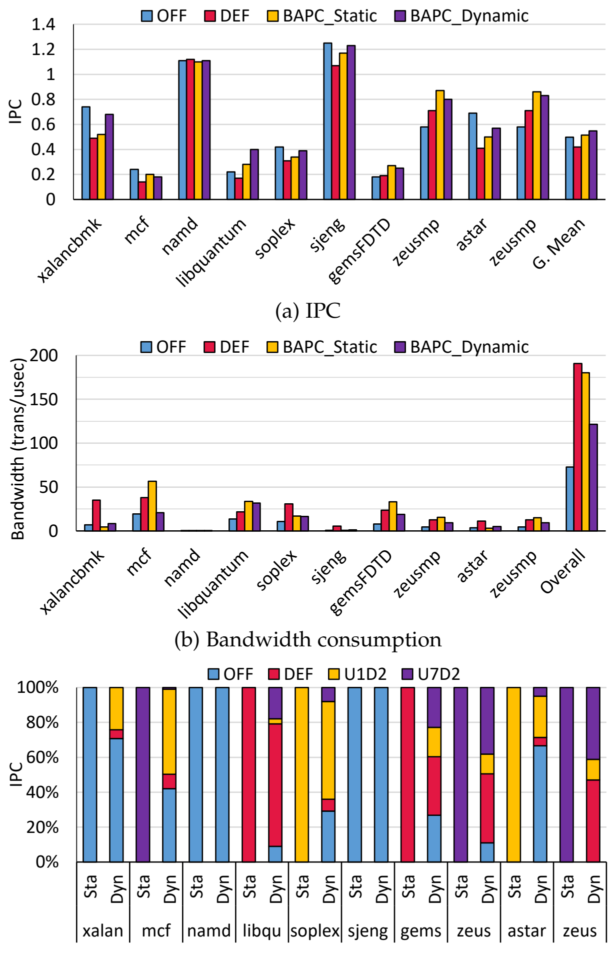


Fig. 9: IPC, bandwidth consumption and prefetch configuration of the applications in workload 58.

indicates that the default prefetch configuration suffers from bandwidth contention. Figure 9a confirms this fact since the applications reach an overall bandwidth utilization of 190 trans/μs for the default prefetch configuration, which eliminate the potential benefits that prefetching usually has.

The static BAPC mechanism selects the best prefetch configuration for each application. For some application this turns into higher prefetch aggressiveness and bandwidth utilization (e. g., *mcf*) while for others prefetching is disabled and bandwidth saved (e.g., *xalancbmk*). The static BAPC mechanism improves the performance of the configuration with prefetching disabled but it still suffers from significant bandwidth contention (overall bandwidth utilization of 180 trans/μs). Thus, the performance benefits it has compared to the prefetching disabled are moderate.

The dynamic BAPC mechanism is able to (i) adapt the prefetch configuration to the dynamic phase behavior of the applications and (ii) reduce the prefetch aggressiveness of some applications as soon as it detects that the bandwidth utilization is excessive and could significantly rise bandwidth contention. As Figure 9c shows, these features make the prefetch configuration of most applications to vary among different prefetch settings. This way turns into important bandwidth savings for several applications that reduce their prefetch aggressiveness for a long time in their execution (e.g., *mcf*, *GemsFDTD*) with minor impact on their performance. Other applications such as *xalancbmk* run with more aggressive prefetch configurations (compared to the static BAPC mechanism) for approximately 30% of the execution time, which significantly improves the performance with a small increase in the bandwidth utilization. In summary, adapting the prefetch configuration to the phase behavior of applications while limiting the bandwidth utilization to avoid bandwidth contention allows the dynamic BAPC mechanism to improve the performance of most applications compared to the static BAPC mechanism.

7 Conclusions

Prefetch engines in current high-performance processors include a wide range of prefetch settings, which makes difficult to dynamically configure them at run-time. This work makes two main contributions. Firstly, we characterize the behavior of a wide set of applications varying the bandwidth contention for the most significant prefetch configurations in the IBM POWER8. We found that the best prefetch setting for each individual application, considering both performance and bandwidth consumption, does not depends only on the application itself but also on the co-running applications competing for memory bandwidth. Secondly, we propose Bandwidth-Aware Prefetch Configuration (BAPC), an scalable and adaptive prefetch algorithm. The dynamic version of BAPC chooses the prefetch setting that better fits the behavior of each application in order to achieve a good trade-off between performance and bandwidth. Experimental results show that BAPC excels in high-contention bandwidth scenarios. Compared to default prefetching, BAPC outperforms by 12%, 15%, and 16% for workload mixes consisting of 6, 8, and 10 applications, while dramatically reducing the bandwidth consumption.

Although the focus of this paper has been on the IBM POWER8, the fundamentals of the characterization study and the devised approach could be generally adapted to other processors implementing programmable prefetchers.

Acknowledgments

This work was partially supported by Ministerio de Ciencia, Innovación y Universidades and the European ERDF under Grant RTI2018-098156-B-C51, Generalitat Valenciana under Grant AICO/2019/317, and Universitat Politècnica de València (PAID-06-18) under grant SP20180140.

References

[1] J.-L. Baer and T.-F. Chen, “An effective on-chip preloading scheme to reduce data access penalty,” in *ACM/IEEE Conference on Supercomputing*, 1991, pp. 176–186.
[2] J. W. Fu, J. H. Patel, and B. L. Janssens, “Stride directed prefetching in scalar processors,” *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2, pp. 102–110, 1992.
[3] D. Joseph and D. Grunwald, “Prefetching using markov predictors,” in *24th International Symposium on Computer architecture*, 1997, pp. 252–263.
[4] G. B. Kandiraju and A. Sivasubramaniam, “Going the distance for TLB prefetching: an application-driven study,” in *29th Annual International Symposium on Computer Architecture*, 2002, pp. 195–206.
[5] K. J. Nesbit and J. E. Smith, “Data cache prefetching using a global history buffer,” in *10th International Symposium on High Performance Computer Architecture*, 2004, pp. 96–96.

[6] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. Chuang, R. L. Scott, A. Jaleel, S. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 626–637.

[7] S. Byna, Y. Chen, and X.-H. Sun, "Taxonomy of data prefetching for multicore processors," in *Journal of Computer Science and Technology*, vol. 24, no. 3, 2009, pp. 405–417.

[8] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *42st Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 316–326.

[9] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared resource management for multi-core systems," in *38th Annual International Symposium on Computer Architecture*, 2011, p. 141–152.

[10] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-aware DRAM controllers," in *41st IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 200–209.

[11] M. Torrens, "Improving prefetching mechanisms for tiled cmp platforms," in *PhD. Thesis. Universitat Politècnica de Catalunya*, 2016.

[12] C. Ortega, M. Moreto, M. Casas, R. Bertran, A. Buyuktosunoglu, A. E. Eichenberger, and P. Bose, "libprism: An intelligent adaptation of prefetch and smt levels," in *International Conference on Supercomputing*, 2017, pp. 28:1–28:10.

[13] V. Jiménez, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, P. Bose, F. P. O'Connell, and B. G. Mealey, "Adaptive prefetching on POWER7: improving performance and power consumption," *ACM Transactions on Parallel Computing*, vol. 1, no. 1, pp. 1–25, 2014.

[14] V. Jiménez, A. Buyuktosunoglu, P. Bose, F. P. O'Connell, F. J. Cazorla, and M. Valero, "Increasing multicore system efficiency through intelligent bandwidth shifting," in *21st IEEE International Symposium on High Performance Computer Architecture*, 2015, pp. 39–50.

[15] A. Valero, J. Sahuquillo, S. Petit, P. López, and J. Duato, "Combining recency of information with selective random and a victim cache in last-level caches," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 3, pp. 1–20, 2012.

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

[17] M. Li, G. Chen, Q. Wang, Y. Lin, P. Hofstee, P. Stenström, and D. Zhou, "Pater: A hardware prefetching automatic tuner on IBM POWER8 processor," *Computer Architecture Letters*, vol. 15, no. 1, pp. 37–40, 2016.

[18] K. Nesbit, A. Dhodapkar, and J. Smith, "Ac/dc: an adaptive data cache prefetcher," in *13th International Conference on Parallel Architecture and Compilation Techniques*, 2004, pp. 135–145.

[19] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *13st International Conference on High-Performance Computer Architecture*, 2007, pp. 63–74.

[20] W. Heirman, K. D. Bois, Y. Vandriessche, S. Eyerman, and I. Hur, "Near-side prefetch throttling: adaptive prefetching for high-performance many-core processors," in *27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 28:1–28:11.

[21] B. Panda and S. Balachandran, "Expert prefetch prediction: An expert predicting the usefulness of hardware prefetchers," *Computer Architecture Letters*, vol. 15, no. 1, pp. 13–16, 2016.

[22] X. Zhuang and H. S. Lee, "Reducing cache pollution via dynamic data prefetch filtering," *IEEE Trans. Computers*, vol. 56, no. 1, pp. 18–31, 2007.

[23] X. Dang, X. Wang, D. Tong, Z. Xie, L. Li, and K. Wang, "An adaptive filtering mechanism for energy efficient data prefetching," in *18th Asia and South Pacific Design Automation Conference*, 2013, pp. 332–337.

[24] V. Selfa, J. Sahuquillo, M. E. Gómez, and C. G. Requena, "Efficient selective multicore prefetching under limited memory bandwidth," *Journal of Parallel and Distributed Computing*, vol. 120, pp. 32–43, 2018.

[25] B. Hall, P. Bergner, A. S. Housfater, M. Kandasamy, T. Magno, A. Mericas, S. Munroe, M. Oliveira, B. Schmidt, W. Schmidt *et al.*, *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8*. IBM Redbooks, 2017.

[26] J. Feliu, S. Petit, J. Sahuquillo, and J. Duato, "Cache-Hierarchy Contention Aware Scheduling in CMPs," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, 2014, pp. 581–590.

[27] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *13th International Symposium on High Performance Computer Architecture*, 2007, pp. 63–74.

[28] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Addressing fairness in smt multicores with a progress-aware scheduler," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 187–196.

[29] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

Carlos Navarro received the BS and MS degrees in computer engineering from the Universitat Politècnica de València (UPV), Spain, in 2018 and 2019, respectively. He is currently working towards a PhD degree at the Department of Computer Engineering (DISCA) of the same university. His PhD research focuses on prefetching strategies in commercial multicore processors.

Josué Feliu received his MSc and PhD degrees in computer engineering from the UPV, Spain, in 2012 and 2017, respectively. He is currently working as a postdoctoral researcher at the Department of Computer Engineering of the same university. His research interests include scheduling strategies and performance modeling for multicore and multi-threaded processors. He was awarded the "IEEE TCSC Outstanding Ph.D Dissertation Award" in 2017.

Salvador Petit (M'07) received the PhD degree in computer engineering for the UPV, Spain. Since 2009, he has been an Associate Professor with DISCA Department at UPV, where he has taught several courses on computer organization. He has authored over 100 refereed conference and journal papers. His current research interests include multithreaded and multicore processors, memory hierarchy design, GPU architecture, and resource management.

Maria E. Gomez received her M.S. and Ph.D. degrees in Computer Engineering from the UPV, Spain, in 1996 and 2000, respectively. She joined the DISCA department at UPV in 1996 where she is currently a Full Professor. She has published more than 70 conference and journal papers. She has served on program committees for several major conferences. Her research interests are on processor architecture and interconnection networks.

Julio Sahuquillo (M'04) received the BS, MS, and PhD degrees from the UPV, Spain, all in computer engineering. He is a Full Professor with DISCA department at the UPV. He has taught several courses on computer architecture. He has authored over 150 refereed conference and journal papers. His research interests include processor microarchitecture, memory hierarchy design, GPU architecture, and system resource management.