Adaptive Prefetching on POWER7: Improving Performance and Power Consumption

VÍCTOR JIMÉNEZ, Barcelona Supercomputing Center and Universitat Politècnica de Catalunya
FRANCISCO J. CAZORLA, Barcelona Supercomputing Center and Spanish National Research Council
ROBERTO GIOIOSA, Pacific Northwest National Laboratory
ALPER BUYUKTOSUNOGLU and PRADIP BOSE, IBM T. J. Watson Research Center
FRANCIS P. O'CONNELL and BRUCE G. MEALEY, IBM Systems and Technology Group

Hardware data prefetch engines are integral parts of many general purpose server-class microprocessors in the field today. Some prefetch engines allow users to change some of their parameters. But, the prefetcher is usually enabled in a default configuration during system bring-up, and dynamic reconfiguration of the prefetch engine is not an autonomic feature of current machines. Conceptually, however, it is easy to infer that commonly used prefetch algorithms—when applied in a fixed mode—will not help performance in many cases. In fact, they may actually degrade performance due to useless bus bandwidth consumption and cache pollution, which in turn, will also waste power. We present an adaptive prefetch scheme that dynamically modifies the prefetch settings in order to adapt to workloads' requirements. We use a commercial processor, namely the IBM POWER7 as a vehicle for our study. First we characterize—in terms of performance and power consumption—the prefetcher in that processor using microbenchmarks and SPEC CPU2006. We then present our adaptive prefetch mechanism showing performance improvements with respect to the default prefetch setting up to 2.7X and 1.3X for single-threaded and multiprogrammed workloads, respectively. Adaptive prefetching is also able to reduce power consumption in some cases. Finally, we also evaluate our mechanism with SPECjbb2005, improving both performance and power consumption.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*Design studies*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; B.3.m [**Memory Structures**]: Miscellaneous

General Terms: Algorithms, Measurement, Performance

Additional Key Words and Phrases: Adaptive system, prefetching, performance, power consumption

This work is supported by a Collaboration Agreement between IBM and BSC. It is also supported by the Ministry of Science and Technology of Spain under contract TIN-2007-60625 and grant JCI-2008-3688.
This work was done while R. Gioiosa was at Barcelona Supercomputing Center.
Author's addresses: V. Jiménez, Barcelona Supercomputing Center, Jordi Girona, 29, 08034 Barcelona, Spain; email: victor.javier@bsc.es; F. J. Cazorla, Barcelona Supercomputing Center, Jordi Girona, 29, 08034 Barcelona, Spain; R. Gioiosa, Pacific Northwest National Laboratory, 902 Battelle Blvd., Richland, WA 99352; A. Buyuktosunoglu and P. Bose, IBM T. J. Watson Research Center, 1101 Kitchawan Rd., Yorktown Heights, NY 10598; F. P. O'Connell and B. G. Mealey, IBM Systems and Technology Group, 11400 Burnet Rd., Austin, TX 78758.

DOI: http://dx.doi.org/10.1145/2588889

ACM Reference Format:
Jiménez, V., Cazorla, F. J., Gioiosa, R., Buyuktosunoglu, A., Bose, P., O'Connell, F. P., and Mealey, B. G. 2014. Adaptive prefetching on POWER7: Improving performance and power consumption. *ACM Trans. Parallel Comput.* 1, 1, Article 4 (May 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2588889

1. INTRODUCTION

Hardware data prefetch is a well-known technique to help alleviate the so-called *memory wall* problem [Wulf and McKee 1995]. Many general purpose server-class microprocessors in the field today rely on data prefetch engines to improve performance for memory-intensive workloads. Some prefetch engines allow users to change some of their parameters. In current commercial systems, however, the hardware prefetcher is typically enabled in a default configuration during system bring-up, and dynamic reconfiguration of the prefetch engine is not an autonomic feature. Nonetheless, commonly used prefetch algorithms—when applied in a fixed, nonadaptive mode—will not help performance in many cases. In fact, they may actually degrade it due to useless bus bandwidth consumption and cache pollution. In this article, we present an adaptive prefetch scheme that dynamically adapts the prefetcher configuration to the running workload, aiming to improve performance; we also show some cases where our mechanism reduces memory power consumption too.

We use the IBM POWER7 [Sinharoy et al. 2011] as the vehicle for this study, since: (i) this represents a state-of-the-art high-end processor, with a mature data prefetch engine that has evolved significantly since the POWER3 time-frame; and (ii) this product provides facilities for accurate measurement of performance and power metrics. POWER7 contains a programmable prefetch engine that is able to prefetch consecutive data blocks as well as those separated by a non-unit, constant stride [Sinharoy et al. 2011]. The processor system is provided to customers with a default prefetch setting that is targeted to improve performance for most applications. But users can manually override the default setting via the operating system, if needed. Users can specify some parameters such as the prefetch depth and whether strided prefetch and prefetch for store operations should be enabled or not. Workloads present different sensitivities to changing the prefetch configuration, even within the class of scientific-engineering applications, which are generally amenable to data prefetch. While the optimal prefetch setting—if known—can lead to a significant performance improvement, the corollary to this, as we show later in this article, is that blindly setting a configuration may reduce performance and waste power consumption.

Our contributions are summarized below. Note that this work expands our recent work [Jiménez et al. 2012], considering both performance and power consumption, as well as enlarging the benchmark set used for characterizing the prefetcher in IBM POWER7, and evaluating our adaptive prefetch mechanism.

— We first provide a motivation for adaptive prefetching, showing how the different prefetch configurations in POWER7 affect performance and power consumption for several workloads. To that end, we use both microbenchmarks and the SPEC CPU2006 benchmark suite [Henning 2006].
— We present a runtime-based adaptive prefetch mechanism capable of improving performance via dynamically setting the optimal prefetch configuration, without the need for a priori profile information. We evaluate the performance benefits of adaptive prefetching. Our adaptive scheme increases performance up to 2.7X and 1.3X compared to the default prefetch configuration for single-threaded and multiprogrammed workloads, respectively. We also show that our mechanism is able to reduce memory power consumption in some cases.

— We evaluate the impact of our adaptive prefetch mechanism on a Java server-side workload (SPECjbb2005). For that benchmark, adaptive prefetching is able to both improve performance by 21% and reduce memory power consumption by 22%.
— We also study the implementation of such an adaptive prefetch scheme within the OS kernel. After implementing our adaptive mechanism into the Linux kernel, we have observed similar performance improvements to those obtained by the userspace implementation.

This article is organized as follows. Section 2 provides background for reading the article. Section 3 describes the POWER7 processor, providing information on the different knobs that control the prefetcher. It also characterizes the effect of the different knobs on performance and power consumption. Section 4 describes the methodology that we use in this article. Section 5 shows the implementation of an adaptive prefetch mechanism and evaluates its impact on performance and power consumption. Section 6 shows a possible OS-based implementation. Finally, Section 7 presents the conclusions of this article.

2. RELATED WORK

There is a significant record of past research in data prefetch (e.g., [Baer and Chen 1991; Jouppi 1990; Palacharla and Kessler 1994]). Most of the initial proposals were based on sequential prefetchers. They prefetch sequential memory blocks relying on the fact that many applications exhibit spatial locality. Although sequential prefetchers work effectively in many cases, there are applications with nonsequential data access patterns that do not benefit from sequential prefetching. That motivated the research on more complex prefetchers that try to capture the nonsequential nature of these applications. Prefetch techniques targeting pointer-based applications have been studied [Ebrahimi et al. 2009; Roth et al. 1998; Yang and Lebeck 2000]. Joseph and Grunwald [1997] study Markov-based prefetchers and present solutions to limit the bandwidth devoted to prefetching. [Solihin et al. 2002] use a user-level memory thread in order to prefetch data, delivering significant speedups even for applications with irregular accesses. Yet, most of these prefetch designs have not been implemented in a real processor. Limit studies and prefetch analytical models have been presented in Emma et al. [2005] and Srinivasan et al. [2004].

With the advent of CMP processors, interaction between threads is taken into account when designing a prefetch system. Ebrahimi et al. [2011] and Lee et al. [2008] study the effect of thread-interaction on prefetch, and propose techniques to design prefetch systems that improve throughput or fairness. The impact of prefetching and bandwidth partitioning in CMPs has been studied in Liu and Solihin [2011].

Although there are lots of studies on prefetching based on simulators, there are very few works that deal with hardware-based measurement and characterization. Wu and Martonosi [2011] characterize the prefetcher of an Intel Nehalem processor and provide a simple algorithm to dynamically control whether to turn the prefetcher on or off. Their study, however, is solely oriented towards reducing intra-application cache interference without taking actual system performance into consideration. In Liao et al. [2009] construct a machine learning model that dynamically modifies the prefetch configuration of the machines in a data center (based on Intel Core2 processors). They improve performance for some applications by enabling/disabling prefetch. In our case, however, the POWER7 processor not only allows us to enable/disable prefetch, but it exposes a set of knobs to fine tune the prefetch configuration.

There are other examples of adaptive mechanisms for controlling thread execution rate. For instance, different instruction fetch policies in the context of SMT processors have been studied in [Cazorla et al. 2006; Choi and Yeung 2006], aiming to increase

Table I.

Notation used in this article for referring to prefetch configurations. We use tags (W/S) to indicate whether *prefetch on stores* (W) or *stride-N* (S) are enabled. Prefetch depth can be set to *default* (D) or to any value in the range 2–7 (*shallowest-deepest*). The special configuration where depth is 001 turns off the prefetcher (O). Table I(b) shows some examples with this notation.

Shortname	DSCR value	Description
O	xx001	Off (prefetch disabled)
D	xx000	Default depth
2	xx010	Shallowest
3	xx011	Shallow
4	xx100	Medium
5	xx101	Deep
6	xx110	Deeper
7	xx111	Deepest
W	x1xxx	Prefetch on stores
S	1xxxx	Stride-N

(a) Notation

Shortname	Depth	Prefetch on stores	Stride-N
D	Default	No	No
WD	Default	Yes	No
SD	Default	No	Yes
SWD	Default	Yes	Yes
S2	Shallowest	No	Yes
S3	Shallow	No	Yes
7	Deepest	No	No
S7	Deepest	No	Yes
SW7	Deepest	Yes	Yes

(b) Examples

throughput or provide quality of service (QoS). Similarly, Boneti et al. [2008] explore the usage of the dynamic hardware priorities present in IBM POWER5 processor for controlling thread resource balancing and prioritization. Qureshi and Patt [2006] study the problem of last-level cache partitioning between multiple applications for improving throughput. Moreto et al. [2009] present a similar technique, but focus on achieving QoS for the co-running applications.

3. THE POWER7 PROCESSOR

The IBM POWER7 [Sinharoy et al. 2011] processor is an eight-core chip where each core can run up to four threads. Each core contains two 32KB L1 caches (for instructions and data), plus a 256KB L2 cache. The processor contains an on-chip 32MB L3 cache. Each core has a private 4MB portion of the L3 cache, although it can access the rest of portions from other cores—with higher latency. A core can switch between single-thread (ST), two-way SMT (SMT2), and four-way SMT (SMT4) execution modes.

3.1. POWER7 Prefetcher

POWER7's prefetcher [Cain and Nagpurkar 2010; Hur and Lin 2006; IBM 2010] is programmable and allows users to set different parameters (knobs) that control its behavior: (i) *prefetch depth*, how many lines in advance to prefetch, (ii) *prefetch on stores*, whether to prefetch store operations, and (iii) *stride-N*, whether to prefetch streams with a constant stride larger than one cache block. The prefetcher is controlled via the data stream control register (DSCR). The Linux kernel exposes the register to users through the sys virtual filesystem [Mochel 2005], allowing them to set the prefetch setting on a per-thread basis.

Table I(a) describes the possible prefetch configurations and introduces the notation that will be used throughout the article. Prefetch depth can take values from 2 (*shallowest*) to 7 (*deepest*). Additionally, there are two special values that can be used: 001b (O) and 000b (D). The former disables the prefetcher, while the latter is the system-predefined default depth. In POWER7 the default depth corresponds to depth 5 (*deep*) [Abeles et al. 2010], being automatically selected when the system boots. *Prefetch on stores* (W) and *stride-N* (S) can only be enabled or disabled; they are disabled in the default configuration. Therefore, the default configuration corresponds to configuration 5 (using our notation). Every knob in the prefetcher can be independently configured. Table I(b) shows some examples of the possible combinations that can be

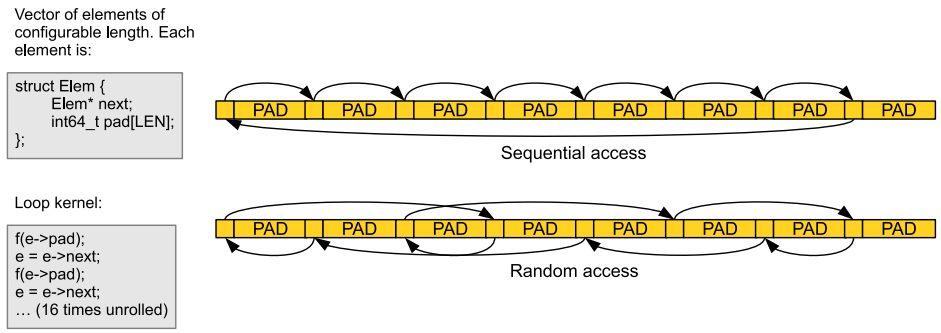

Fig. 1. Microbenchmarks description. The microbenchmarks perform an array traversal either in sequential or random order. The distance between accesses is a configurable parameter. Depending on function f, the accesses to every array element can be loads, stores or both.

formed by setting values for each prefetch knob. Additionally, it shows how the short-names that we use in the article are constructed.

3.2. Impact of Prefetch Settings on Microbenchmarks

In order to understand the behavior of the multiple knobs available in POWER7's prefetcher, we use several microbenchmarks and characterize their effect on performance, memory bandwidth and power consumption.

Real applications present phases and significant dynamic variations during their execution, complicating the task of fine-grain architectural characterization. In addition to that, OS interferences and asynchronous I/O services further complicate the analysis. Microbenchmarks with well-defined characteristics simplify this problem by allowing us to understand the behavior of the different architectural components in isolation.

We developed a set of synthetic microbenchmarks that stress the prefetcher, caches and memory subsystem in different ways. By using them, we can understand the behavior of the prefetcher and its interaction with the rest of the memory hierarchy. The basic structure for all the microbenchmarks consists of an array traversal following a given order and bringing lines from a given point in the memory hierarchy to levels closer to the CPU. Figure 1 shows the implementation details of the microbenchmarks as well as two access patterns (sequential and random traversal). Each element of the array is composed of a pointer to the following element—the next element will depend on the type of traversal—and a padding area. The length of the padding area will determine how consecutive lines are accessed. For instance, if the size of the element structure equals the size of a cache line, every step in the traversal will touch a line. This design, when applied within a sequential traversal, will bring adjacent lines from the memory to the low-level caches. If the padding size is bigger, however, two consecutive accesses will not touch adjacent lines. Although sequential prefetching does not help with this access pattern, we will see that if stride-N is enabled, the prefetcher is able to improve performance.

In this characterization we use three microbenchmarks based on the scheme presented in Figure 1: seq-bench, seq-bench-stride, and rnd-bench. The first one performs sequential accesses to consecutive cache lines. The second one is similar but the stride between two accesses is larger than a cache line. This creates an access pattern that a sequential prefetcher cannot identify. Finally, the last one performs random accesses, therefore it does not benefit from prefetch. All the microbenchmarks are mainly composed of memory load operations, and they are heavily memory-bound workloads.

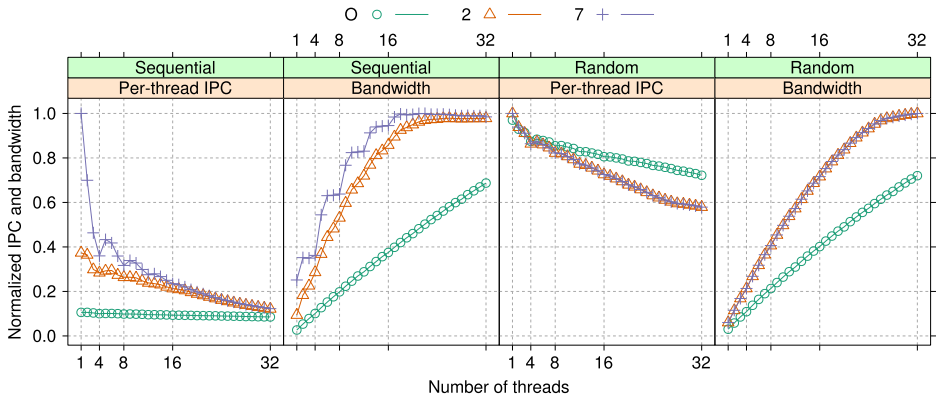

Fig. 2. Prefetch depth effect characterization. Both sequential and random microbenchmarks are used to show the effect of prefetch depth on performance and memory bandwidth. Threads are bound to contexts in an increasing order (the first four threads go to the first core, the next four ones go to the second core, and so on). Values are normalized to the maximum value observed in each plot.

Performance Results. Figure 2 shows the results of running an increasing number of threads (from 1 to 32) under different prefetch configurations, varying prefetch depth. The left part of the figure shows per-thread IPC and memory bandwidth for seq-bench. This workload accesses consecutive memory blocks, and hence prefetch really helps in this case. Prefetch depth significantly affects performance too, with the deepest configuration (7) achieving a 2.6X speedup over the shallowest one (2) for the single-thread case. As more threads run, memory bandwidth consumption significantly increases, and prefetch depth does not make a significant difference after eight threads are running in the system—with the exception of depth 2. After that point, the effect of prefetch depth is somehow limited, but there is still a large performance gap between enabling and disabling prefetch. If more threads are added, at some point memory bandwidth saturates and performance asymptotically converge to the same performance as when the system is not using prefetching. Although this example helps us to understand the effect of prefetch depth on both performance and memory bandwidth, we must bear in mind that it is an extreme case, since the workload is mainly composed of operations that continuously access memory.[1] When more realistic workloads (e.g., SPEC CPU2006) are used, pressure on bandwidth is not so high, and prefetch depth keeps helping beyond the early saturation point seen in this example.

The right part of Figure 2 shows the same experiment with a benchmark that accesses memory positions in random order. In this case, prefetch cannot help, since the workload's access pattern is not sequential. In fact, if prefetch is enabled, bandwidth consumption increases up to 1.5X compared to the case when prefetch is disabled. Upon encountering a cache miss, the prefetcher sends an L3 prefetch for the next cache line. Since those prefetches are useless, they do not contribute to increase performance, but they actually increase memory bandwidth and create more cache conflicts. Because of all these factors, disabling prefetch actually provides the best performance, especially when the number of threads increases.

Figure 3 shows the effect of stride-N for several choices of prefetch depth on the seq-bench-stride microbenchmark. We only display the results for depths 2 and 7 in order to ease the comprehension of the figure—the remaining depth values would lie

[1]POWER7-based systems contain *two* memory controllers, however ours is a low-end system with only *one* controller. This could explain why bandwidth is saturated with fewer threads.

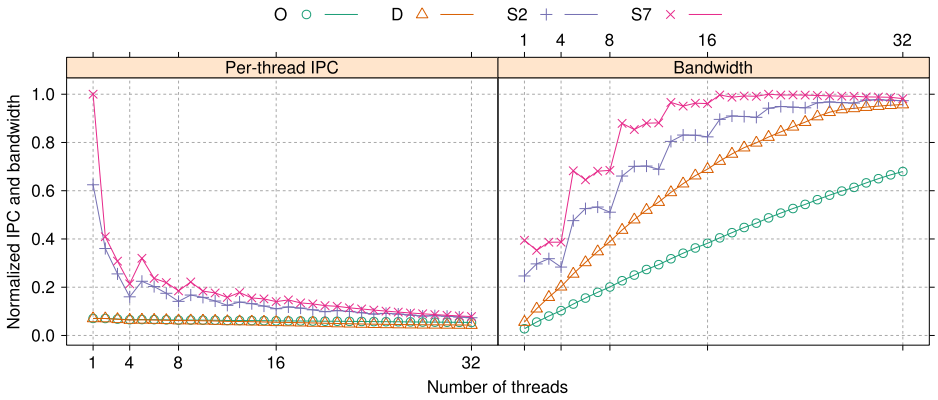

Fig. 3. Stride-N and prefetch depth effect characterization. A sequential strided microbenchmark is used to show the effect of stride-N and prefetch depth on performance and memory bandwidth. Threads are bound to contexts in an increasing order (the first four threads go to the first core, the next four ones go to the second core, and so on). Values are normalized to the maximum value observed in each plot.

in between 2 and 7. As it was expected, the default prefetch configuration (D) does not improve performance for this benchmark. Since accesses to memory are sequential, but they are not to adjacent cache lines, sequential prefetching does not help. The performance for the default configuration (D) is exactly the same as when prefetch is disabled. As we can see in Figure 3, however, the default configuration consumes significantly more bandwidth than turning prefetch off (O), without obtaining any performance benefit. Once stride-N is enabled (configurations S2 and S7), the prefetcher is able to identify the strided access pattern, and a significant speedup is achieved. The effect of prefetch depth is similar to the one observed for seq-bench (Figure 2): when the number of threads is low, increasing prefetch depth achieves a significant speed up. But, as the thread count increases, the impact of prefetching considerably reduces.

Power Consumption Results. Figure 4 shows both memory and total system power consumption for the same experiments shown in Figure 2. In all the cases, power consumption is significantly lower when prefetch is disabled. For seq-bench there is up to 30% memory power consumption difference between enabling and disabling prefetch. In terms of total system power consumption, the difference is still very significant (up to 10%). We must remember, however, that this power consumption reduction comes at the cost of a significant decrease in performance (see Figure 2). We computed energy efficiency using *energy-delay product*, and the results show that when both performance and power consumption are taken into account, disabling prefetch is not an efficient decision for seq-bench-like workloads.

The power consumption results for rnd-bench are similar to the ones observed for seq-bench. In this case the maximum observed difference between enabling and disabling prefetch is 18% for memory power consumption (5% for total system power). But as Figure 2 shows, a benchmark with a random access pattern does not benefit from prefetch, and performance is typically better when prefetch is disabled. Therefore, this case is a win-win situation. Disabling prefetch both improves performance and reduces power consumption, boosting system efficiency.

Power consumption results for the case when stride-N is enabled are similar to the ones presented in Figure 4. Because of that, they are not presented in the article.

Overall, we have seen that prefetch depth can significantly influence performance as long as memory bandwidth is not under a lot of pressure. When bandwidth gets saturated due to a high amount of demand loads generated by the benchmarks running

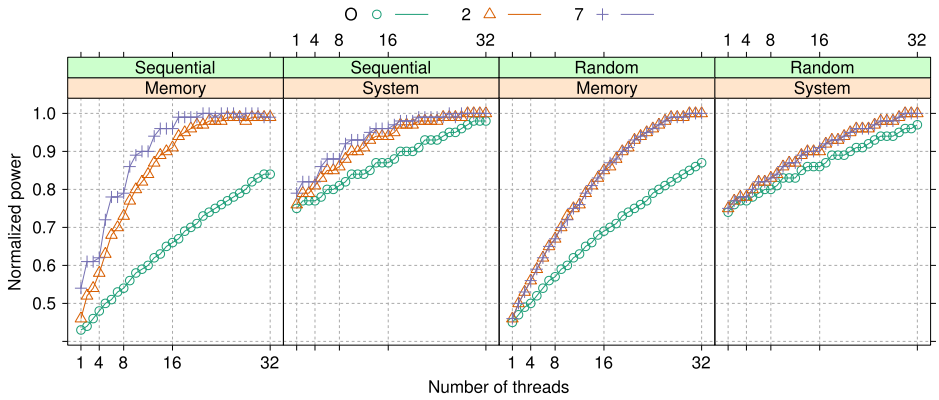

Fig. 4. Memory and total system power consumption both for sequential and random microbenchmarks. Threads are bound to contexts in an increasing order (the first four threads go to the first core, the next four ones go to the second core, and so on). Values are normalized to the maximum value observed in each plot.

on the system, prefetch does not help as much anymore. Additionally, if a workload generates many useless prefetches, bandwidth consumption will increase, which may hurt system performance. We have also seen that prefetch typically increases power consumption. When prefetch is useful, power consumption increases with respect to prefetch aggressiveness. When prefetches are not useful, they may decrease system performance and waste power at the same time. All these observations are useful to understand the results with real benchmarks in the following sections.

3.3. Impact of Prefetch Settings on SPEC CPU2006

In the previous section we have studied the effect of prefetch settings on performance and power consumption for a set of microbenchmarks. In this section we conduct a similar study with more realistic workloads, using the SPEC CPU2006 benchmark suite.

Prefetching affects workloads in different ways, depending on their nature. Some experience a significant speedup when prefetch is used, while others are totally insensitive. We classify the benchmarks in four different groups, according to the way prefetch affects their performance when running in single-thread mode on our POWER7 system.

(i) *Prefetch-Insensitive (PI)*. This type of benchmark is insensitive to prefetch. It does not suffer any significant performance variation no matter whether prefetch is enabled or not. Additionally, the various configurations (e.g., depth, stride-N and prefetch-on-stores) do not affect its performance (e.g., sjeng and gamess).
(ii) *Prefetch-Friendly (PF)*. Enabling prefetch positively affects the performance of the benchmarks in this group. But, they are not affected when the prefetch setting is varied (e.g., zeusmp and cactusADM).
(iii) *Config-Sensitive (CS)*. For the benchmarks in this group, the performance also increases when prefetch is enabled. Moreover, changing the prefetch configuration affects their performance too (e.g., enabling stride-N improves the performance with respect to the default configuration; this is the case for mcf and milc).
(iv) *Prefetch-Unfriendly (PU)*. For this type of benchmark, enabling prefetch negatively affects its performance (e.g., omnetpp and povray).

Figure 5 shows the performance for SPEC CPU2006 benchmarks—representatives of each different class—running in single thread mode under several prefetch settings.

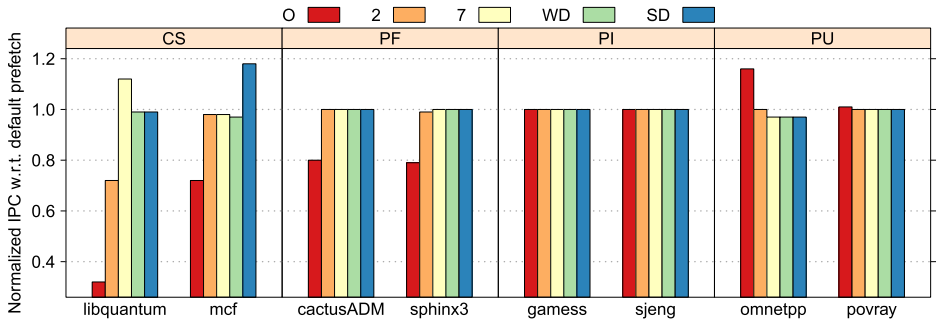

Fig. 5. Effect of prefetch on performance for single-threaded runs. Multiple prefetch configurations are used in order to show the effect of each prefetch knob: depth (2–7), prefetch on stores (WD), and stride-N (SD)—refer to Table I for notation on prefetch configurations.

Table II.

Benchmark classification based on how their performance is affected by the prefetch settings when running in single-thread mode.

Classes	Benchmarks		
Prefetch-insensitive	perlbench	bzip2	gamess
	gromacs	namd	gobmk
	sjeng	h264ref	tonto
	astar	xalancbmk	
Prefetch-friendly	gcc	zeusmp	cactusADM
	dealII	calculix	hmmer
	GemsFDTD	lbm	wrf
	sphinx3		
Config-sensitive	bwaves	mcf	milc
	leslie3d	soplex	libquantum
Prefetch-unfriendly	omnetpp	povray	

We use the default prefetch configuration (D) as the baseline to normalize IPC. That figure easily visualizes prefetching impact on the different classes that we use to classify the benchmarks (Table II contains the exact classification for all the SPEC CPU2006 benchmarks).

In terms of power consumption, Figure 6 shows CPU and memory power consumption for the same benchmarks appearing in Figure 5. The values are normalized to the ones obtained with the default prefetch configuration. In terms of CPU power, there is not too much variation when using the most aggressive prefetch setting (SW7). When disabling prefetch (O), CPU power consumption decreases up to 3% (for libquantum). The reduction of memory power consumption is much more significant—especially for config-sensitive, prefetch-friendly and prefetch-unfriendly benchmarks. For instance, power consumption for libquantum decreases 15% but at the expense of reducing its IPC more than 60% (see Figure 5). Perhaps the most interesting cases are povray, and especially omnetpp. For the latter, disabling prefetching reduces memory power consumption 10% while, at the same time, performance increases close to 20%. That is a win-win situation, caused by avoiding useless bandwidth consumption due to inefficient prefetches. Power consumption is decreased for povray too, although in this case the reduction is more modest (5%).

Overall, as results in Figures 5–6 show, an adaptive prefetch mechanism could tune the prefetcher for every particular benchmark in order to find the prefetch

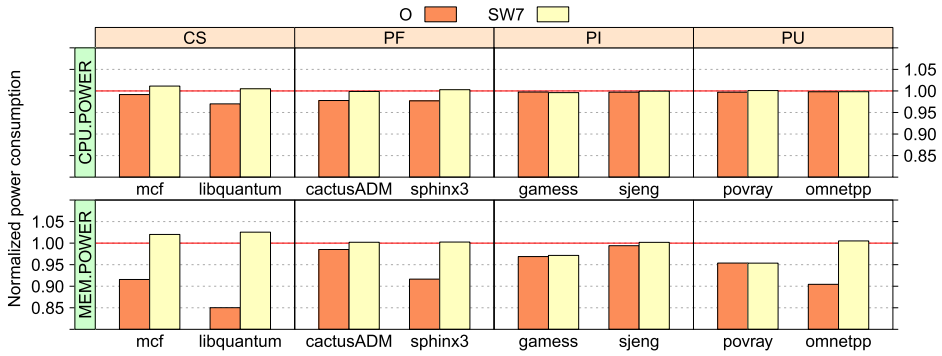

Fig. 6. Effect of prefetch on CPU and memory power consumption for single-threaded runs. The values are normalized to the ones obtained with the default prefetch configuration.

configuration that leads to its optimal performance, potentially saving power consumption at the same time.

4. METHODOLOGY

We use an IBM BladeCenter PS701 to conduct all the experiments, including the ones in the previous section. This system contains one POWER7 processor running at 3.0 GHz and 32 GB of DDR3 SDRAM running at 800 MHz. The operating system is SUSE Linux Enterprise Server 11 SP1. We use IBM XL C/C++ 11.1 and IBM XL Fortran 13.1 compilers to compile all the SPEC CPU2006 benchmarks. We disable compiler-generated prefetch instructions in order to avoid interactions between these instructions and the hardware prefetcher. We also use SPECjbb2005 [SPEC 2005] for evaluating adaptive prefetching. The Java VM is IBM J9 VM build 2.4. For collecting information from the performance counters we use *perf*, the official implementation in the mainstream Linux kernel [de Melo 2010].

Power measurements are obtained using the IBM Automated Measurement of Systems for Temperature and Energy Reporting software [Floyd et al. 2011; Lefurgy et al. 2007]. The software connects to the EnergyScale microcontroller to download real-time power, temperature, and performance measurements of POWER7 microprocessor and server. The software samples sensors at 1-ms granularity. By using this software we can access multiple sensors in the system, making it possible to sample total system power, chip power and memory power.

5. ADAPTIVE PREFETCHING

In Section 3.3 we have seen that different applications derive maximum performance benefit from different prefetch settings. In that approach, users need to profile applications prior to running them in order to determine the best prefetch setting for each application. We refer to this method as the *best static configuration* approach or, simply, the static approach. In this approach, a priori profiling yields the optimal prefetch configuration for a given application, and all future runs of this application would use this optimal configuration to achieve its efficiency target. Note that in this approach, the prefetch configuration is statically fixed for the duration of the application run. A truly dynamic adaptation of the data prefetch algorithm presents the promise of two potential benefits: (i) users would be able to avoid the per-application profiling step; and, (ii) dynamic phase changes within the same application would trigger adaptation of the prefetch parameters in order to further maximize the targeted efficiency metric.

Algorithm 1 Base adaptive prefetch algorithm

```
1: for all t in threads do
2:    for all ps in pref_settings do
3:       set_prefetch(cpu(t), ps)
4:       wait T_e ms
5:       ipc[ps] = read_pmcs()
6:    end for
7:    best_ps = arg max_ps(ipc)
8:    set_prefetch(cpu(t), best_ps)
9: end for
10: wait T_r
```

5.1. Basic Adaptive Algorithm

Algorithm 1 contains two configurable parameters, T_s and T_r. The former specifies the interval length to be used during the exploration phase (line 4). The latter is the amount of time that the best settings found during the exploration phase will be used before a new exploration phase starts (line 10). In our implementation we use the interval lengths $T_e = 10ms$ and $T_r = 100ms$. This granularity is a good compromise between adaptability and overhead. It is actually a typical value for sampling-based approaches in the OS and runtime realms [Isci et al. 2005]. For instance, the Linux kernel allows the user to choose the granularity of the timer tick from 1 ms up to 10 ms. A finer granularity would introduce a significant overhead in the system.

This first algorithm is the base for the other two presented in this article. It, however, suffers from two potential problems: the effect of phase changes and the impact of "inefficient" prefetch settings (for a particular workload). Next, we examine and present solutions for these two problems.

5.2. Impact of Phase Changes

It is well known that applications present phases during their execution [Denning 1968]. They actually present phases at different levels, ranging from the microsecond to the millisecond level (some phases may even last for some seconds). Our adaptive mechanism periodically samples performance for different settings, attempting to find the best setting for that particular interval. We must, however, take care of possible phase changes that may occur between different samples in the exploration phase. Otherwise, we could attribute a performance change to the effect of a given prefetch setting when the real reason is an underlying phase change between measurements.

In order to alleviate this problem we use a moving average buffer (MAB) [Abraham and Ledolter 1983] that keeps the last m IPC samples for every prefetch setting and thread under control of the adaptive prefetch runtime. We then compare the performance of prefetch settings by using the mean of the values in the buffer, instead of using individual measurements. We evaluate the effect of using buffers of different sizes on the IPC variability between consecutive samples. Figure 7 shows the normalized IPC variability as we increase the buffer size from 1 (i.e., no buffer is used) up to 32. IPC variability is computed with the following equation:

$$variability = \frac{1}{n-1} \sum_{i=1}^{n-1} |IPC_{i+1} - IPC_i|, \tag{1}$$

where IPC is an array with all the n IPC samples for a given workload execution. Variability is then normalized to the average IPC for every workload. For clarity reasons the figure is split into two. Figure 7(a) contains the results for SPEC INT benchmarks and Figure 7(b) does so for SPEC FP benchmarks. As it can be seen in the figure, most

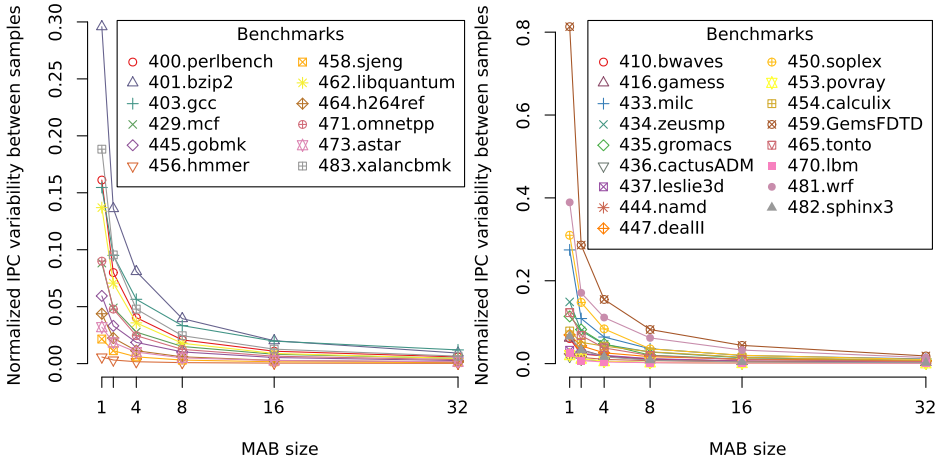

Fig. 7. Effect of changing the buffer size on intersample IPC variability. IPC variability (see Equation (1)) is normalized to the average IPC for each benchmark.

Algorithm 2 Adaptive prefetch with MAB

```
1: for all t in threads do
2:     for all ps in pref_settings do
3:         set_prefetch(cpu(t), ps)
4:         wait T_e ms
5:         push(ipc_mab[t,ps], read_pmcs())
6:         ipc_mean[ps] = mean(ipc_mab[t,ps])
7:     end for
8:     best_ps = arg max_ps(ipc_mean)
9:     set_prefetch(cpu(t), best_ps)
10: end for
11: wait T_r
```

of the benchmarks present a small to moderate variation when MAB is not used. A few of them (bzip2, perlbench, wrf, and GemsFDTD), however, have quite a high variation. As an example, let us examine bzip2. The average IPC variability between consecutive samples is 30% when MAB is not used. As the buffer size increases the variability is reduced, reaching 2% for a buffer containing the last 32 samples. By using a moving average buffer, we are able to significantly reduce the impact that phase changes may have on the exploration step of the adaptive prefetch mechanism.

Algorithm 2 presents the new version of the algorithm, using the moving average buffer. The algorithm is very similar to the one presented in the previous section. The only differences are on lines 5, 6, and 8, where the buffer is actually used. The operation of pushing a new sample into the buffer (line 5) is implemented using a circular buffer. Thus, when the buffer is full and a new sample is added, the oldest one is removed from the buffer.

5.3. Impact of "Inefficient" Prefetch Settings

The base adaptive prefetch algorithm iterates along a set of prefetch settings during the exploration phase. After the exploration phase is over, the runtime lets the threads run for a certain amount of time with the best setting found. Depending on the workload, there could be a significant performance variation between different settings used in the exploration phase. For instance, for bwaves, disabling the

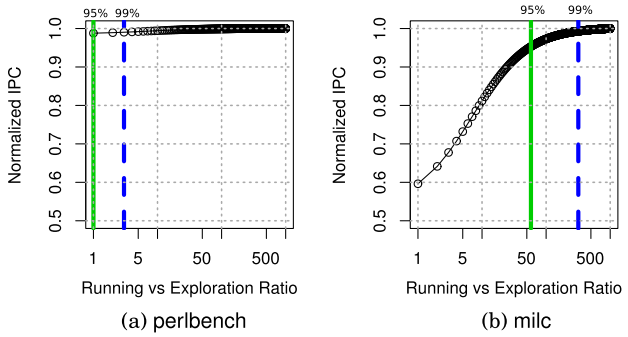

Fig. 8. Effect of exploration/running ratio on expected performance. Values are normalized to the maximum values observed for each workload.

prefetch reduces its performance 78% with respect to the best setting. Such a significant slowdown may actually impact overall performance if the exploration phase is executed too often. Therefore, for this particular workload disabling prefetch would be an inefficient prefetch setting (it is important to mention that an inefficient setting for one workload may be the best one for another workload; thus settings' efficiency is workload-dependent).

In order to quantify the effect of inefficient settings, we model the expected performance, $\widehat{IPC}$, based on the ratio of exploration and running phases' length. We use the following equation for the model:

$$\widehat{IPC} = \sum_{i=1}^{n} \frac{L_e}{L_t} \times IPC_i + \frac{L_r}{L_t} \times \max_i(IPC_i), \tag{2}$$

where L_e and L_r are the lengths of the exploration and running phases, respectively, and $L_t = L_e + L_r$. IPC is a set containing the average IPC values for each prefetch setting for a given workload.

Figure 8 shows the expected performance for two different types of workloads as the ratio L_r/L_e increases. The solid, green vertical line determines the running-exploration ratio such that the expected performance is within 5% of the best achievable performance (i.e., if there is no exploration phase and the best prefetch setting is used during all the interval). The dashed, blue vertical line is equivalent to the previous one, but it marks the point where the expected performance is within 1% of the best achievable performance. We use two benchmarks, perlbench and milc, to construct an illustrative example. The results for all the other SPEC CPU2006 benchmarks are similar to either one of these two.

Figure 8(a) shows the results for perlbench. This workload is mostly insensitive to prefetching and, thus, the expected performance follows a very flat curve. In order not to lose more than 1% of performance, it suffices to set a running phase four times longer than a single exploration interval. For these types of workloads, since they do not really suffer from inefficient prefetch settings, the running-exploration ratio is not so important. This totally changes for a different type of workload such as milc. Figure 8(b) shows the results for this workload. In this case the curve is not flat anymore. Indeed if we are not willing to pay a performance drop bigger than 5% we must use a running phase at least 50 times longer than a single exploration interval. For a tighter 1% bound, the ratio would increase up to approximately 400. Using such a large value for all the possible workloads would imply a drastic reduction in the number of

times that an exploration phase is triggered. Thus, the adaptability of our mechanism would be significantly reduced.

In order to avoid this issue we decided to introduce a new feature in our adaptive prefetch scheme. This feature removes inefficient prefetch settings from the set containing all the settings to be tried during the exploration phase. We call this feature *prefetch setting drop*. Settings are "dropped" for a certain amount of time based on their inefficiency and then, they are considered again to be selected in a future exploration phase. The exact number of iterations, IT_i, that a given setting, i, will be dropped is given by the following equation:

$$IT_i = DF \times |MAB| \times \left(\frac{max_i(IPCi)}{IPC_i} - 1 \right), \tag{3}$$

where DF is the *drop factor*, $|MAB|$ is the size of the moving average buffer and the last term is a measure of the slowdown experienced when using setting i. If the performance for setting i is equal to the best performance observed, the last term becomes zero and the setting is actually not dropped at all, so it will be used in the next exploration phase. The slowdown term in the last equation penalizes inefficient settings proportionally to the measured slowdown. Thus, settings that significantly deviate from the best setting's performance will be penalized more than the others. The equation drops settings proportionally to the size of the moving average buffer too. After a setting is dropped, its MAB is reset, because by the time the setting is considered again in the exploration phase, the contents of the buffer may not be valid anymore. Moreover, the adaptive mechanism does not give a prediction for a setting until its associated buffer is full—doing so would be equivalent to not using a buffer. Therefore, $|MAB|$ exploration phases are necessary before the algorithm can decide whether a prefetch setting that has just been reconsidered again for inclusion continues to be an inefficient setting and, consequently, must be dropped once more. The bigger the size of the moving average buffer, the more potentially harmful effect that an inefficient setting may have. Thus, Equation (3) includes a term that drops settings proportionally to the size of the moving average buffer.

In Equation (3) the drop factor, DF, is the only parameter that the adaptive prefetch mechanism's designer or the end-user must select a value for. Its value will depend on the workloads that the end-user will ultimately execute on the system. Based on mathematical performance modeling and empirical analysis, it is possible to select a default value for that parameter. We use a similar approach as we did to determine the effect of the exploration-execution ratio on performance. In this case, we model the effect of changing the drop factor on the expected performance. We use the following equation to model the impact on performance of different drop factor values for the case of two prefetch settings:

$$\widehat{IPC}_i = \frac{t_1}{T} \times IPC_{best} + \frac{t_2}{T} \times \alpha IPC_{best}, \tag{4}$$

where t_1 and t_2 correspond to the amount of time that setting one and two are respectively selected. Their values are $|MAB| + IT_i$ and $|MAB|$, respectively. Finally, T is the total interval time $(t_1 + t_2)$ and α is the reduction in performance of setting two compared to the first one. Figure 9 shows normalized expected performance for several drop factor values, for the case of two prefetch settings. One of the settings corresponds to the best setting in a given interval (performance = 1.0). We include results for different performances (α) for the second setting, ranging from 1% to 50% slowdown.

As it can be observed in Figure 9, settings that are close to the best one do not reduce performance significantly and, thus, they do not need to be dropped for a long time—if at all. As the performance of the second setting decreases, the impact on performance

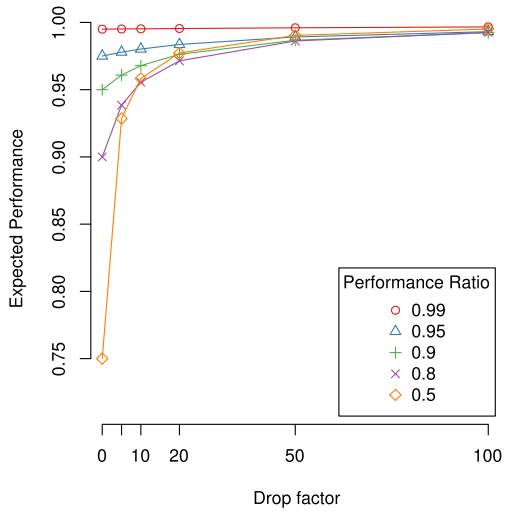

Fig. 9. Effect of drop factor on expected performance. Values are normalized to the best possible performance.

Algorithm 3 Adaptive prefetch with MAB and inefficient setting drop.

```
1:  for all t in threads do
2:    for all ps in pref_settings do
3:      drop_iter[t,ps] = max(0, drop_iter[t,ps] − 1)
4:      if drop_iter[t,ps] = 0 then
5:        set_prefetch(cpu(t), ps)
6:        wait T_e ms
7:        push(ipc_mab[t,ps], read_pmcs())
8:        ipc_mean[ps] = mean(ipc_mab[t,ps])
9:      end if
10:   end for
11:   best_ps = arg max_ps(ipc_mean)
12:   set_prefetch(cpu(t), best_ps)
13:   for all ps in pref_settings do
14:     SL = (ipc_mean[best_ps] / ipc_mean[ps] − 1)
15:     drop_iter[t,ps] = DF × |MAB| × SL
16:   end for
17: end for
```

becomes much more noticeable. For instance, if the performance for the second setting is 50% compared to the best setting, not using the drop feature would lead to an estimated overall performance of 75% compared to when just the best setting is used. As the drop factor increases, the impact of inefficient settings clearly reduces and the expected performance tends to converge to the performance obtained with the best prefetch setting.

Algorithm 3 presents the latest version of the adaptive prefetch mechanism, both including the moving average buffer and the drop feature. As it can be seen there is no running phase in this algorithm. The running phase is not necessary anymore since the drop feature removes inefficient settings, thus, allowing us to perform a continuous exploration. Before trying a prefetch setting, the algorithm decrements the number of drop iterations for that setting, and it only actually considers the setting if it is not dropped (lines 3–4). In lines 13–16 the algorithm computes the number of iterations

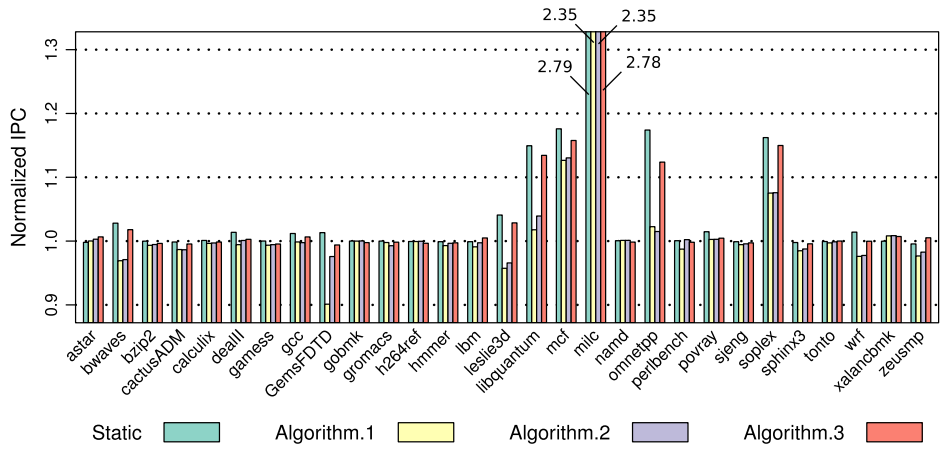

Fig. 10. Performance results for single-threaded workloads normalized to the ones obtained with the default prefetch configuration.

that inefficient settings will be dropped. We select $DF = 100$ based on the previous analysis and on empirical evaluation, obtaining good performance for all the benchmarks both for single-threaded and multiprogrammed workloads, as we will see in the next section.

5.4. Results

In this section we evaluate the performance benefits of using adaptive prefetching. We use both single-threaded workloads as well as multiprogrammed workloads composed of random SPEC CPU2006 benchmark pairs.

5.4.1. Single-Threaded Workloads. Figure 10 shows the results for single-threaded workloads. We present results for all the SPEC CPU2006 benchmarks. Performance values are normalized to the ones obtained with the default prefetch configuration. As it can be seen in the figure, many of the benchmarks do not experience any performance variation. That is especially true for prefetch-insensitive workloads. In that case, neither the best static nor the adaptive approaches improve performance. It is important to notice that while the first and the second algorithm may experience a performance decrease compared to the default configuration—due to, for instance, inefficient settings—that is not the case for the third algorithm. Algorithm 3 does not perform worse than the default configuration for any of the benchmarks. That is an important observation, since otherwise it may not be "safe" to unconditionally enable adaptive prefetching.

We can observe the effect of the moving average buffer especially in the case of GemsFDTD. This benchmark is the one that suffered the most from intersample variability (see Figure 7). By using a MAB we can reduce the impact of IPC variability between samples and improve performance.

If we look at config-sensitive workloads we observe that adaptive prefetching performs nearly as well as the best static approach. SPEC CPU2006 benchmarks present little variability in terms of which prefetch setting they most benefit from along their execution. Because of this, it is typically not possible for dynamic prefetching to beat the static approach—we look at this in more detail in Section 5.4.2. The speedups obtained with the adaptive scheme are, however, very significant (in the order of 15% for mcf, soplex, and libquantum). In the case of milc we observe a large speedup of 2.7X. While all those workloads benefit from prefetch and they see their performance

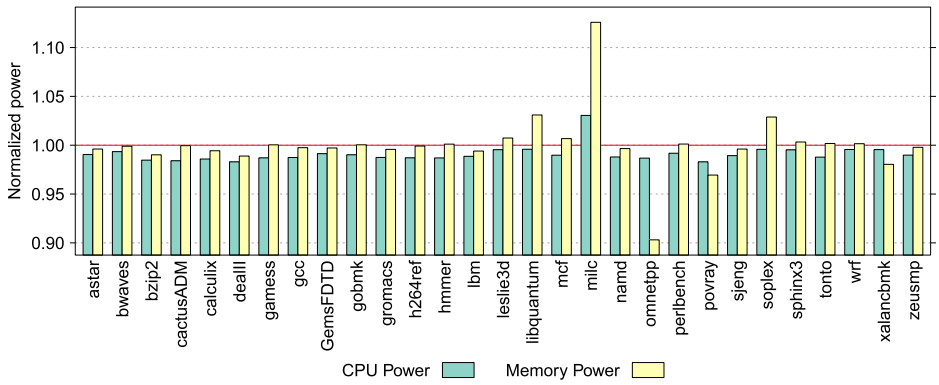

Fig. 11. CPU and memory power consumption results for single-threaded workloads using Algorithm 3. The values are normalized to the ones obtained with the default prefetch configuration.

increased when the right setting is selected for them, omnetpp behaves in a completely different way, and it actually benefits from disabling prefetch. By profiling this benchmark we have seen that it spends a significant percentage of its execution time traversing a heap. A heap is a tree-like data structure, and when traversing it, accesses between nodes are separated by a variable stride. This access pattern is very difficult for a sequential prefetcher, even if stride-N is enabled. In fact, prior research already showed that omnetpp does not benefit from prefetch [Ebrahimi et al. 2009; Lee et al. 2008]. When prefetch is disabled during all the execution (static approach), performance for omnetpp increases 17%. Adaptive prefetching detects that, and turns off prefetch most of the time, significantly improving performance too.

Figure 11 shows CPU and memory power consumption results for all the benchmarks when running under Algorithm 3. CPU power consumption is slightly lower for all benchmarks except for milc. That benchmark experiences a 2.8X speedup when running under adaptive prefetching. Selecting the right prefetch setting reduces the impact of cache misses, increasing both CPU and memory activity in a very significant manner. In terms of memory power consumption, prefetch-insensitive and prefetch-friendly benchmarks do not experience any variation, consuming the same amount for both the default configuration and adaptive prefetching. Performance for config-sensitive benchmarks increases when the right prefetch setting is used. That extra performance implies accessing memory more intensively. Because of that, memory power consumption increases. It significantly does for milc (up to 15%) and more modestly for libquantum and soplex (up to 3%). In all the cases, the performance increase surpasses the increment in power consumption. For omnetpp, power consumption actually decreases under adaptive prefetching. Our mechanism effectively detects that disabling prefetching is the best setting for that benchmark. By doing so, useless bandwidth consumption is eradicated, reducing memory power consumption in turn. We also observe a memory power reduction for povray. Being a prefetch-unfriendly workload, disabling prefetch helps as well, reducing power consumption 3%. Another interesting case is xalancbmk. That benchmark presents two different phases. During the first one, prefetching—especially when stride-N is enabled—helps significantly. In the second one, disabling prefetch is slightly better in terms of performance. Doing so also reduces bandwidth consumption to some degree. That reduction translates into a 2% memory power decrease for adaptive prefetching compared to the default setting.

Overall, the significant speedups for single-threaded workloads, together with the fact that performance does not decrease when compared to the default configuration,

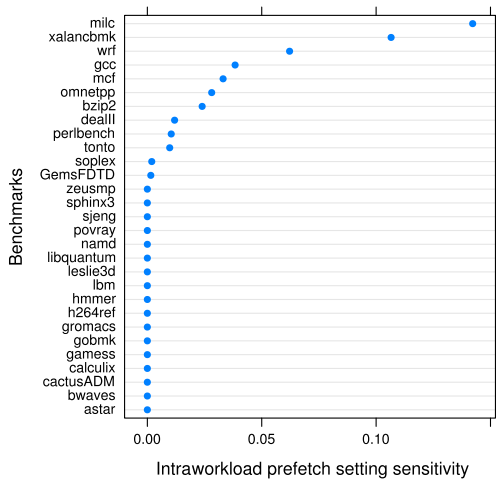

Fig. 12. Intraworkload prefetch setting sensitivity for all the SPEC CPU2006 benchmarks.

converts adaptive prefetching into a very useful mechanism to improve performance for memory intensive workloads. Additionally, memory power consumption is reduced for prefetch-unfriendly workloads such as omnetpp and povray, adding further value to our adaptive solution.

5.4.2. Composite Workloads. As shown in the previous sections, our adaptive scheme is able to find, without user intervention, the best prefetch setting for all SPEC CPU2006 benchmarks, with similar performance speedups to the best static approach. For an application that benefits from multiple "best" prefetch settings over its full execution period, however, the dynamic approach generally performs better. We use the term *intraworkload prefetch setting sensitivity* to refer to the degree of potential improvement that applications may have due to benefiting from multiple prefetch settings within their execution. In the previous sections we have pointed out that a single SPEC CPU2006 benchmark does not benefit from multiple prefetch settings, thus they have a low intraworkload prefetch setting sensitivity.

Figure 12 shows the sensitivity for all benchmarks. We compute the sensitivity as the ratio of time where a prefetch setting different from best static setting obtains a better performance compared to the best static one. Most benchmarks present a very low sensitivity (under 5%). And the only three benchmarks with relatively higher sensitivity only experience a slight increase in their performance during less than 15% of their execution. Therefore, we conclude that SPEC CPU2006 benchmarks do not present a high intraworkload prefetch setting sensitivity.

It is, however, conceptually easy to imagine the existence of applications that would benefit from different prefetch settings during their execution. For instance, a scientific application that retrieved a large amount of data from the Internet, uncompressed the data and finally processed it, would have three very different macro-phases. Moreover, each one of these phases may benefit from a different prefetch setting. In such a scenario, the best static approach could easily perform worse than a dynamic mechanism.

In order to demonstrate the potential benefits of an adaptive scheme compared to a static one, we construct some *composite workloads* by stitching together two SPEC CPU2006 benchmarks, one running after the other. Table III shows the speedup obtained by adaptive prefetching compared to the best static approach. As we can observe, there are significant performance improvements for workloads with a higher intraworkload prefetch setting sensitivity. As these results show, the adaptive prefetch

Table III.

Performance increase for adaptive prefetching compared to the static approach for composite workloads.

Workload	IPC speedup (%)
bwaves-omnetpp	9.1
mcf-omnetpp	8.9
milc-omnetpp	10.5
libquantum-omnetpp	7.7

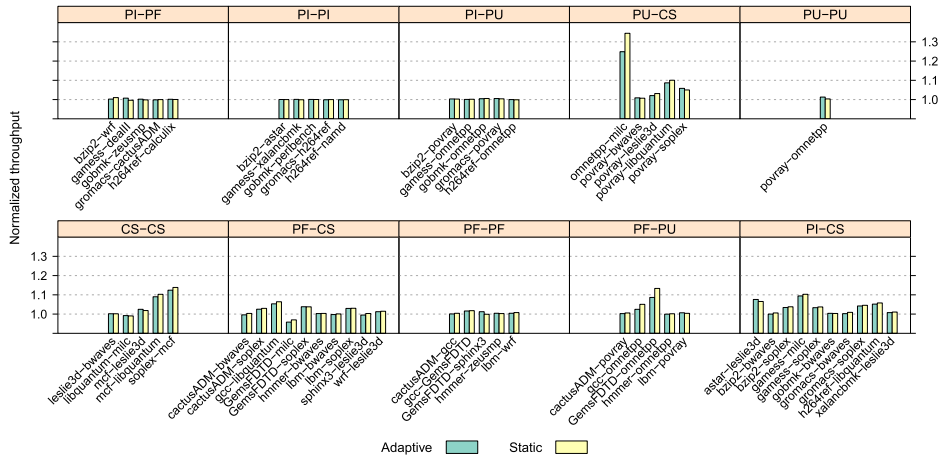

Fig. 13. Performance results for both the static and adaptive approaches for mixed-workloads. Each workload is composed of two different benchmarks from different classes (PI=prefetch-insensitive, PF=prefetch-friendly, PU=prefetch-unfriendly, CS=config-sensitive). Four copies of each benchmark are run at the same time. Results are normalized to the ones obtained with default prefetching.

mechanism is able to find the best prefetch setting for each of the macro-phases, thus increasing performance compared to a static approach.

5.4.3. Multiprogrammed Workloads. In this section we compare adaptive prefetching against the default configuration and the static approach for multiprogrammed workloads. Since, as we have seen in Section 5.4.1, the performance for Algorithm 3 is much better than the other two, in this section we only show results for the third algorithm. The results in Figure 13 are normalized to the case where all the benchmarks run with the default prefetch setting. We construct random pairs in such a way that all the benchmark types listed in Table II are represented. Each workload is composed of eight threads, four from a benchmark class and four from the other class. Each thread runs on a different core. We show results for five random workloads for each class combination except for PF-CS and CS-CS where we use ten random workloads since the result space and the performance variability are larger for these combinations. For PU-PU there is only one result, since there are only two benchmarks in PU class.

Looking at the results we observe that, as it was the case with single-threaded workloads, there is not too much difference in performance for workloads composed of prefetch-insensitive or prefetch-friendly benchmarks (PI-PI, PI-PF or PF-PF classes). For config-sensitive workloads, however, we observe very significant speedups (over 10%) for some pairs. Throughput goes up to 30% for the pair omnetpp-milc. In this case the adaptive mechanism disables prefetch for omnetpp and enables stride-N for milc, boosting the performance of both workloads. It is also important to notice that virtually in

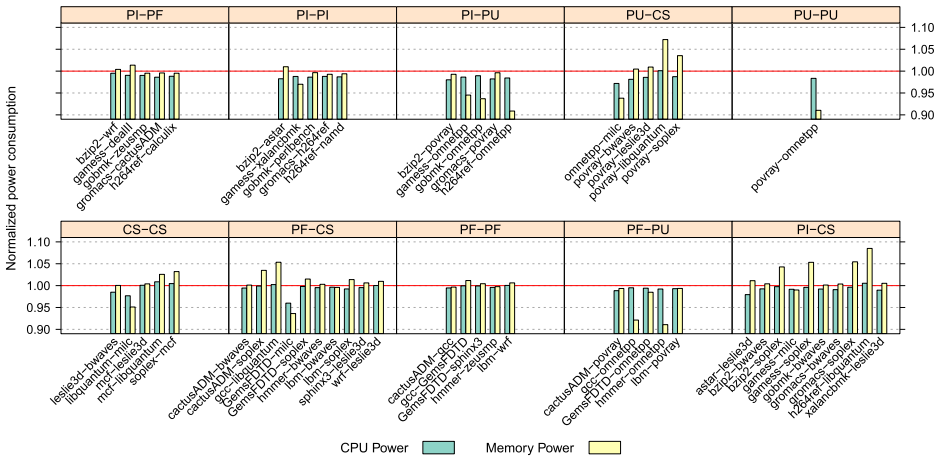

Fig. 14. CPU and memory power consumption results for the adaptive approach for mixed-workloads (same pairs as in Figure 13). Values are normalized to the ones obtained with the default prefetch configuration.

no case the performance achieved by the adaptive prefetch mechanism is lower than the baseline (using default prefetch for all the threads). The only two cases where this happens are for the pairs GemsFDTD-milc (in PF-CS class) and libquantum-milc (in CS-CS class). The reason for these results is the small absolute IPC for milc. When it runs together with other higher-IPC benchmarks, the total throughput may not increase that much—it may actually decrease—when using the adaptive approach. If we look at the individual IPC values, however, the results show that the adaptive mechanism actually improves performance. Let us examine the GemsFDTD-milc case in more detail. For that pair, adaptive prefetching worsens total throughput 4% compared to the baseline. When using the baseline, IPC values are 0.61 and 0.18 for GemsFDTD and milc, respectively. Adaptive prefetching selects different prefetch settings, and the IPC values change to 0.37 and 0.34 for the same benchmarks. These results show that GemsFDTD suffers a 35% slowdown, but the speedup for milc is almost 2X, easily compensating the slowdown for GemsFDTD. In addition to throughput, we have used other metrics such as the harmonic speedup in order to obtain performance measurements that combine both throughput and fairness between threads in each pair. Our results show that the adaptive mechanism always obtain a better performance compared to the baseline when using the harmonic speedup metric.

We observe that the static approach always obtains a performance equal or slightly higher than the adaptive one. As we pointed out in the previous section, virtually no SPEC CPU2006 benchmark benefits the most from more than a single prefetch setting. In such a case, the static approach always obtains the best possible performance. With our adaptive scheme, however, the user gets the benefit of autonomic performance boost across all workloads (compared to the default configuration), without the need to invest into a priori characterization of each and every workload.

Figure 14 shows CPU and memory power consumption for the same set of pairs that we used in Figure 13. As it was the case with single-threaded experiments, power consumption does not significantly vary for pairs where both benchmarks are either prefetch-insensitive or prefetch-friendly. Config-sensitive benchmarks, such as libquantum and soplex, experience significant speedups when the right prefetch setting is selected by our adaptive mechanism. That extra performance is delivered through an increase in memory bandwidth usage, and therefore, memory power consumption

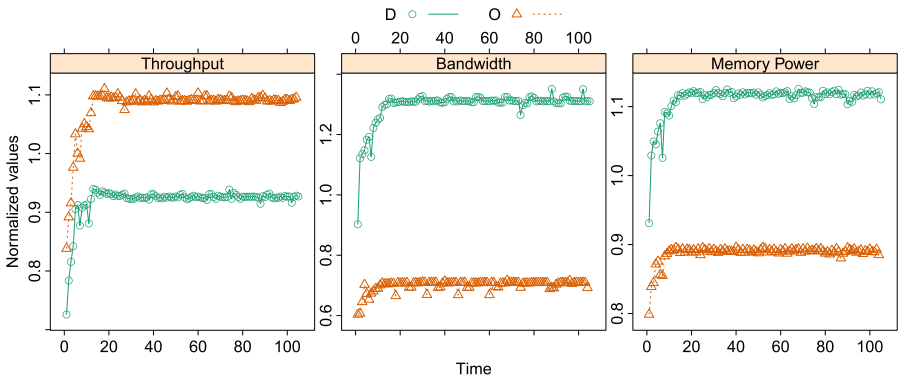

Fig. 15. Performance and power characterization for SPECjbb2005 along its execution for eight warehouses (i.e., threads). Individual thread values are first aggregated, and then they are normalized, dividing them by the mean of all the samples. In this way we keep the same ratio between both prefetch configurations as in the original values.

increases too. As Figure 14 shows, memory power consumption can increase up to 10% for these kinds of benchmarks. An interesting example is milc; this benchmark considerably benefits from enabling stride-N, resulting in a significant performance increment. As we can see in Figure 14, the pairs containing milc experience a power consumption reduction when they run under our adaptive mechanism. Adaptive prefetching enables stride-N most of the time for milc, effectively capturing that benchmark's access pattern, and increasing prefetching efficiency. When we use the default prefetch configuration, the prefetcher fails to capture the strided pattern, and the bandwidth consumption due to demand loads increases. Yet, (useless) prefetches are still generated, thus consuming memory bandwidth—and increasing power consumption in turn. Actually, we already observed this effect for seq-bench-stride microbenchmark (see Figure 3). Finally, we also observe how adaptive prefetching is able to reduce power consumption for prefetch-unfriendly benchmarks. Pairs where omnetpp appears, experience memory power consumption reductions close to 10%. These results demonstrate the potential of our adaptive prefetch scheme, not only at improving performance, but at reducing memory power consumption as well.

5.4.4. Java Business Workloads. So far we have evaluated our adaptive prefetch mechanism using SPEC CPU2006, a benchmark suite mainly composed of HPC simulation kernels and some integer workloads. Those are, however, just a fraction of the representative workloads running on real systems. Therefore, in addition to SPEC CPU2006, we have also evaluated our mechanism using SPECjbb2005 [SPEC 2005], a server-side, Java business application that models a three-tier client/server system. This type of application is commonly used nowadays in areas such as banking, wholesale suppliers or data warehousing.

In all the experiments, we run SPECjbb using eight warehouses; each warehouse is executed by a different thread. Thus, we have eight threads in total, mapping each one of them onto a different core. We have tried other numbers of warehouses, obtaining similar results. Typical SPECjbb executions consist of multiple steps where the number of warehouse is increased from 1 to the number of CPUs in the system. The reason to do that is to study how the system scales as more warehouses are executed. In our case, however, we are not studying the scaling capabilities, but just how different prefetch settings affect performance and power consumption for SPECjbb. Thus, we just execute the last step when all the cores are used.

Table IV.

Throughput and memory power consumption evaluation for SPECjbb2005. Results are normalized to the ones obtained with the default prefetch setting.

	Throughput	Power Consumption
Static	22.4%	−23.1%
Adaptive	21.1%	−21.9%

Figure 15 shows the results of executing SPECjbb with different prefetch configurations: default (D) and off (O). As we can observe in the figure, throughput increases 19% when prefetching is disabled compared to using the default prefetch configuration. SPECjbb is a prefetch-unfriendly benchmark, thus benefiting from disabling data prefetching in the same way omnetpp (from SPEC CPU2006) does. In the same figure we also notice that bandwidth consumption increases 56% when prefetching is enabled. Since SPECjbb is a prefetch-unfriendly benchmark, that extra bandwidth consumption is basically wasted due to inefficient prefetches. Even if the increased bandwidth consumption does not translate into extra performance—the opposite is actually true in this case—more frequent accesses to the memory subsystem incur into a significant memory power consumption overhead (22% increase).

In such a scenario, adaptive prefetching has the potential to both improve performance and reduce power consumption at the same time. That is a very much desired win-win situation. We have evaluated the impact of using our adaptive prefetch mechanism while running SPECjbb. Table IV contains the results, showing total throughput and memory power consumption. All the values are normalized with respect to the ones obtained when using the default prefetch configuration. The static approach, as expected, significantly increases performance by 22.4% and reduces power consumption by 23.1%. Our adaptive prefetch mechanism effectively detects that disabling prefetching is the optimal choice for this benchmark, and it obtains similar results: 21.1% performance speedup and 21.9% power reduction.

6. OS-BASED IMPLEMENTATION

The presented implementation of the adaptive prefetch is based on a user-level runtime. Compared to an OS implementation, a user-level runtime provides the maximum flexibility and portability. An OS-based implementation would provide several advantages, though. For instance, the overhead for reading performance counters as well as for changing the DSCR register would be reduced, since it would not be necessary to change the privilege mode to do so.

Therefore, besides evaluating the runtime-based mechanism, we studied the implementation of adaptive prefetch within the Linux OS. For that purpose, we have implemented OS-based adaptive prefetch algorithms similar to the runtime-based ones.

We rely on the *timer interrupt* in order to divide the execution of threads into intervals containing exploration and running phases. At each timer interrupt a reference to the thread running on the current context is first obtained (see Algorithm 4). Then the behavior of the algorithm depends on the current phase: i) If the exploration phase is active, the performance for the current prefetch setting (*curr_ps*) is recorded and the next setting is selected (lines 5–6). In case no more settings are available, the algorithm starts the running phase, after selecting the best setting found during the exploration phase (lines 8–11). ii) If the running phase is active, the running quantum is first reduced (line 14). That quantum determines how long a running phase will be. A larger value will reduce the effect of inefficient prefetch settings at the expense of a coarser adaptability.

Algorithm 4 OS-based implementation of Algorithm 1

```
1: ct = get_current_running_thread()
2: if mode = EXPLORATION then
3:    perf[ct, curr_ps[ct]] = read_ipc()
4:    if curr_ps[ct] ≠ last_ps() then
5:       curr_ps[ct] = next_ps(curr_ps[ct])
6:       set_dscr(ct, curr_ps[ct])
7:    else
8:       best_ps = arg max_ps(perf[ct])
9:       set_dscr(ct, best_ps)
10:      run_quantum[ct] = RUN_QUANTUM
11:      mode = RUNNING
12:   end if
13: else if mode = RUNNING then
14:   run_quantum[ct] = run_quantum[ct] − 1
15:   if run_quantum[ct] = 0 then
16:      curr_ps[ct] = first_ps()
17:      set_dscr(ct, curr_ps[ct])
18:      mode = EXPLORATION
19:   end if
20: end if
```

Using OS-based algorithms we have observed similar results to the ones obtained at user-level. These promising results encourage us to further pursue this path. We leave, however, the exploration of other OS-based adaptive schemes for future work.

7. CONCLUSIONS

Prefetch engines in current server-class microprocessor are getting more and more sophisticated. The IBM POWER7 processor contains a programmable hardware data prefetcher, allowing users to control different knobs in order to adapt the prefetcher to workload requirements. In this article we present an adaptive prefetch mechanism capable of boosting performance by leveraging these knobs. We evaluate its impact on performance for single-threaded and multiprogrammed workloads, showing that significant speedups can be obtained with respect to the default prefetch setting. We also show how our adaptive mechanism reduces power consumption for prefetch-unfriendly benchmarks. We compare the adaptive scheme to an approach where applications are first profiled and the best prefetch setting found is used for future executions. Our dynamic approach, however, frees users from profiling every application in order to find the best static prefetch setting.

Although we use POWER7-specific measurements and analysis in this article, the basic insights gleaned generally also apply to other (non-POWER) systems that use programmable hardware data prefetch engines.

REFERENCES

J. Abeles, L. Brochard, L. Capps, et al. 2010. Performance guide for HPC applications on IBM POWER 755 system. https://www.power.org/events/Power7/Performance_Guide_for_HPC_Applications_on_Power_755-Rel_1.0.1.pdf.

B. Abraham and J. Ledolter. 1983. *Statistical Methods for Forecasting*. Wiley.

J. L. Baer and T. F. Chen. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. ACM, 176–186.

C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C. Y. Cher, and M. Valero. 2008. Software-controlled priority characterization of POWER5 processor. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. IEEE, 415–426.

H. W. Cain and P. Nagpurkar. 2010. Runahead execution vs. conventional data prefetching in the IBM POWER6 microprocessor. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 203–212.

F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero, 2006. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Trans. Comput. 55*, 7, 785–799.

S. Choi and D. Yeung. 2006. Learning-based SMT processor resource distribution via hill-climbing. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. IEEE, 239–251.

A. C. de Melo. 2010. Performance counters for Linux. http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf.

P. J. Denning. 1968. The working set model for program behavior. *Commun. ACM 11*, 5, 323–333.

E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. 2011. Prefetch-aware shared resource management for multi-core systems. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ACM, 141–152.

E. Ebrahimi, O. Mutlu, and Y. N. Patt. 2009. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *Proceedings of the 15th International Symposium on High-Performance Computer Architecture*. 7–17.

P. G. Emma, A. Hartstein, T. R. Puzak, and V. Srinivasan. 2005. Exploring the limits of prefetching. *IBM J. Res. Dev. 49*, 1, 127–144.

M. Floyd, M. Allen-Ware, and K. Rajamani. 2011. Introducing the adaptive energy management features of the POWER7 chip. *IEEE Micro 31*, 2, 60–75.

J. L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comp. Arch. News 34*, 4, 1–17.

I. Hur and C. Lin. 2006. Memory prefetching using adaptive stream detection. In *Proceedings of the 39th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 397–408.

IBM. 2010. Power ISATM Version 2.06 Revision B. https://www.power.org/wp-content/uploads/2012/07/PowerISA_V2.06B_V2_PUBLIC.pdf.

C. Isci, A. Buyuktosunoglu, and M. Martonosi. 2005. Long-term workload phases: Duration predictions and applications to DVFS. *IEEE Micro 25*, 5, 39–51.

V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O'Connell. 2012. Making data prefetch smarter: Adaptive prefetching on POWER7. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. ACM, 137–146.

D. Joseph and D. Grunwald. 1997. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. ACM, 252–263.

N. P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. ACM, 364–373.

C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. 2008. Prefetch-aware DRAM controllers. In *Proceedings of the 41st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 200–209.

C. Lefurgy, X. Wang, and M. Ware. 2007. Server-level power control. In *Proceedings of the 4th International Conference on Autonomic Computing*. IEEE, 4–14.

S. W. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou 2009. Machine learning-based prefetch optimization for data center applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 1–10.

F. Liu and Y. Solihin. 2011. Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. ACM, 37–48.

P. Mochel. 2005. The sysfs filesystem. In *Proceedings of the Annual Linux Symposium*.

M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. 2009. FlexDCP: A QoS framework for CMP architectures. *SIGOPS Oper. Syst. Rev. 43*, 2, 86–96.

S. Palacharla and R. E. Kessler. 1994. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*. IEEE, 24–33.

M. K. Qureshi and Y. N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 423–432.

A. Roth, A. Moshovos, and G. S. Sohi. 1998. Dependence based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 115–126.

B. Sinharoy, R. Kalla, W. J. Starke, et al. 2011. IBM POWER7 multicore server processor. *IBM J. Res. Dev. 55*, 3, 1–29.

Y. Solihin, J. Lee, and J. Torrellas. 2002. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. ACM, 171–182.

SPEC. 2005. SPECjbb2005. http://www.spec.org/jbb2005/.

V. Srinivasan, E. S. Davidson, and G. S. Tyson. 2004. A prefetch taxonomy. *IEEE Trans. Comput. 53*, 2, 126–140.

C. J. Wu and M. Martonosi. 2011. Characterization and dynamic mitigation of intra-application cache interference. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2–11.

W. A. Wulf and S. A. McKee. 1995. Hitting the memory wall: Implications of the obvious. *SIGARCH Comp. Arch. News 23*, 20–24. 1.

C. L. Yang and A. R. Lebeck. 2000. Push vs. pull: Data movement for linked data structures. In *Proceedings of the 14th International Conference on Supercomputing*. ACM, 176–186.

Received February 2013; revised October 2013; accepted October 2013