Making Data Prefetch Smarter: Adaptive Prefetching on POWER7

Víctor Jiménez
Barcelona Supercomputing Center
Barcelona, Spain
victor.javier@bsc.es

Roberto Gioiosa*
Pacific Northwest National Laboratory
Richland, WA, USA
roberto.gioiosa@pnnl.gov

Francisco J. Cazorla
Spanish National Research Council and Barcelona Supercomputing Center
Barcelona, Spain
francisco.cazorla@bsc.es

Alper Buyuktosunoglu
IBM T. J. Watson Research Center
Yorktown Heights, NY, USA
alperb@us.ibm.com

Pradip Bose
IBM T. J. Watson Research Center
Yorktown Heights, NY, USA
pbose@us.ibm.com

Francis P. O'Connell
IBM Systems and Technology Group
Austin, TX, USA
oconnell@us.ibm.com

ABSTRACT

Hardware data prefetch engines are integral parts of many general purpose server-class microprocessors in the field today. Some prefetch engines allow the user to change some of their parameters. The prefetcher, however, 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. In this paper, we present an adaptive prefetch scheme that dynamically modifies the prefetch settings in order to adapt to the workload requirements. We implement and evaluate adaptive prefetching in the context of an existing, commercial processor, namely the IBM POWER7. Our adaptive prefetch mechanism improves performance with respect to the default prefetch setting up to 2.7X and 30% for single-threaded and multiprogrammed workloads, respectively.

Categories and Subject Descriptors

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

General Terms

Algorithms, Measurement, Performance

Keywords

Adaptive system, prefetching, performance

*This work was done while the author was at BSC.

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.

1. INTRODUCTION

Hardware data prefetch is a well-known technique to help alleviate the so-called *memory wall* problem [31]. 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 the user to change some of their parameters. In 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 of current machines. Nonetheless, commonly used prefetch algorithms, when applied in a fixed, non-adaptive mode will not help performance in many cases; and, in fact, they may actually degrade it due to useless bus bandwidth consumption and cache pollution. In this paper, we present an adaptive prefetch scheme that dynamically adapts the prefetcher configuration to the running workload, aiming to improve performance. We use the IBM POWER7 [27] 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 metrics through a user-visible performance counter interface.

POWER7 contains a programmable prefetch engine that is able to prefetch consecutive data blocks as well as those separated by a non-unit stride [27]. The processor system is provided to customers with a default prefetch setting that is targeted to improve performance for most applications. However, if needed, the user can manually override the default setting via the operating system. The user 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 that 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 paper, is that blindly setting a configuration may reduce performance.

Overall, the main contributions of the paper are:

- We first provide a motivation for adaptive prefetching, showing how the different prefetch configurations in POWER7 affect the performance for several workloads. To that end, we use the SPEC CPU2006 benchmark suite [14].
- 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 30% compared to the default prefetch configuration for single-threaded and multiprogrammed workloads, respectively.
- We also study the implementation of such an adaptive prefetch scheme within the OS kernel. By implementing our adaptive mechanism into the Linux kernel, we have observed similar performance improvements to those obtained by the userspace implementation.

This paper is organized as follows: Section 2 provides background for reading the paper. 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. Section 4 describes the methodology that we use in this paper. Section 5 shows the implementation of an adaptive prefetch mechanism and evaluates its impact on performance. Section 6 shows a possible OS-based implementation. Finally, Section 7 presents the conclusions of this paper.

2. RELATED WORK

There is a significant record of past research in data prefetch (e.g., [5, 18, 24]). 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 non-sequential data access patterns that do not benefit from sequential prefetching. That motivated the research on more complex prefetchers that try to capture the non-sequential nature of these applications. Prefetch techniques targeting pointer-based applications have been studied in [12, 26, 32]. Joseph and Grunwald study Markov-based prefetchers in [17]. Solihin et al. used a user-level memory thread in order to prefetch data, delivering significant speedups even for applications with irregular accesses [28]. Yet, most of these prefetch designs have not been implemented in a real processor. Limit studies and prefetch analytical models have been presented in [13, 29].

With the advent of CMP processors, interaction between threads is taken into account when designing a prefetch system. Ebrahimi et al. [11] and Lee et al. [19] study the effect of thread-interaction on prefetch and propose techniques to design prefetch systems that improve throughput and/or fairness. Liu and Solihin study the impact of hardware prefetching and bandwidth partitioning in CMPs [21].

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 characterize the prefetcher of an Intel Nehalem processor and provide a simple algorithm to dynamically control whether to turn the prefetcher on or off [30]. However, their study is solely oriented towards reducing intra-application cache interference without taking actual system performance. Liao et al. construct a machine learning model that dynamically modifies the prefetch configuration of the machines in a data center (based on Intel Core2 processors) [20]. They improve performance for some applications by enabling/disabling prefetch. In our case, however, the POWER7 processor not only allows us to enable/disable the prefetch, but it exposes a set of knobs to fine tune 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 are studied in [8, 9] to increase throughput and/or provide quality of service (QoS). Similarly, Boneti et al. [6] explore the usage of the dynamic hardware priorities present in IBM POWER5 processor for controlling thread resource balancing and prioritization. Qureshi and Patt [25] study the problem of last-level cache partitioning between multiple applications for improving throughput. Moreto et al. [23] present a similar technique but focus on achieving QoS for the co-running applications.

3. THE POWER7 PROCESSOR

The IBM POWER7 [27] 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 [2, 7, 15] is programmable and allows the user 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 stride larger than one cache block. The prefetcher is controlled via the data stream control register (DSCR). The Linux kernel exposes the register to the user through the `sys` virtual filesystem [22], allowing the user to set the prefetch setting on a per-thread basis.

Table 1a describes the possible prefetch configurations and introduces the notation that will be used throughout the paper. The 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*) [3], being automatically selected when the

Table 1: Notation used in this paper 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 *default* (D) or any value in the range 2-7 (*shallowest-deepest*). The special configuration where depth is 001 turns off the prefetcher (O). Table 1b shows some examples with this notation.

(a) 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

(b) Examples

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

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 prefetch can be independently configured. Table 1b shows some examples of the possible combinations that can be done by setting values for each prefetch knob. Additionally, it shows how the shortnames that we use along the paper are constructed.

3.2 Impact of Prefetch Settings

In the previous section we have presented the design of the prefetch engine in the POWER7, describing the different available settings. In this section we look at the effect of using settings different from the default one, motivating the need for an adaptive prefetch mechanism.

Figure 1 shows the performance of all the benchmarks in the SPEC CPU2006 suite running in single thread mode under several prefetch settings. We use the default prefetch configuration (D) as the baseline to normalize IPC. 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 (Table 2 contains the exact classification for all the SPEC CPU2006 benchmarks): 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. However, 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), and iv) *prefetch-unfriendly (PU)*; for this type of benchmark, enabling prefetch negatively affects its performance (e.g., omnetpp).

Table 2: Benchmark classification based on how their performance is affected by prefetch when running in single-thread mode.

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

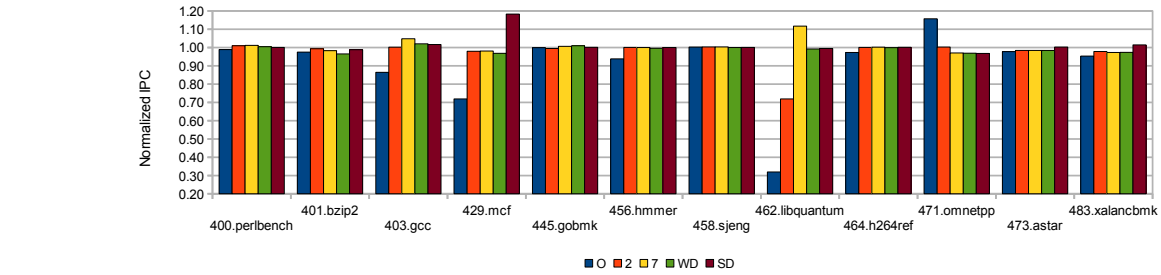
Overall, as results in Figure 1 show, an adaptive prefetch mechanism could tune the prefetch for every particular benchmark in order to find the prefetch configuration that leads to its optimal performance.

4. METHODOLOGY

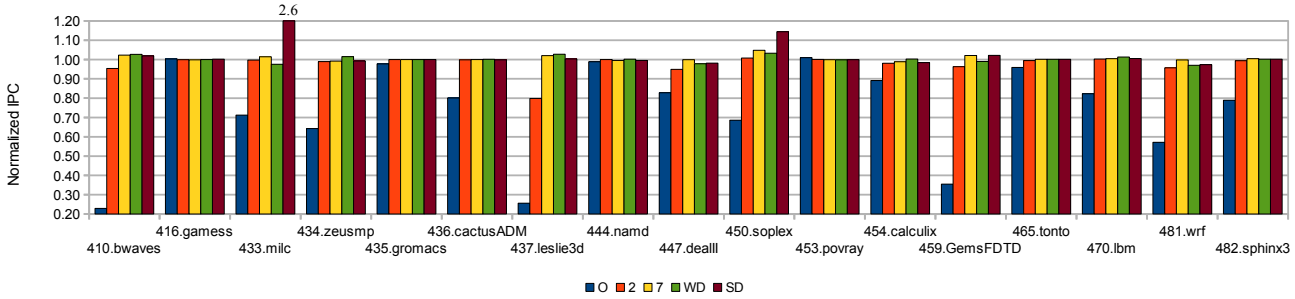
We use an IBM BladeCenter PS701 to conduct all the experiments, including the ones in the previous section. This system contains one POWER7 processor running at 3.0 GHz and 32 GB of DDR3 SDRAM running at 800 MHz. The operating system is SUSE Linux Enterprise Server 11 SP1. We use the IBM XL C/C++ 11.1 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. For collecting information from the performance counters we use *perf*, the official implementation in the mainstream Linux kernel [1].

5. ADAPTIVE PREFETCHING

In Section 3.2 we have seen that different applications derive maximum performance benefit from different prefetch settings. In that approach, a user needs 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.



(a) SPECINT



(b) SPECFP

Figure 1: 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 1 for notation on prefetch configurations.

5.1 Basic Adaptive Algorithm

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
```

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 [16]. 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 paper. 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.

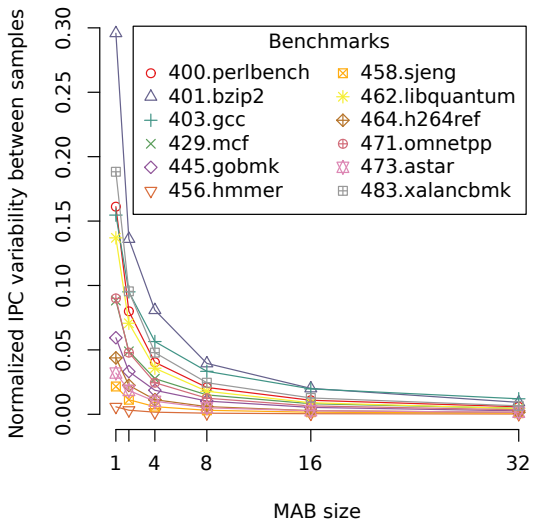
5.2 Impact of Phase Changes

It is well-known that applications present phases during their execution [10]. 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.

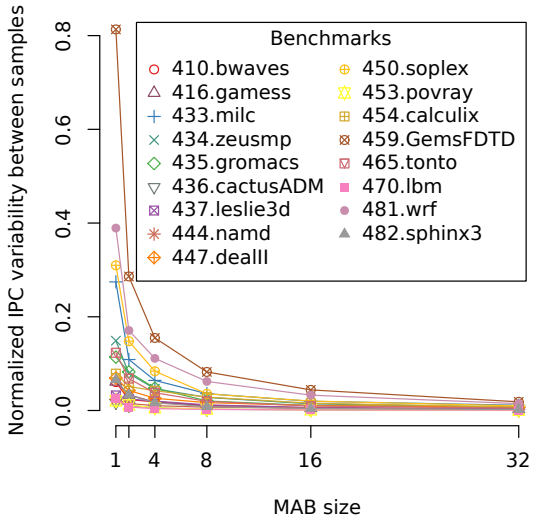
In order to alleviate this problem we use a moving average buffer (MAB) [4] that keeps the last m IPC samples for every prefetch setting and process under control of the adaptive prefetch runtime. We then compare the performance of prefetch settings by using the mean of the values in the buffer, instead of using individual measurements. We evaluate the effect of using buffers of different sizes on the IPC variability between consecutive samples. Figure 2 shows the normalized IPC variability as we increase the buffer size from 1 (i.e., no buffer is used) up to 32 positions. IPC variability is computed with the following equation:

$$variability = \frac{1}{n-1}\sum_{i=1}^{n-1}|IPC_{i+1} - IPC_i| \qquad (1)$$

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 2a contains the results for SPEC INT benchmarks and Figure 2b does so for SPEC FP benchmarks. As it can be seen in the figure, most 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



(a) SPEC INT



(b) SPEC FP

Figure 2: Effect of changing the buffer size on inter-sample IPC variability. IPC variability (see Equation 1) is normalized to the average IPC for each benchmark.

the impact that phase changes may have on the exploration phase of the adaptive prefetch mechanism.

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
```

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.

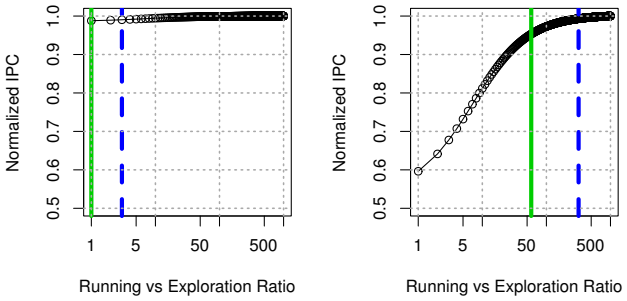
5.3 Impact of "Inefficient" Prefetch Settings

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 the different settings used in the exploration phase. For instance, for bwaves, disabling the prefetch reduces its performance by 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:

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

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.



(a) perlbench (b) milc

Figure 3: Effect of exploration/running ratio on expected performance. Values are normalized to the maximum values observed for each workload (i.e., $max_i(IPC_i)$).

Figure 3 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 3a 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 with setting 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 3b 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 dropping*. 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:

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

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 included 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. We believe, however, that it is possible to select a default value for that parameter, based on mathematical performance modeling and empirical analysis. 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:

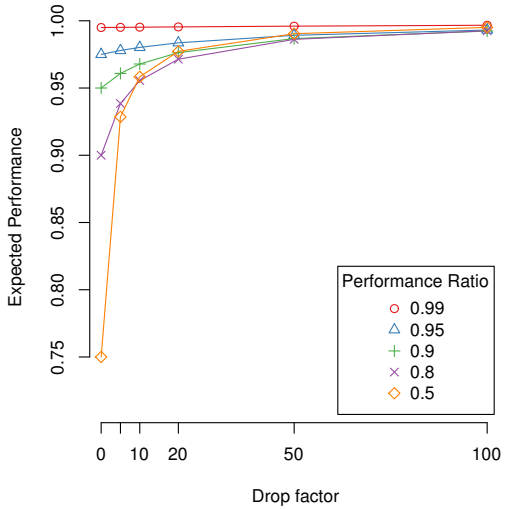


Figure 4: Effect of drop factor on expected performance. Values are normalized to the best possible performance.

$$\widehat{IPC_i} = \frac{t_1}{T} \times IPC_{best} + \frac{t_2}{T} \times \alpha IPC_{best} \quad (4)$$

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 4 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 4, 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 becomes much more noticeable. For instance, if the performance for the second setting is 50% compared to the best setting, not using the dropping 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 dropping feature. As it can be seen there is no running phase in this algorithm. The running phase is not necessary anymore since the dropping 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

Algorithm 3 Adaptive prefetch with MAB and inefficient setting dropping.

```
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
```

(lines 3-4). In lines 13-16 the algorithm computes the number of iterations 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 5 shows the results for single-threaded workloads. We present results for all the SPEC CPU2006 benchmarks. The performance values are normalized to the ones obtained with the default prefetch configuration. As 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 inter-sample variability (see Figure 2). 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 good 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 5.4.3). 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 increased when the right setting is selected for them, omnetpp behaves in a completely different way, and it actually benefits from disabling the 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 prefetch, even if stride-N is enabled. In fact, prior research already showed that omnetpp does not benefit from prefetch [12, 19]. 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.

Overall, these significant speedups, together with the fact that performance does not decrease when compared to the default configuration, converts adaptive prefetching in a very useful mechanism to improve performance for memory intensive workloads.

5.4.2 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 the previous section, 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 6 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 2 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 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%

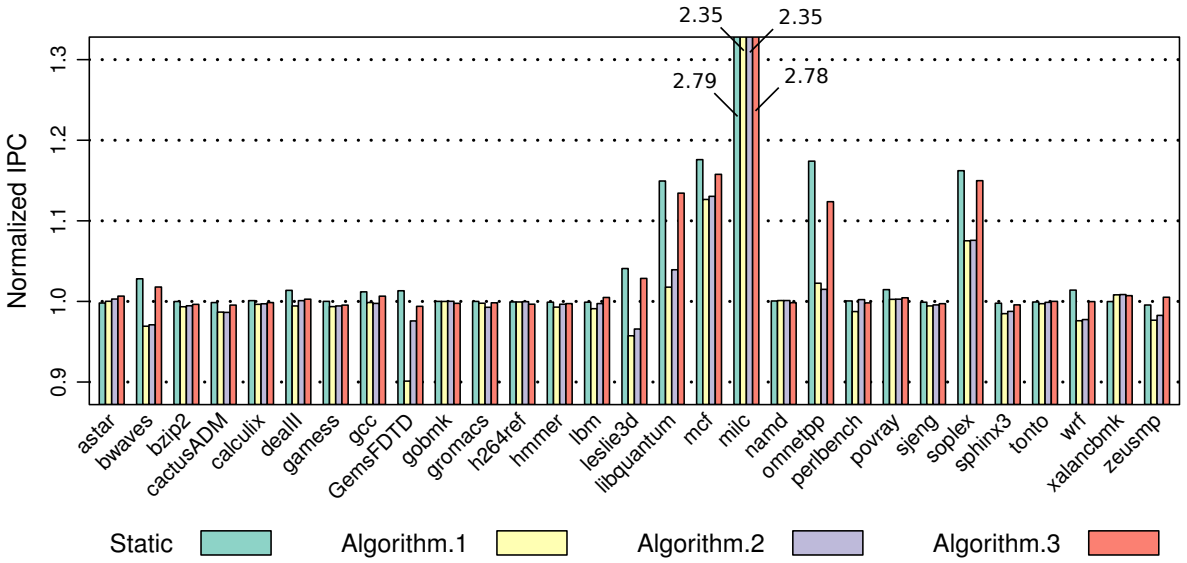


Figure 5: Results for single-threaded workloads normalized to the ones obtained with the default prefetch configuration.

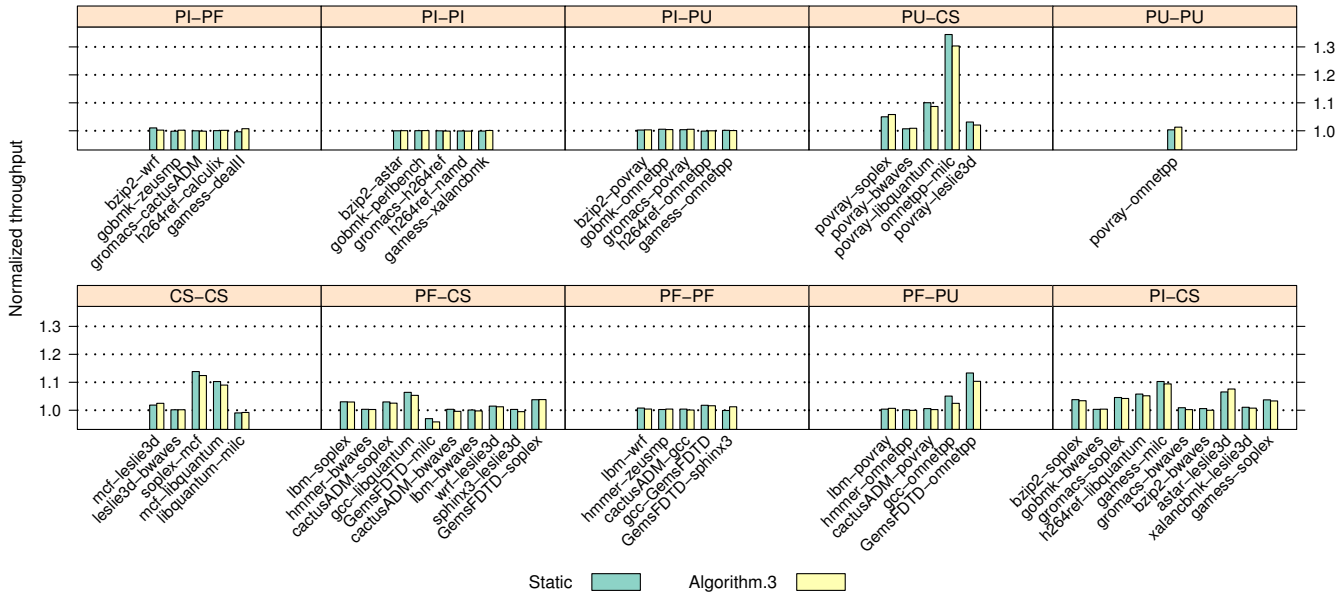


Figure 6: Performance results for both the static and adaptive approaches for mixed-workloads. Each workload is composed of two different benchmarks from different classes (PI=prefetch-insensitive, PF=prefetch-friendly, PU=prefetch-unfriendly, CS=config-sensitive). Four copies of each benchmark are run at the same time. Results are normalized to the ones obtained with default prefetching.

compared to the baseline. When using the baseline the 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.

5.4.3 *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 *intra-workload 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 intra-workload prefetch setting sensitivity.

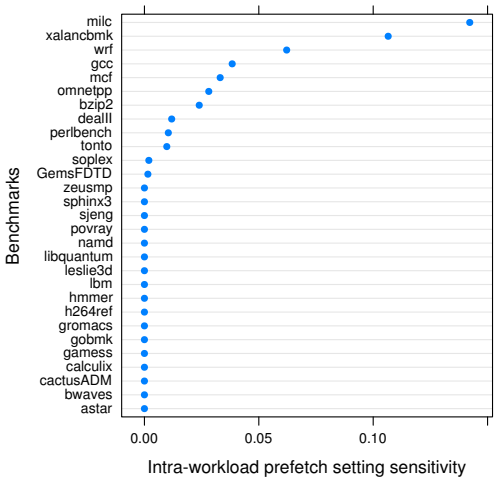


Figure 7: Intra-workload prefetch setting sensitivity for all the SPEC CPU2006 benchmarks.

Figure 7 shows the sensitivity for all the 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. The figure shows that most benchmarks present a very low sensitivity (under 5%). The only three benchmarks with relatively higher sensitivity, however, 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 intra-workload 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 definitely perform worse than a dynamic mechanism.

Table 3: 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

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 3 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 intra-workload prefetch setting sensitivity. As these results show, the adaptive prefetch mechanism is able to find the best prefetch setting for each of the macro-phases, thus increasing performance compared to a static approach.

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 actually implemented OS-based adaptive prefetch algorithms similar to the runtime-based ones.

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
```

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.

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 as 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 the user to control different knobs in order to adapt the prefetcher to workload requirements. In this paper we present an adaptive prefetch mechanism capable of boosting performance by leveraging on 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 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 paper, the basic insights gleaned generally also apply to other (non-POWER) systems that use programmable hardware data prefetch engines.

8. ACKNOWLEDGMENTS

This work was supported by a Collaboration Agreement between IBM and BSC. It has also been supported by the Ministry of Science and Technology of Spain under contract TIN-2007-60625 and grant JCI-2008-3688. We also acknowledge Ramon Bertran and Lluís Vilanova for developing data processing tools that significantly simplified handling all the experimental data necessary for this paper.

9. REFERENCES

[1] Performance Counters for Linux. `https://perf.wiki.kernel.org`.

[2] Power ISA^{TM} Version 2.06 Revision B. `https://www.power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf`.

[3] J. Abeles et al. 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`.

[4] B. Abraham and J. Ledolter. *Statistical Methods for Forecasting*. Wiley series in probability and mathematical statistics: Applied probability and statistics. Wiley, 1983.

[5] J. L. Baer and T. F. Chen. An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty. In *Proc. ACM/IEEE Conf. Supercomputing*, SC, pages 176–186, 1991.

[6] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C. Y. Cher, and M. Valero. Software-Controlled Priority Characterization of POWER5 Processor. In *Proc. 35th Int'l Symp. Comp. Arch.*, ISCA, pages 415–426, 2008.

[7] H. W. Cain and P. Nagpurkar. Runahead Execution vs. Conventional Data Prefetching in the IBM POWER6 Microprocessor. In *Proc. Int'l Symp. Perf. Analysis of Systems Software*, ISPASS, pages 203–212, 2010.

[8] F. J. Cazorla et al. Predictable Performance in SMT Processors: Synergy between the OS and SMTs. *IEEE Trans. Comput.*, 55(7):785–799, July 2006.

[9] S. Choi and D. Yeung. Learning-Based SMT Processor Resource Distribution via Hill-Climbing. In *Proc. 33rd Int'l Symp. Comp. Arch.*, ISCA, pages 239–251, 2006.

[10] P. J. Denning. The Working Set Model for Program Behavior. *Commun. ACM*, 11(5):323–333, May 1968.

[11] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Prefetch-Aware Shared Resource Management for Multi-Core Systems. In *Proc. 38th Int'l Symp. Comp. Arch.*, ISCA, pages 141–152, 2011.

[12] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. In *Proc. 15th Int'l Symp. High Perf. Comp. Arch.*, HPCA, pages 7–17, 2009.

[13] P. G. Emma, A. Hartstein, T. R. Puzak, and V. Srinivasan. Exploring the limits of prefetching. *IBM J. R&D*, 49(1):127–144, January 2005.

[14] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comp. Arch. News*, 34(4):1–17, September 2006.

[15] I. Hur and C. Lin. Memory Prefetching Using Adaptive Stream Detection. In *Proc. 39th Int'l Symp. on Microarchitecture*, MICRO, pages 397–408, 2006.

[16] C. Isci, A. Buyuktosunoglu, and M. Martonosi. Long-Term Workload Phases: Duration Predictions and Applications to DVFS. *IEEE Micro*, 25(5):39–51, September 2005.

[17] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. In *Proc. 24th Int'l Symp. Comp. Arch.*, ISCA, pages 252–263, 1997.

[18] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proc. 17th Int'l Symp. Comp. Arch.*, ISCA, pages 364–373, 1990.

[19] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-Aware DRAM Controllers. In *Proc. 41st Int'l Symp. Microarch.*, MICRO, pages 200–209, 2008.

[20] S. W. Liao et al. Machine Learning-Based Prefetch Optimization for Data Center Applications. In *Proc. Int'l Conf. High Perf. Comp. Networking, Storage and Analysis*, SC, pages 1–10, 2009.

[21] F. Liu and Y. Solihin. Studying the Impact of Hardware Prefetching and Bandwidth Partitioning in Chip-Multiprocessors. In *Proc. Int'l Conf. Measur. and Model. of Comp. Sys.*, SIGMETRICS, pages 37–48, 2011.

[22] P. Mochel. The sysfs Filesystem. *Proc. Annual Linux Symp.*, 2005.

[23] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. FlexDCP: a QoS Framework for CMP Architectures. *SIGOPS Oper. Syst. Rev.*, 43(2):86–96, April 2009.

[24] S. Palacharla and R. E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proc. 21st Int'l Symp. Comp. Arch.*, ISCA, pages 24–33, 1994.

[25] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proc. 39th Int'l Symp. Microarch.*, MICRO, pages 423–432, 2006.

[26] A. Roth, A. Moshovos, and G. S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. 8th Int'l Conf. Arch. Support for Prog. Lang. and Operat. Sys.*, ASPLOS, pages 115–126, 1998.

[27] B. Sinharoy et al. IBM POWER7 multicore server processor. *IBM J. R&D*, 55(3):1–29, May-June 2011.

[28] Y. Solihin, J. Lee, and J. Torrellas. Using a User-Level Memory Thread for Correlation Prefetching. In *Proc. 29th Int'l Symp. Comp. Arch.*, ISCA, pages 171–182, 2002.

[29] V. Srinivasan et al. A prefetch taxonomy. *IEEE Trans. Comp.*, 53(2):126–140, February 2004.

[30] C. J. Wu and M. Martonosi. Characterization and Dynamic Mitigation of Intra-Application Cache Interference. In *Proc. Int'l Symp. Perf. Analysis of Systems and Software*, ISPASS, pages 2–11, 2011.

[31] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comp. Arch. News*, 23:20–24, March 1995.

[32] C. L. Yang and A. R. Lebeck. Push vs. Pull: Data Movement for Linked Data Structures. In *Proc. 14th Int'l Conf. Supercomputing*, ICS, pages 176–186, 2000.