Increasing Multicore System Efficiency through Intelligent Bandwidth Shifting

Victor Jimenez
Alper Buyuktosunoglu
Pradip Bose
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
{victorjj,alperb,pbose}@us.ibm.com

Francis P. O'Connell*
Apple Inc.
Lone Star Design Center
Austin, TX 78746
francis_oconnell@apple.com

Francisco Cazorla†
Mateo Valero
Barcelona Supercomputing Center
Barcelona, Spain
{francisco.cazorla,mateo.valero}@bsc.es

***Abstract*—Memory bandwidth is a crucial resource in computing systems. Current CMP/SMT processors have a significant number of cores and they can run many threads concurrently. This large thread count adds high pressure to the memory bus, which demands high bandwidth to service memory requests from the cores. Hardware data prefetching is a well-known technique for hiding memory latency. Due to its speculative nature, however, in some situations prefetching does not effectively work, wasting memory bandwidth and polluting the caches. Data prefetching efficiency depends on the prefetching algorithm. It also depends on the characteristics of the applications running on the system. In this paper we propose an online bandwidth shifting mechanism that dynamically assigns bandwidth to applications according to their prefetch efficiency. This mechanism maximizes the utilization of memory bandwidth, thereby improving system performance and/or reducing memory power consumption. To the best of our knowledge, this solution is the first to not require hardware support. We evaluate the benefits of using our bandwidth shifting mechanism on a real system—the IBM POWER7. We obtain speedups in the order of 10-20% (in one instance, speedup exceeds 1.6X). Our mechanism does not generate a significant degree of unfairness among the applications. In many cases individual thread performance increases by 10-35%, while virtually no thread experiences a slowdown larger than 5%.**

I. Introduction

Current CMP/SMT processors contain a significant number of cores (e.g., IBM POWER7 has 8 cores), and given the current trends future processors will have a larger core count. These processors can execute many threads concurrently, which can stress the bandwidth between processor and memory [31].

Data prefetching is a well-known technique for hiding memory latency. The technique relies on the fact that many applications exhibit spatial locality (i.e., once a given memory address is accessed, it is very likely that surrounding addresses will be accessed in the near future). Upon a data cache miss for a given address, the prefetcher may speculatively bring consecutive blocks corresponding to the addresses that the application is likely to access in the future. More complex prefetch implementations may detect access patterns to non-consecutive data (e.g., pointer-based list traversal).

While it helps performance, data prefetching may not be accurate because of its speculative nature, bringing useless data from the memory into the processor's caches. When this happens, memory bandwidth is wasted and cache pollution may occur. This problem is exacerbated when a processor supports the simultaneous execution of multiple threads. Bandwidth can easily be saturated, even without the presence of data prefetching. Enabling data prefetch in such a case may degrade system performance, since prefetches will fight with demand loads for the scarce available bandwidth.

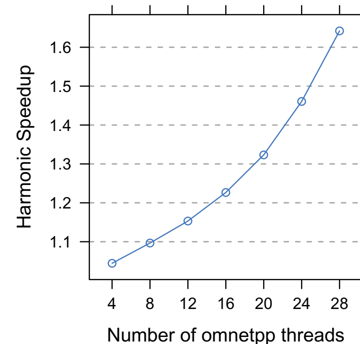


Fig. 1. Effect of bandwidth shifting on system performance when a prefetch-efficient benchmark (bwaves) and a prefetch-inefficient one (omnetpp) run together. The X axis shows the number of omnetpp threads (x). The number of bwaves threads is $32 - x$.

Although data prefetching efficiency depends on the algorithm used by the prefetcher, ultimately it depends on the characteristics of the applications running on the system. Some applications present memory access patterns that are amenable to prefetching while others contain nearly random access patterns for which no prefetcher can be completely accurate. Because the impact of data prefetching depends on the nature of the applications running on the system, it is necessary to intelligently assign prefetching resources to the applications—especially under a constrained bandwidth scenario. This approach will attempt to maximize the utilization of memory bandwidth, potentially improving system performance and/or reducing power consumption (e.g., by turning off the prefetcher for applications that are not amenable to prefetching).

To the best of our knowledge, this paper is the first solution that addresses prefetch bandwidth management for CMP processors *without requiring hardware support*. Because of its design, our solution should work on any multicore system with a programmable prefetch engine—most modern processors allow users to control prefetching in different ways.

*This work was done while Francis P. O'Connell was at IBM. †Francisco Cazorla is also with the Spanish National Research Council (IIIA-CSIC).

978-1-4799-8930-0/15/$31.00 ©2015 IEEE

Figure 1 shows an illustrative example of the effect of bandwidth shifting on system performance. In this example we run two benchmarks—bwaves (prefetch friendly) and omnetpp (prefetch unfriendly). For every execution (represented as a tick in the X axis) we run 32 processes in total: x omnetpp copies and $32 - x$ bwaves copies. We compute the system speedup using the harmonic speedup between two configurations: 1) both benchmarks using the most aggressive prefetch setting, and 2) bwaves keeps using the most aggressive setting, but prefetching is disabled for omnetpp. Our bandwidth shifting mechanism would effectively shift prefetch resources from the prefetch-friendly to the prefetch-unfriendly benchmark. As the number of omnetpp copies increases, the benchmark keeps adding pressure to the available memory bandwidth thus taking bandwidth away from bwaves. If we shift bandwidth between both benchmarks by disabling prefetching for omnetpp, we observe very significant speedups. This is especially noticeable as the number of omnetpp copies increases, since prefetches issued for that benchmark saturate the bandwidth to memory. When we intelligently shift bandwidth between the applications, for 28 omnetpp threads the system speedup exceeds 60%. As Figure 1 demonstrates, there is ample room for an intelligent bandwidth shifting mechanism that takes bandwidth resources away from prefetch-inefficient workloads, and gives those resources to more efficient workloads.

Overall, the main contributions of this paper are:

- We provide a motivation for prefetch-based bandwidth shifting and a characterization of the performance-bandwidth trade-off for multiple benchmarks.
- We introduce a metric that estimates prefetch usefulness for a given thread based solely on performance counters commonly available in current processors.
- We present a novel bandwidth shifting mechanism capable of significantly improving system performance by taking bandwidth away from benchmarks that do not use prefetching in an efficient way and giving it to prefetch-efficient benchmarks. The mechanism does not require any hardware support, and it is able to obtain up to 18.5% speedup (10-11% on average).
- We study the impact of bandwidth shifting in extreme cases where one benchmark is highly prefetch-efficient and the other uses prefetching inefficiently. Our results show that bandwidth shifting achieves much larger speedups (>1.6X).
- We also evaluate the impact of the bandwidth shifting mechanism on power consumption.

This paper is organized as follows: Section II describes the POWER7 processor, and presents a characterization of the performance-bandwidth impact of prefetching on a diverse set of benchmarks. Section III describes the methodology that we use in this paper. Section IV shows the implementation of the bandwidth shifting mechanism. Section V evaluates the impact of bandwidth shifting on performance and power consumption. Section VI provides the related work for this paper. Finally, Section VII presents the conclusions of this paper.

II. The IBM POWER7 Processor

The POWER7 [33] 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) modes.

Implemented within the load-store unit (LSU), the data prefetching unit (DPU) [2], [15], [16] contains twelve prefetch request queues (PRQs) plus associated logic that are capable of detecting and prefetching load, store, and load-to-store streams. As in previous POWER implementations, the DPU is able to detect sequential storage reference patterns, but is augmented with a "stride-N" logical subunit. The stride-N subunit detects streams which have regular access patterns, but do not fetch from consecutive cache lines in memory. The DPU can detect strides up to 8KB in length, with a 32B granularity. The detection is handled in a four entry buffer which examines the stride between the address of the current cache line miss and those from the previous four cache line misses. When a pattern is detected, a data stream is created in a PRQ. From this point forward, the data stream is treated just like any other data stream, with the distinction that subsequent prefetch requests may fetch from non-consecutive cache lines. The detection hardware is unique compared to traditional stride-N approaches in that the pattern can be detected across multiple load instructions in the code sequence. This is required for proper detection of unrolled loops and complex conditional load structures.

The POWER7 DPU uses two types of prefetches to optimize the retrieval of data via prefetching. L3-prefetches prefetch data from memory (or other caches) into the L3 cache and L1-prefetches prefetch data into the L1 data cache. The core generates both types of prefetches to optimally cascade data from high latency DRAM into the L3 and from the L3 into the L1 data cache. POWER7 DPU is programmable and allows users to set different parameters (knobs) that control its behavior and determine the aggressiveness of the prefetch engine (i.e., how many lines ahead to prefetch). The L1 and L3 prefetchers cannot be independently controlled. Prefetch settings are controlled via the data stream control register (DSCR). The Linux kernel exposes the register to users through the sys virtual filesystem [25], allowing them to set the prefetch setting on a per-thread basis. In this paper we utilize two different settings which we refer to as ON—prefetch is configured in its most aggressive mode—and OFF—no data prefetches are issued at all (neither L3 nor L1 prefetches). We explored the inclusion of intermediate settings with moderate aggressiveness but we did not observe any benefits.

A. Effect of Prefetching on Performance and Bandwidth

In order to study the potential for a bandwidth shifting mechanism and to better understand the inherent trade-offs to such a mechanism, in this section we look at the effect of prefetching on performance and bandwidth for different mix of benchmarks running concurrently on a system.

Figure 2 shows the throughput and memory bandwidth consumption for a subset of the SPEC CPU2006 benchmarks (the rest of the benchmarks in the suite have a similar behavior to one of the benchmarks shown in the figure).

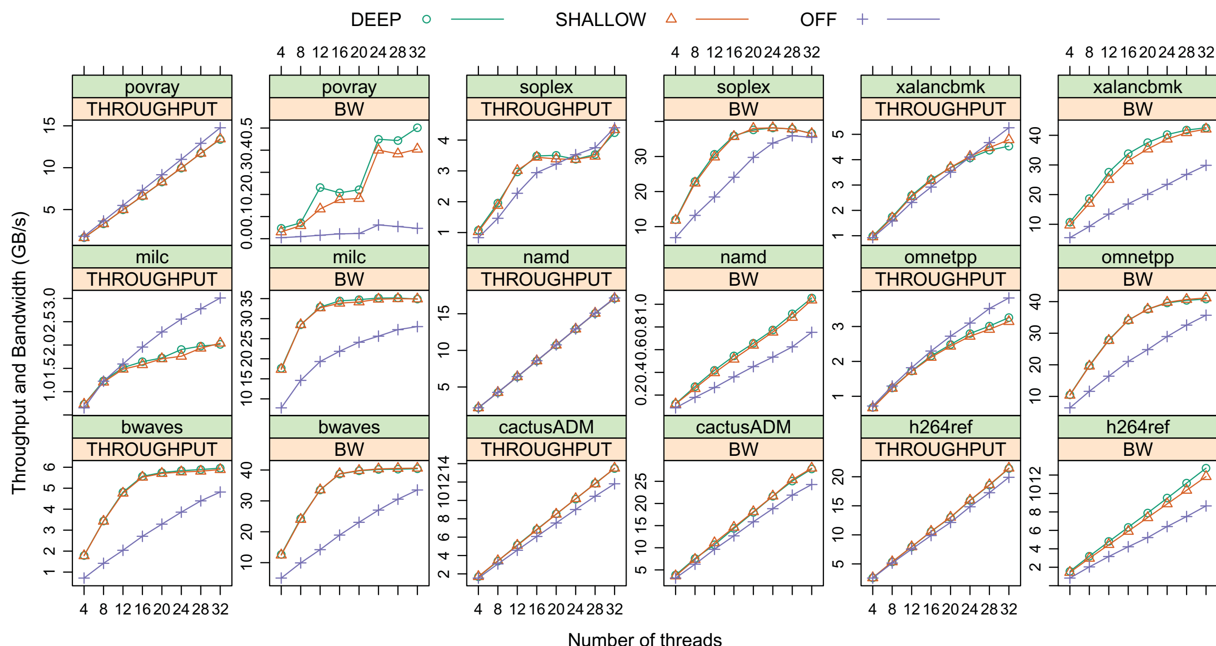


Fig. 2. Throughput and memory bandwidth consumption characterization for a subset of the benchmarks. This subset is representative of all the benchmarks used in this paper (i.e., the curves for the benchmarks not shown here match one of the benchmarks shown in the figure).

The results show throughput and bandwidth values for an increasing thread count, from 4 threads (using only one core) to 32 threads (using all 8 cores). Results are shown for three different prefetch configurations: DEEP, SHALLOW and OFF. In the first two settings prefetching is enabled, but with various aggressiveness configurations. For DEEP the prefetcher uses the longest prefetch distance available, while for SHALLOW it uses the shortest one. Setting OFF simply turns off the prefetcher.

As the figure shows, bandwidth and performance of certain benchmarks saturate and level off when we use more than 16 threads. In this study we use a low-end POWER7 system, where the maximum available bandwidth is 40GB/s per socket and the DRAM clock frequency is 800MHz. These results might differ for higher-end systems, which have more than double this memory bandwidth per socket. It should be noted that the goal of this paper is to evaluate the benefits of the memory bandwidth shifting idea, not to measure the effectiveness of the existing data prefetch mechanisms in this particular IBM POWER7 machine.

Some benchmarks use prefetching in a very efficient way. High performance computing (HPC) applications such as bwaves and cactusADM are good representatives of prefetch-efficient workloads. Their speedup when prefetching is enabled is linearly proportional to the extra bandwidth consumption. Prefetching is critical for these applications to obtain high performance. Other benchmarks such as h264ref benefit from prefetching, but the performance benefit they obtain does not compensate the extra bandwidth utilized. Benchmarks like this one do not utilize prefetching as efficiently as benchmarks such as bwaves do.

Benchmarks such as xalancbmk, milc and soplex are high bandwidth consumers—all of them reaching 40 GB/s (the bandwidth limit in our system) when the thread count is high. When the thread count is relatively small, prefetching increases performance since memory bandwidth is not saturated. As the number of threads running on the system increases, bandwidth to memory becomes saturated, and at some point, prefetching stops being useful. After a certain thread count (which depends on the benchmark) prefetching may degrade performance.

Other benchmarks such as omnetpp and povray simply do not benefit from prefetching for any thread count. Even if the total bandwidth consumption of povray is very low, Figure 2 shows that the bandwidth consumption significantly increases (in relative terms) when we enable prefetching. Performance, on the contrast, decreases when prefetching is turned on. omnetpp has a similar behavior, but this benchmark consumes a large amount of bandwidth. It saturates the available bandwidth with useless prefetches that do not benefit itself, and degrade the performance of other workloads running on the system. Another benchmark with a similar behavior is namd. When prefetching is turned off, however, its performance increase is barely noticeable.

Our findings show that the efficiency of prefetching on applications significantly varies depending on the access patterns of these applications. Bandwidth saturation is another important parameter that determines the effectiveness of prefetching—an application that benefits from prefetching when there is plenty of bandwidth available might be negatively affected by prefetching when bandwidth is scarce. All these observations suggest that a dynamic—online—mechanism, such as the one we are presenting in this paper, is required in order to use bandwidth in a more efficient way.

Because we do not observe significant differences when varying the prefetch aggressiveness, in the design of the bandwidth shifting mechanism we only consider two settings:

ON and OFF. [1] We use DEEP as the configuration when prefetch is enabled (ON).

III. 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 64 GB of DDR3 SDRAM running at 800 MHz. The maximum bandwidth achievable by this system is approximately 40 GB/s. 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. Although compiler-generated prefetches [3], [26] may improve the performance of some applications, because of their static nature, they are not a suitable instrument for dynamically adapting to the mix of applications running on a system. Our solution is actually orthogonal to software prefetching. The interaction between hardware and software prefetching has already been studied in the past [2]. We also use Graph500 [27] for evaluating our bandwidth shifting mechanism. Graph500 is a representative example of a new class of server applications: analytics. We run all benchmarks until completion. Each benchmark may run for a different amount of time. Because of this, we restart benchmarks that finish early until all benchmarks have fully completed their execution at least once [38]. For collecting information from the performance counters we use *perf*, the official implementation in the mainstream Linux kernel [8]. The default page size in Linux for POWER is 64 kilobytes. This helps prefetching since it is not necessary to restart the streams after crossing the boundary of relatively small 4 kilobytes pages.

Power measurements are obtained using the IBM Automated Measurement of Systems for Temperature and Energy Reporting software [13], [21]. 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.

IV. Intelligent Bandwidth Shifting

Our intelligent bandwidth shifting mechanism dynamically takes prefetch resources away from prefetch-inefficient threads and gives those resources to more efficient threads, effectively shifting bandwidth between the threads running on the system. Giving that extra bandwidth to the threads that use prefetching efficiently leads to system (global) speedups.

In order to decide which threads use prefetching efficiently, we must first define a metric to estimate the prefetch usefulness (PU) level for a given thread. We define that metric as:

$$PU = \frac{IPC_{on}/BW_{on}}{IPC_{off}/BW_{off}} \quad (1)$$

[1]Using SHALLOW instead of DEEP makes a difference when the system runs few threads in single-threaded mode. But, we are interested in more realistic cases where many threads run on the system.

where IPC_{on} and IPC_{off} are the instructions per cycle when the prefetch is on and off, respectively. Similarly, BW_{on} and BW_{off} refer to the memory bandwidth consumption for the same configurations. All these values are dynamically obtained while applications are running on the system by sampling per-thread IPC and bandwidth from the performance monitoring unit (PMU) in POWER7. The rationale behind this metric is to compare the increase in performance to the increase in bandwidth when going from prefetching disabled to enabled. The theoretical range of values for this metric is $(0, 1]$. On the one hand, workloads with a prefetch usefulness close to 0 experience a very significant performance decrease or a large increase in bandwidth consumption (without a proportional increase in performance) when prefetching is enabled. On the other hand, prefetch-efficient workloads that obtain a prefetch usefulness equal to 1 have a proportional increase in performance and bandwidth when prefetching is turned on. This implies that for every unit of bandwidth consumed by prefetching there is a linear increase in performance. This indeed is the upper limit for prefetch usefulness. As a proof, let us consider a memory-bound program that traverses an array of size N bytes. With no prefetching it takes T_{off} seconds to execute. Therefore, bandwidth consumption is N/T_{off} bytes/second. Let us now assume a perfect prefetch engine (in terms of coverage, accuracy and timeliness). Such a prefetcher does not waste any data, thus only N bytes are moved from memory into the processor as well. The difference is that in this case data is not transferred because of demand misses, but because of prefetch actions. When prefetching is used, the time the program takes to complete is T_{on} ($T_{on} < T_{off}$). Bandwidth consumption is N/T_{on} bytes/second. Since execution time is inversely proportional to IPC we have the following:

$$\frac{IPC_{on}/BW_{on}}{IPC_{off}/BW_{off}} = \frac{T_{off}/T_{on}}{BW_{on}/BW_{off}} = \frac{T_{off}/T_{on}}{\frac{N/T_{on}}{N/T_{off}}} = 1 \quad (2)$$

This case represents the upper limit since in any other case where the prefetch engine moved useless data, bandwidth would proportionally increase more than performance and prefetch usefulness would be less than 1.

We explored the potential of extending the PU metric with some extra information such as estimators of cache pollution. But we decided to keep the metric as simple as possible for three reasons: 1) to increase the portability across different platforms, 2) due to the fact that in some systems obtaining an estimate of cache pollution may not be feasible or it may require reading a significant number of events from the performance monitoring unit (PMU)—incurring a high cost, and 3) because based on empirical observation we concluded that the effect of bandwidth saturation was a much more important factor to be addressed than cache pollution. Although we do not directly measure cache pollution or cache interference between threads, our algorithm dynamically recomputes PU for every thread running on the system. Therefore, our mechanism naturally adapts to application phases and changes in the thread mix. Our approach is also compatible with using extra information that might be potentially available in future processors.

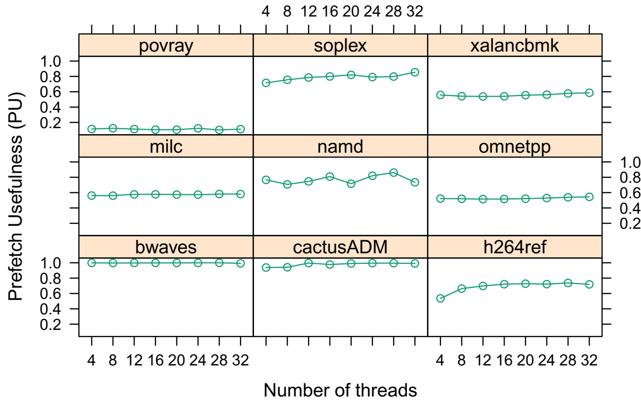


Fig. 3. Prefetch usefulness characterization for the benchmarks shown in Figure 2.

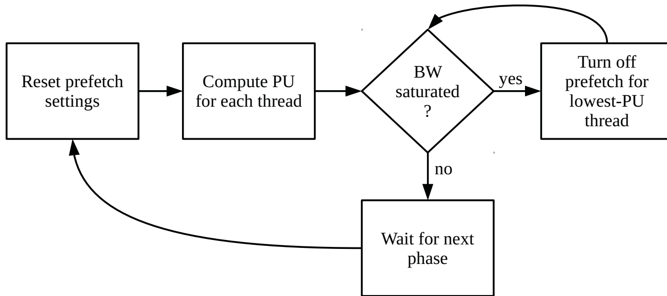


Fig. 4. Base bandwidth shifting algorithm.

Figure 3 shows a prefetch usefulness characterization for the benchmarks shown in Figure 2. Prefetch-efficient benchmarks such as bwaves and cactusADM consistently reach a prefetch usefulness of 1 for any number of threads (that is the largest prefetch usefulness that a benchmark could obtain). Benchmarks such as soplex and h264ref make a moderately efficient usage of prefetching. Other benchmarks such as omnetpp, milc and xalancbmk do not use prefetching in an efficient way. Therefore, it is typically better to take prefetching bandwidth away from them and to give it to more efficient workloads when bandwidth is scarce. Finally, povray is by far the most inefficient benchmark in terms of prefetching usage. Because of its low bandwidth consumption, however, this benchmark does not negatively affect other benchmarks running on the system.

A. Mechanism Description

Figure 4 shows the base implementation of the intelligent bandwidth shifting algorithm. It uses a fully online approach—no offline profiling step is required at all. The algorithm behaves in an iterative way. At the beginning of an iteration, the prefetch setting for every thread is reset using the most aggressive prefetch configuration. After that step, the algorithm computes the prefetch usefulness for every thread, and keeps the results in a table. [2] It does so by sequentially turning on and off prefetching for each thread, and measuring IPC and bandwidth in both configurations. By doing this process in a sequential manner, our mechanism is able to indirectly account for the interferences between threads at the different levels of the cache hierarchy. The algorithm samples performance counters with 1ms granularity. Therefore, computing prefetch usefulness for all the threads takes 64ms. The distance between two sampling steps or phases is 100ms. This sampling granularity may seem coarse compared to hardware-based solutions, but it is common for software-based ones. In fact, the Linux kernel cannot sample PMCs at a granularity smaller than 1ms (when sampling events from different groups). While hardware-based solutions are able to exploit the dynamic behavior of shorter phases, real applications present phases lasting just a few nanoseconds all the way up to the multi-seconds range. Therefore, our solution is able to adapt to the longer application's phases. But, more important, it can adapt to changing conditions in the system (e.g., when new threads are spawned and the workload mix changes). The sampling overhead is negligible since the runtime spends most of the time sleeping. We conducted tests where the runtime sampled PMCs but did not take any bandwidth shifting action. The results showed no measurable slowdown.

This is a common sampling granularity for OS and runtime adaptive solutions. Finer granularities might not be accurate and they may create significant overhead in the system. In the next step the algorithm checks the total bandwidth consumption in the system. If the threads running on the system do not saturate the total bandwidth capacity[3], the algorithm does nothing and it just waits for the next phase or iteration. If bandwidth is saturated, shifting bandwidth from low to highly-efficient threads will typically improve system performance. Our mechanism therefore turns prefetch off for the thread with the lowest prefetch usefulness. The algorithm then checks whether the bandwidth is still saturated. While it is, the algorithm will keep turning prefetch off for the running threads, based on their prefetch usefulness—going from low to high values.

Our bandwidth shifting mechanism is implemented as a runtime that monitors the running threads (reading the PMCs) and controls the prefetchers (through an OS-interface exposed in sysfs). The scheme, however, is not restricted to this implementation. For instance, an OS-level implementation would also be possible.

B. Guard Mechanism

We have observed an unlikely situation where, due to a lack of hardware resources, a thread with a higher PU cannot take over the bandwidth left unused when turning prefetch off for a lower PU thread. Threads running on a system share hardware resources in the memory hierarchy and this limits their individual peak bandwidth (e.g., there is a limit on the number of simultaneous prefetch streams that threads can allocate). Current hardware does not expose such information to the software. Yet, it is essential to prevent the algorithm from taking a decision that may lead to a system performance decrease. We thus extend our base algorithm with a guard

[2] We actually do not just store the last read sample. Instead, we use an exponentially-weighted moving average (EWMA) in order to limit the noise coming from micro-phases during the execution. This approach is similar to others used in prior work [17].

[3] Bandwidth is determined to be saturated once it reaches 90% of the peak achievable bandwidth. We have conducted experiments and determined that this threshold is the turning point where system performance degrades if prefetch bandwidth is not carefully managed.

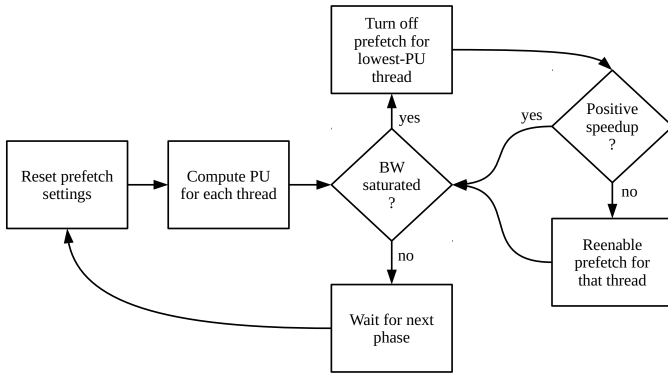


Fig. 5. Enhanced bandwidth shifting algorithm with a guard mechanism.

mechanism that increases performance up to 33% compared to the base algorithm.

Figure 5 shows the new version of the algorithm, including the guard mechanism. The behavior for the first steps is equivalent to the previous version. When the algorithm decides to turn prefetching off for a given thread, however, system performance is measured before and after disabling prefetching for that thread. If there is a negative global speedup, prefetching is restored for that thread—and the decision to turn it off again is not considered again until the next iteration. Otherwise, the algorithm behaves as the base one and prefetching is kept turned off until another iteration starts.

V. Results

In this section we evaluate the performance and power impact of our bandwidth shifting mechanism. Our mechanism targets improving system performance but it considers individual thread performance as well. Therefore, we use harmonic speedup (HS) as the metric to measure performance since it both captures individual thread performance and global system performance. We compute harmonic speedup using the following equation:

$$HS = \frac{\#threads}{\sum_{i}^{\#threads} \frac{time_{i,bw-shifting}}{time_{i,baseline}}} \qquad (3)$$

where $time_{i,bw-shifting}$ is the execution time for thread i when the bandwidth shifting mechanism is applied, and $time_{i,baseline}$ is the execution time for thread i when using the baseline prefetch configuration. In all our evaluations we run all the threads until the last one completes its execution. Before reaching that point we re-execute threads that finish earlier, thus keeping the number of threads constant during the execution. We use a configuration where prefetching is enabled for all the threads running on the system as the baseline for the evaluation of bandwidth shifting. In all the following figures we use this baseline for computing the harmonic speedup and for normalizing the power consumption.

A. Random Workloads

We use workloads composed of multiple randomly-selected benchmarks to evaluate the benefits of using our bandwidth shifting mechanism. We construct workloads containing eight benchmarks each. In total 32 threads are executed since we use all four SMT contexts, effectively running 4 threads per core (we run 4 copies of the same benchmark in each core). Evaluating bandwidth shifting with all the possible combinations of eight benchmarks is not feasible due to the vast exploration space. Instead, we create multiple groups of workloads with different characteristics in terms of memory bandwidth usage and prefetch efficiency. We use these groups to show the benefits of bandwidth shifting under different scenarios. We construct the following four groups by constraining the benchmarks that can be part of the different workloads:

- Random: Workloads in this group are constructed in a purely random way—no constraints are enforced. This translates into workloads where bandwidth consumption and prefetch efficiency vary across all the possible range. We use this group to demonstrate that our bandwidth shifting mechanism does not degrade performance for workloads that, a priori, should not significantly benefit from such a mechanism.
- MI-PE-high: Workloads in this group are memory intensive and they contain a high percentage of benchmarks that are prefetch efficient.
- MI-PE-mix: Workloads in this group are memory intensive and they are composed of benchmarks with mixed prefetch efficiency levels.
- MI-PE-low: Workloads in this group are memory intensive and they contain a high percentage of benchmarks that are not prefetch efficient.

Table I shows the exact benchmarks used in every workload for all the workload groups. In the following figures we utilize the workload name as displayed in this table to identify individual workloads.

1) Performance Evaluation: Figure 6 shows the performance impact of using our bandwidth shifting mechanism for all the workload groups. We study the benefits of using bandwidth shifting independently for each group.

a) Random group: Workloads in this group typically do not contain more than one prefetch inefficient benchmark. Since they are also composed of multiple benchmarks with low memory bandwidth consumption, the speedups obtained when using the bandwidth shifting mechanism are not large. Except for workload 10, speedups are always below 5%—the average speedup is 2.5%. Workload 10 is one of the workloads with highest bandwidth consumption in this group. It additionally contains two inefficient benchmarks (milc and astar). Because of this, the bandwidth shifting mechanism is able to obtain a 7% speedup. We observe a small performance degradation for workload 8. The slowdown is, however, less than 0.5%. No other workload suffers any performance degradation, so we can effectively consider that our bandwidth shifting mechanism works efficiently even in cases where the potential benefits are not expected to be large.

b) Memory intensive, high prefetch efficiency: Most workloads in this group contain less than three prefetch-inefficient benchmarks. But compared to the previous group, there is more room for effectively shifting bandwidth from prefetch-inefficient benchmarks to other benchmarks that use prefetching more efficiently. A 6.5% average speedup reflects that observation. In this group no workload suffers a performance degradation—the minimum speedup is 1.5%

TABLE I
BENCHMARK COMBINATIONS USED FOR CREATING THE RANDOM WORKLOADS.

WL1	WL2	WL3	WL4	WL5	WL6	WL7	WL8	WL9	WL10
sjeng	dealII	tonto	GemsFDTD	gromacs	GemsFDTD	perlbench	libquantum	h264ref	milc
tonto	milc	namd	milc	Graph500	soplex	lbm	zeusmp	mcf	zeusmp
zeusmp	libquantum	cactusADM	h264ref	soplex	milc	bwaves	gobmk	soplex	astar
hmmer	zeusmp	soplex	h264ref	bzip2	soplex	wrf	gromacs	hmmer	GemsFDTD
h264ref	calculix	namd	perlbench	xalancbmk	calculix	omnetpp	xalancbmk	libquantum	gcc
gobmk	leslie3d	povray	sphinx3	gamess	h264ref	perlbench	dealII	Graph500	soplex
hmmer	GemsFDTD	omnetpp	soplex	bwaves	xalancbmk	gamess	mcf	hmmer	lbm
h264ref	bzip2	mcf	sjeng	sjeng	tonto	dealII	wrf	wrf	mcf

(a) Random

WL1	WL2	WL3	WL4	WL5	WL6	WL7	WL8	WL9	WL10
leslie3d	astar	gcc	milc	lbm	Graph500	astar	sphinx3	xalancbmk	Graph500
xalancbmk	GemsFDTD	soplex	milc	lbm	mcf	leslie3d	lbm	libquantum	leslie3d
xalancbmk	mcf	libquantum	omnetpp	astar	libquantum	bwaves	sphinx3	gcc	Graph500
leslie3d	xalancbmk	GemsFDTD	leslie3d	lbm	Graph500	omnetpp	sphinx3	Graph500	xalancbmk
mcf	milc	gcc	astar	astar	leslie3d	milc	milc	mcf	bwaves
gcc	libquantum	GemsFDTD	bwaves	bwaves	gcc	astar	milc	gcc	milc
gcc	sphinx3	omnetpp	omnetpp	xalancbmk	GemsFDTD	astar	GemsFDTD	GemsFDTD	omnetpp
leslie3d	mcf	leslie3d	libquantum	sphinx3	xalancbmk	libquantum	milc	mcf	libquantum

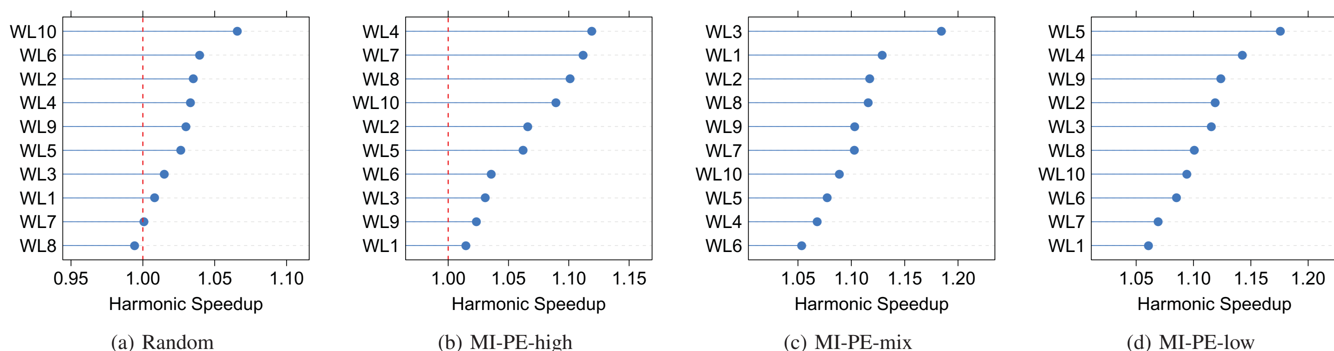
(b) MI-PE-high

WL1	WL2	WL3	WL4	WL5	WL6	WL7	WL8	WL9	WL10
xalancbmk	omnetpp	omnetpp	gcc	milc	Graph500	gcc	GemsFDTD	sphinx3	omnetpp
astar	sphinx3	omnetpp	astar	sphinx3	soplex	leslie3d	Graph500	astar	leslie3d
mcf	bwaves	milc	gcc	milc	Graph500	astar	milc	lbm	astar
soplex	milc	sphinx3	Graph500	GemsFDTD	astar	milc	Graph500	milc	xalancbmk
milc	soplex	bwaves	leslie3d	libquantum	xalancbmk	mcf	omnetpp	astar	Graph500
omnetpp	mcf	milc	Graph500	astar	libquantum	astar	gcc	GemsFDTD	Graph500
Graph500	xalancbmk	sphinx3	omnetpp	gcc	GemsFDTD	leslie3d	Graph500	milc	sphinx3
lbm	xalancbmk	milc	mcf	gcc	soplex	milc	astar	leslie3d	lbm

(c) MI-PE-mix

WL1	WL2	WL3	WL4	WL5	WL6	WL7	WL8	WL9	WL10
gcc	GemsFDTD	xalancbmk	milc	gcc	astar	libquantum	milc	milc	leslie3d
astar	Graph500	gcc	milc	omnetpp	omnetpp	astar	leslie3d	GemsFDTD	xalancbmk
gcc	milc	omnetpp	omnetpp	lbm	xalancbmk	milc	milc	libquantum	gcc
Graph500	Graph500	GemsFDTD	astar	milc	astar	gcc	milc	omnetpp	bwaves
leslie3d	omnetpp	milc	omnetpp	omnetpp	xalancbmk	xalancbmk	Graph500	xalancbmk	omnetpp
Graph500	gcc	astar	GemsFDTD	mcf	Graph500	bwaves	xalancbmk	milc	Graph500
omnetpp	Graph500	sphinx3	Graph500	omnetpp	libquantum	xalancbmk	Graph500	milc	astar
mcf	astar	omnetpp	milc	milc	gcc	gcc	soplex	astar	omnetpp

(d) MI-PE-low



(a) Random (b) MI-PE-high (c) MI-PE-mix (d) MI-PE-low

Fig. 6. Performance results for randomly-constructed workloads.

(workload 1). Workloads 4 and 7 contain multiple copies of prefetch-inefficient, memory-intensive benchmarks such as milc, omnetpp and astar. The bandwidth shifting mechanism obtains speedups slightly over 10% for these two cases.

c) Memory intensive, mixed prefetch efficiency: Workloads in this group contain approximately 50% prefetch-inefficient benchmarks. The potential for bandwidth shifting in this scenario seems ample and the results confirm that intuition. The average speedup for this group of workloads is 10%. This is a very significant performance increase, especially considering that we are conducting our experiments on a real machine with all the software stack running on it. The highest speedup (18.5%) is achieved for workload 3. That workload contains a large number of prefetch-inefficient benchmarks plus some highly efficient ones. In such a scenario, the bandwidth shifting mechanism obtains the best results by giving the valuable bandwidth to the benchmarks that better use it. All the workloads experience speedups when run under the control of the bandwidth shifting mechanism. The smallest speedup is 5.4%.

d) Memory intensive, low prefetch efficiency: Workloads in this group are composed of approximately 70% prefetch-inefficient workloads. It may seem as if bandwidth shifting could potentially obtain higher speedups for this group compared to the previous one. When most benchmarks in a workload are prefetch-inefficient, however, there are not so many good candidates to shift the bandwidth to. Nonetheless, the bandwidth shifting mechanism performs very successfully in this group too. The average speedup is 11%. The minimum and maximum speedups are 6% and 18%, respectively.

According to these results, we conclude that the bandwidth shifting mechanism works efficiently across a wide range of different scenarios. Even if, as expected, large speedups are not obtained for workloads with a low degree of memory intensity and few prefetch-inefficient benchmarks, performance for these workloads is not degraded in virtually any case. When we use bandwidth shifting under the presence of memory-intensive, prefetch-inefficient workloads, such a dynamic mechanism certainly helps at improving performance—the maximum speedup obtained being 18.5%.

2) Fairness Evaluation: The main goal of this paper is to present a mechanism for improving global system performance by intelligently allocating prefetch bandwidth among the different applications. For such kind of mechanism, however, it is important to assess its impact on fairness. Because of the nature of our mechanism, even if the global performance increases, some applications may experience speedups while others may experience slowdowns. Therefore, we study the impact of our mechanism on the fairness among the running benchmarks. In order to do that, we look at the individual speedups experienced by all the benchmarks composing the workloads.

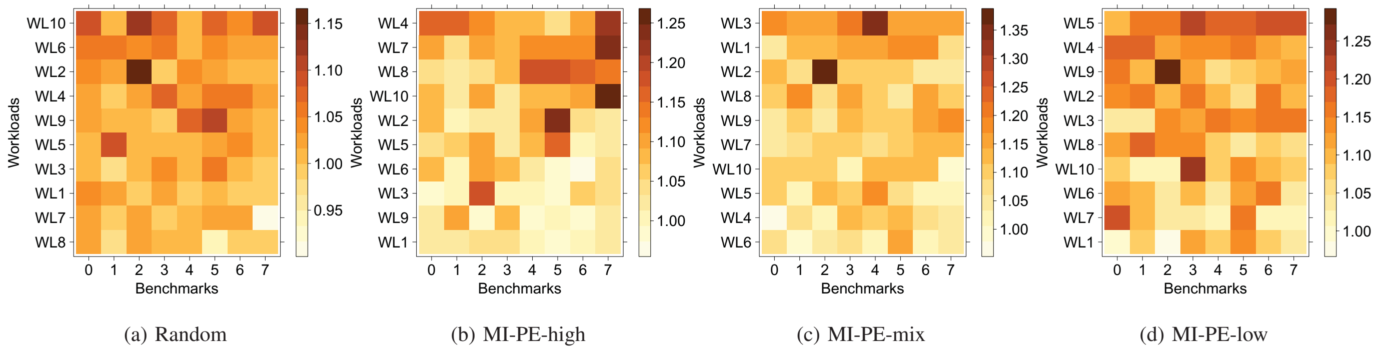
Figure 7 contains a heat map showing individual benchmark speedup for each workload group. Lighter colors represent lower speedups—or even slowdowns—and darker ones stand for higher speedups. For instance, the row labeled WL3 in Figure 7b shows the individual speedups for all benchmarks in workload 3 (from the MI-PE-high group). Such workload contains a heterogeneous mix of workloads: some such as libquantum and leslie3d are prefetch-efficient while omnetpp is very inefficient. Other benchmarks such as gcc, GemsFDTD and soplex are sensitive to prefetching but to a much lesser degree. Our mechanism shifts bandwidth away from omnetpp so that libquantum and leslie3d can benefit from that extra bandwidth, obtaining 18% and 4% speedups, respectively. At the same time omnetpp benefits as well, since turning prefetch off for that benchmark avoids the generation of a high number of useless prefetches. In the figure we observe a dark square—representing the high speedup for libquantum—and two smaller speedups—omnetpp and leslie3d. The rest of the benchmarks experience $\pm 1\%$ speedups.

TABLE II
SUMMARY OF INDIVIDUAL SPEEDUPS FOR THE DIFFERENT WORKLOAD GROUPS.

	Random	MI-PE-high	MI-PE-mix	MI-PE-low
Min. speedup	0.92	0.97	0.98	0.99
Max. speedup	1.15	1.25	1.36	1.27
Avg. speedup	1.03	1.07	1.11	1.11

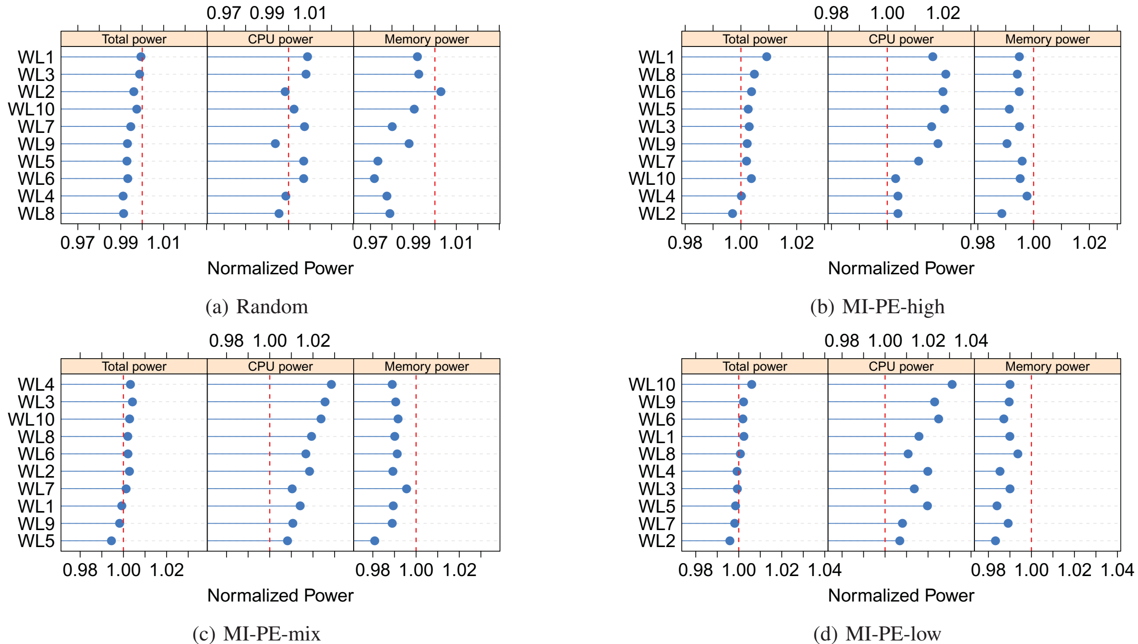
Looking at Figure 7 we observe that in general there is not a high degree of unfairness. Some benchmarks such as bwaves and libquantum consistently obtain very significant individual speedups up to 36%—most of the darkest squares in the figure are related to these two benchmarks. Other benchmarks experience a slowdown since bandwidth is taken away from them, but the maximum performance degradation is 8% in the most extreme case—below 3% in the average case. Table II shows these observations. It contains a summary of the individual speedups obtained for each workload group. The results show that our bandwidth shifting mechanism—with the help of the guard feature—achieves to maintain a good level of fairness between the different benchmarks running on the system. As future work, we plan to extend the guard mechanism to keep fairness under a certain threshold if desired.

3) Power Consumption Evaluation: Figure 8 shows the power consumption impact of using our bandwidth shifting mechanism for all the workload groups. In general, total power consumption is not significantly affected—all values are within 1% of the power consumption obtained when the bandwidth shifting mechanism is not utilized. For virtually all workloads memory power consumption decreases (up to 3%). The reason behind that reduction is that the bandwidth shifting mechanism effectively turns off prefetch for those benchmarks that are not efficiently using prefetching. Even if prefetch-efficient benchmarks benefit from that extra available bandwidth—performance actually increases for them—in some cases, these benchmarks might not be able to utilize all the bandwidth freed by the shifting mechanism. On the other hand, CPU power consumption typically increases for most benchmarks. By shifting bandwidth to the more efficient benchmarks, useless prefetches and cache misses due to cache pollution are avoided. That increases CPU utilization for the benchmarks running on the system while at the same time it increases power consumption in the cores.



(a) Random (b) MI-PE-high (c) MI-PE-mix (d) MI-PE-low

Fig. 7. Individual speedups for results in Figure 6. Each square in a plot displays the individual speedup for a benchmark within a workload. The speedup degree is shown with a color scale, going from light colors—lower speedup—to dark colors—higher speedup.



(a) Random (b) MI-PE-high

(c) MI-PE-mix (d) MI-PE-low

Fig. 8. Power consumption results for random workloads. Values are normalized to the case where the most aggressive prefetch setting is used for all the benchmarks.

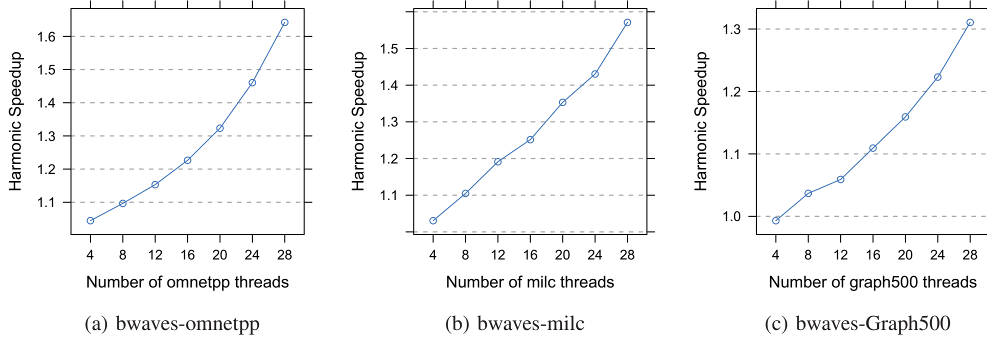
B. Limit Studies

In this section we look at the potential of bandwidth shifting in more extreme cases where we run two benchmarks with very different characteristics together. Figure 9 shows the results of these experiments. In all the cases we run one benchmark that uses prefetching in a very efficient way (bwaves) and a benchmark that is very prefetch-inefficient (omnetpp, milc and Graph500). It is important to note that even if we only use bwaves as the representative of prefetch-efficient benchmarks, similar results are obtained with other benchmarks such as leslie3d or libquantum.

In Figure 9 the X axis shows the number of copies running for the prefetch-inefficient benchmark (x). In all the cases we run 32 copies in total, therefore the number of copies for the prefetch-efficient benchmark is $32 - x$. All the combinations show a similar trend: when the number of prefetch-inefficient threads is low (4) most of the bandwidth is consumed by the prefetch-efficient benchmark. Therefore, there is limited potential for bandwidth shifting in this scenario. In all cases the speedup is well below 10%. As we increase the number of prefetch-inefficient threads, however, the pressure on bandwidth increases and the impact of bandwidth shifting on performance is much more significant. In the most extreme case—when 28 omnetpp copies are run—bandwidth shifting achieves a speedup over 1.6X.

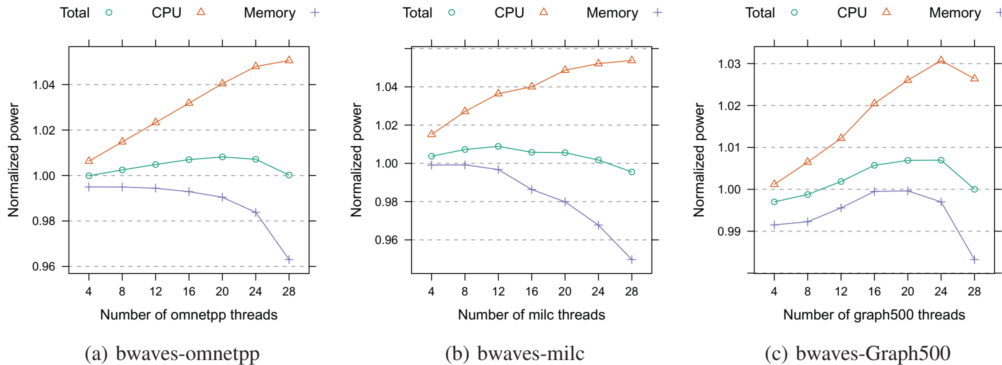
While we have seen very significant speedups in the case of randomly constructed workloads, the potential for bandwidth shifting is even higher for mixed workloads with applications that exploit prefetching in a very efficient way and applications that are prefetch unfriendly.

Figure 10 shows the power consumption for the experiments shown in Figure 9. In terms of total power consumption, the variation due to using bandwidth shifting is rather small (<1%). If we look at the CPU and memory power consumption we observe higher power variations in the order of 4-5%. As it was the case with random workloads, memory power consumption tends to go down when bandwidth shifting is used. This decrease is much more significant as the number of prefetch-inefficient threads gets larger. For instance, in Figure 10a we observe that when 28 omnetpp copies are run,



(a) bwaves-omnetpp (b) bwaves-milc (c) bwaves-Graph500

Fig. 9. Performance results for an increasing number of copies of prefetch-inefficient benchmarks.



(a) bwaves-omnetpp (b) bwaves-milc (c) bwaves-Graph500

Fig. 10. Power consumption results for an increasing number of copies of prefetch-inefficient benchmarks. Figure 9 shows the performance results for the same set of experiments.

memory power consumption goes down 4% with respect to the case where no bandwidth shifting is used. The reason for that is a total bandwidth decrease. As bandwidth shifting turns off prefetching for omnetpp, its bandwidth consumption is almost halved. The only 4 bwaves copies running on the system cannot consume enough bandwidth—even when prefetching is enabled. Since the total bandwidth to memory decreases, and active power consumption in DRAM is proportional to bandwidth consumption, using bandwidth shifting reduces memory power consumption. The same trend can be observed for the other two experiments.

CPU power consumption goes up when using bandwidth shifting. The reason is again a more effective usage of core resources since the cores do not need to wait so long for memory requests to come back from the last level cache or DRAM. All three experiments show similar results where CPU power consumption increases up to 5%. It is important to note that such a power consumption increase comes with a very significant performance speedup in the order of 1.3-1.6X.

VI. Related Work

This section provides references to related and complementary works in the field of hardware data prefetching. Table III provides a comparison with respect to the closest works available in the literature. To the best of our knowledge, our solution is the first to work on a real CMP systems without requiring hardware support.

TABLE III
Summary of differences w.r.t. prior work.

Work	Targets global system performance	Requires HW support
[6], [7], [29], [36]	No	Yes
[17], [22], [40]	No	No
[9], [10]	Yes	Yes
This work	Yes	No

A. General Prefetching

There is a significant record of past research in data prefetch (e.g., [1], [14], [19], [30], [34]). Most of the initial proposals were based on sequential prefetchers, which rely on applications exhibiting 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. This has 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 [5], [11], [32], [39], [41]. In [18] the authors study Markov-based prefetchers and present solutions to limit the bandwidth devoted to prefetching. Another interesting work uses a user-level memory thread in order to prefetch data, delivering significant speedups even for applications with irregular accesses [35]. Limit studies and prefetch analytical models have

TABLE IV
PERFORMANCE BENEFITS W.R.T. [17].

Workload Group	Average Speedup	Maximum Speedup
Random	0.99	1.04
MI-PE-high	1.03	1.09
MI-PE-mix	1.11	1.19
MI-PE-low	1.10	1.17

been presented in [12], [37].

Most general-purpose processors contain a prefetch engine based on some of these works. Our solution is orthogonal to them since their objective is to improve the accuracy of the algorithms implemented in a single prefetcher.

B. Filtering Useless Prefetches

Several works that attempt to reduce the number of useless prefetches sent to memory have been presented in the past [4], [20], [23], [26], [28], [42]. Even when these filtering techniques are used, applications still have different prefetch-efficiency degrees. Therefore, our bandwidth shifting mechanism can be complementary used to further improve performance.

C. Local Adaptive Prefetching

Using a prefetch engine that implements a fixed algorithm is suboptimal since the prefetch-efficiency of applications may change during the different phases of execution. Several adaptive solutions that attempt to dynamically change the prefetch configuration exist in prior research [6], [7], [17], [29], [36]. The objective of these solutions, however, is not to maximize global system performance. They are either designed for single-threaded processors or they only attempt to locally increase performance of individual cores. Therefore, while they may improve performance for a particular core, system performance may decrease. On the contrary, our solution maintains a global system view and increases performance for the whole system. For comparison purposes we have actually implemented the mechanism proposed in [17].

We have confirmed that it increases system performance for relatively simple workloads such as the ones used in [17] (only workloads composed of two different benchmarks were used). For more complex workloads such as the ones used to evaluate the mechanism proposed in this paper (where up to eight different benchmarks are used in each workload), bandwidth shifting obtains 12.3% speedup on average—maximum speedup is 19%—compared to the mechanism presented in [17].

Table IV shows more detailed results on the performance benefits of using our bandwidth shifting solution compared to the work presented in [17]. Most benchmarks in the "random" workload group are either not memory intensive or prefetch-efficient. Because of that, the potential for bandwidth shifting is not large. In this case our solution performs very similar to [17]. But, as workloads contain more memory-intensive or prefetch-unfriendly benchmarks, our solution clearly outperforms [17]. These results demonstrate that the lack of a global system vision in the solution presented in [17] prevents that solution from scaling to systems with a large core count and from handling complex workloads.

D. CMP-Aware Adaptive Prefetching

With the advent of CMP processors, interaction between threads must be taken into account when designing a prefetch system. [10] and [9] study the effect of thread-interaction on prefetch, and propose techniques to design prefetch systems that improve throughput or fairness. Despite the similarities to our paper, their solution requires costly extra hardware—amounting to multiple kilobytes—whereas our solution works with most modern general-purpose processors. In a related work [24], the authors study the impact of prefetching and bandwidth partitioning in CMPs. But their work only presents an analytical model and no mechanism to exploit their observations is included.

E. Solutions for Real Systems

Although there is a significant number of studies on prefetching based on simulators, there are very few works that deal with hardware-based measurement and characterization. In [40] the authors 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 [22] the authors construct a machine learning model that dynamically modifies the prefetch configuration of the machines in a data center (based on Intel Core2 processors). Although they improve performance for some applications by enabling/disabling prefetch, their work only focuses on how to improve the performance for a single application without taking into account the performance/bandwidth trade-offs that appear when multiple applications are executed concurrently. A dynamic mechanism for optimizing individual thread performance on a real system is presented in [17]. As already noted in Section VI-C, however, this mechanism is not CMP-aware and this fact hinders its potential for workloads composed of different benchmarks.

VII. CONCLUSIONS AND FUTURE WORK

Effectively managing memory bandwidth consumption in highly-threaded CMP/SMT systems is becoming paramount. In this paper we present an intelligent bandwidth shifting mechanism that assigns bandwidth resources to applications in such a way that prefetch-inefficient applications do not waste these resources while prefetch-efficient ones benefit from the extra available resources. To the best of our knowledge, our solution is the first one to address this important problem without requiring hardware support. We assess the impact of our bandwidth shifting mechanism using an extensive evaluation. We obtain significant speedups—in the order of 10-20% for randomly built workloads and over 1.6X for more extreme cases where one benchmark uses prefetching in a very efficient way while the other is very prefetch-inefficient.

ACKNOWLEDGMENTS

This work has been partially sponsored by Defense Advanced Research Projects Agency (DARPA), Microsystems Technology Office (MTO), under contract no. HR0011-13-C-0022. The views expressed are those of the authors and do not reflect the official policy or position of the Department of

Defense or the U.S. Government. This document is: Approved for Public Release, Distribution Unlimited.

This work has also received funding from: the Spanish Ministry of Science and Innovation under grant TIN2012-34557 and the HiPEAC Network of Excellence; and the European Research Council under the European Unions 7th FP (FP/2007- 2013) / ERC GA n. 321253. Additional support was received from a joint study agreement between IBM and BSC (number W1361154).

REFERENCES

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

[2] 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*, ser. ISPASS, 2010, pp. 203–212.

[3] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," in *Proc. 4th Int'l Conf. Arch. Support for Prog. Lang. and Operat. Sys.*, ser. ASPLOS. ACM, 1991, pp. 40–52.

[4] M. J. Charney and T. R. Puzak, "Prefetching and memory system behavior of the SPEC95 benchmark suite," *IBM J. R&D*, vol. 41, pp. 265–286, 1997.

[5] R. Cooksey, S. Jourdan, and D. Grunwald, "A Stateless, Content-directed Data Prefetching Mechanism," in *Proc. 10th Int'l Conf. Arch. Support for Prog. Lang. and Operat. Sys.*, ser. ASPLOS. ACM, 2002, pp. 279–290.

[6] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors," in *Proc. 22nd Int'l Conf. Parallel Processing*, vol. 1, 1993, pp. 56–63.

[7] ——, "Sequential Hardware Prefetching in Shared-Memory Multiprocessors," *Trans. Parallel and Dist. Sys.*, vol. 6, no. 7, pp. 733–746, 1995.

[8] A. C. de Melo, "Performance Counters for Linux," 2010. [Online]. Available: http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf

[9] 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.*, ser. ISCA. ACM, 2011, pp. 141–152.

[10] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated Control of Multiple Prefetchers in Multi-core Systems," in *Proc. 42nd Int'l Symp. Microarch.*, ser. MICRO 42. ACM, 2009, pp. 316–326.

[11] 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.*, ser. HPCA, 2009, pp. 7–17.

[12] P. G. Emma, A. Hartstein, T. R. Puzak, and V. Srinivasan, "Exploring the limits of prefetching," *IBM J. R&D*, vol. 49, no. 1, pp. 127–144, January 2005.

[13] M. Floyd, M. Allen-Ware, K. Rajamani, B. Brock, C. Lefurgy, A. Drake, L. Pesantez, T. Gloekler, J. Tierno, P. Bose, and A. Buyuktosunoglu, "Introducing the Adaptive Energy Management Features of the POWER7 Chip," *IEEE Micro*, vol. 31, no. 2, pp. 60–75, March-April 2011.

[14] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride Directed Prefetching in Scalar Processors," in *Proc. 25th Int'l Symp. Microarch.*, ser. MICRO. IEEE Computer Society Press, 1992, pp. 102–110.

[15] I. Hur and C. Lin, "Memory Prefetching Using Adaptive Stream Detection," in *Proc. 39th Int'l Symp. on Microarchitecture*, ser. MICRO. IEEE Computer Society, 2006, pp. 397–408.

[16] IBM, "Power ISA™ Version 2.06 Revision B," 2010. [Online]. Available: https://www.power.org/wp-content/uploads/2012/07/PowerISA_V2.06B_V2_PUBLIC.pdf

[17] V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O'Connell, "Making Data Prefetch Smarter: Adaptive Prefetching on POWER7," in *Proc. 21st Int'l Conf. Parallel Arch. and Compilation Techniques*, ser. PACT. ACM, 2012, pp. 137–146.

[18] D. Joseph and D. Grunwald, "Prefetching using Markov Predictors," in *Proc. 24th Int'l Symp. Comp. Arch.*, ser. ISCA. ACM, 1997, pp. 252–263.

[19] 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.*, ser. ISCA. ACM, 1990, pp. 364–373.

[20] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-Aware DRAM Controllers," in *Proc. 41st Int'l Symp. Microarch.*, ser. MICRO. IEEE Computer Society, 2008, pp. 200–209.

[21] C. Lefurgy, X. Wang, and M. Ware, "Server-Level Power Control," in *Proc. 4th Int'l Conf. on Autonomic Computing*, ser. ICAC. IEEE Computer Society, 2007, pp. 4–14.

[22] S. W. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, "Machine Learning-Based Prefetch Optimization for Data Center Applications," in *Proc. Int'l Conf. High Perf. Comp. Networking, Storage and Analysis*, ser. SC. ACM, 2009, pp. 1–10.

[23] W.-F. Lin, S. K. Reinhardt, D. Burger, and T. R. Puzak, "Filtering Superfluous Prefetches Using Density Vectors," in *Proc. Int'l Conf. Comp. Design: VLSI in Computers & Processors*, ser. ICCD. IEEE Computer Society, 2001, pp. 124–132.

[24] F. Liu and Y. Solihin, "Studying the Impact of Hardware Prefetching and Bandwidth Partitioning in Chip-Multiprocessors," in *Proc. Int'l Conf. Meas. and Model. of Comp. Sys.*, ser. SIGMETRICS. ACM, 2011, pp. 37–48.

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

[26] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," in *Proc. 5th Int'l Conf. Arch. Support for Prog. Lang. and Operat. Sys.*, ser. ASPLOS. ACM, 1992, pp. 62–73.

[27] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," *Craig User's Group, CUG*, May 2010.

[28] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt, "Using the First-level Caches As Filters to Reduce the Pollution Caused by Speculative Memory References," *Int. J. Parallel Program.*, vol. 33, no. 5, pp. 529–559, Oct. 2005.

[29] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "AC/DC: An Adaptive Data Cache Prefetcher," in *Proc. 13th Int'l Conf. Parallel Arch. and Compilation Techniques*, 2004.

[30] S. Palacharla and R. E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," in *Proc. 21st Int'l Symp. Comp. Arch.*, ser. ISCA. IEEE Computer Society Press, 1994, pp. 24–33.

[31] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling," in *Proc. 36th Int'l Symp. Comp. Arch.*, ser. ISCA. ACM, 2009, pp. 371–382.

[32] 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.*, ser. ASPLOS. ACM, 1998, pp. 115–126.

[33] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams, "IBM POWER7 multicore server processor," *IBM J. R&D*, vol. 55, no. 3, pp. 1–29, May-June 2011.

[34] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, 1978.

[35] Y. Solihin, J. Lee, and J. Torrellas, "Using a User-Level Memory Thread for Correlation Prefetching," in *Proc. 29th Int'l Symp. Comp. Arch.*, ser. ISCA. ACM, 2002, pp. 171–182.

[36] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *Proc. 13th Int'l Symp. High Perf. Comp. Arch.*, ser. HPCA. IEEE Computer Society, 2007, pp. 63–74.

[37] V. Srinivasan, E. S. Davidson, and G. S. Tyson, "A prefetch taxonomy," *IEEE Trans. Comp.*, vol. 53, no. 2, pp. 126–140, February 2004.

[38] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernández, and M. Valero, "FAME: FAirly MEasuring Multithreaded Architectures," in *Proc. 16th Int'l Conf. Parallel Arch. and Compilation Techniques*, ser. PACT, 2007, pp. 305–316.

[39] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, "Guided Region Prefetching: A Cooperative Hardware/Software Approach," in *Proc. 30th Int'l Symp. Comp. Arch.*, ser. ISCA. ACM, 2003, pp. 388–398.

[40] 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*, ser. ISPASS. IEEE Computer Society, 2011, pp. 2–11.

[41] C. L. Yang and A. R. Lebeck, "Push vs. Pull: Data Movement for Linked Data Structures," in *Proc. 14th Int'l Conf. Supercomputing*, ser. ICS. ACM, 2000, pp. 176–186.

[42] X. Zhuang and L. H.-H. S., "A hardware-based Cache Pollution Filtering Mechanism for Aggressive Prefetches," in *Proc. 32nd Int'l Conf. Parallel Processing*, 2003, pp. 286–293.