Adaptive and application dependent runtime guided hardware prefetcher reconfiguration on the IBM POWER7

David Prat, Cristobal Ortega, Marc Casas, Miquel Moretó, Mateo Valero

Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya
08034, Barcelona, Spain

ABSTRACT

Hardware data prefetcher engines have been extensively used to reduce the impact of memory latency. However, microprocessors' hardware prefetcher engines do not include any automatic hardware control able to dynamically tune their operation. This lacking architectural feature causes systems to operate with prefetchers in a fixed configuration, which in many cases harms performance and energy consumption.

In this paper, a piece of software that solves the discussed problem in the context of the IBM POWER7 microprocessor is presented. The proposed solution involves using the runtime software as a bridge that is able to characterize user applications' workload and dynamically reconfigure the prefetcher engine. The proposed mechanisms has been deployed over OmpSs, a state-of-the-art task-based programming model. The paper shows significant performance improvements over a representative set of microbenchmarks and High Performance Computing (HPC) applications.

1. INTRODUCTION

Hardware data prefetch is a performance optimization technique that helps to alleviate the so-called *Memory Wall* [24] problem by taking advantage of applications' spatial locality when accessing to memory. Although some contemporary processors come with a set of knobs that adjust different parameters of the hardware prefetcher, their tuning is left to programmer's responsibility being them set to a default configuration when the system boots up. Unfortunately, apart from being a source of detriment for application's performance, in some cases, this default configuration can suppose a waste of consumed power. For example: prefetching a great amount of data in each memory request may involve bringing unnecessary data that not only wastes power by overloading memory bandwidth, but also pollutes cache memory hierarchy potentially reducing the effective cache space, which can impact performance in multicore environments.

The IBM POWER7 microprocessor [11] provides the user with the possibility to enable/disable the hardware prefetcher, also to tune the depth of each prefetch operation, to find store prefetch streams of data and to find strides in data accesses, which are gaps of a given fixed size in a data stream. Over this paper, it will be shown how different applications can benefit from this sort of knobs and it will be made evident that hardware prefetcher configuration can not be left to randomness nor default values but it needs of an algorithm that finds a balance in power-performance depending on each application workload. To provide the algorithm with data to determine which configuration to choose, placed in the runtime, a dynamic mechanism that can track performance of multithreaded workloads will be constructed. Specifically, the dynamic mechanism collects performance counters at task level thus being possible to adjust the prefetcher configuration for each code region delimited by the programmer.

This paper is organized as follows: Section 2 describes the IBM POWER7 main characteristics and the prefetcher reconfigurability. Section 3 describes the proposed dynamic mechanism that finds the best prefetcher configuration at runtime. Next, Section 4 consists in an evaluation of the proposed solutions by means of analyzing performance metrics of selected representative benchmarks. Section 5 summarizes the related work and, finally, Section 6 presents the conclusions of this paper.

2. BACKGROUND

The IBM POWER7 [11] is an 8-way issue superscalar symmetric multiprocessor based on the Power Architecture. Its main specifications include: 8 cores with 4-way SMT; for each core, two separated L1 caches of 32KB, one for data and other for instructions, plus a 256KB L2 cache. Furthermore, there is an on-chip 32MB L3 shared cache where each core has its private 4MB portion, being able to access other portions though at a cost of higher latency. The IBM POWER7 reconfigurability allows the end-user to choose the SMT degree, it can be set to single-thread, two-way and four-way. There is also the possibility to change the priority in the decoded instructions of each thread and there are also different

Table 1: Hardware prefetcher configurations

DSCR	Description	DSCR	Description
xx001	Off (disabled)	xx101	Deep
xx000	Default (Deep)	xx110	Deeper
xx010	Shallowest	xx111	Deepest
xx011	Shallow	x1xxx	Prefetch on stores
xx100	Medium	1xxxx	Stride-N

knobs associated to the hardware data prefetcher that control its operation mode.

The IBM POWER7's hardware data prefetcher is programmable per each SMT hardware thread, which means that there are 32 configuration registers accessible from the Operating System (OS). They are denoted as Data Stream Control Register (DSCR). They operate independently, which means that while one hardware thread is executing aggressively prefetching data, another one can be running with the prefetcher disabled. It is possible to enable or disable the prefetcher engine in each thread as well as to change the depth of each prefetcher operation. Moreover, detecting store data streams and strided accesses can also be enabled. Table 1 shows how to do it by writing the DSCR. Bits first to third are called default prefetcher depth (DPFD) where their value represent, in each case, the number of lines each prefetch operation brings from main memory to cache. The fifth bit, called Stride-N Stream Enable (SNSE), only has some effect in its activation if the fourth bit, called Store Stream Enable (SSE), is also enabled and the hardware data prefetcher is enabled.

In this paper, OmpSs [5], a state-of-the-art task-based programming model is used. This programming model, similarly to the recent OpenMP 4.0 standard, lets the programmer to specify sequential regions of code with their data dependencies. These code regions are called tasks and can run once their input and control dependencies are satisfied. The OmpSs runtime system orchestrates the parallel execution of the different tasks while makes sure all the dependences are satisfied.

3. ADAPTIVE PREFETCHER

In this paper, an adaptive prefetcher mechanism able to operate at runtime is proposed. Performance metrics associated to application's execution will be used to choose the most suitable configuration. The mechanism operates in two phases: During the *exploration* phase, each prefetcher configuration is evaluated in terms of performance improvement. During the *stable* phase, the best prefetcher configuration found in the exploration phase is used for another amount of consecutive tasks. A very similar approach about hardware prefetcher reconfiguration at runtime was recently presented by Jimenez et al. [8]. In their work, a fixed time of 10ms for the exploration phases as well as 100ms for the stable phases were proposed. These values were chosen empirically and aimed to mitigate the problem that appears when one prefetcher configuration is chosen but the application phase changes to another one that can benefit more from a different prefetcher configuration. However, their mechanism selects the same prefetcher configuration for all threads of an application in the stable phase. In this paper, a more powerful technique is presented, using granularity at OmpSs task level to characterize these phases and allowing to have different prefetcher configurations per task type, even if they run simultaneously.

The lengths of exploration and stable phases are important parameters of the adaptive mechanism. These lengths are measured in terms of number of task instances that are executed in the corresponding phase. The execution time of each task instance is measured and stored internally in the runtime system metadata. Regarding exploration phases, it is necessary to run enough experiments to filter the measurement noise while keeping the exploration phase as short as possible. First experiments are about finding optimal values for exploration and stable phases. Section 4.2 explains in detail the exploration we do in this paper regarding exploration phases' lengths.

The impact of having different optimal prefetcher configurations for different task types instead of having a task agnostic mechanism is also evaluated in Section 4.3. Different OmpSs tasks may have different kinds of workloads and therefore they can benefit more from different prefetcher configurations. However, that difference may not be large enough to compensate the additional overhead that this task type aware mechanism has.

The trade-offs between performance improvement and power consumption in terms of memory bandwidth usage are explored in Section 4.4. The paper presents and evaluates a solution based on an ϵ parameter configurable by the user to determine what percentage of difference in the IPCs of one prefetcher configuration with respect to another one less aggressive is needed to choose it as optimal. In this case, it is important to see for each application the relation between the aggressiveness of the prefetcher configuration, the used bandwidth and the execution time. When the adaptive prefetcher mechanism chooses aggressive prefetcher configurations, higher bandwidth rates are consumed. However, consuming more bandwidth with small performance improvement may not be worth.

4. RESULTS EVALUATION

4.1 Experiments Setup

In this work the used system has been an IBM BladeCenter PS701; which basically is a blade containing one socket with an 8 core IBM POWER7 running at 3.0 GHz. Although the POWER7 has two quad-channel memory controllers, the PS701 uses a single memory controller offering up to 40GB/s of bandwidth. The OS is SUSE Linux Enterprise Server 11 SP3. Applications have been compiled with Mercurium 1.99.1 source-to-source compiler using as back-end compilers IBM XL C/C++ 11.1 and IBM XL Fortran 13.1. Prefetcher instructions added by the back-end compiler have been disabled as only the dynamic mechanism within the runtime will be in charge of configuring the prefetcher. Prefetcher configurations' performance monitoring has been done by collecting hardware counters using PAPI. Because bandwidth results involve dealing with shared performance counters, perf, the original implementation from Linux kernel, has been used.

Regarding the used benchmarks, applications written in OmpSs programming model have been chosen from different sources as well as own written codes aiming to do stress tests in the system. Benchmarks are briefly explained here.

- **Dotproduct** (DP): an OmpSs implementation of a dot product of two vectors a and b with a stride K ($DP_K = \sum_i a[K \cdot i] \cdot b[K \cdot i]$). This microbenchmark was specifically created to test the hardware prefetcher in a controlled environment.
- **Jacobi**: it computes the solution of a linear system obtained from a stencil scheme via the Jacobi iterative method.
- **K-means**: it performs K-means clusterings, that is, partitions n observations into k clusters in which each observation belongs to the cluster with the nearest mean.
- **Knn**: an implementation of a machine learning non-parametric method used for classification and regression called k-nearest neighbors.
- **Specfem3D**: this application simulates a 3D seismic wave propagation in any region of the Earth based on the spectral-element method [18].
- **Heat**: it solves linear systems that come from heat distribution problems.

By means of the Dotproduct benchmark, we validate the expected behavior of the IBM POWER7 prefetcher. With a linear access pattern ($K = 1$), enabling the prefetcher halves execution time. When the stride equals the cache line size, the aggressiveness of the prefetcher is critical, obtaining 5x speedups with the deepest prefetcher with respect to disabling it. When the stride is larger than twice the size of the cache line, the SNSE bit has to be set to observe performance improvements. Finally, these benchmarks are not sensitive to the SSE bit, since they accumulate the result in a single variable. Instead, if we compute the addition of two vectors and store it in a target vector, then the SSE bit significantly improves performance when activated. In the remaining of the paper, we assume $K = 1$ for the Dotproduct benchmark.

4.2 Impact of Phases Lengths

The first step to deploy a successful adaptive technique is to evaluate the impact of phases lengths and figure out their optimal values. Exploration phases have to be wide enough to make sure that the phase is representative and thus the optimal prefetcher configurations can be extrapolated.

We tested Dotprod, Jacobi, Spefem3D and Heat with 1 and 8 threads computing for each case the relative IPC error of setting the prefetcher beforehand with respect to use the dynamic reconfiguration. So the idea is, for different lengths of exploration phases, to compute the relative error of the IPC in exploration phases with respect to executions when setting the prefetcher beforehand.

In general, results for lengths of 2, 4, 8, 16, 32 and 2500 tasks in each prefetcher configuration did not show a significant improvement in the relative error but, few applications showed a sensitive drop in the error when using lengths starting at 8 and 16 tasks. Regarding the biggest length, while sometimes benefiting from it, the error suffered from a high increment in general; that is because many OmpSs applications do not have so many task instances for some task types thus it is not possible to try all prefetcher configurations.

We did not observe a high correlation between execution times and different orders of magnitude in lengths of exploration phases. For this we decided to choose a length that allowed most of OmpSs applications to execute several times exploration phases.

4.3 Impact of Classifying Task Types

Next experiment consists in comparing performance of two versions of the dynamic mechanism: The first one classifies different task types when choosing the best prefetcher configuration, which implies gathering statistics for task types separately. The second approach treats all task types in the same way.

Separating different task types can give better results because some types may benefit more from a given prefetcher configuration whereas other task types may benefit more from a different configuration. However, this difference could not be enough to compensate an additional complexity of dealing with task types separately.

Figure 1 shows results of this experiment considering different applications and different requested thread numbers. Speedups are calculated with respect to the version that does not classify by task type. Additionally, when we apply the task type aware approach we deploy an additional optimization that consists in saving

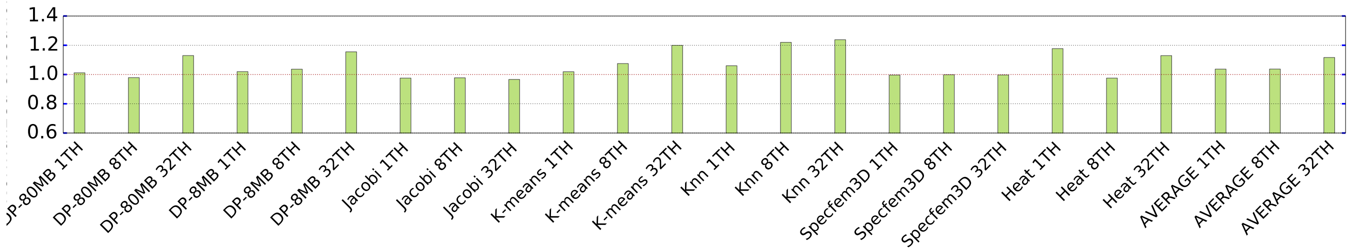

Figure 1: Performance speedup when classifying statistics per task type

the prefetcher configuration per thread in the runtime meta-data, which avoids consulting and modifying the prefecther status very often. The Dotproduct benchmark only has one task type, so the speedup observed when considering the 32 threads executions is obtained from this additional configuration. We included Dotproduct in this set of experiments to evaluate these extra benefits, which are orthogonal to the task type aware approach. The rest of applications, which have more than one kind of task, show different behaviors. While Jacobi and Specfem3D do not present speedups, Kmeans, Knn and Heat have speedups starting at parallel level of 1 thread. The classifying version of the dynamic mechanism is considered as a useful improvement since it provides significant performance benefits.

4.4 IPC Driven Power-Performance Optimization

Having determined acceptable lengths for exploration and stable phases for any OmpSs application and with a dynamic mechanism classifying performance statistics of tasks by their type, the third experiment consists in saving power by reducing the aggressiveness of the prefetcher when this one does not bring a considerable gain in performance. This is done through a configurable parameter ϵ that represents, in terms of percentages, the difference in the IPC of one aggressive configuration with respect to another one less aggressive. When the difference is smaller than the ϵ, the most aggressive configuration is not considered to be better. The method starts from the disabled prefetcher configuration, and goes through more aggressive configurations step by step.

Figure 2 shows execution slowdowns considering ϵ values of 0, 10, 20, 30, 40 and 80%. Dotproduct application results show a drop in the performance when ϵ sets an 80% of difference in the IPC. This 80% of difference in the IPC is observed when passing from prefetcher disabled to enabled in the shallowest configuration, which is the first step of increasing prefetcher aggressiveness. Jacobi presents nearly a 10% of slowdown for 1 and 8 threads configurations when setting ϵ to 10%. Specfem3D interestingly shows a drop in the performance nearly in each step that heightens ϵ, showing a strong correlation between the depth of the prefetcher and the obtained IPC.

Regarding the memory bandwidth usage, both Dotproduct and Specfem3D show a reduction that is consistent with the performance drop, meaning in these cases the applications fully exploit the extra bandwidth used by the prefetchet. Jacobi, knn and Heat do not show a consistent reduction in the used bandwidth and they neither suffer performance slowdowns when choosing less aggressive configurations. Finally, K-means application does not suffer from performance slowdowns in the execution time although the bandwidth usage gets significantly reduced when ϵ increases. Therefore, in this particular application, the adaptive mechanism successfully selected the less aggressive prefecther configuration that provides maximum performance, avoiding the spending of useless memory bandwidth.

5. RELATED WORK

There have been many works that have dealt with data prefetch [1, 10, 17]. First attempts were based on sequential prefetchers, this approach suggests to prefetch memory blocks sequentially. Despite being effective in these cases, this solution is not able to yield performance when the application does not follow a sequential data access pattern. Due to this, further research in prefetchers was done to try to capture the non-sequential nature of those applications. Prefetch techniques aimed to deal with pointer-based applications have been studied [6, 20, 25]. Solihin et al. [21] made use of a user-level memory thread to do prefetching, getting in the applications with irregular accesses significant speedups. Joseph and Grundwald [9] worked on Markov-based prefetchers. Although most of these works about prefetching have not been put into practice with real processors, limit studies and prefetch analytical models have been proposed [7, 22].

A further step in data prefetching is to consider the interaction between threads that take place in the CMP processors. Ebrahimi et al. [13] and Lee et al. [12]

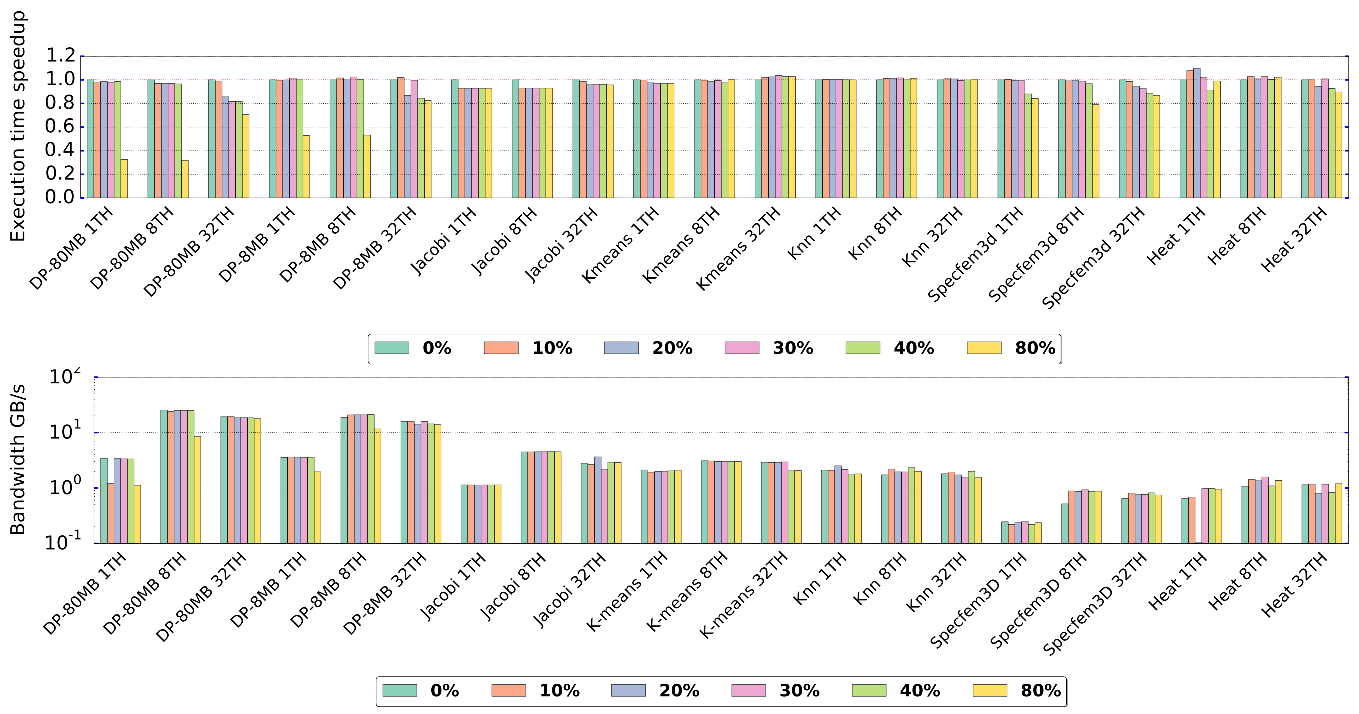

Figure 2: Performance speedup and bandwidth when ϵ increases

study the effect of thread-interaction on prefetch and design prefetch systems that improve throughput and fairness. Liu and Sohilin [15] present a study about the impact prefetching has and bandwidth partitioning in CMPs. Although there are many sutudies about data prefetching on top of simulators, there are very few works that make use of real processors. For instance, Wu and Martonosi [23] characterize the prefetcher of an Intel Nehalem processor and provide a straightforward algorithm that can control dynamically the activation and deactivation of the prefetcher. Nevertheless, their work only contemplates intra-application cache interference obviating actual system performance. Liao et al. [14] build a machine learning model that dynamically modifies the prefetch configuration of the machines in a data center (based on Intel Core2 processors). Their work also bases its approach on turning on and off the prefetcher.

Beyond enabling and disabling the prefetcher, there are other kind of works targeted to control thread execution rate. For example, playing with fetch policies within a SMT processor has been studied [3, 4]. They aim to increase throughput and/or provide quality of service (QoS). In the same line, the work of Boneti et al. [2] study the usage of the dynamic hardware priorities in the IBM POWER5 processor aiming to yield performance from resource balancing and prioritization. Qureshi and Patt [19] study how to improve throughput through solving the problem of partitioning the last-level cache for multiple applications. Moretó et al. [16] show a similar solution based on achieving QoS for multiple applications running at the same time.

6. CONCLUSIONS

Contemporary microprocessors are being designed with reconfigurability features and increasingly more capable of counting different events by means of hardware counters. In this paper, a portable solution implemented within a runtime smartly reconfigures the hardware prefetcher making use of hardware counters. A dynamic mechanism makes the process of reconfiguration automatic. Once it has enough collected performance data from different configurations, it calls to an algorithm that is in charge of determining which one is the most power-performance efficient. This process is repeated with a given timing during the application execution. A series of experiments have shown that sensitivity in performance is nearly negligible when collecting great amounts of data from performance counters; this can be attributed to the fact that few OmpSs tasks contain performance data that turns out to be representative enough. Additionally, OmpSs task types classification has a positive impact in performance because different OmpSs task types may benefit from different prefetcher configurations as task types may determine different kinds of workloads in the machine. Finally, a proposal for saving power in the cases in which aggressive prefetcher configurations do not come with a substantial speedup has proved to be potentially useful and reaped good results. The underlying idea is to set an IPC percentage threshold that limits the aggressiveness of chosen prefetcher configurations.

Acknowledgements

This work has been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557, the HiPEAC Network of Excellence, by the European Research Council under the European Union's 7th FP, ERC Grant Agreement n. 321253, and by a joint study agreement between IBM and BSC (number W1361154). Miquel Moreto has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship number JCI-2012-15047. With the support of the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (Contract 2013 BP_B 00243)

7. REFERENCES

[1] J. Baer and T. Chen. An effective on chip preloading scheme to reduce data access penalty. *In SC*, pages 176–186, 1991.

[2] C. Boneti, F. Cazorla, R. Gioiosa, A. Buyuktosunoghu, C. Y. Cher, and M. Valero. Software-controlled priority characterization of POWER5 processor. *In ISCA*, pages 415–426, 2008.

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

[4] S. Choi and D. Yeung. Learning-based smt processor resource distribution via hill-climbing. *In ISCA*, pages 239–251, 2006.

[5] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parall. Proc. Lett.*, 21(2):173–193, 2011.

[6] E. Ebrahimi, O. Mutlu, and Y. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hibrid prefetching systems. *In HPCA*, pages 7–17, 2009.

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

[8] 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 PACT*, pages 137–146, 2012.

[9] D. Joseph and D. Grundwald. Prefetching using markov predictors. *In ISCA*, pages 252–263, 1997.

[10] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *In ISCA*, pages 364–373, 1990.

[11] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. POWER7: IBM's next-generation server processor. *IEEE MICRO*, 30(2):7–15, 2010.

[12] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware dram controllers. *In MICRO*, pages 200–209, 2008.

[13] E. C. J. Lee, O. Mutlu, and Y. Patt. Prefetch-aware shared resource management for multi-core systems. *In ISCA*, pages 141–152, 2011.

[14] 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 SC*, pages 1–10, 2009.

[15] F. Liu and Y.Solihin. Studying the impact of prefetching and bandwidth partitioning in chip-multiprocessors. *In SIGMETRICS*, 2011.

[16] M. Moretó, F. Cazorla, A. Ramirez, R. Skellariou, and M.Valero. FlexDCP: a QoS framework for CMP architectures. *SIGOPS Oper. Sys. Rev.*, 43(2):86–96, April 2009.

[17] S. Palacharla and R. Kessler. Evaluating stream buffers as secondary cache replacement. *In ISCA*, pages 24–33, 1994.

[18] A. T. Patera. A spectral element method for fluid dynamics: Laminar flow in a channel expansion. *Journal of Computational Physics*, 54:468–488, 1984.

[19] M. K. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *In MICRO*, pages 423–432, 2006.

[20] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linkded data structures. *In ASPLOS*, pages 115–126, 1998.

[21] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. *In ISCA*, pages 171–182, 2002.

[22] V. Srinivasan, E. S. Davidson, and G. S. Tyson. A prefetch taxonomy. *IEEE Trans. Comp.*, 53(2):126–140, February 2004.

[23] C.-J. Wu and M. Martonosi. Characterization and dynamic mitigation of intra-application cache interference. *In ISPASS*, pages 2–11, 2011.

[24] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

[25] C. Yang and A. Lebeck. Push vs pull: Data movement for linked structures. *In ICS*, pages 176–186, 2000.