Adaptive Runtime-Assisted Block Prefetching on Chip-Multiprocessors

Victor Garcia[1,2], Alejandro Rico[1,2], Carlos Villavieja[3], Paul Carpenter[2], Nacho Navarro[1,2], and Alex Ramirez[1,2]

[1] Universitat Politecnica de Catalunya, Barcelona, Spain
[2] Barcelona Supercomputing Center, Barcelona, Spain
[3] University of Texas at Austin, Austin, TX, USA

Abstract. Memory stalls are a significant source of performance degradation in modern processors. Data prefetching is a widely adopted and well studied technique used to alleviate this problem. Previous solutions range from hardware controlled prefetchers to software-based prefetchers. Among the latter we find a variety of schemes, including runtime-directed prefetching and more specifically runtime-directed block prefetching.

This work proposes a hybrid prefetching mechanism that integrates a software driven block prefetcher with existing hardware prefetching techniques. Our runtime-assisted software prefetcher brings large blocks of data on-chip with the support of a low cost hardware engine, and synergizes with existing hardware prefetchers that manage locality at a finer granularity. The runtime system that drives the prefetch engine dynamically selects which cache to prefetch to.

Our evaluation on a set of scientific benchmarks obtains a maximum speed up of 32% and 10% on average compared to a baseline with hardware prefetching only. As a result, we also achieve a reduction of up to 18% and 3% on average in energy-to-solution.

1 Introduction

Modern high-performance processors incur large latencies in accessing off-chip main memory, causing CPU stalls and reducing performance [12, 7]. Many processors include latency hiding mechanisms to reduce the number of stall cycles; examples include non-blocking caches, out-of-order execution and data prefetching. The size of on-chip memories keeps increasing, but current memory hierarchies still work at the granularity of a cache line. This is problematic for software-based prefetching mechanisms because one prefetch instruction must be executed per cache line requested, adding significant instruction overhead [2].

Block prefetching is a good solution to this problem. Some proposals rely on compiler analysis [5], others on manual insertion of prefetch directives in the code [1] and others use a runtime system to guide the prefetch engine [8]. While all approaches are valid, compiler analysis is still limited, and manually inserting prefetch instructions in the code is difficult and time-consuming. Using a runtime system to guide prefetching, on the other hand, is a simple and efficient way of performing block prefetching. A runtime system can see further into the future than current compilers are able to, has dynamic information of the application and requires minimal user intervention.

This work presents a hybrid prefetching scheme that integrates a runtime-assisted block prefetcher with existing prefetching mechanisms. The runtime system guides a prefetch engine in bringing large blocks of data on-chip. Once the data is on-chip, hardware and software-based prefetching mechanisms manage locality at cache line granularity by bringing data closer to the CPU.

The runtime system leverages its information about application schedule to decide when to start prefetching. In addition, it compares the task input data and cache sizes to dynamically select the best prefetch destination for each task.

The main contributions of this work are:

- A new block prefetcher guided by the runtime system that integrates with existing hardware prefetchers to effectively reduce memory access time.
- A mechanism that uses runtime schedule and cache information to dynamically decide when to prefetch and which cache to prefetch to.
- An implementation of a hardware block prefetch engine called Multi-core Data Transfer Engine (MDTE).

2 Background And Motivation

Traditional software and hardware prefetching techniques work at a cache line granularity. This is especially problematic for software-based prefetchers, where an additional instruction must be executed per cache line requested. The effect on the instruction cache and the resulting overhead caused by these prefetch instructions can be significant [2]. In order to maximize memory bandwidth and avoid unnecessary overheads, it is more beneficial to use block transfers than to work at a cache line basis [5]. In addition, it allows for better overlapping of data transfer and computation.

Previous block prefetching proposals have relied either on the compiler or on the programmer to insert prefetch instructions in the code. Manually inserting prefetch instructions is time consuming and error prone, while compilers require complex program analysis and lack any form of dynamic feedback. We argue that in contrast, runtime-assisted prefetching is the simplest and most effective way of performing block prefetching.

Using a runtime system to guide the prefetch engine has multiple advantages, specially those found on task-based programming models that follow a data-flow model. First, it requires minimal user intervention and does not rely on complex compiler analysis. Second, if the runtime system has knowledge of *what* data is accessed by each task, it can prefetch that data without speculation, decreasing cache pollution. Third, the runtime system is in charge of scheduling tasks. Having knowledge of the execution flow simplifies the timeliness considerations of prefetching, since it is known *when* and *where* the data is required. Fourth, if the runtime system has knowledge of the data used by each task and it is provided a map of the cache hierarchy, it can dynamically choose which cache level to use as a destination for the prefetched data, prefetching always as close as possible to the processing elements without evicting useful data.

3 Related Work

The benefit of prefetching large blocks of data instead of individual cache lines was first noted by Gornish et al. [5]. In their approach, the compiler performs static program dependence analysis on array references in nested loops, inserting a block prefetch command before the data is referenced. Our proposal, in contrast, exploits the runtime system's knowledge of the upcoming task schedule to control the block prefetcher, and it is not restricted to nested loops.

Wall [11] presented a study on the effect of different code optimizations on the memory subsystem, including software block prefetching using the MOV instruction. This approach requires the programmer to insert MOV instructions by hand, and, as the author found, in some cases it may not work well with other compiler optimizations.

ARM includes a block prefetcher in their Cortex-A8 and Cortex-A9 processors [1]. Their Preload Engine, as it is named, allows the user to load selected regions of memory into the L2 cache. The Preload Engine expects the programmer to add load directives by hand, requiring a good understanding of the code and some knowledge of the underlying architecture. ARM's Preload Engine is attached to the cores, and is only able to direct the data transfers to the last level L2 cache. By targeting a task-based programming model we simplify this process, leaving the decisions to the runtime system that is able to dynamically decide when to initiate the prefetch and where to prefetch into.

Papaefstathiou et al. [8] also propose software prefetching for task-based programming models. However there are several differences with our approach. First, whereas their proposal is an alternative to traditional hardware prefetchers, we propose a hybrid hardware-software prefetching scheme where software prefetching complements the hardware prefetch engines found in most processors nowadays. Second, whereas Papaefstathiou et al. evaluate their approach using a simple in-order processor, our evaluation uses an advanced out-of-order processor that can hide on itself some memory latency. We therefore establish that the approach is also applicable to high-performance processors implementing aggressive instruction-level parallelism techniques. Third, they propose a prefetch engine per core, while our proposed hardware engine (MDTE) may be shared by multiple cores, reducing chip area and power consumption. Finally, while their approach prefetches only to the Last Level Cache, we believe a key aspect of runtime-assisted prefetching is leveraging all the information the runtime system has by letting it dynamically choose the prefetch destination.

4 Runtime-Assisted Block Prefetching

This section describes the implementation details of the runtime-assisted block prefetcher, as well as the accompanying hardware support, the Multi-core Data

Transfer Engine (MDTE). We also introduce the multi-core architecture targeted in this work.

4.1 Target Architecture

Figure 1a shows a high-level overview of the target architecture, a Chip Multiprocessor with a unified shared address space. All cores have private L1 and L2 caches and are connected through a crossbar to a shared Last Level Cache/L3 (LLC). The MDTE can be placed next to a core's L2 or the shared LLC. We let the runtime system decide which one to use in every case.

4.2 Prefetch Commands

Prefetch commands are simple instructions that reference a contiguous block of memory. They contain a starting address and data size. They are generated by the runtime system based on a task's input data and have unrestricted length. These commands initially contain logical addresses, but the physical pages they map to may not be contiguous in memory and therefore are later split at page boundaries. Splitting prefetch commands and the address translation is performed in the MDTE (see Section 4.4 for details).

4.3 ISA extensions

In order to enable the runtime system to issue prefetch commands we extend the ISA with the following user mode instruction:

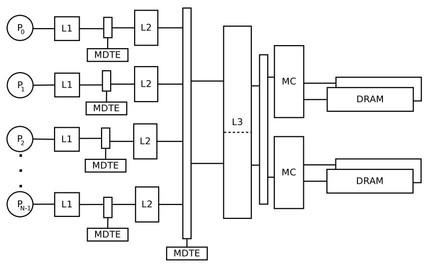
prefetchX r1, r2

r1 is the register holding the base address of the block to be prefetched, r2 is the register holding the size of the block in bytes, and X takes the value of the cache level to which the prefetch command is to be sent. In this manner, the instruction *prefetch2 r1, r2* would send a prefetch command with the address in r1 and the size in r2 to the MDTE corresponding to the core's L2 cache. In our target architecture one bit in the instruction word is enough to specify whether the prefetch instruction is directed to the L2 or the L3.

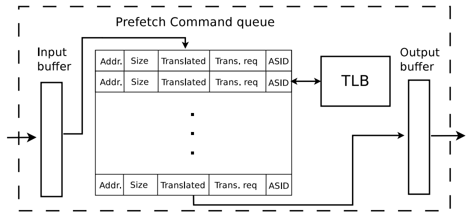
4.4 MDTE Architecture

The MDTE is a programmable DMA-like controller that receives and processes the prefetch commands generated by the runtime system. It does not require any modification to existing caches. Figure 1b shows its design. The main components are:

– An input buffer to store the received prefetch commands until they are queued.



(a) Targeted multi-core architecture.



(b) MDTE components

– A prefetch command queue where commands are inserted in FIFO order. Each command in the queue can prefetch up to one memory page. Each entry in the queue holds the starting address, size, address space identifier (ASID), a translated bit and a translation requested bit.
– A Translation Lookaside Buffer (TLB) to speed up address translation.
– An output buffer storing translated commands until they are sent to memory.

The MDTE reads the input buffer for new commands. When a new command is received, it is split into page aligned commands and enqueued in the prefetch command queue. New commands are discarded when the queue is full. The commands received contain logical addresses that need to be translated. There are two main advantages to delaying the translation until the command arrives at the MDTE: First, if address translation were to be done at the core's MMU, a prefetch command for a big block of data (e.g. a few megabytes) would be split into a large number of page-sized prefetch commands. These would have to travel to the corresponding MDTE, increasing traffic on the interconnect and reducing available bandwidth. Second, address translation at the MMU's is in the critical path. The additional translations would delay the translation of demand requests, further degrading performance.

The MDTE contains a TLB to speed up address translation and reduce the traffic caused by the translation requests. The impact of adding these TLBs is not significant since they need not be very large (see Table 1). We also use a TLB directory to minimize the overhead of TLB shootdowns [10].

Once a translation response is received, the prefetch command is updated and moved to the output buffer. Commands from the output buffer are sent to their target cache where they are issued one cache line at a time in round robin fashion.

4.5 Prefetch Consideration: Timeliness

An important aspect of any prefetch mechanism is deciding *when* to issue a prefetch request. In our implementation, prefetching for a task is triggered right before the execution of the preceding task begins, in the following manner: when task A completes, the core executes the runtime scheduler to obtain the next two ready tasks B and C. The core then executes the instruction to prefetch the inputs of task C, an operation that represents an overhead in the order of tens of assembly instructions and is negligible compared to the cost of running the scheduler algorithm. After executing the prefetch instruction, the core begins executing task B while the data for task C is being prefetched, successfully overlapping data movement with computation. At that point task C is pinned to the hardware thread executing task B, disabling work stealing and guaranteeing that task C will be scheduled to execute on the core whose caches hold the prefetched data. By doing so the runtime system implicitly applies an affinity-based scheduling policy, allowing for simpler scheduler algorithms.

4.6 Prefetch Consideration: Destination

Another important aspect to determine is *where* to send the prefetch commands to, i.e., the prefetch destination.

It is always desirable to allocate the prefetched data as close to the processor as possible without affecting the performance of the current task. Although the runtime system does not know exactly the content of each cache, it has knowledge of the input data used by each task. Using that information it is able to approximate where the prefetched data can be placed without evicting the working set of the current task. The runtime system can then dynamically decide the best prefetch destination before issuing the prefetch command.

We initially attempt to prefetch data into the private L2 cache (L1 caches are too small for block prefetching). Once the runtime system estimates the L2 cache cannot hold more data without evicting the current task's working set, we direct the remaining prefetch commands to the shared MDTE.

4.7 Coordinating Hardware, Software Prefetch and Demand Loads

The main goal of our mechanism compared to previous work is to bring data on-chip at a coarser granularity (blocks vs cache line) with the help of the runtime system, and combine it with other traditional hardware and/or software prefetching mechanism to move data closer to the core, i.e. L1 or L2 caches.

Unfortunately, prefetching has potentially a high cost in terms of bandwidth usage and network contention, specially if multiple and simultaneous prefetching mechanism are used. Throttling policies [3] can be used to coordinate them, slowing or even stopping completely one of the prefetch engines in order to maintain fairness or avoid contention on shared resources.

We take into account some priority considerations to ensure that requests in the critical path are always processed first. Demand requests generated by the CPU are always prioritized over prefetch requests. This ensures no prefetch instruction will delay a CPU request. Also, software prefetches are not as time sensitive as hardware prefetches, since the data prefetched is only required for the next task which is usually hundreds of thousands or millions of cycles in the future. Hardware prefetch engines predict future accesses and generate requests for data that will be needed in the near future, and therefore are prioritized over the runtime-generated prefetches.

5 Evaluation Methodology

We use a trace-driven cycle-accurate simulator that models the timing of an out-of-order processor, cache hierarchy, interconnection network and the off-chip memory [9]. Our simulation framework uses the dynamic binary instrumentation tool Pin [6] to obtain the traces. The out-of-order cores are configured with a reorder buffer of 128 entries. The configuration parameters of the cache hierarchy are shown in Table 1. The cache line size is 128 bytes divided into 16 sub-blocks of 8-bytes each for all cache levels. All caches are inclusive, non-blocking and implement an LRU replacement policy. The bandwidth of all on-chip network links is 8 bytes per cycle with a latency of 3 cycles. The MDTEs are implemented as described in Section 4.4 and configured using the parameters shown in Table 1. For energy estimations we use CACTI version 6.5 with the memory parameters specified in Table 1, and technology parameters based on ITRS predictions for a 32nm technology.

Table 1: Memory hierarchy configuration parameters

Parameter	Value	Parameter	Value
Cache (L1/L2/L3)		DRAM DIMM	
Size (KB)	32/256/2048 per core	autoprecharge	disabled
Latency (cycles)	2/12/45	data rate (MT/s)	1600
Associativity	2/8/16	bursts per access	8
MSHR entries	8/32/8 per core	^{t}RCD,^{t}RP,CL,^{t}RC,^{t}WR,^{t}WTR	[1]
MDTE (L2/L3)		Memory Controller	
TLB size	16/16	Access queue size	128
Prefetch queue size	256/1024	Number of DIMMs	4

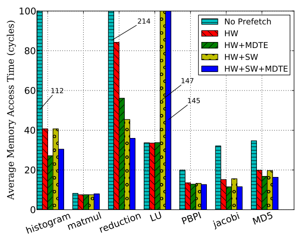
Workloads The proposed hybrid prefetching mechanism targets scientific codes with regular data structures such as those found in high performance computing. These applications can benefit both from our runtime directed software prefetching and from hardware-based prefetching techniques. In order to evaluate our proposal we use a set of scientific benchmarks including PBPI, a parallel implementation of Bayesian phylogenetic inference method for DNA sequence data [4], an implementation of the MD5 hashing algorithm, and a set of kernels representing algorithms commonly found on scientific applications: histogram, matrix multiplication, vector reduction, LU decomposition of a sparse matrix and an implementation of a jacobi method. All applications were compiled for x86-64 with the GCC compiler version 4.6.3 using the -O3 optimization flag.

6 Experimental Evaluation

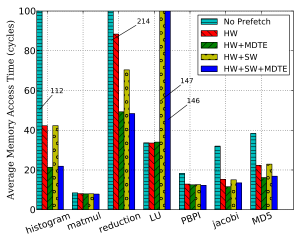
6.1 Hardware Prefetchers

We first explored the effectiveness of the standalone hardware prefetchers for each of the benchmarks. We evaluated a next-line configuration that prefetches

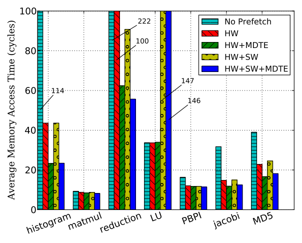
[1] DRAM timing parameter values match the Micron DDR3-1600 specification.



(c) 4 cores

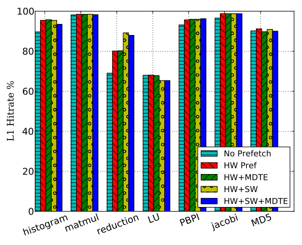


(d) 8 cores

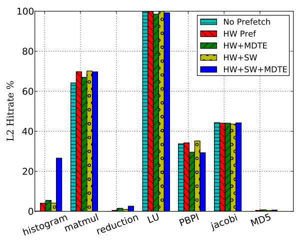


(e) 16 cores

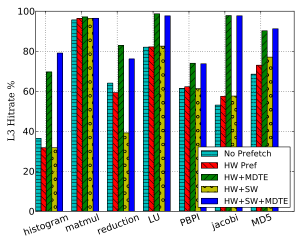
Fig. 1: Average memory access time for 4, 8 and 16 cores and multiple prefetch configurations



(a) L1 hit rate



(b) L2 hit rate



(c) L3 hit rate

Fig. 2: Cache hit rates for the execution with multiple prefetch configurations

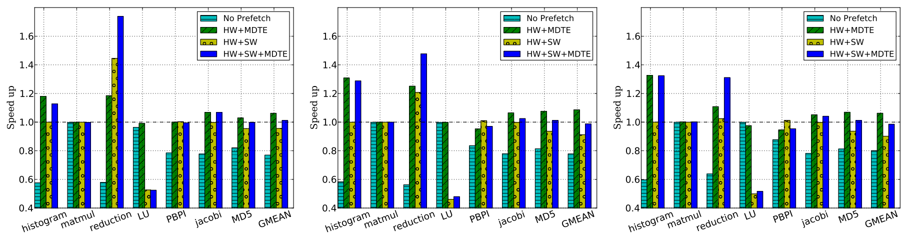
the next N lines after a cache miss, as well a reference prediction table based stride prefetcher. We tested different configurations of prefetch degree, and found N=2 to be optimal for both prefetchers.

For the rest of this section, we use the best standalone hardware prefetch configuration as the baseline for each benchmark. This configuration is labelled as “HW” on the figures.

6.2 Compiler Based Software Prefetching

We also evaluate our proposal against other traditional software prefetching techniques by compiling every benchmark with the GCC flag *-fprefetch-loop-arrays*. With this optimization the compiler attempts to insert ISA specific prefetch instructions in loops that traverse large data arrays.

We first execute the benchmarks compiled with the prefetch flag in conjunction with every hardware prefetcher and select the best performing; this configuration is labelled on the figures as “HW+SW”. Then we take this configuration and run it with the proposed runtime-assisted block prefetcher (labelled as “HW+SW+MDTE”).



(a) 4 cores (b) 8 cores (c) 16 cores

Fig. 3: Application speed up normalized to the execution with the best hardware prefetcher standalone

6.3 Performance Analysis

Average Memory Access Time Figure 1 shows how for six of the seven benchmarks the MDTE is able to reduce the Average Memory Access Time (AMAT). As expected, applications that display a high AMAT (even with hardware prefetching) benefit more from our software block prefetcher. In particular, *jacobi*, *MD5*, *reduction* and *histogram* obtain on the 8 cores configuration an AMAT reduction of 18%, 28%, 48% and 49% respectively over executions with the best hardware prefetching configuration only. On the other hand, the benefit obtained by our hybrid scheme is limited to a 5% AMAT reduction for *PBPI*. The reason is that the AMAT for this application is already very low (20 cycles) with no prefetching mechanism, and it is even further reduced to 14 cycles by the hardware prefetcher. Since the latency of our L2 caches is 12 cycles and we model out-of-order cores that can hide some of that latency, the benefit attainable is very limited.

Cache Hit Rates The cache hit rates shown in Figure 2 explain why *matmul* barely obtains any AMAT reduction and *LU* slightly increases it. Our implementation of matrix multiply uses blocking and the BLAS library. These commonly used optimizations fully exploit the size of the L1 cache, obtaining 99.9% L1 hit rate. *LU* factorization also uses blocking, with a block size of 128 KB that fits comfortably in the L2 cache. Prefetching provides no additional benefit after the initial cold state of the caches, and can even hurt performance by causing additional contention on the interconnection network and on the memory controllers, as is the case for *LU*. Nevertheless, L3 cache hit rate is significantly increased in all benchmarks with our hybrid approach compared to the execution with only the baseline hardware prefetcher, reducing memory access time whenever an application does not display such high L1 or L2 hit rates.

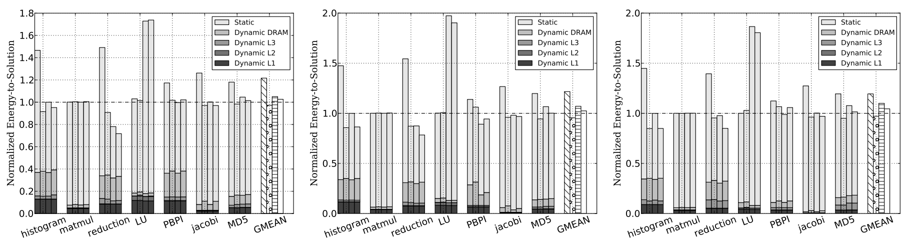
Performance Evaluation Figure 3 shows the speed up over the execution with the best hardware prefetch configuration standalone. On the 4 core system, our hybrid hardware + MDTE configuration obtains a 19% speed up over execution with the best hardware prefetcher standalone for *histogram* and *reduction*, and over 2x compared to execution with no prefetch. *jacobi* achieves a 7% speed up while *PBPI* does not improve over hardware prefetching only. *matmul* and *LU* do not obtain any benefit out of software block prefetching, with a slight performance degradation on *LU*. *reduction* obtains an even larger speed up when the compiler inserts prefetch instructions, reaching almost an 80% increase on the configuration with the best hardware prefetcher, compiler-inserted prefetch instructions and our proposed block prefetcher working together. This is due to the large L1 hit rate increase caused by the compiler-inserted prefetches. On average, the hybrid hardware + MDTE configuration obtains an 8% speed up over the baseline. Although the configuration including compiler-inserted prefetch instructions may perform best in some benchmarks, in others such as *LU* the performance drop is considerable, and overall the best results are obtained with hardware prefetching + MDTE.

On a system with 8 cores we double the number of L3 banks and memory controllers. In this context our hybrid prefetching scheme shines obtaining a 30% and 25% speed up in *histogram* and *reduction* respectively, with an average of 10% for all benchmarks. The configuration with compiler-inserted prefetch instructions experiences a large drop on the speed up observed on *reduction* with the 4 core configuration. The additional traffic caused by these prefetch instructions saturates the interconnect network and memory controllers, diminishing the benefits obtained. *PBPI* suffers a small performance degradation because, as explained before, block prefetching does not provide any benefit over an already low AMAT, and because, as in the case of *LU*, the overhead caused by the prefetch requests travelling through the memory subsystem is non-negligible.

These results are maintained on the 16 core configuration with one exception: *reduction* loses about 10% performance gain with our hybrid HW + MDTE configuration. The reason is that the LLC saturates with the increased number of requests and our throttling mechanism stops all prefetching. More complex throttling policies could be applied to lessen the impact of the increased traffic, and are left for future work.

The performance results acknowledge the hypothesis of this work: the runtime-assisted MDTE brings data on-chip in advance (as confirmed by the increased L3 hit rates), and the hardware prefetcher brings the data closer to the cores (hit rates in L1 and L2 are kept). The synergy between the MDTE and the stock hardware and software prefetchers translates into the increased performance shown in Figure 3.

Energy Consumption Figure 4 shows energy-to-solution for every benchmark. We see how energy consumption is dictated primarily by static power, and therefore by execution time. The increase in power caused by the MDTEs has been included in the dynamic power of the cache level they are attached to, i.e., L2 for the private MDTEs and L3 for the shared. The speed ups obtained using our hybrid prefetching scheme translate into energy-to-solution gains of 3% on average for all benchmarks. On all but two benchmarks we consume less energy by using our hybrid scheme compared to hardware prefetching only. On the best performing benchmark, *reduction* with an 8 core configuration, we obtain an 18%



(a) 4 cores (b) 8 cores (c) 16 cores

Fig. 4: Energy consumption normalized to the execution with the best hardware prefetcher standalone. From left to right for each benchmark: no prefetch, hardware + MDTE prefetch, hardware + software prefetch, hardware + software + MDTE prefetch.

decrease in energy-to-solution compared to the best hardware prefetch configuration standalone. *PBPI* and *LU* see an slight increase in energy consumption of 4% and 2% respectively due to an increase in execution time.

7 Conclusions

In this work we propose a hybrid hardware and software block prefetching scheme. We have demonstrated that by using a runtime system to guide a block prefetch engine we increase L3 cache hit rates and therefore reduce large off-chip access latencies. This approach is simpler and more robust than manually inserting prefetch instruction in the code or relying on complex compiler analysis. For best results, we combine our runtime-guided block prefetcher with other traditional hardware and software prefetching techniques that manage locality at cache line granularity, moving the data closer to the CPU and increasing L1 and L2 cache hit rates. We apply throttling mechanisms to coordinate the prefetchers and reduce the overhead caused by the prefetch engines.

By using a runtime system with knowledge of the upcoming task schedule and accessed data, we prefetch only data that will be used, avoiding cache pollution. In addition we let the runtime system leverage this information to dynamically make decisions such as prefetch destination and timeliness. Our proposal benefits memory-sensitive applications and does not harm compute-bound applications.

The evaluation on a set of scientific workloads shows that our hybrid prefetching scheme is able to obtain up to 32% performance improvement with an average of 10% compared to the execution with hardware prefetching only. The performance benefits offset the increased power from the extra hardware and the increase in dynamic power caused by prefetch activity, leading to a reduction of up to 18% with an average of 3% in energy-to-solution.

Bibliography

[1] ARM. Cortex-a9 technical reference manual, 2008.

[2] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. ISCA '94, pages 223–232, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[3] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *SIGARCH Comput. Archit. News*, 38(1):335–346, Mar. 2010.

[4] X. Feng, K. W. Cameron, and D. A. Buell. Pbpi: a high performance implementation of bayesian phylogenetic inference. SC '06, New York, NY, USA, 2006. ACM.

[5] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *In International Conference on Supercomputing*, pages 354–368, 1990.

[6] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[7] M. R. Martonosi. Analyzing and tuning memory performance in sequential and parallel programs. Technical report, Stanford, CA, USA, 1994.

[8] V. Papaefstathiou, M. G. Katevenis, D. S. Nikolopoulos, and D. Pnevmatikatos. Prefetching and cache management using task lifetimes. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 325–334, New York, NY, USA, 2013. ACM.

[9] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero. On the simulation of large-scale architectures using multiple application abstraction levels. *ACM Trans. Archit. Code Optim.*, 8(4):36:1–36:20, Jan. 2012.

[10] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. PACT '11, pages 340–349, Washington, DC, USA, 2011. IEEE Computer Society.

[11] M. Wall. Using block prefetch for optimized memory performance, 2001.

[12] 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.