International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

Adaptive Runtime-Assisted Block Prefetching on Chip-Multiprocessors

Victor Garcia · Alejandro Rico · Carlos Villavieja · Paul Carpenter · Nacho Navarro · Alex Ramirez

the date of receipt and acceptance should be inserted later

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. Prefetching can be performed by the hardware, or be initiated and controlled by software. Among software controlled prefetching we find a wide variety of schemes, including runtime-directed prefetching and more specifically runtime-directed block prefetching.

This paper 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.

Keywords cache memories, prefetch, task based programming models

Victor Garcia, Nacho Navarro
Universitat Politecnica de Catalunya, Barcelona, Spain
E-mail: vgarcia@ac.upc.edu

Victor Garcia, Alejandro Rico, Paul Carpenter, Nacho Navarro
Barcelona Supercomputing Center, Barcelona, Spain

Carlos Villavieja
Google Inc.

Alex Ramirez
NVIDIA Corporation

1 Introduction

Modern high-performance processors incur large latencies in accessing off-chip memory, causing CPU stalls and reducing performance [36, 23]. 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. Although the size of on-chip memories keeps increasing, current memory hierarchies continue working 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 [7].

Block prefetching is a good solution to this problem. Some proposals rely on compiler analysis [18], others on manual insertion of prefetch directives in the code [1] and others use a runtime system to guide the prefetch engine [26]. 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.

In particular, the runtime systems of task-based programming models provide a perfect opportunity for dynamic block prefetching. On these programming models computation is divided into tasks that can be executed concurrently. The runtime system knows exactly when a task is going to execute, the data that it is going to access (as specified by the user, see Section 2.3) and the CPU on which it will be scheduled. The runtime system is therefore able to perform data prefetching while minimizing (if not completely avoiding) commonly associated problems such as prefetch mispredictions and cache pollution.

This paper 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 on-chip large blocks of data. Once the data is on-chip, other prefetching mechanisms are used to 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

This section discusses the motivation for block prefetching and explains why it is best performed by a runtime system.

2.1 Block Prefetching

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 [7]. 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 [18]. Transfering larger blocks of data allows also 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.

2.2 Runtime-Directed Prefetching

Using a runtime system to guide the prefetch engine has multiple advantages, specially those found on task-based programming models (see Section 2.3).

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 only that data without speculation, decreasing cache pollution. Third, the runtime system is in charge of scheduling work. 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. The advantage of this approach is two-fold: it brings the data as close to the processing elements as possible, and it also guarantees that no data of the current task will be evicted by the prefetched data.

2.3 Task-Based Programming Models

In a task-based programming model the programmer divides the work into multiple tasks that can be executed concurrently. These task are enqueued into a task queue from where they are pulled by the runtime system and scheduled

```
#pragma omp task in(a, b) inout(c)
void sgemm_t(float a[M][M], float b[M][M],
             float c[M][M]);

#pragma omp task inout(a)
void spotrf_t(float a[M][M]);

#pragma omp task in(a) inout(b)
void strsm_t(float a[M][M], float b[M][M]);

#pragma omp task in(a) inout(b)
void ssyrk_t(float a[M][M], float b[M][M]);

-----------------------------------------

float A[N][N][M][M]; // NxN blocked matrix,
                     // with MxM blocks
for (int j = 0; j<N; j++) {
   for (int k = 0; k<j; k++)
      for (int i = j+1; i<N; i++)
         sgemm_t(A[i][k], A[j][k], A[i][j]);

   for (int i = 0; i<j; i++)
      ssyrk_t(A[j][i], A[j][j]);

   spotrf_t(A[j][j]);

   for (int i = j+1; i<N; i++)
      strsm_t(A[j][j], A[i][j]);
}
```

Fig. 1: Example code for a task-based

based on the available resources. In this manner, the runtime system can see the *future* simply by looking at the task queue, and hence can effectively perform block prefetching for the upcoming tasks.

Cilk [17], OpenMP [9], Sequoia [15], OmpSs [13], StarPU [2], X10 [6], Chapel [5] and Intel TBB [27] are examples of task-based programming models. Some of these allow the programmer to specify additional information for each task, such as the device where it can run or the input and output data used [9, 13, 2]. Figure 1 shows a code snippet of a Cholesky factorization programmed in a task-based programming model. Pragma annotations are used to identify and declare tasks. The keywords **in**, **out** and **inout** are used to specify input and output dependencies, corresponding to the read-only, write-only and read-write task data respectively. This information allows the runtime system to prefetch only data that is known to be needed. In addition, data locality can be better exploited by taking informed scheduling decisions.

3 Related Work

Hardware and software prefetching techniques have been studied extensively [10, 33, 11, 25, 24, 31, 4]. Hardware-controlled prefetchers are highly effective for applications with regular data access patterns [4]; they have been integrated into all modern high-performance processors, including Intel Core i3/i5/i7, AMD Opteron and IBM POWER, and many embedded and mobile processors, such as ARM's Cortex-A9 and Cortex-A15.

Most software-based prefetchers require executing one prefetch instruction per cache line prefetched, adding a non-negligible overhead and straining the instruction cache [7]. The benefit of prefetching large blocks of data instead of individual cache lines was first noted by Gornish et al. [18]. 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 [35] 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.

Lu et al. [21] propose a dynamic optimization system that uses hardware profile information gathered at run-time to dynamically insert software prefetch instructions in the code. They take into account the variability and impact of micro-architectural constraints and memory behavior on the performance and effectiveness of software prefetching. Even with this dynamic behavior, their proposal relies on data access pattern detection. The speculative nature of this approach can be more error-prone, as the runtime system may incorrectly insert prefetch directives for data that will not be used, with all the negative effects it can cause such as additional contention in the interconnect and cache pollution. In addition, the performance monitoring and dynamic recompilation adds significant overhead. In our approach, the runtime system only prefetches data that is declared to be an input of a task, and so it will never fetch data that is not needed.

Papaefstathiou et al. [26] also propose a software prefetching and cache management mechanism 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 brings data on-chip to hide the large DRAM latencies, and hardware prefetching moves the data closer to the processing units. We evaluate multiple hardware prefetching configurations and perform an extensive design space exploration of parameters such as prefetch degree and distance, and evaluate the best configuration on each case running in conjunction with our proposed software prefetching technique. 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 where there is lower benefit from additional prefetching. 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. Additionally, grouping prefetch commands in a common engine allows for the coordination of priorities among the cores, and also allows us to introduce effective throttling mechanisms. 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

Previous runtime-assisted block prefetching proposals target simple in-order cores [26]. In contrast, we aim to validate that this technique is also effective reducing memory access time when using out-of-order cores.

Figure 2 shows a high-level overview of the architecture targeted in this work. The cores have private L1 and L2 caches. All the cores are connected through a crossbar to a shared Last Level Cache/L3 (LLC), itself connected to off-chip main memory. The MDTE can be placed next to a core's L2 or the shared LLC. If placed next to a private cache it will only process prefetch commands from that core. If placed next to the LLC it can receive and process prefetch commands from every core. Our proposed technique would work with only the shared MDTE, but ideally we also want private MDTEs to let the runtime system decide which one to use in every case.

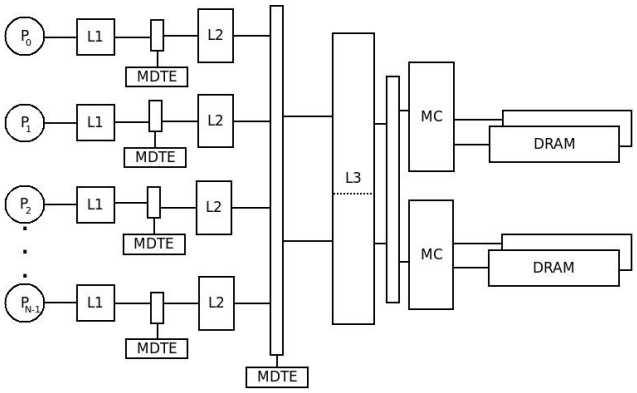


Fig. 2: Targeted multi-core architecture.

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 since the physical pages they map to may not be contiguous in memory, the need to be split at page boundaries. Splitting prefetch commands and 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 order to send a prefetch command to the LLC's, i.e., L3 in Figure 2, the runtime system would issue the instruction *prefetch3 r1, r2*. If the runtime system has not been provided with a cache hierarchy map and there is no L3 cache in the system, the instruction is ignored. In the targeted architecture (Figure 2), one bit in the instruction word is enough to specify whether the prefetch instruction targets the L2 or the L3.

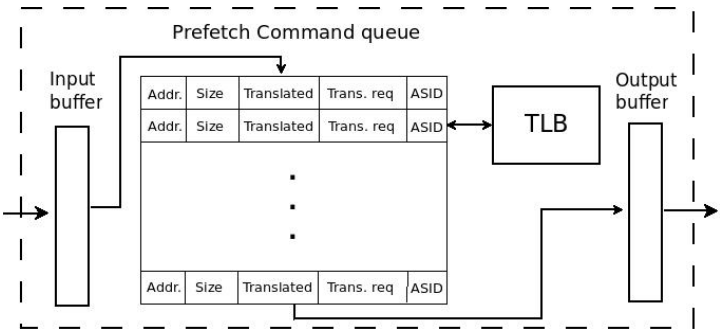


Fig. 3: Multi-core Data Transfer Engine components

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 3 shows its design. The main components are:

- An input buffer to store the received prefetch commands until they are queued.
- 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 to store 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 [34].

Once a translation response is received, the prefetch command is updated and moved to the output buffer. Interrupts and exceptions can modify the logical to physical address mapping, rendering the prefetches useless. In these situations we flush the TLB and the entries in the prefetch command queue whose translation has been requested, as well as the translated commands from the output buffer.

On every cycle at most one request will be issued, either a prefetch command or a translation request. 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. These prefetches coexist with hardware-based prefetch requests but are much less time sensitive, hence the need for some form of coordination. See Section 4.7 for more details.

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

The private MDTEs will always forward the translated commands to the private cache they are attached to, and the shared MDTE to the LLC. Thus, 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

```
prefData_next = 0
while input_next > 0:
  capacity = size_L2 - input_curr - prefData_next
  if capacity > 0 then:
    L2 prefetch up to capacity bytes
    increase prefData_next
    decrease input_next
  else
    L3 prefetch input_next bytes
  endif
```

Fig. 4: Algorithm used by the runtime system to decide the prefetch destination.

L2 cache cannot hold more data without evicting the current task's working set, we direct the remaining prefetch commands to the shared MDTE.

Figure 4 summarizes the algorithm used by the runtime system to decide the prefetch destination. The amount of data that can be placed in the L2 is calculated as: $capacity = size_{\text{L2}} - input_{\text{curr}} - prefData_{\text{next}}$, where $size_{\text{L2}}$ is the size of the L2 cache, $input_{\text{curr}}$ the size of the input data from the task currently executing and $input_{\text{next}}$ from the task that will be executed next. $prefData_{\text{next}}$ represents the amount of data already prefetched from the next task.

Figure 5 shows the destination of the prefetched data for two executions of the same benchmark with two different cache configurations. In this example, for simplicity, all tasks have 160 KB of input data.

The caches are initially assumed to hold old data, so the data for task 1 is always placed in the L2. On a system with a 128 KB L2 cache, only 128 KB of data fit; the remaining 32 KB are then prefetched into the L3 cache. When the runtime system begins prefetching for task 2, the L2 is filled with tasks' 1 working set, therefore the 160 KB of data are prefetched into the L3. This behavior repeats until the end of execution.

On a system with a 256 KB L2 cache, the 160 KB of input data from task 1 are initially placed on the L2. When the runtime system begins prefetching for task 2, 96 KB of it's input data are prefetched into the L2 and the remaining 64 KB into the L3. On this configuration the working set of the currently executing task co-exists with a portion of the following tasks input data.

The L3 cache is assumed to be large enough to hold the working set of each of the executing tasks plus the prefetched data. As it will be further discussed in Section 5.2, it is usually desirable to divide computation into small enough tasks to improve load balancing. Table 2 shows the average task input data size for our workloads, and Table 1 the configuration parameters of the simulated architecture. This shows that even for tasks with the largest input data size, the L3 cache is large enough to fit all the required data.

Since the runtime system can be informed of the characteristics of the memory hierarchy, if the ratio of task input data to last-level cache size were

to change, it would be trivial to modify the runtime system to stop prefetching when necessary.

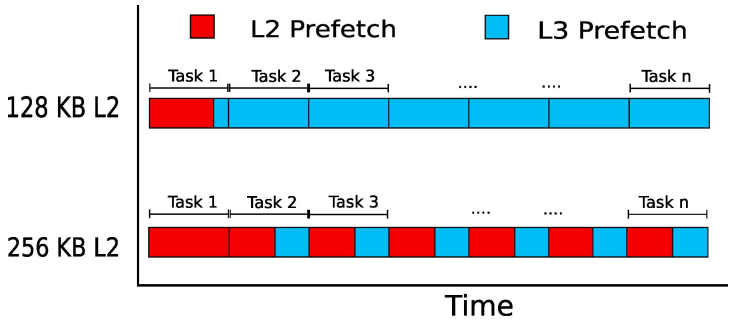


Fig. 5: Prefetch destination of the input data for each task for two runs with different L2 configurations. Input data size: 160 KB.

4.7 Coordinating Hardware, Software Prefetch and Demand Loads

The main goal of our mechanism compared to previous prefetching 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. the 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 [14] 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. The first such consideration is that 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 (see Table 2). 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.

In addition, while demand requests are always prioritized, in-flight prefetches may still stall the memory subsystem if any of the hardware structures becomes full (input buffers, MSHR queues, etc). We apply a simple throttling policy to deal with this issue. Any time that a cache level is unable to process a

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

new request, prefetch issue is stopped in that cache until demand requests can again be successfully processed. By doing so we give time to the in-flight requests to complete and we avoid getting the hardware structures filled with new prefetch requests that would further stall demand requests.

5 Evaluation Methodology

In this section we describe the simulation infrastructure and the benchmarks used to evaluate our proposed mechanism.

5.1 Simulation Infrastructure

We use a trace-driven cycle-accurate simulator that models an x86 multi-core processor [28]. We model the timing of an out-of-order processor, cache hierarchy, interconnection network and the off-chip memory. Our simulation framework uses the dynamic binary instrumentation tool Pin [22] 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.

5.2 Workloads

We evaluate our proposal using a set of scientific benchmarks including PBPI, a parallel implementation of Bayesian phylogenetic inference method for DNA

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

Benchmark	Input size	T. creation	T. duration
histogram	256KB	18μs	546μs
matmul	128KB	14μs	631μs
reduction	256KB	17μs	145μs
LU	128KB	16μs	1000μs
PBPI	200KB	13μs	114μs
jacobi	258KB	15μs	245μs
MD5	512KB	14μs	2021μs

Table 2: Benchmarks evaluated, average task input size, average task creation overhead and average execution time per task

sequence data [16], an implementation of the MD5 hashing algorithm, and a set of kernels representing algorithms commonly found on scientific applications. The full list can be found on Table 2. All applications were compiled for x86-64 with the GCC compiler version 4.6.3 using the -O3 optimization flag.

In this work we target scientific codes such as those used in high performance computing (HPC). HPC applications usually operate on regular data structures and can therefore benefit both from our runtime directed software prefetching and from hardware-based prefetching techniques. Our runtime directed prefetching scheme also works on applications with less regular data structures as long as the tasks' input and output data is specified as described in Section 2.3.

An important aspect to consider in high performance computing is the granularity at which the work is divided. In order to fully utilize all the system's processing elements and maximize the benefits of the cache hierarchy, the programmer must choose an appropriate block or task size to work with. This decision is usually taken considering the size of the cache memories and the number of processing elements. To improve load balancing, it is usually desirable to split computation into small tasks, allowing the scheduler to keep the processing elements busy at all times. On the other hand, working at a too small granularity adds non-negligible overheads in the form of thread or task creation. There is plenty of literature on the topic of how to best choose this parameter and the impact it has on the overall system performance [8, 20, 29, 30, 32]. We create tasks as small as possible to obtain good load balancing and exploit L1 cache locality, but keeping the overhead of task creation relatively small over the total execution time.

Table 2 shows the average size of the inputs for each task, the average overhead of task creation and the average execution time per task. These numbers were obtained on a 16-core, dual-socket AMD Opteron 6128 machine running at a frequency of 2.4 GHz.

6 Experimental Evaluation

We evaluated the MDTE using the seven scientific benchmarks shown in Table 2 and three different configurations of 4, 8 and 16 cores. Each core has a

Benchmark	Best HW pref.
histogram	L1 Nextline + L2 Stride
matmul	L1 Stride
reduction	L1 Nextline
LU	L2 Nextline
PBPI	L1 Nextline + L2 Stride
jacobi	L1 Nextline + L2 Stride
MD5	L1 Nextline + L2 Stride

Table 3: Best standalone hardware prefetch configuration

private L1 and a private L2 cache, and all the cores share the L3 LLC. The LLC is multi-banked, with an 8 MB bank per each 4 cores. We also add an additional memory controller per each additional LLC bank to sustain the traffic generated by the out-of-order cores.

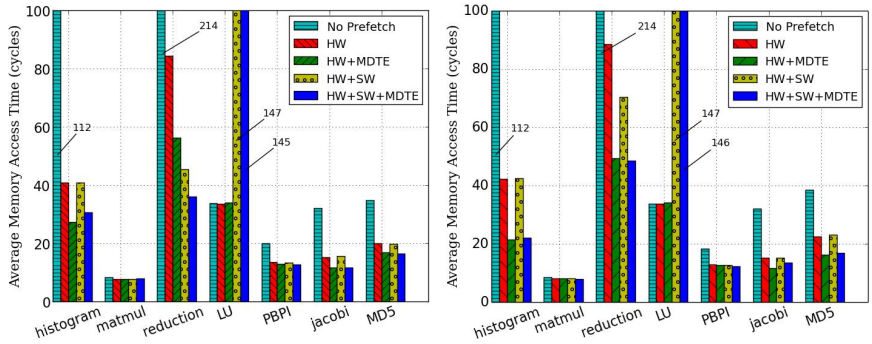
6.1 Hardware Prefetchers

We first explored the effectiveness of the standalone hardware prefetchers for each of the benchmarks. Table 3 shows which hardware prefetching mechanisms works better in each case. The *Next-line* configuration prefetches the next N lines after a cache miss. The *Stride* configuration is a reference prediction table based stride prefetcher [3]. We evaluated a range of values for the prefetch degree and found N=2 to be optimal for both configurations. We then executed the benchmarks with all the hardware prefetching configurations combined with our software prefetching mechanism. For all benchmarks but one, the hardware prefetch configuration that obtains better results standalone is also the best configuration in our hybrid hardware + software approach. The exception is LU, where every hardware + software configuration degrades performance by at least 5% over no prefetching. For the rest of this section, we use the best standalone hardware prefetch configuration shown in Table 3 as the baseline for each benchmark. This configuration is labelled as "HW" on the figures.

6.2 Compiler Based Software Prefetching

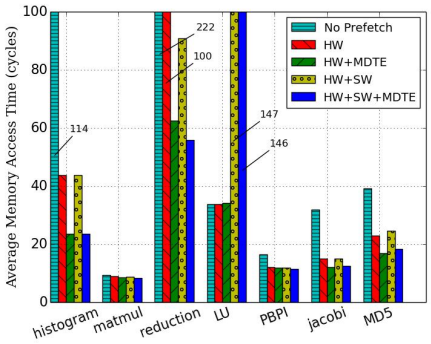
We evaluate our proposal against other traditional software prefetching techniques by compiling every benchmark with the GCC flag *-fprefetch-loop-arrays*. With this optimization the compiler attempts to insert ISA specific prefetch instructions in loops that traverse large data arrays.

As stated before, our hybrid approach combines runtime-assisted block prefetching with other traditional prefetching mechanisms that move data closer to the cores once it is brought on-chip by the MDTE. Therefore we not only use this configuration to compare our proposal against, but we also evaluate the impact of combining both. We first execute the benchmarks compiled with the prefetch flag in conjunction with every hardware prefetcher



(a) 4 cores

(b) 8 cores



(c) 16 cores

Fig. 6: Average memory access time for 4, 8 and 16 cores and multiple prefetch configurations

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").

6.3 Performance Analysis

6.3.1 Average Memory Access Time

Figure 6 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.

6.3.2 Cache Hit Rates

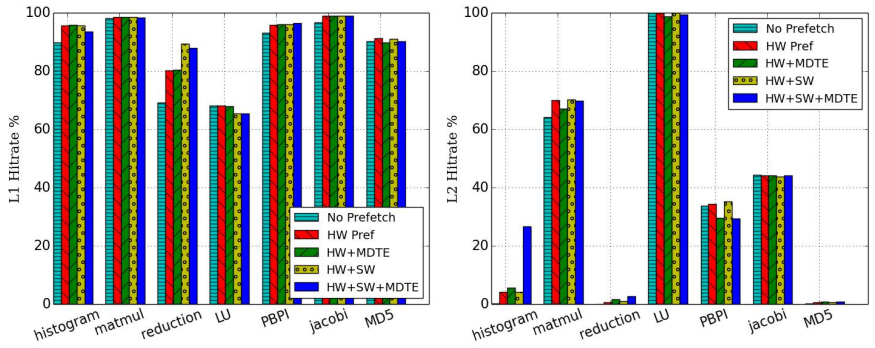
The cache hit rates shown in Figure 7 explain why *matmul* barely obtains any AMAT reduction and *LU* slightly increases it. Our implementation of matrix multiply uses blocking and the BLAS library. These commonly used optimizations fully exploit the size of the L1 cache, obtaining 99.9% L1 hit rate. *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.

6.3.3 GCC-Based Software Prefetching

Executing the benchmarks compiled with the GCC prefetch flag has mixed results, including a large degradation in performance of up to 50% on *LU*. *reduction*, *PBPI*, *jacobi* and *MD5* obtain the best results with the software + hardware configuration, while *histogram* barely improves and *matmul* sees no benefit. As explained in GCC's documentation [12], compiling with the prefetch flag may generate better or worse code and is highly dependent on the structure of loops, hence it is an unreliable mechanism to consistently improve performance. Nevertheless, our proposed technique is designed to work in conjunction with any other fine-grained prefetching mechanism, so it is at the discretion of the user whether to use GCC-based software prefetching or not.

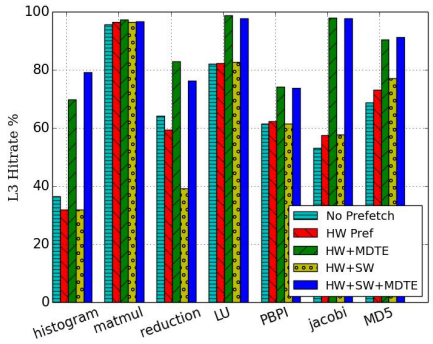
6.3.4 Performance Evaluation

Figure 8 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



(a) L1 hit rate

(b) L2 hit rate

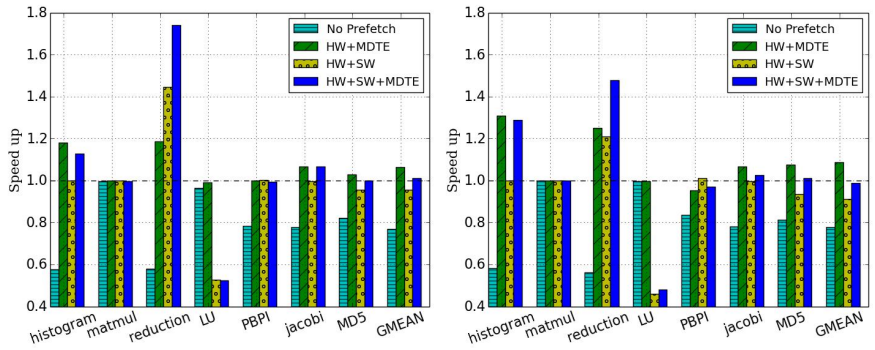


(c) L3 hit rate

Fig. 7: Cache hit rates for the execution with multiple prefetch configurations

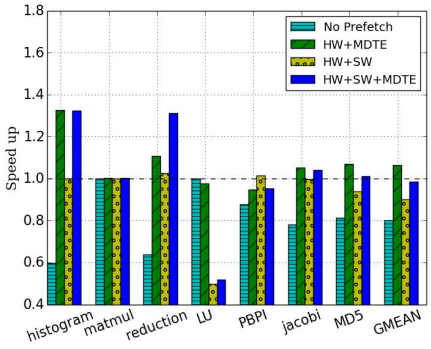
and our proposed block prefetcher working together. This is due to the large L1 hit rate increase caused by the compiler-inserted prefetches. On average, the hybrid hardware + MDTE configuration obtains an 8% speed up over the baseline. Although the configuration including compiler-inserted prefetch 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*,



(a) 4 cores

(b) 8 cores



(c) 16 cores

Fig. 8: Application speed up normalized to the execution with the best hardware prefetcher standalone

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 8.

6.4 Energy Consumption

Prefetching is usually considered a trade-off between performance and energy consumption, especially on speculative hardware based prefetchers [19]. Runtime-directed prefetching however brings only data known to be needed, and the additional hardware required to support our software block prefetcher has an almost negligible cost in area and power.

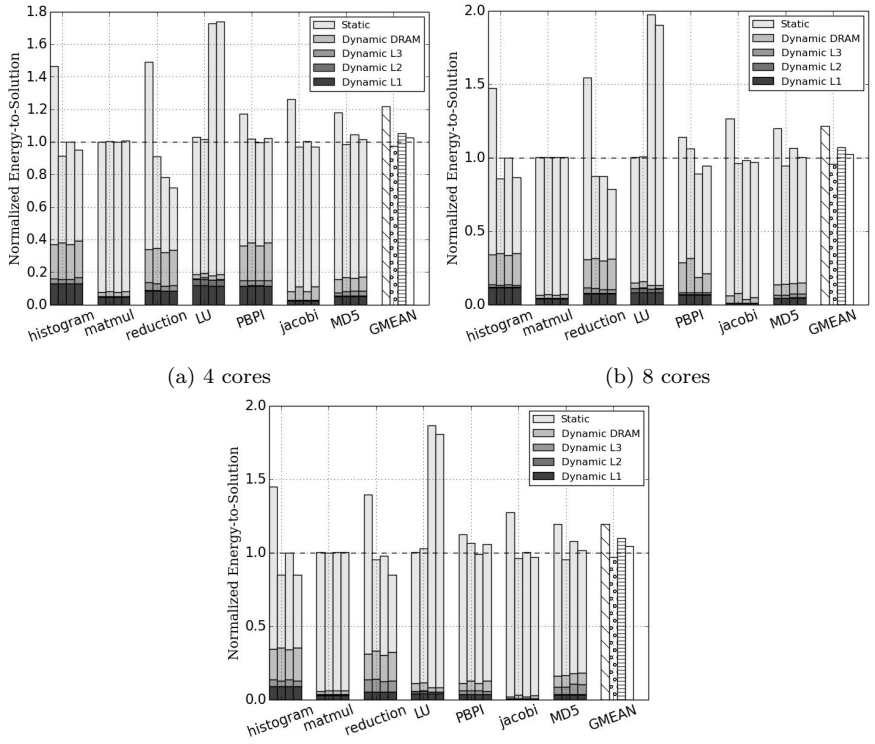
Figure 9 shows energy-to-solution for every benchmark. We see how energy consumption is dictated primarily by static power, and therefore by execution time. The increase in power caused by the MDTEs has been included in the dynamic power of the cache level they are attached to, i.e., L2 for the private 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% 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 paper 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.



(a) 4 cores

(b) 8 cores

(c) 16 cores

Fig. 9: 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.

8 Acknowledgements

This work has been partially supported by an FPI-UPC grant, the Consolider program of the Spanish Ministry of Economy and Competitiveness (TIN2012-34557), the Mont-Blanc project (ICT-FP7-288777), and the European Network of Excellence HIPEAC-3 (ICT-287759).

References

1. ARM. Cortex-a9 technical reference manual, 2008.
2. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, Feb. 2011.
3. J.-L. Baer and T.-F. Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.*, 44(5):609–623, May 1995.

4. S. Byna, Y. Chen, and X.-H. Sun. A taxonomy of data prefetching mechanisms. ISPAN '08, pages 19–24, Washington, DC, USA, 2008. IEEE Computer Society.
5. B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, Aug. 2007.
6. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.
7. 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.
8. I.-H. Chung and J. Hollingsworth. A case study using automatic performance tuning for large-scale scientific programs. HPDC '06, pages 45–56, 2006.
9. O. Consortium. Openmp website.
10. F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *ICPP 1993*, volume 1, pages 56–63, 1993.
11. F. Dahlgren and P. Stenstrom. Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In *HPCA 1995. Proceedings.*, pages 68–77, 1995.
12. G. developers. Gcc documentation.
13. 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. *Parallel Processing Letters*, 21(2):173–193, 2011.
14. 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.
15. K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.
16. 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.
17. M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.
18. 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.
19. Y. Guo, P. Narayanan, M. Bennaser, S. Chheda, and C. Moritz. Energy-efficient hardware data prefetching. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(2):250–263, 2011.
20. D. Lowenthal and M. James. Run-time selection of block size in pipelined parallel programs. In *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pages 82–87, Apr.
21. J. Lu. *Design and implementation of a lightweight runtime optimization system on modern computer architectures.* PhD thesis, Minneapolis, MN, USA, 2006. AAI3220014.
22. 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.
23. M. R. Martonosi. Analyzing and tuning memory performance in sequential and parallel programs. Technical report, Stanford, CA, USA, 1994.
24. T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12:87–106, 1991.

25. K. Nesbit and J. Smith. Data cache prefetching using a global history buffer. In *Software, IEE Proceedings-*, page 96, feb. 2004.
26. 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.
27. J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.
28. 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.
29. A. Rico, A. Ramirez, and M. Valero. Available task-level parallelism on the cell be. *Sci. Program.*, 17(1-2):59–76, Jan. 2009.
30. E. Rothberg, J. P. Singh, and A. Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. ISCA '93, pages 14–26, New York, NY, USA, 1993. ACM.
31. Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 171–182, Washington, DC, USA, 2002. IEEE Computer Society.
32. S. Tandri and T. Abdelrahman. Automatic partitioning of data and computations on scalable shared memory multiprocessors. ICPP '97, pages 64–73, 1997.
33. D. M. Tullsen and S. J. Eggers. Effective cache prefetching on bus-based multiprocessors. *ACM Trans. Comput. Syst.*, 13(1):57–88, Feb. 1995.
34. 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.
35. M. Wall. Using block prefetch for optimized memory performance, 2001.
36. 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.