Prefetching and Cache Management Using Task Lifetimes

Vassilis Papaefstathiou*
FORTH-ICS
Heraklion, Crete, Greece
papaef@ics.forth.gr

Manolis G.H. Katevenis*
FORTH-ICS
Heraklion, Crete, Greece
kateveni@ics.forth.gr

Dimitrios S. Nikolopoulos
Queen's University of Belfast
Belfast, United Kingdom
d.nikolopoulos@qub.ac.uk

Dionisios Pnevmatikatos†
FORTH-ICS
Heraklion, Crete, Greece
pnevmati@ics.forth.gr

ABSTRACT

Task-based dataflow programming models and runtimes emerge as promising candidates for programming multicore and manycore architectures. These programming models analyze dynamically task dependencies at runtime and schedule independent tasks concurrently to the processing elements. In such models, cache locality, which is critical for performance, becomes more challenging in the presence of fine-grain tasks, and in architectures with many simple cores.

This paper presents a combined hardware-software approach to improve cache locality and offer better performance is terms of execution time and energy in the memory system. We propose the explicit bulk prefetcher (EBP) and epoch-based cache management (ECM) to help runtimes prefetch task data and guide the replacement decisions in caches. The runtime software can use this hardware support to expose its internal knowledge about the tasks to the architecture and achieve more efficient task-based execution. Our combined scheme outperforms HW-only prefetchers and state-of-the-art replacement policies, improves performance by an average of 17%, generates on average 26% fewer L2 misses, and consumes on average 28% less energy in the components of the memory system.

Categories and Subject Descriptors

B.3.2 [**Design Styles**]: Cache memories; C.1.4 [**Parallel Architectures**]; D.1.3 [**Concurrent Programming**]

General Terms

Design, Measurement, Performance

*Also, with the Computer Science Department (CSD), University of Crete, Heraklion, Greece.
†Also, with the ECE Department, Technical University of Crete, Chania, Greece.

ICS'13, June 10–14, 2013, Eugene, Oregon, USA.

Keywords

Task-based Programming; Prefetching; Cache Management

1. INTRODUCTION

One common approach to parallel programming is to decompose a program into a set of tasks and distribute them among the processing elements. Many task-based programming models have been proposed in the literature [4, 9, 18] and promise to ease programming effort by abstracting out the elements of parallel programming that are traditionally considered hard and time consuming, such as scheduling, synchronization, and locality optimizations. This paper considers a class of emerging task-based dataflow programming models where the memory footprint of each task is declared by the programmer and the runtime software automatically detects task dependencies based on these footprints and schedules independent tasks concurrently [2, 6, 28, 33]. In this context, we focus on the memory behavior of fine-grain tasks and their impact on cache locality, which greatly affects performance. The use of fine-grain tasks can unleash large amounts of parallelism and has the potential to allow many-core computing resources to be utilized.

Each task has its own unique memory footprint and the associated data will eventually be transferred into the portion of the underlying memory hierarchy that is closest to the core that executes this task, i.e. L1 and/or L2 cache. Given that tasks are separate units of work, each with its own memory footprint, reuse of cache contents among tasks is a difficult problem that locality-aware schedulers are trying to alleviate [16]. However, task data reuse is not always possible, and depends on the distance between producer and consumer tasks, which is an intrinsic characteristic of each application. Moreover, several types of applications do not benefit from a single type of scheduler and their behavior may be incompatible with specific scheduling algorithms. Often times, applications benefit from load-balancing and work-stealing, which makes locality a conflicting goal; the most extreme case of locality scheduling dictates that all tasks execute sequentially in a single core.

The task-based dataflow programming models use the memory footprints of tasks to build dependency graphs (DAGs) and maintain significant amount of information that is used to dynamically drive runtime decisions. However, the underlying memory architecture is still agnostic of what runtimes are trying to achieve. Therefore, hardware decisions are based on rather simplistic assumptions. Our thesis is that

the runtime software maintains important semantic knowledge regarding the execution sequence and memory footprints of tasks, which can be shared with the underlying hardware to achieve an effective hardware-software synergy.

To this end, we propose architectural support and runtime co-design to improve cache locality and optimize task-based execution. We introduce the *Explicit Bulk Prefetcher (EBP)*, a programmable prefetch engine that can be utilized by the runtime software to prefetch task data. EBP is reminiscent of an RDMA (remote direct memory access) engine, but it is designed for cache-based architectures, integrates with the local cache hierarchy of each core, and offers a low overhead memory-mapped interface that can be used at *user-level*. EBP enables runtime software to prefetch task data in bulk before each task executes and perform common optimizations such as double-buffering. This form of software-directed prefetching can overcome some of the challenging issues with hardware-only prefetchers such as *timeliness*, *accuracy*, and *access pattern prediction*.

Although prefetching has the potential to improve cache locality and hide memory latency, its effectiveness is affected by the ability of the cache to keep the prefetched data. When applying double-buffering optimizations, prefetching can pollute the cache and evict useful data. To address the latter issues and shortcomings, we propose *Epoch-based Cache Management (ECM)*, a mechanism that allows software to expose its knowledge of tasks to the cache hierarchy, assign cache resources to them, and isolate the effects of prefetching. ECM is based on the notion of *Epoch*, which is defined by software as the lifetime of a task, i.e. the time period during which a task executes. ECM offers a memory-mapped interface that allows software to advance epochs, i.e. signal the beginning of new tasks, and assign quotas to epochs, i.e. declare the space a task is allowed to allocate in the cache. All data accessed (or prefetched) by a task is associated with an epoch number in the cache. ECM guides the cache replacement policy, by allowing it to distinguish between data belonging to different tasks. The hardware cost of ECM is very small.

The contributions of this paper are the following:

- We propose the *Explicit Bulk Prefetcher (EBP)*, a programmable prefetch engine that allows software to accurately prefetch data ahead of time and improve cache locality in task-based programs.
- We introduce *Epoch-based Cache Management (ECM)*, a lightweight mechanism to guide cache replacement decisions, assign local cache resources to tasks, and isolate the effects of prefetching.
- We evaluate EBP and ECM using a task-based runtime and a set of benchmark applications. We demonstrate that EBP in conjunction with ECM outperforms HW-only prefetchers, improves performance by an average of 17%, generates on average 26% fewer L2 misses, and consumes on average 28% less energy in the components of the memory system.

The rest of this paper is organized as follows: Section 2 introduces the task-based dataflow programming model. Section 3 explains the proposed architectural support. Section 4 provides the experimental methodology. Section 5 presents our evaluation. Section 6 discusses related work. Finally, Section 7 offers our conclusions.

2. TASK DATAFLOW PROGRAMMING

The task-based dataflow programming models aim to simplify parallel programming by discovering task dependencies at runtime and dynamically extracting task parallelism. Such models require the programmer (or the compiler) to identify tasks (functions) that may run in parallel, annotate the memory footprint of their arguments (addresses), and declare the side-effect of each task argument in memory (read/write). Representative examples of such programming models include SmpSS/OmpSS [28, 29], BDDT [32], Legion [6], Serialization Sets [2] and other proposals that follow similar concepts and techniques [7, 10, 33].

The underlying runtime libraries use the memory footprints and the side-effects of each task argument to identify task dependencies and build dependency graphs as directed acyclic graphs (DAGs) at runtime. Independent tasks are immediately scheduled for execution on the available cores, while dependent tasks are kept in internal data structures and queues, waiting until all of their dependencies are satisfied. Tasks in this model may execute out-of-order using the scheduling techniques followed in processors, while respecting task dependencies in the same way that processors respect register dependencies, i.e. true dependencies (RAW), anti-dependencies (WAR) and output dependencies (WAW). Deterministic execution and the preservation of task dependencies are guaranteed by the order in which the main application thread issues tasks, similarly to the order that instructions are issued in a sequential program.

In this paper, we consider the programming model proposed in [28], which is based on *C pragmas* and follows syntax similar to the popular OpenMP task pragmas [4]. We use the same task constructs and multi-dimensional memory regions syntax as in [29]. The side-effects of task arguments may be declared as one of: *(i) input:* for read-only arguments, *(ii) output:* for write-only arguments, and *(iii) inout:* for arguments that are both read and written. A code example illustrating matrix multiplication with our task dataflow programming model is shown in Listing 1.

```
void matmul_block( double A[N][N],
                   double B[N][N],
                   double C[N][N] )
{
  for (int i=0 ; i < L ; i++)
    for (int j=0 ; j < L ; j++)
      for (int k=0 ; k < L ; k++)
        C[i][j] += A[i][k] + B[k][j];
}

double A[N][N], B[N][N], C[N][N];
...
for (int i=0 ; i < N ; i+=L)
  for (int j=0 ; j < N ; j+=L)
    for (int k=0 ; k < N ; k+=L)
      #pragma xss task input(A{0:L}{0:L}) \
                       input(B{0:L}{0:L}) \
                       inout(C{0:L}{0:L})
      matmul_block( &A[i][k],
                    &B[k][j],
                    &C[i][j] );

#pragma xss waitall
...
```

Listing 1: A matrix multiplication example using the task dataflow programming model

3. ARCHITECTURAL SUPPORT

Emerging task-based dataflow programming models require the memory footprint of each task to be declared. This requirement provides unique opportunities for optimizations in the memory system. The runtime system maintains important knowledge about the task execution sequence and task memory footprints, which can guide the resource allocation decisions of the underlying hardware.

To this end, we propose a set of hardware mechanisms which can improve task-based execution with low hardware and software overhead. We propose the *Explicit Bulk Prefetcher (EBP):* a hardware mechanism that allows the runtime to accurately prefetch task data (Section 3.1) and *Epoch-based Cache Management (ECM):* a hardware mechanism to guide the cache replacement decisions (Section 3.2).

3.1 Explicit Bulk Prefetcher

Task-based programming models with annotated memory footprints allow the runtime to know before-hand which data will be used by a task before that task executes. This observation offers the opportunity to prefetch task data in a timely fashion and improve cache locality. The *Explicit Bulk Prefetcher (EBP)* is a hardware unit that allows software to explicitly prefetch memory ranges that correspond to task arguments. Software can utilize EBP to prefetch data for the next task(s) waiting in the scheduling queue, effectively applying double- or multi- buffering, in order to minimize cache misses and improve task execution.

EBP is a programmable prefetch engine that offers a memory-mapped interface and accepts a set of commands at user-level. We design EBP as a per-core engine that operates on a private coherent L2 cache. We target L2 caches since their large capacity – when compared to L1 caches – makes them more suitable for bulk prefetching and because processor pipelines and critical paths are less susceptible to changes in the L2 cache. Moreover, we choose a memory-mapped interface instead of a register-mapped interface, in order to avoid ISA changes and make our design less intrusive. EBP is reminiscent of existing RDMA engines [3, 22].

The memory-mapped interface offered by EBP, is designed to allow memory ranges to be specified with virtual addresses, thus offering fast user-level access with low software overhead. In order to translate virtual addresses[1] and ensure protection, EBP requires access to the local TLBs; typically the 2nd level TLB. The use of virtual addresses provides EBP with the capability to prefetch across page boundaries, which is a common limiting factor in HW prefetchers. In addition, EBP can trigger page-table walking hardware early and minimize, or even hide, the effect of TLB misses.

The EBP engine supports *2D memory ranges* with a constant stride in order to minimize the number of required prefetch operations in common array patterns, such as blocking/tiling. The interface defines the following memory-mapped registers to initiate prefetch operations:

- *Address:* The starting virtual address for a prefetch.
- *Block Size:* The size (bytes) of each block.
- *Block Number:* The number of blocks to prefetch.
- *Block Stride:* A constant stride (measured in bytes) used for the calculation of the next block address.
- *Epoch:* This field is used by ECM as described later.

[1]We assume that the L2 cache is physically tagged.

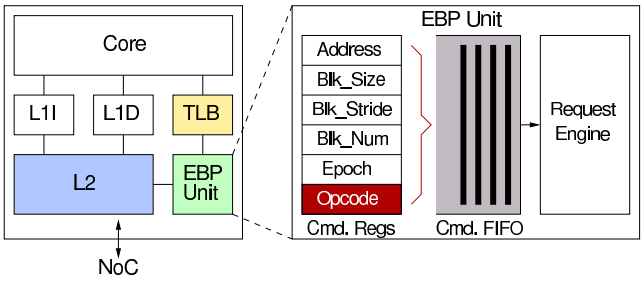

Figure 1: EBP Overview

- *Opcode:* This field marks whether this data will be used as Read-Only or Read-Write. Based on this value, the associated cache-lines are requested with the proper coherence permissions: *Shared* or *Exclusive*.

Upon writing the "Opcode" register, all command fields are atomically enqueued in a "Command FIFO" and each command is served in-order by the internal "Request Engine" (Figure 1). The "Request Engine" converts each memory range into multiple cache-line aligned requests, performs address translation, and probes the cache. If a cache-line is present in the cache with the appropriate coherence permissions, then the request is skipped. If the cache-line is not present, or present with "limited" permissions, then a new request is sent to the coherence directory to fetch or upgrade the cache-line. In case a cache set is full, an old cache-line is evicted to make space. The intermediate "Command FIFO" supports multiple outstanding prefetch operations (32 in our implementation). The "Request Engine" also supports multiple outstanding cache-line requests, the number of which is however limited by the number of miss status handling registers (MSHRs). We assume that up to 8 outstanding requests can be issued without occupying all MSHRs.

3.2 Epoch-based Cache Management

Current cache replacement policies base their decisions on the recent history of referenced cache-blocks and try to predict which blocks will be referenced in the near future. They typically assume that the most recent or the most referenced blocks should remain in the cache [21] and try to optimize this behavior. However, the behavior of task-based execution models is substantially different, since after the lifetime of a task, the reference history of many cache-blocks used by the completed task may be useless and can negatively affect performance.

The replacement decisions become even more challenging in the presence of prefetching [36], e.g. when utilizing EBP. The use of EBP has the potential to effectively hide memory latency when the software can initiate it ahead of time, before data is requested by a task, for example when the runtime software uses double-buffering. However, the effectiveness of EBP is also affected by the ability of the cache to keep the prefetched data. Prefetching is known to cause cache pollution, therefore double-buffering data for the next task may evict data needed by the current task. Likewise, the current task may evict data prefetched for the next task.

These inefficiencies offer an opportunity to improve performance by making the cache aware of the tasks' lifetimes and datasets. We propose *Epoch-based Cache Management*

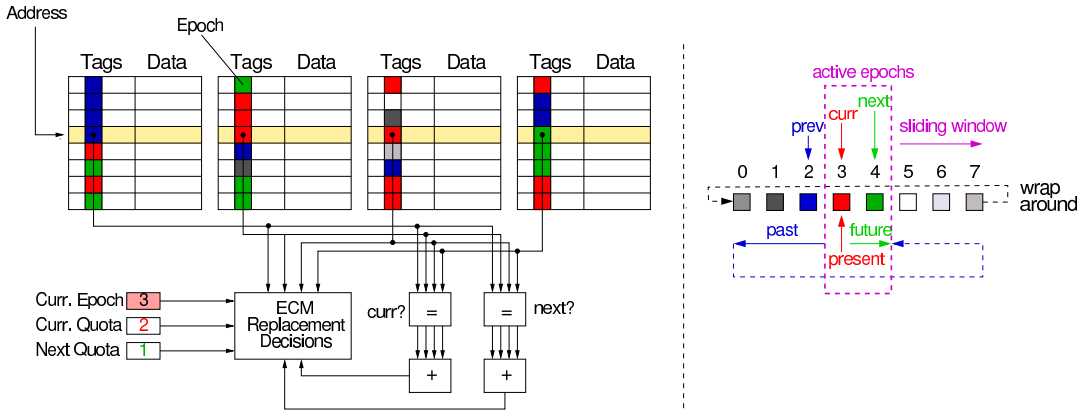

Figure 2: ECM Overview

(ECM), a hardware mechanism that allows software to expose the knowledge of tasks and their requirements to the cache hierarchy in order to improve replacement decisions and optimize cache locality.

ECM is based on the notion of *Epoch*, which is defined as the time period during which a task executes, which we refer to as the *task lifetime*. Epochs are tracked locally on each core, in the form of a memory-mapped register visible to the local cache. Every memory access that arrives from the processor, is augmented in hardware with the current epoch and marks the corresponding cache-line. The epoch number is kept in the tag of each cache-line and occupies a few bits, e.g. 3 bits to support 8 epochs. ECM only requires the software to advance the epoch register at the beginning of a new task. The epochs are free to wrap around without any special handling.

The epoch number is essentially an short identifier that allows the cache to distinguish between data that belong to different tasks while maintaining a short history, i.e. data accessed by the last 8 tasks. The replacement policy can use the epoch numbers contained in the tags of each set and the current epoch register, in order to quickly filter old data and decide which cache-lines to victimize when needed. This strategy effectively prioritizes data used by the current task. When all cache-lines in a set belong to the current epoch, the replacement policy operates as it would do without epochs and uses the reference state of cache-lines (e.g. LRU bits).

However, when the runtime employs double-buffering optimizations and uses EBP, the cache must also handle another active task context, i.e. the next task and its data. The cache has to ensure that data between these active task contexts do not interfere in a destructive manner. To address the latter issue, we introduce software-controlled *Quotas* for epochs and their corresponding tasks. ECM offers a set of memory-mapped quota registers that enables software to assign a portion of cache space for each "active" epoch. We consider active epochs to be the "current" and "next" epoch, which correspond to the current task and the first waiting task in the processor's task queue. Older epochs do not have quotas. The software assigns quotas, expressed in number of bytes for the active epochs, using the memory footprint of each task, in order reserve cache space for the task. We implement this scheme in private L2 caches.

The underlying hardware mechanism uses the quotas to construct flexible and lightweight partitions for each epoch. When a quota is assigned, the byte quantity is converted into equivalent number of ways, depending on the size and associativity of the cache. The scheme rounds up the quota to the closest multiple of equivalent cache ways and handles cases of over-booking; the sum of quotas cannot exceed the number of cache ways. ECM enforces the quotas in a best-effort manner and guarantees that each active epoch can allocate *at least* its assigned quota (ways) per set. However, an active epoch is allowed to allocate more than the assigned ways in a set, when another active epoch does not fully utilize its quota. Moreover, an active epoch is also free to use cache-lines that belong to old epochs.

The replacement policy counts the allocated ways for the active epochs in a per-set basis and based on the quotas, decides whether an epoch can allocate more space. When a cache set is fully utilized with cache-lines that belong to active epochs, the replacement policy selects a victim that belongs to the requesting epoch, by consulting only the reference state bits (e.g. LRU bits) that belong to this epoch. Cache-line allocation for an active epoch is not tied to specific cache ways but is instead dynamically selected. An overview of ECM is illustrated in Figure 2.

When EBP is used in conjunction with ECM, each prefetch request is augmented with the "Epoch" field of the prefetch command, to signify whether this request belongs to the current or the next epoch. In addition, EBP probes ECM to discover whether its quota has exceeded, i.e. the corresponding set is full with cache-lines that belong to active epochs. When a set is full, EBP throttles prefetching by skipping requests destined to this specific set, in order to avoid evicting cache-lines that were recently prefetched.

ECM can be easily implemented at low hardware cost, and in fact we have already implemented it along with EBP in an FPGA prototype [25]. Adding 3 epoch bits per tag in a 256KB 8-way set-associative L2 cache has a memory overhead of only 0.5%.

4. EXPERIMENTAL METHODOLOGY

4.1 Simulation Infrastructure

We use an execution-driven simulation framework that couples Pin [24] and GEMS [26] to model the memory hierarchy in detail. We use Garnet [1] for NoC modeling and Dramsim2 [31] for the off-chip DRAM memory controllers.

Master Core	out-of-order superscalar at 2 GHz, 4-wide, 128-entry ROB, 96-entry LD/ST queue,
Worker Cores	up to 64, in-order at 2 GHz
L1 Caches	32KB, 4-way, 64-byte block, 1 port, 1-cycle, LRU, split I/D
L2 Caches	private 256KB, 8-way, 64-byte block, 2-port, 8-cycles, NRU, 16 MSHRs, unified, coherent, inclusive
Coherence	MESI directory per memory controller, 4 virtual networks, 10-cycles, non-blocking
L2 Prefetcher	PC-based stride, 64 streams, degree of 4
NoC	2D Mesh at 2 GHz, 2 cores per node, 16-byte control packets, 80-byte data packets, 16-byte links, 1-cycle link, 5-stage routers, 4 virtual networks, 4 VCs per virtual network
DRAM	4 dual-channel memory controllers, 16GB DDR3 SDRAM, PC3-15400, 8 banks, FR-FCFS scheduling policy
EBP	32 commands, 8 outstanding requests
ECM	8 epochs using 3 bits per L2 tag

Table 1: Detailed architectural parameters for the simulated system.

Power estimation is for 32nm technology, uses Cacti 6.5 [17] for the caches and directories, Orion 2.0 [23] for the on-chip interconnect, and the infrastructure of Dramsim2 for the off-chip DRAM power. Our framework supports the x86 ISA and simulates user-level application and library code.

We model a tiled manycore architecture with directory-based cache coherence and distributed directories (per memory controller). Our design uses up to 64 in-order cores with a two-level private cache hierarchy, similar to the Intel Xeon-Phi Coprocessor [19]. We also add an out-of-order superscalar core for the role of the master processor that runs the main application thread and spawns tasks. Further details for simulation parameters appear in Table 1.

For the evaluation of our proposed architectural support we also implement a per-core PC-based stride prefetcher [5] that prefetches cache-lines into the L2 cache. We also model a state-of-the-art prefetch-aware replacement policy [36] for the purpose of comparison with ECM.

4.2 Runtime Software

We implement an in-house runtime system for the task-based dataflow programming model that supports the basic task spawning and waiting constructs as discussed in Section 2. It supports two-dimensional address ranges for declaring task footprints and uses a block-based approach with arbitrary granularity for dynamic dependence analysis [32]. Pragmas are converted into calls to the underlying runtime library using an in-house source-to-source compiler that is based on CIL [27].

The runtime, in its current form, can utilize an arbitrary number of worker threads but only a single master thread can spawn tasks. To issue tasks, the compiler generated code builds task descriptors containing: *(i)* the function pointer of each task, *(ii)* the base memory pointer for each argument, *(iii)* the dimensions of each argument, and *(iv)* the argument type describing the memory side-effects (input, output, inout). Upon task spawning, the runtime analyzes inter-task dependencies using internal metadata for the application memory and decides whether tasks are eligible for immediate execution. Ready (independent) tasks are scheduled to worker threads, while dependent tasks are kept in internal data structures until all their dependencies are satisfied. Eventually, all tasks become eligible for scheduling. The runtime fully supports out-of-order execution of tasks. The implementation utilizes private per-worker FIFO queues to schedule tasks and follows a *round-robin lowest occupancy first* scheduling policy to achieve load balancing in the presence of unbalanced tasks.

When the runtime executes on architectures that implement EBP and ECM, it utilizes our hardware support to optimize task-based execution. It prefetches task data in a double-buffered fashion using EBP and leverages ECM to manage the local cache resources. The runtime advances ECM epochs on task boundaries and assigns epoch quotas to tasks based on their memory footprint.

4.3 Benchmarks

We use six widely used benchmark applications, after converting them to the task-based dataflow programming model using *C pragmas*. Most of these benchmarks originate from the SmpSS distribution and are sketched in [29]. All benchmarks accept a block size as input parameter to allow controlling the memory footprint of tasks and, in effect, the task granularity. The benchmarks are the following:

Matrix Multiplication: An implementation of dense matrix multiplication using the BLAS [8] package. We run the benchmark with 1000×1000 double precision matrices.

Jacobi: The iterative 5-point stencil linear equation solver. We run Jacobi with 1000×1000 double precision matrices.

FFT: The 2D Fast Fourier Transform uses the FFTW [13] package for 1D FFT computations and performs transposition and twiddling in-place. We run FFT with 1M points.

Bitonic Sort: A comparison-based sorting kernel. The implementation uses Quicksort to create an initial bitonic sequence and then performs a logarithmic number of merge phases. We sort 1M long integers.

Cholesky: The Cholesky decomposition of a positive definite matrix into the product of a lower triangular matrix and its conjugate transpose. The implementation uses routines from the BLAS [8] package. We run Cholesky with a 1280×1280 double precision matrix.

Sparse LU: The LU factorization of a sparse matrix as the product of a lower triangular matrix and an upper triangular matrix. We run Sparse LU with a 1280×1280 double precision matrix.

5. EVALUATION

5.1 Performance Analysis

We evaluate the performance of our proposed architectural support (EBP and ECM) by comparing it to a baseline without and with HW-only prefetching. We run the benchmarks with block sizes that generate fine-grain tasks, each with a memory footprint size approximately equal the L1 cache size (32KB). Figure 3 shows percentage of performance improvement (execution time) over the baseline without prefetching for runs with up to 64 worker cores, for three configurations: *(i)* hardware L2 prefetcher (HWP), *(ii)* EBP, and *(iii)* EBP together with ECM. All measurements include the software runtime overhead. We also plot the speedup achieved by the baseline without prefetching, normalized to single core serial execution without runtime overhead (ignore pragmas), to show how each benchmark scales with the number or cores in our implementation.

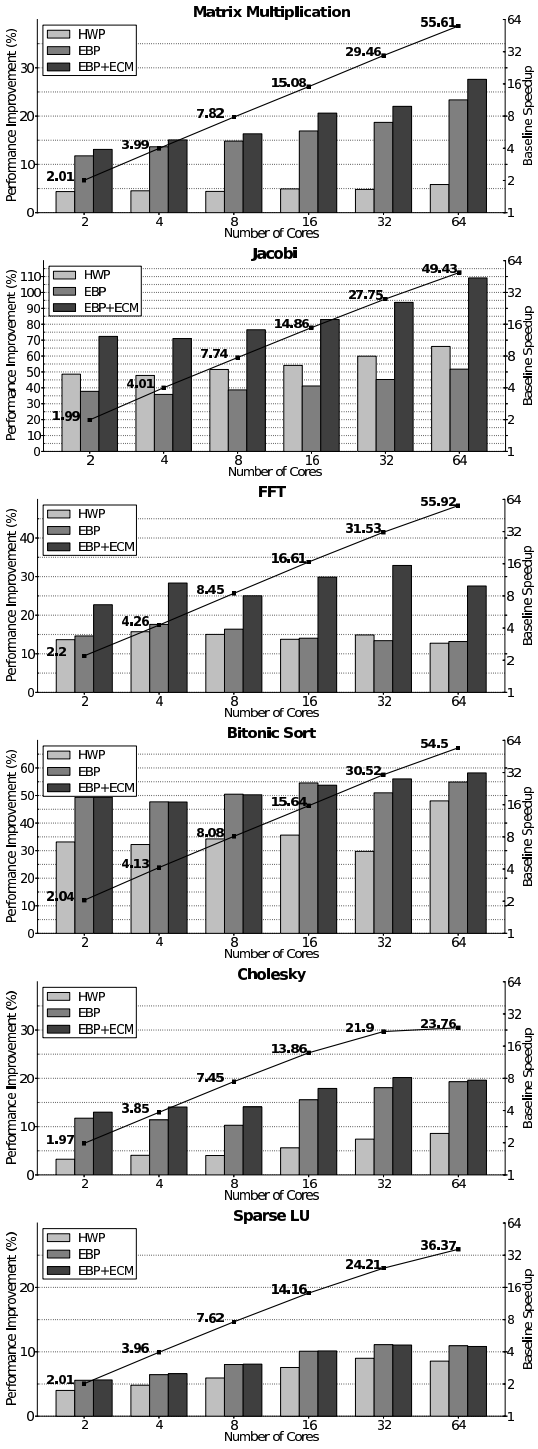

Figure 3: Performance improvement over the baseline without prefetching for each of the following configurations: (i) Hardware Prefetcher (HWP), (ii) Explicit Bulk Prefetcher (EBP), (iii) Explicit Bulk Prefetcher with Epoch-based Cache Management (EBP+ECM). The line shows the speedup of the baseline, normalized to the single core serial code execution time that ignores the task pragmas (no runtime overhead).

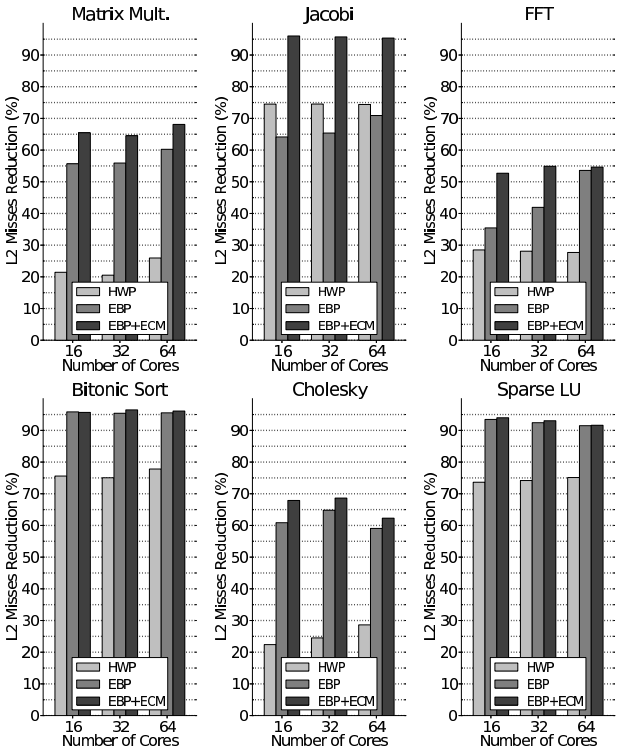

Figure 4: Reduction in the number of L2 misses over the baseline without prefetching when running with 16, 32, and 64 worker cores (higher is better).

We observe that the configuration with EBP in conjunction with ECM always outperforms HWP. For the benchmarks that scale almost linearly (Matrix multiplication, Jacobi, FFT, and Bitonic) the improvement over HWP for the higher core counts (>=16) ranges from 15% in FFT to 43% in Jacobi, which is the most memory intensive benchmark. Bitonic is 26% faster in 32 cores but appears 10% faster in 64 cores because the master thread saturates and cannot generate enough tasks for double-buffering. Matrix multiplication is consistently better by more than 16%. For benchmarks that do not scale linearly and are more compute intensive (Cholesky, Sparse LU) the improvement ranges from 3% in Sparse LU up to 11% in Cholesky. EBP alone (without ECM) is not always better than HWP. In Jacobi it performs worse than HWP by more than 13% and in FFT it achieves almost the same performance. In the rest of the benchmarks, EBP alone performs either worse than EBP+ECM (5% in Matrix multiplication) or achieves the same performance (Sparse LU). On average, for the 64-core setup, EBP+ECM is *17% faster* than the HW-only prefetcher and *13% faster* than EBP alone.

To gain more insight into the latter results, we study the impact of EBP and ECM in the number of L2 misses. Figure 4 presents the reduction in L2 misses over the baseline without prefetching for the three configurations and for setups with high cores counts (>=16). EBP with ECM significantly reduces the number of L2 misses, outperforms HWP, and achieves reduction of more than 90% on half of the benchmarks (Jacobi, Bitonic, Sparse LU). HWP performs relative well in a number of benchmarks (Jacobi, Bitonic,

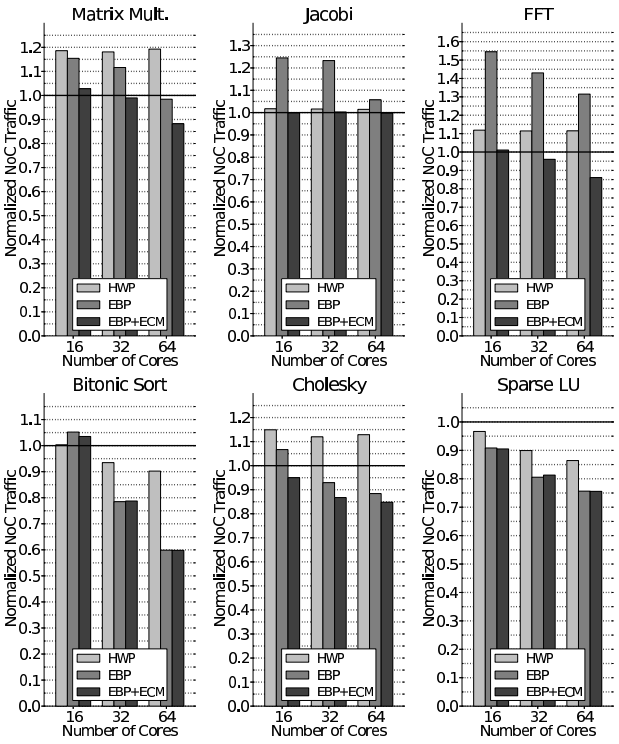

Figure 5: NoC traffic normalized to the baseline without prefetching when running with 16, 32, and 64 worker cores (lower is better).

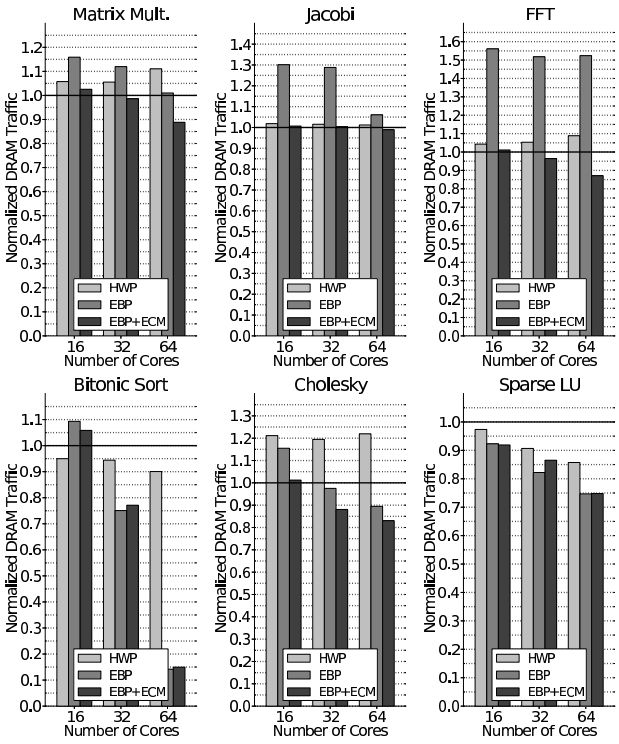

Figure 6: DRAM traffic normalized to the baseline without prefetching when running with 16, 32, and 64 worker cores (lower is better).

Sparse LU) and reduces the associated L2 misses by up to 75%, however, in the rest of the benchmarks it cannot identify the memory access patterns accurately and the reduction in L2 misses falls below 30%. On the other hand, EBP with ECM consistently provides more than 18% additional reduction in L2 misses over HWP. The additional reduction in L2 misses ranges from over 18% (Jacobi, FFT, Bitonic, Sparse LU) to 44% (Matrix multiplication, Cholesky). EBP alone cannot always perform better than HWP (e.g. Jacobi) and is less effective than EBP+ECM. When EBP prefetches data and requires L2 replacements, the replacement policy cannot distinguish between old task data, current task data and next task data, thus the victim selection is not optimal, causes interference, and reduces the effectiveness of prefetching. On average, for the 64-core setup, EBP+ECM reduces L2 misses by an *additional 26%* over the HW-only prefetcher and an *additional 7%* over EBP alone.

5.2 Memory Traffic Analysis

We study the implications of EBP and ECM on the memory system by measuring the on-chip and off-chip traffic. Given that the memory traffic is sensitive to task-scheduling, i.e. which task is executed on which core and in what order, we use a fixed off-line schedule that was captured when running the application on the baseline system. We replay the same schedule in the runtime for each of the three configurations. We present the results for on-chip network (NoC) traffic in Figure 5 and for off-chip memory (DRAM) traffic in Figure 6 for setups with high core counts (>=16). The traffic volumes are normalized to the traffic generated by the baseline without prefetching for each configuration.

The results indicate that EBP with ECM does not generate excessive traffic when compared to the baseline without prefetching. In some cases, EBP with ECM generates even less traffic than the baseline. On the contrary, HWP results in increased traffic on most benchmarks because of inaccuracy in its predictions and the associated cache pollution. The improved traffic behavior observed in the 64-core setups is partly attributed to the increased on-chip cache size when the number of cores increases; on high core counts a larger portion of the benchmarks' datasets fits on-chip[2]. In the presence of prefetching (either HWP or EBP), data from the producers' caches are transferred earlier to the consumers' caches (before they are evicted), thus saving traffic. However, the improved behavior of EBP+ECM is also because of the smarter cache management of the task datasets and the throttling technique employed (Section 3.2).

EBP+ECM generates less on-chip traffic on all benchmarks when compared to both HWP and EBP alone, Figure 5. On the other hand, EBP alone generates significantly more traffic in a number of benchmarks (Matrix multiplication, Jacobi, FFT) which can be up to 50% higher than the baseline (e.g. FFT on 16 cores). EBP alone prefetches the requested cache-lines blindly in the cache and incurs destructive interference between data belonging to the active task contexts (current and next). When EBP prefetches cache-lines that map to fully-occupied sets with data from active tasks, the replacement policy has to evict useful data, whereas ECM throttles prefetching for these sets by consult-

[2]We did not use larger datasets because of the prohibitive simulation times.

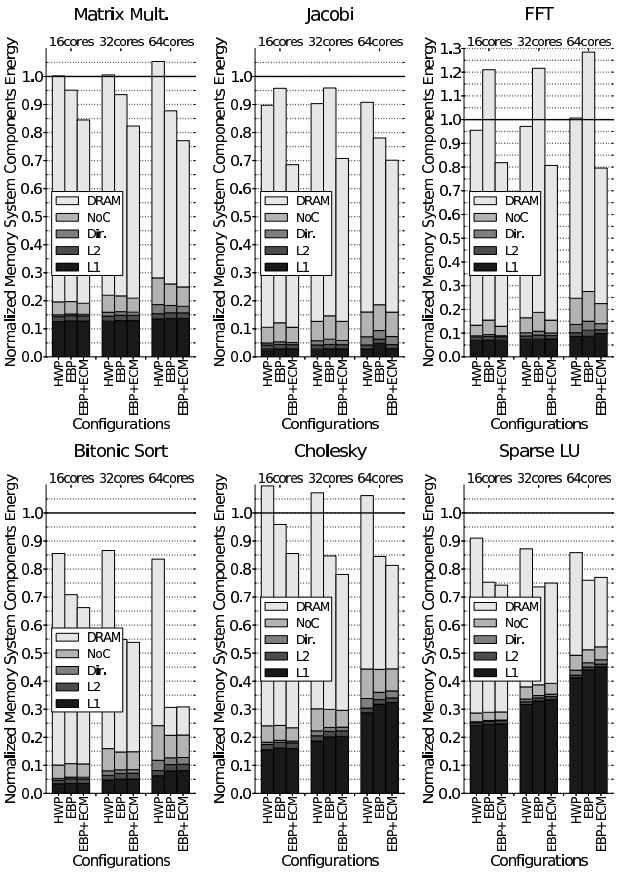

Figure 7: Dynamic energy consumption of the memory system's components. The energy is normalized to the baseline without prefetching when running with 16, 32, and 64 worker cores (lower is better).

ing the epochs and their associated quotas. On average, for the 64-core setup, EBP+ECM generates *21% less* on-chip traffic than the HW-only prefetcher and *11% less* traffic than EBP alone.

The off-chip DRAM traffic shows a similar trend with on-chip traffic (Figure 6) and the pathological cases of EBP described earlier are again clearly shown. On average, for the 64-core setup, EBP+ECM generates *28% less* off-chip DRAM traffic than the HW-only prefetcher and *15% less* traffic than EBP alone.

Bitonic presents a noteworthy behavior on 64 cores, where EBP reduces off-chip traffic by more than 80% over the baseline. This behavior is explained by the nature of comparison-based sorting which leads to a coherence pattern where data come as shared (to compare) and then upgrade to exclusive (to swap). In Bitonic, the application has marked this data as *inout* and EBP fetches them directly in exclusive mode. This avoids writebacks when data is dirty in another cache and downgrades due to the reads required for comparisons, assuming a MESI cache coherence protocol.

5.3 Power Analysis

Following the methodology discussed in Section 5.2, we use the simulator infrastructure to measure the dynamic energy consumption[3] of the memory system components (L1, L2, Directory, DRAM) and the on-chip interconnect (NoC). We present our results using breakdowns in Figure 7. In almost all the benchmarks, the dynamic energy consumption is dominated by the off-chip DRAM, while the second major component is the L1 cache. The most important finding is the reduction of DRAM energy when prefetching is used (either HWP or EBP). This behavior is explained by the open-row management and the scheduling policy of the memory controller (FR-FCFS [30]). With prefetching, the memory controller has the potential to serve more requests from open rows and reduce the number of row activations and precharges, which contribute significantly to DRAM energy. With EBP, task data is requested in bulk and close in time, thus offering the memory controller more opportunities to exploit open row buffer locality. Moreover, the significant reduction in on-chip and DRAM traffic we observe when using EBP+ECM (Section 5.2) offers additional energy savings. On average, for the 64-core setup, EBP+ECM consumes *28% less* energy than the HW-only prefetcher and *11% less* energy than EBP alone.

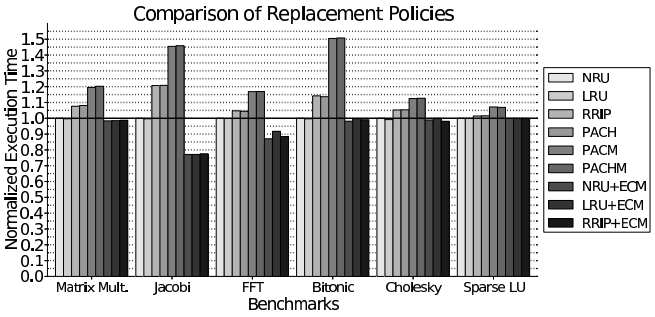

Figure 8: Comparison of replacement policies when EBP is utilized, running with 16 worker cores.

5.4 Comparing Cache Replacement Policies

We explore a number of candidate replacement policies for use with EBP and present our findings for 16-cores in Figure 8. We examine the typical NRU and LRU replacement policies, RRIP [21], and the state-of-the-art prefetch-aware replacement policy called PACMan [36] in its three variants: PACH, PACM, PACHM[4]. Moreover, given that ECM is versatile and can be used in conjunction with many replacement policies, we also evaluate three variants with ECM: NRU+ECM, LRU+ECM, RRIP+ECM. We observe that when EBP is utilized, the PACMan variants perform worse than NRU and LRU, whereas the ECM variants can be up to 22% faster than LRU (Jacobi). In PACMan, the reference history that is accumulated during a task lifetime is useless for the next task and requires several misses before cache-lines with old task data become candidates for replacement. This behavior appears to have a negative effect on performance. On the other hand, the ECM epochs help to effectively filter "old" cache-lines and allow the data prefetched for the next task to be preserved in the cache.

[3]Note that we do not include static energy measurements which are influenced by execution time, thus our comparisons are conservative.

[4]For RRIP and PACMan we use 2-bit values and the version without set-dueling.

6. RELATED WORK

There is a vast amount of previous work on hardware prefetchers that try to predict memory access patterns and prefetch data without any software guidance such as [11, 12]. However, we propose a hardware-software approach for prefetching which is based on the fully-accurate knowledge about task memory footprints, known a-priori by the runtime software (before tasks start executing).

Guided region prefetching (GRP) [34] is a related scheme that augments load instructions with compiler generated hints to improve the accuracy of a hardware prefetcher. GRP requires sophisticated compiler analysis, the hints are not always accurate, and the prefetches are triggered by L2 misses during the execution of code. Our approach differs from GRP, since we use fully accurate memory footprints and the data are fetched early before task execution with the potential to hide all L2 misses.

Most pertinent to this work is Streamware [14] and the related architectural support [15] which target stream processing. They propose a software programmable Stream-Load-Store (SLS) hardware unit, that resembles EBP, and is used by the runtime software. However, they do not tackle the problem of cache pollution and interference due to prefetching, as we propose with the use of ECM. ARM's Preload Engine (PLE) [3], which is also similar to EBP, does not address cache pollution either.

"KILL" [20] and "Evict-me" [35] are two related schemes that try to improve cache replacement decisions and reduce pollution in the presence of prefetching. Both of them are based on sophisticated compiler analysis and instruction hints (using ISA modification) that are used in conjunction with the replacement policy. These approaches try to identify which data will not be used in the future and mark this data appropriately. In the task-based execution context, this approach could only be useful for the data of the current task, while there can be no information about the behavior of the next task, the data of which can be prefetched. In addition, when a task completes, such schemes would require massive marking (or eviction) of each task's dataset. The latter could be an erroneous behavior if some of the data is reused by the next task. ECM on the other hand, easily handles all these cases and only requires software to advance the local epoch. Moreover, the epoch quotas offer an additional criterion to throttle prefetching.

PACMan [36] is also a relevant prefetch-aware cache management scheme that builds on top of RRIP [21]. PACMan tries to reduce cache pollution and prefetch interference by handling demand and prefetch requests separately. To achieve this effect, PACMan modifies the cache insertion and hit promotion policies. Although this scheme might perform well for intra-task prefetching (when data for the current task is prefetched), it can not handle inter-task prefetching. ECM addresses prefetching across tasks and helps throttling prefetches to reduce traffic and pollution. We evaluated the performance of PACMan as a candidate cache replacement scheme for use with EBP in Section 5.4.

7. CONCLUSIONS

This paper presented a hardware-software approach to improve cache locality and optimize the execution of fine-grain tasks. We proposed the *Explicit Bulk Prefetcher (EBP)* and *Epoch-based Cache Management (ECM)* to allow runtime software to prefetch task data and guide replacement decisions in caches. Using EBP and ECM the runtime software can exploit its available information about task memory footprints and task lifetimes, and expose this information to the memory hierarchy with minimal software overhead. ECM assists the cache replacement policy to select victims and removes cache pollution and prefetch interference when EBP is used. We show that EBP in conjunction with ECM outperforms HW-only prefetchers, improves performance by an average of 17%, generates on average 26% fewer L2 misses, and consumes on average 28% less energy in the components of the memory system.

8. ACKNOWLEDGMENTS

We thankfully acknowledge the support of the European Commission under the 7th Framework Programs through the ENCORE (FP7-ICT-248647) and HiPEAC3 (FP7-ICT-287759) projects. This research is also supported by EPSRC through the GEMSCLAIM project (grant EP/K017594/1).

9. REFERENCES

[1] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *Proc. of IEEE Int. Symp. on Performance Analysis of Systems and Software*, ISPASS '09, pages 33–42, 2009.

[2] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: a dynamic dependence-based parallel execution model. In *Proc. of the ACM Symp. on Principles and Practice of Parallel Programming*, PPoPP '09, pages 85–96, 2009.

[3] ARM Ltd. Cortex A9 Preload Engine. http://infocenter.arm.com/.

[4] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Trans. Parallel Distributed Systems*, 20(3):404–418, 2009.

[5] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. of the ACM/IEEE Conf. on Supercomputing*, Supercomputing '91, pages 176–186, 1991.

[6] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: expressing locality and independence with logical regions. In *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66–77, 2012.

[7] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword. Synchronization via scheduling: techniques for efficiently managing shared state. In *Proc. of the ACM Conf. on Programming Language Design and Implementation*, PLDI '11, pages 640–652, 2011.

[8] L. S. Blackford et al. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28:135–151, 2001.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. of the ACM Symp. on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, 1995.

[10] R. Bocchino et al. A type and effect system for deterministic parallel java. In *Proc. of the ACM Conf.*

on Object Oriented Programming Systems Languages and Applications, OOPSLA '09, pages 97–116, 2009.

[11] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Stealth prefetching. In *Proc. of the Int. Conf. on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 274–282, 2006.

[12] P. Diaz and M. Cintra. Stream chaining: exploiting multiple levels of correlation in data prefetching. In *Proc. of the Int. Symp. on Computer architecture*, ISCA '09, pages 81–92, 2009.

[13] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231, 2005.

[14] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: programming general-purpose multicore processors using streams. In *Proc. of the Int. Conf. on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 297–307, 2008.

[15] J. Gummaraju, M. Erez, J. Coburn, M. Rosenblum, and W. J. Dally. Architectural support for the stream execution model on general-purpose processors. In *Proc. of the Int. Conf. on Parallel Architecture and Compilation Techniques*, PACT '07, pages 3–12, 2007.

[16] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. Slaw: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proc. of the ACM Symp. on Principles and Practice of Parallel Programming*, PPoPP '10, pages 341–342, 2010.

[17] HP Laboratories. Cacti 6.5: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. http://www.hpl.hp.com/research/cacti/.

[18] Intel Corporation. Intel Threading Building Blocks (TBB). http://www.threadingbuildingblocks.org.

[19] Intel Corporation. The Intel Xeon Phi Coprocessor. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[20] P. Jain, S. Devadas, D. Engels, and L. Rudolph. Software-assisted cache replacement mechanisms for embedded systems. In *Proc. of the IEEE/ACM Int. Conf. on Computer-Aided Design*, ICCAD '01, pages 119–126, 2001.

[21] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proc. of the Int. Symp. on Computer Architecture*, ISCA '10, pages 60–71, 2010.

[22] J. A. Kahle et al. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Develeopment*, 49(4/5):589–604, 2005.

[23] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration. In *Design, Automation & Test in Europe Conference*, DATE '09, pages 423–428, 2009.

[24] 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 *Proc. of the ACM Conf. on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[25] S. Lyberis, G. Kalokerinos, M. Lygerakis, V. Papaefstathiou, D. Tsaliagkos, M. Katevenis, D. Pnevmatikatos, and D. Nikolopoulos. Formic: Cost-efficient and scalable prototyping of manycore architectures. In *Proc. of the IEEE Symp. on Field-Programmable Custom Computing Machines*, FCCM '12, pages 61–64, 2012.

[26] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[27] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proc. of the Int. Conf. on Compiler Construction*, CC '02, pages 213–228, 2002.

[28] J. Perez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proc. IEEE Conf. on Cluster Computing*, CLUSTER '08, pages 142 –151, 2008.

[29] J. M. Perez, R. M. Badia, and J. Labarta. Handling task dependencies under strided and aliased references. In *Proc. of the ACM Int. Conf. on Supercomputing*, ICS '10, pages 263–274, 2010.

[30] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proc. of the Int. Symp. on Computer architecture*, ISCA '00, pages 128–138, 2000.

[31] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *Comp. Arch. Letters*, 10(1):16 –19, 2011.

[32] G. Tzenakis, A. Papatriantafyllou, J. Kesapides, P. Pratikakis, H. Vandierendonck, and D. S. Nikolopoulos. BDDT: block-level dynamic dependence analysis for deterministic task-based parallelism. In *Proc. of the ACM Symp. on Principles and Practice of Parallel Programming*, PPoPP '12, 2012.

[33] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. A unified scheduler for recursive and task dataflow parallelism. In *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, PACT '11, pages 1–11, 2011.

[34] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *Proc. of the Int. Symp. on Computer architecture*, ISCA '03, pages 388–398, 2003.

[35] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, PACT '02, pages 199–208, 2002.

[36] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer. PACMan: prefetch-aware cache management for high performance caching. In *Proc. of the IEEE/ACM Int. Symp. on Microarchitecture*, MICRO-44, pages 442–453, 2011.