2015

Runtime-Driven Shared Last-Level Cache Management for Task-Parallel Programs

Abhisek Pan

Vijay Pai

Runtime-Driven Shared Last-Level Cache Management for Task-Parallel Programs

Abhisek Pan
Purdue University,
Department of Electrical and Computer Engineering,
West Lafayette, IN, USA.
pana@purdue.edu

Vijay S. Pai
Purdue University, Google
Department of Electrical and Computer Engineering,
West Lafayette, IN, USA.
vpai@purdue.edu

ABSTRACT

Task-parallel programming models with input annotation-based concurrency extraction at runtime present a promising paradigm for writing parallel programs for today's multi or many-core heterogeneous systems. Through management of dependencies, data-movements, task assignments, and orchestration, these models markedly simplify the programming effort for parallelization while exposing higher levels of concurrency. In addition, the use of a runtime platform enables innovations in the hardware-software interface that allows the hardware to be highly responsive to the characteristics of the application and vice versa.

In this paper, we show that, for task-parallel applications running on multicores with a shared last-level cache (LLC), the concurrency extraction framework can be used to substantially improve the efficiency of the shared LLC. We develop a task-based cache partitioning technique that leverages the dependence tracking and look-ahead capabilities of the runtime. Based on the input annotations for future tasks, the runtime instructs the hardware to prioritize data blocks with future reuse and evict blocks with no future reuse. These instructions allow the hardware to preserve all the blocks for a subset of the future tasks, thus creating partitions for tasks rather than threads. This leads to a considerable improvement in cache efficiency over what is achieved by existing thread-centric cache management policies. Thread-centric cache management policies fail to track the complex patterns of data-reuse among tasks that can be assigned to arbitrary cores and hence replace blocks for all future tasks resulting in poor overall hit-rates. The proposed hardware-software technique leads to a mean improvement of 18% in application performance and a mean reduction of 26% in misses over an LRU-replacement based LLC for a set of input-annotated task-parallel programs using the OmpSs programming model implemented on the NANOS++ runtime. In contrast, the state-of-the-art thread-based partitioning scheme suffers an average performance *loss* of 2% and an average *increase* of 15% in misses over the baseline.

1. INTRODUCTION

Current architectural trends of rising on-chip core counts and worsening power-performance penalties of off-chip memory accesses have made the shared last-level caches (LLC) one of the major determinants of multicore performance. Traditional thread-agnostic Least Recently Used (LRU)-based cache replacement schemes have been found to be ineffective for shared LLCs since they are neither able to prevent the destructive interference of high-demand low-reuse threads on other threads, nor are they adept in handling thrashing or scan-type access streams [17, 19, 20, 32, 36].

Shared LLC management techniques for multicore processors have focussed on either multiprogramming workloads or multithreaded applications. Partitioning techniques for multiprogramming workloads have focussed on managing contention among applications through explicit partitioning of the cache among co-running applications for throughput or fairness improvement [16, 18, 22, 32, 36]. Proposals for replacement policy modification have concentrated on tuning the replacement policy for better prediction of future reuse of cached data, with separate parameter tuning for each application[13, 19, 20, 21, 31]. Researchers have also proposed dynamic partitioning policies for multithreaded programs with static thread assignments and mapping, with an aim to ensure balanced progress for all threads while optimizing throughput [26, 27].

In this work we focus on shared LLC managment for an alternative model of concurrency management for parallel programs – task-based parallism. As the number of on-chip cores increase, runtime-managed task-based programming models have become an important vehicle for expressing parallelism. Through management of dependencies, task assignments, and orchestration, these models markedly simplify the programming effort for parallelization while exposing higher levels of concurrency. These task-based models are especially important for today's many-core heterogeneous chips because of their ability to simplify the effort for parallelization by relieving the programmer of complex scheduling, load-balancing, and data-movement considerations.

Recently researchers have proposed dependency-aware task-parallel models where the programmer specifies the input and output for each task through pragmas or code snippets, and the runtime uses this information to build the task dependency graph and schedule tasks for execution once the dependencies are resolved [4, 6, 12, 3, 29]. These models further ease programming effort through automatic handling of synchronization and forcing deterministic execution, and at the same time improve performance by exposing higher levels of concurrency than what is usually extracted by the programmer [4]. Additionally researchers have exploited the information tracked by the runtime to improve the efficiency of hardware optimizations such as prefetching and coherence for private caches [24, 28].

Figure 1 illustrates the two competing models to express parallelism. In the thread-based model, parallelism is ex-

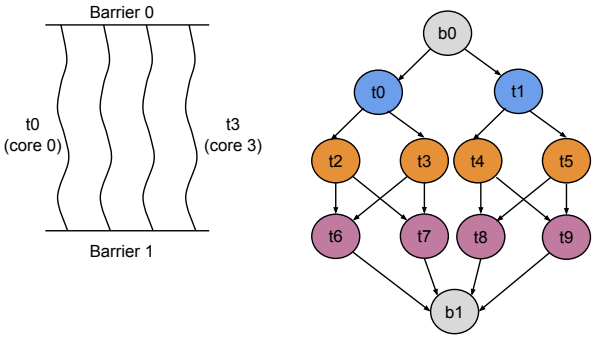


Figure 1: Parallel programming models: thread-based vs. task-based

pressed through a few concurrent long running threads (usually equal to the number of logical cores available), all of which synchronize at barriers. In contrast, execution in task-based models proceeds through a series of relatively small units of concurrency called tasks. All tasks which are independent of each other can be executed in parallel as long as there are enough resources available.

While partitioning the ways of a shared LLC among threads have been shown to improve the performance for thread-based applications, such thread-centric partitioning techniques are not well-suited for task-parallel applications, since these schemes are effective only under the following two conditions:

- LLC associativity much higher than number of threads, in order to create imbalance of allocation among the threads, and
- Long-running pinned threads with substantial intra-thread data reuse, in order to build per-thread data-reuse models.

In thread-centric partitioning models, cache blocks are tagged with the identity (id) of the thread or core that allocates the block. Since there is substantial intra-thread data reuse, data brought in by one thread can be reasonably expected to be reused by the same thread. Hence it makes sense to use thread-based partitions, where data-blocks allocated by one thread are protected at the expense of the blocks brought in by other threads. However, a thread-based partitioning paradigm does not work well for task-parallel programs running on higher number of on-chip cores with fine-grained tasks and dynamic task-core assignments [2, 5, 9]. Tasks have much shorter lifetimes than threads, and there is not much intra-task data-reuse at the LLC level. Hence effective management of cached data for task-parallel applications require tracking of inter-task data reuse and creating partitions that encompass multiple groups of tasks.

This paper presents a hardware-software technique to efficiently manage the shared LLC for applications using the dependency-aware task-parallel model on multicores. As mentioned above, in task-based applications, the dominant form of data-reuse at the LLC level is of the inter-task variety. Hence for task-based applications, in order to partition the cache space among tasks, inter-task reuse of data blocks needs to be tracked. For the dependency-aware task-parallel programming models, the runtime already tracks the inter-task data-reuse among the tasks as part of dependency resolution. Moreover, when a data-block is allocated by a task, the runtime can identify which task will reuse the block in the future. Our technique leverages the reuse-tracking and look-ahead capabilities of the runtime in order to create a task-based partition for the LLC.

With multiple tasks simultaneously accessing the LLC, a large percentage of the data blocks are evicted before they can be reused by future tasks, often leading to poor utilization. Our technique address this inefficiency in two steps. First, when a data block is accessed by a task, the runtime communicates to the hardware the identity of the next task that is going to reuse the data-block. This allows the hardware to group the cache-resident data blocks by the identity of the task-ids that will reuse the data-block. Second, during victim selection for replacement, the replacement engine uses the task-data mapping to create partitions for future tasks - it attempts to retain all the blocks to be reused by a subset of the future tasks at the cost of evicting blocks belonging to other tasks. This considerably increases the cache utilization for the tasks in the preferred subset, which in turn leads to better LLC utilization overall, and hence reduced execution time. The runtime also detects blocks which will no longer be reused by any future tasks and instructs the hardware to de-prioritize these blocks.

The task-aware cache management framework is developed as an extension of the NANOS++ implementation of the OmpSs programming model [12]. The framework does not require any change in the OmpSs API and hence is transparent to the application programmer. OmpSs is a task-based programming model which requires the programmer to annotate each task with the data objects that the task is going to read from or write to. OmpSs provides compiler directives for creating these annotations. The runtime evaluates the annotations at task-creation time and builds a task-dependency graph based on these annotations. If a task is found to be dependent on a previously created task, it is added as a successor to the previous task. A task is scheduled for execution once all its dependencies are resolved. We extend the dependence-resolution framework of the runtime to record, for each created task, the mapping of data objects to the successor tasks who are going to use the objects next. At the start of execution of a task, the runtime communicates to the hardware this stored task-data mapping. This id of the future task that is going to use the block is stored with the tag of the block in the LLC. During replacement, if the replacement engine finds that all blocks in a set are tagged with future task-ids, it replaces the LRU block and considers the task-id of this block to be de-prioritized. So henceforth all blocks belonging to this task are replaced before any block belonging to another task. If no blocks belonging to the de-prioritized task is found in a set, another future task is de-prioritized. This allows at least some of the future tasks to preserve all their blocks in the cache. The extra hits obtained through this preservation usually outweighs the extra misses suffered by the de-prioritized tasks, leading to an overall improvement in cache efficiency and program performance. The runtime decides on the candidate tasks for prioritization based on the size of the data accessed by the tasks. Only the more prominent tasks (in terms of data used) are selected as candidate for prioritization.

Our hardware-software shared LLC management technique leads to a 18% increase in performance and a 26% reduction in misses over a LRU-replacement based LLC for a set of input-annotated task-parallel programs us-

ing the OmpSs programming model implemented on the NANOS++ runtime.

The rest of the document is organized as follows: Section 2 provides a brief background on dependence-aware task-parallel models. Section 3 motivates the need for task-based cache management approaches for task-parallel applications. The proposed hardware-software technique is discussed in Section 4. Section 5 outlines the experimental framework. Section 6 analyses the performance of the proposed technique. Section 7 discusses the implementation challenges. Section 8 reviews the related work, and finally Section 9 concludes.

2. DEPENDENCE-AWARE TASK PARALLELISM

Dependence-aware task-parallel models are a class of task-parallel models that extract concurrency among tasks at runtime based on programmer-provided inputs and outputs for each task [4, 6, 12, 3, 29]. Such a model substantially simplifies the programming effort required to parallelize a sequential application. All the programmer needs to do is to encapsulate sequential blocks of computation (such as functions) into tasks and specify the data to be used by each task. The input-output data can be specified in the form of compiler directives (OpenMP tasks [3], OmpSs [12]), or code snippets (serialization sets [4]). During program execution, the runtime thread first evaluates the data-specification clauses for the tasks it encounters in order to build a task-dependence graph. Tasks are inserted in the task-dependence graph in program order but are executed out of order. The task-dependence graph is used to schedule tasks for execution once all dependencies are resolved.

The OmpSs programming model works with C programs and allows the programmer to create tasks from functions or code blocks using the *task* directive. The Mercurium compiler interprets the task directives to create actual tasks. The NANOS++ runtime system is responsible for the actual execution of the application, and manages all aspects of execution such as thread-pool management, target-device management, dependence resolution, and scheduling. Listing 1 shows the skeleton code for an implementation of the Fast Fourier Transform (FFT) kernel in OmpSs.

2.1 Specifying Data Dependencies

	00	01	10	11
00	0000	0001	0010	0011
01	0100	0101	0110	0111
10	1000	1001	1010	1011
11	1100	1101	1110	1111

Figure 2: 2-dimensional 4x4 array repesented in row-major order.

The *task* directive in OmpSs can be appended with *in, out, inout, or concurrent* dependence clauses to specify which data objects a task depends on (for example Listing 1, line 1). Currently the dependency clauses can efficiently express data objects ranging from simple variables to segments of multidimensional arrays [30]. A multidimensional array segment, known as a *region*, represent a discontiguous region of memory made from a set of contiguous memory segments [30]. Since the dependencies are computed at task creation time, dependencies can be tracked through pointer aliasing as well.

```
1  #pragma omp task inout(data[0;FFT_BS][0;N_SQRT])
2  static void FFT1D (long N_SQRT, long FFT_BS,
3                     double _Complex data[N_SQRT][
       N_SQRT]) { ; }
4  #pragma omp task inout(data[0;FFT_BS][0;N_SQRT])
5  static void FFT1D_2 (long N_SQRT, long FFT_BS,\
6                       double _Complex data[N_SQRT][
       N_SQRT]) { ; }
7  #pragma omp task inout(data[0;TR_BS][0;TR_BS])
8  void trsp_blk(long N, long N_SQRT, long TR_BS,
9                double _Complex data[N_SQRT][N_SQRT
       ]) { ; }
10 #pragma omp task inout(data1[0;TR_BS][0;TR_BS],
       data2[0;TR_BS][0;TR_BS])
11 void trsp_swap (long N, long N_SQRT, long TR_BS,
12                 double _Complex data1[N_SQRT][
       N_SQRT],
13                 double _Complex data2[N_SQRT][
       N_SQRT]) { ; }
14 #pragma omp task inout(data[0;TR_BS][0;TR_BS])
15 void tw_trsp_blk(long N, long N_SQRT, long TR_BS,
       long JJ,
16           double _Complex data[N_SQRT][N_SQRT]) { ;
       }
17 #pragma omp task inout(data1[0;TR_BS][0;TR_BS],
       data2[0;TR_BS][0;TR_BS])
18 void tw_trsp_swap (long N, long N_SQRT, long TR_BS
       , long JJ, long J,
19                    double _Complex data1[N_SQRT][
       N_SQRT],
20                    double _Complex data2[N_SQRT][
       N_SQRT]) { ; }
21
22 void FFT_1D (long N, long N_SQRT, long FFT_BS,
       long TR_BS,
23              double _Complex A[N_SQRT][N_SQRT]) {
24     // Transpose
25     for (long JJ=0; JJ<N_SQRT; JJ+=TR_BS) {
26         trsp_blk (N, N_SQRT, TR_BS, &A[JJ][JJ]);
27         for (long J=JJ+TR_BS; J<N_SQRT; J+=TR_BS)
28             trsp_swap (N, N_SQRT, TR_BS, &A[JJ][J
       ], &A[J][JJ]);
29     }
30     // First FFT round
31     for (long J=0; J<N_SQRT; J+=FFT_BS)
32         FFT1D(N_SQRT, FFT_BS, &A[J][0]);
33     // Twiddle and Transpose
34     for (long JJ=0; JJ<N_SQRT; JJ+=TR_BS) {
35         tw_trsp_blk(N, N_SQRT, TR_BS, JJ, &A[JJ][
       JJ]);
36         for (long J=JJ+TR_BS; J<N_SQRT; J+=TR_BS)
37             tw_trsp_swap(N, N_SQRT, TR_BS, JJ, J,
       &A[JJ][J], &A[J][JJ]);
38     }
39     // Second FFT round
40     for (long J=0; J<N_SQRT; J+=FFT_BS)
41         FFT1D_2(N_SQRT, FFT_BS, &A[J][0]);
42     // Transpose
43     for (long JJ=0; JJ<N_SQRT; JJ+=TR_BS) {
44         trsp_blk (N, N_SQRT, TR_BS, &A[JJ][JJ]);
45         for (long J=JJ+TR_BS; J<N_SQRT; J+=TR_BS)
46             trsp_swap(N, N_SQRT, TR_BS, &A[JJ][J],
        &A[J][JJ]);
47     }
48 }
```

Listing 1: Skeleton code snippet for FFT implementation in OmpSs.

Internally, the runtime represents each *region* in a compact form. For 64-bit virtual addresses, a *region* is represented by an ordered sequence of digits such that each digit can

be 0,1, or X(unknown). This in turn can be represented by a pair of 64 bit binary fields, called the *value* and the *mask* respectively. A *one* in the *mask* field denotes that the bit in the *value* field at the corresponding position is known, otherwise the bit-value is at that position is unknown, and the corresponding position in the *value* field is set to *zero* by convention (more details in [30]). For example, if we consider a 2-dimensional 4x4 array represented in a 4-bit virtual address space (Figure 2), a *region* that consists of two ranges $<0x2 - 0x3, 0x6 - 0x7>$ can be denoted by the sequence 0X1X, which is equivalent to the <value, mask> pair of <1010, 0010>. The compact representation also allows for inexpensive membership tests – only a couple of operations, a bit-wise AND followed by an equality test, are required to check whether an address belongs to a *region*. We leverage the low storage and computation costs of storing a *region* and testing membership in the proposed cache management framework.

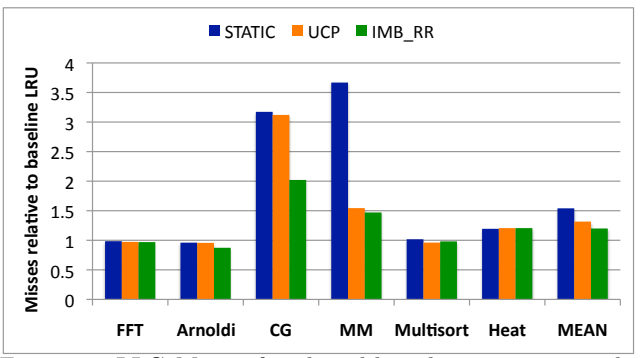


Figure 3: LLC Misses for thread-based partitioning techniques applied to task-based applications running on 16 cores sharing a 32-way 16 MB LLC.

3. CACHE MANAGEMENT FOR TASK-PARALLEL PROGRAMS

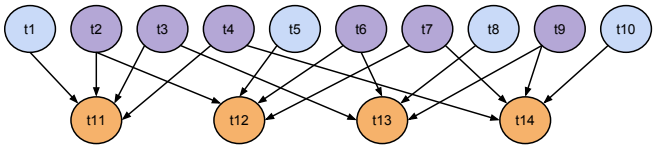
A task is the smallest unit of concurrency in task-parallel applications. The execution of a task-parallel application proceeds through a series of possibly parallel tasks, the dependence among which are determined by the data-objects each tasks reads or writes. Once all dependencies of a task are resolved, it is scheduled for execution, and is executed by an idle thread among a pool of worker threads. A task is usually of much smaller duration than a thread and does not show appreciable data-reuse at the LLC level during execution. The dominant form of data-reuse in such programs are of the inter-task variety, and the data reuse patterns among tasks can be quite complex.

Thread-based dynamic cache partitioning policies attempt to partition the ways of the shared LLC among co-running threads in order to maximize throughput and achieve balanced progress for all threads [26, 27, 32]. We find that such thread-based partitioning schemes are quite ineffective in improving the shared LLC efficiency for task-parallel applications. Figure 3 shows the misses due to three thread-based cache partitioning techniques for a 16 MB, 32-way LLC shared among 16 cores, relative to the baseline Global LRU replacement for a series of task-parallel applications (application details in Section 5). The *STATIC* policy statically partitions the cache ways equally among all threads. The utility-based cache partitioning (*UCP*) policy allocates space to each thread based on a runtime estimate of the marginal utility of the space to each thread with the goal of maximizing total cache throughput [32]. The imbalance-based cache partitioning technique (*IMB_RR*), designed specifically for symmetric multithreaded applications, creates temporary imbalance of allocation among threads to accelerate each thread in turn so that all threads are accelerated in the long run. Cache misses incurred by these schemes is seldom appreciably better than the the baseline and often worse (up to 3.7X worse). On an average, the *STATIC* policy incurs 1.54X, the *UCP* policy incurs 1.31X, and the *IMB_RR* policy incurs 1.15X times the misses incurred by the baseline. In contrast, the well-known *OPTIMAL* replacement policy (due to Belady [7]) incurs only 0.65X as many misses on an average as the baseline.

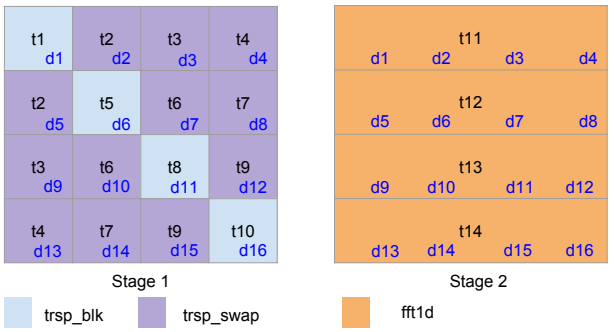
There are a few reasons for the inefficiency of thread-centric techniques for task-parallel applications. These schemes determine the per-thread allocations based on some form of a runtime model of allocation-vs-efficiency for each thread. Building this model requires each thread to be long-running and pinned to a specific core. Since tasks are short-lived and data referenced by a task running on a particular core can be reused by another task on a different core depending on scheduling considerations, such models are not meaningful in a task-based environment. Second, since major data-reuse at the LLC level is *between* tasks and not *within* a task, any runtime model needs to track data-reuse across groups of tasks and distribute the cache space across these groups of tasks. Intra-task data reuse with short reuse distances (due to spatial locality for example) is usually captured at lower levels of the cache hierarchy. Third, thread-based allocation models evaluate the allocation at pre-specified coarse-grain static interval of instructions (tens of millions of instructions), whereas allocation decisions for task-based applications should ideally happen at task-boundaries. Finally the patterns of inter-task data reuse are complex - for example, a single task can consume data generated by multiple producer tasks, individual portions of the data touched by a single task can be reused by different tasks, or a single data-object can be heavily reused by multiple tasks. The thread-based models cannot capture these complex reuse patterns.

As an example, Figure 4 shows the nature of inter-task data reuse for the two-dimensional FFT task-parallel application. Assume that the input matrix is broken into 16 disjoint chunks ($d1 - d16$), which are touched by tasks $t1 - t14$. Figure 4a shows part of the task-dependency graph for the application. Figure 4b shows the mapping of tasks to data. FFT proceeds in alternate stages (see Listing 1), with all tasks in each stage being independent of each other. In the first stage the *trsp_blk* and *trsp_swap* tasks (tasks $t1 - t10$) operate on the smallest chunks. A *trsp_blk* task touches 1 chunk ($t1 \rightarrow d1$), and a *trsp_swap* task touches 2 chunks($t2 \rightarrow d2, d5$). In the second stage however (tasks $t11 - t14$), each *fft1d* task operates on a set complete rows ($t11 \rightarrow d1, d2, d3, d4$). Hence 1 *fft1d* task reuses data block from 4 tasks from the previous stage, and conversely chunks touched by 1 *trsp_swap* task is used by two *fft1d* tasks (for example, blocks touched by $t2$ are used by $t11$ and $t12$). Thread-centric cache management techniques do not capture such reuse relationships among tasks.

Furthermore, the effectiveness of thread-based way-



(a) FFT2D task-dependency graph.



(b) Task-data mapping for FFT2D tasks.

Figure 4: **Inter-task data reuse for FFT2D application.** A single task can consume data generated by more than one tasks, vice versa.

partitioning techniques reduces as the number of cores increase relative to the shared LLC associativity, since the number of possible partitioning configurations decrease.

The goal of this work is to develop a cache management technique that captures the complex data-reuse patterns among tasks and partitions the cache accordingly. Referring to the FFT example, the tasks in the first stage ($t1-t10$) operate in parallel and attempt to bring the entire input matrix in the shared LLC. However if the input matrix, which represents the combined working set for all tasks, is too large to fit in the LLC, the baseline policy starts replacing the earliest blocks touched. This leads to a high percent of miss rates for all future tasks ($t11-t14$). In our scheme, when the first set of tasks ($t1-t10$) touch the data-blocks, the runtime informs the hardware about the identity of the future tasks that would reuse the data blocks. For example, when the chunks in the topmost row are touched by tasks $t1-t4$, they are marked to be touched by task $t11$ next. This information allows the partitioning engine to be smarter about which blocks to replace. The engine essentially partitions the cache for future tasks, so as to preserve all blocks belonging to a single task. By default, the partitioning engine tries to protect all blocks for every future task. However that is not possible if the working set is larger than the LLC capacity. Hence, at the time of replacement, if all the blocks in a set are found to be protected, the engine replaces the LRU block. This also means that the task that owns the LRU block is marked as low-priority. Since this task is marked as low-priority, all the data blocks for this low-priority task becomes candidates for replacement across all sets. If in another set, there are no blocks belonging to this low-priority task, then another task, which owns the LRU block in that set, is identified as low-priority. *This implicitly creates a partition shared by a group of low priority tasks across all sets, while allowing the other tasks to remain at high priority and entirely preserve their data.* Depending on the relative size of the working set and the LLC capacity, a subset of the tasks $t11-t14$ will be protected and the protected tasks will enjoy very high hit-rates in the cache, thus increasing the overall hit-rate. In contrast, if global LRU replacement is used, all future tasks suffer from poor hit-rates, which leads to inefficient cache usage. If the runtime determines that there are no future tasks that will reuse a data block, it instructs the hardware to consider such blocks as dead and hence candidates for immediate eviction. The future task-data mapping is updated by the runtime at the start of each task.

The runtime can also decide not to update the task-data mapping for future tasks that have small memory footprints. In that case the data blocks are marked belonging to a default task (a common task-id is used for all such blocks), and remain at a priority lower than the protected tasks but higher than any de-prioritized task. Only the more prominent tasks (in terms of data used) are selected as candidates for prioritization. This allows us to limit the overheads and achieve better performance by protecting only the tasks that have a high impact on application performance. For applications using matrix-vector computations, tasks that involve only vector-vector computations can be ignored since the memory footprint of these tasks are orders of magnitude smaller than that of the tasks that involve matrix-vector computations. In this work, the candidate tasks are chosen by the programmer and are communicated to the runtime through the priority directive available through the API. However it is possible to let the runtime select such tasks at runtime based on the relative size of the memory footprints of tasks. For applications which have only a single type of task (matrix multiplication) or comparable memory footprints for all tasks (parallel sort), the runtime considers all tasks as candidates for prioritization. In the following section, we describe in detail the hardware and software support required to implement the above technique.

4. HARDWARE-SOFTWARE SUPPORT

The objectives of our hardware-software cache management framework are twofold:

1. For any LLC-resident data block, communicate the identity of the future task(s) that will use the block to the LLC so that updated task-data mapping can be maintained. The runtime communicates the changes in task-data mapping to the hardware only at the start and end of a task execution.
2. Design an LLC replacement engine that uses the task-data mapping to create task-based partitions that attempts to preserve data blocks for as many future tasks as possible. The priority levels of each live task needs to be tracked in order to create the partitions.

In this section we describe the proposed modifications in the runtime, hardware-software interface, cache storage, and the replacement engine required to achieve these goals.

4.1 Runtime Modifications

When a task is created, the dependence analysis engine of the NANOS++ runtime creates a unique id for the task and stores the data regions accessed by the task in a data structure called the *region tree*. Each region is tagged with the last writer task and the reader tasks of the latest produced value. The dependencies for a newly-created task can then be computed by comparing its data regions with the data-regions inserted in the tree by previous tasks. We ex-

tend the state of this newly created task to store a mapping of data-regions accessed by this task to the id of the next future task(s) that will reuse these regions. A special task-id is used to represent the fact that no future task is going to use this block (henceforth called the *dead* task). The mapping is updated as the dependence engine adds future tasks to the tree and computes the dependencies. For example, Figure 5 shows a simple task-dependence graph that the runtime might generate. Assume that all tasks have a read-write relationship with the data regions. Task $t2$ is dependent on $t1$ through region $d1$; task $t3$ is dependent on $t2$ through $d1$, and on $t1$ through $d2$. When $t1$ is created, regions $d1, d2$ are mapped to the dead task. As tasks $t2$ and $t3$ are added, the task-data region mapping for $t1$ is updated.

When this task starts execution, the runtime informs the hardware about the data-blocks and the associated future tasks. The runtime sends this information if the memory footprint of the future task is prominent enough to warrant protection.

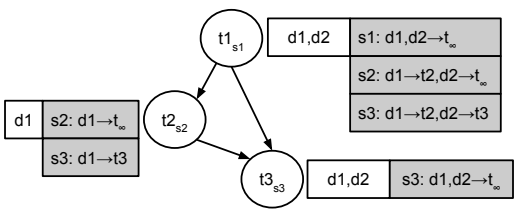


Figure 5: **Task-dependence graph and task-data mapping.** The circles show the task-ids subscripted with task-creation time. White boxes show data regions touched by a task, grey boxes show how task-data mapping changes as more tasks are created. t_∞ denotes the dead task ie. no task is going to use the data in future.

4.2 Interfacing with Hardware

As discussed in Section 2.1, a data region is represented in the OmpSs in a very compact form, through two 64 bit fields, which can represent a set of non-contiguous virtual addresses. Hence the runtime can communicate a task-region pair using two 64-bit fields for region id and a 32 bit field for a task-id. Accordingly, we propose a simple instruction set extension in the form of a memory-mapped interface with specific user-level commands. The fields to communicate for one data region are as follows:

- value (64-bit)
- mask (64 bit)
- software task-id (32 bit)
- group-id(1 bit)

A small per-core hardware engine translates the software task-id to a hardware task-id and stores this mapping in a *Task-Region Table* and determines the future task-id of each memory access instruction. Task-id translation mechanism and the purpose of the *group-id* is explained in Section 4.3. This table is flushed and updated by the runtime at the start of each task. Each memory access instruction during the task execution does a lookup of this table to identify the task-id that the address belongs to. The membership test for an address requires two bitwise logical operations. If no future task-id is found, a special default task-id is assumed. The number of entries in this table is equal to the number of tasks a particular task depends on, and only a few entries are needed. With the use of *composite task-ids* (described in 4.3), we find that 16 entries per core is more than enough. This setup is similar to the well-known programmable memory-access-interception frameworks proposed for transparent memory access monitoring, debugging, or dynamic optimizations [37].

Once a task-id is obtained the task-id is carried with the memory transaction and stored in the memory hierarchy. If the access is a miss in lower-level (L1) cache, the future-task id is sent to the LLC as part of the miss request, and is updated in the tags of the LLC. If the access is hit in the lower-level cache, the task-id is compared with the last task-id. If the ids are found to be different, an id-update request is sent to the LLC to update the tag of the block with the new owner.

At the end of a task, the runtime informs the hardware that the task with that software task-id has finished execution. This allows the hardware to update the status of the corresponding hardware task-id to *Not-Used* (task status values are described in Section 4.4), and also free the hardware task-id for recycling. Figure 6 tabulates the commands sent by the runtime at different stages of execution for the example shown in Figure 5.

Task execution state	Command type	Parameters
t1 start	Update task-data map	d1 → t2
	Update task-data map	d2 → t3
t1 end	Release sw task-id	t1
t2 start	Update task-data map	d1 → t3
t2 end	Release sw task-id	t2
t3 start	Update task-data map	d1 → t_∞
	Update task-data map	d2 → t_∞
t3 end	Release sw task-id	t3

Figure 6: **Hardware-software communication wrt Figure 5.** The runtime sends the new task-data map at the start of execution of each task, and requests release of the software task-id at the end of execution of the task.

4.3 Hardware-Software Task Translation

The runtime communicates the tasks-data mapping to the hardware in terms of the unique task-id that it has determined. While these software task-ids can be directly used in the hardware framework, we propose a translation framework between the software and hardware task-ids (*HW-SW Task Map*) because of two reasons. First, software task-ids are monotonically increasing and are equal to the number of tasks created throughout the application. However at a particular time, only a limited number of tasks are active from the perspective of our framework (currently executing tasks, and the tasks that are directly dependent on these currently executing tasks). So a HW-SW task mapping maintained at the LLC level allows us to recycle hardware task-ids and limit the bit-budget of storing a task-ids. So before the runtime updates the Task-Region Table, the hardware sends a

Task execution state	Command type	Parameters	SW-HW task-id map	Active HW task ids
t1 start	Update task-data map	d1 → t2	t2 : x1	x1
	Update task-data map	d2 → t3	t3 : x2	x1, x2
t1 end	Release sw task-id	t1		x1, x2
t2 start	Update task-data map	d1 → t3	t3 : x2	x1, x2
t2 end	Release sw task-id	t2		x2
t3 start	Update task-data map	d1 → t_∞	t_∞ : x_∞	x2, x_∞
	Update task-data map	d2 → t_∞	t_∞ : x_∞	x2, x_∞
t3 end	Release sw task-id	t3		x_∞

Figure 7: **Hardware-software task-id translation wrt Figure 5.** t1, t2, t3 are software task-ids and x1 and x2 are hardware task-ids.

request to a centralized engine to obtain a hardware task-id. If a hardware id has been already assigned to the requested software id, or a free hardware id is available for allocation, the Task-Region Table is updated, otherwise the update is skipped. The runtime requests to release the hardware-id corresponding to the software id at the end of each task. Figure 7 shows when and where hardware tasks are mapped to software tasks, and also lists the active hardware task-ids required during the execution for the example shown in Figure 5.

Second, hardware task-ids allow us to effectively deal with multiple reader tasks. In most cases a data-region touched by a task has a clearly identifiable single next user task (all cases of WAW, WAR dependencies, and most cases of RAW dependencies). However if a data object written by a task is read by multiple tasks which are independent of each-other, the object has multiple future users, which can all proceed in parallel (Figure 8). Hence the object should be preserved as long as any of these multiple protected tasks are high-priority. Also, the the id change from the multiple tasks ($t1, t2, t3$ in the figure) to the next task ($t5$) should happen only after each of $t1, t2$, and $t3$ has used the data. In order to achieve these goals, we assign a composite task id to this region, and the mapping between composite task-id to its constituent tasks is maintained in the LLC level. The priority of the block is determined to be the highest of all tasks that own the block. Also the ownership transition from a composite task to the next task happens only when all constituent tasks are released. The *group-id* that was mentioned as a part of the interface is used to identify groups of tasks for a single region that make up a composite task. A data-region with a *group-id* of 0 signifies that there are more tasks for this particular data-region. A *group-id* of 1 implies the end of a group of task for a region. In the common case of a single task for a region, the *group-id* is always 1.

4.4 Last-Level Cache Modifications

At the LLC level the future task-ids are stored along with the cache tags. There are two special task-ids, the dead task, and the default task. The partitioning engine maintains a table, indexed by the task-id, called the *Task Status Table*, maintaining the status of the task-ids. A task-id can be in

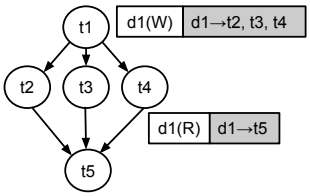


Figure 8: **Task-dependence graph for multiple readers.** All three tasks $t2, t3, t4$ are future users of region $d1$ after it is touched by $t1$. Also all of them have $t5$ as the next future task-id. Hence $t2, t3, t4$ are mapped to a composite task-id in hardware.

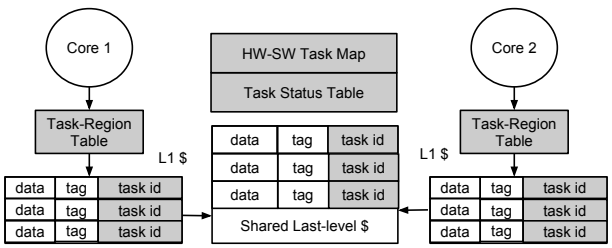


Figure 9: Task-based cache management framework. Additional structures are highlighted in grey.

the following states:

1. High-Priority: The data blocks that belong to this task-id are protected and would not be replaced unless necessary,
2. Not-Used: The task-id is not in use, so the blocks belonging to this task-id will be replaced after low-priority blocks but before high-priority blocks,
3. Low-Priority: At least one block belonging to the task block has been replaced already, and blocks belonging to this tasks are first candidates for replacement.

Hence 2 bits per task-id are required to represent the task status. A hardware task-id can be used to represent either a single task-id or a composite task-id. Hence a third bit is required to identify if the task-id is a composite id. For a non-composite task-id, the status is read directly from the Task-Status Table. Otherwise the composite Task-Status Map is read to find the tasks that the composite task-id belongs to, and the highest priority among all member tasks (again read from the Task-Status Table) is chosen as the block priority. Figure 9 shows a system-level view of the proposed framework, with per-core Task-Region Tables and the Task Status Table at the LLC level.

Algorithm 1 describes the victim selection algorithm used by the replacement engine. The replacement policy is still LRU-based but is modified such that the replacement engine chooses the LRU block based on the following overriding priority order (most-likely to least-likely to be replaced): blocks belonging to the *dead* task, blocks belonging to the low-priority task, blocks not tied to any task(default-task) or blocks with task-ids which are not being used, and blocks belonging to the high-priority task . So all blocks with the lowest priority status will be replaced before any higher-priority block, and within the same priority group, the LRU block will be replaced first.

When a block belonging to a high-priority task is replaced, the task priority in the Task-Status Table changed to a low-

priority task. For a block with a composite task-id, if all the constituent tasks are high-priority, then a randomly chosen task from the group is downgraded. This is the key step that creates the partitioning implicitly, since once a task is downgraded, its blocks will be replaced from all sets, until there is a set where no such blocks are found. Then another task will be downgraded. This creates a common partition for all downgraded tasks, while letting the other tasks preserve their data. The number of tasks that are downgraded are not controlled explicitly, but is decided automatically based on the size of the working set relative to the cache capacity.

5. EVALUATION FRAMEWORK

We evaluate the performance of our hardware-software LLC management technique using the GEMS execution-driven full-system multiprocessor simulator [25]. GEMS is a detailed timing simulator for the memory hierarchy that uses the Simics full-system simulator as its functional simulation engine [14]. We model a multicore chip with a highly-associative shared last-level L2 cache with private L1 caches. Table 1 lists the relevant system parameters. We modify the *Perfect-regions* dependence plugin supplied with the Nanos++ runtime to implement the sofware hints frameowrk, and use the default breadth-first scheduler in our experiments.

Table 1: System Parameters.

Number of Cores	16
Cache Line Size	64 bytes
L1 Cache Associativity	4
L1 Cache Size	256KB
L2 Cache Associativity	32
L2 Cache Size	16 MB
L2 Cache Request Latency	4 cycles
L2 Cache Response Latency	4 cycles
Coherence Protocol	MESI directory
Frequency	1 GHz

Workloads: We use the following seven task parallel applications obtained from the the OmpSs application repository [1]. After warming up the cache until the start of execution of the first batch of tasks, we run the benchmarks to completion.We parallelize the input initialization where appropriate.

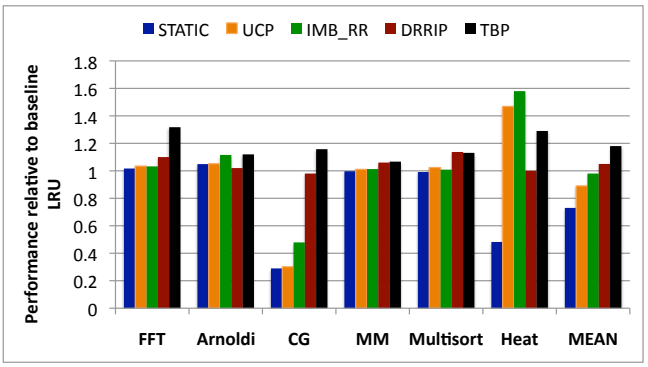
1. FFT: Two dimensional Fast Fourier Transform that includes two phases of 1D FFTs interspersed with sets of transpose and twiddling. We use a 2048 X 2048 double precision matrix as input. Each 1D FFT task performs 128 rows of FFTs and each transposition-twiddling task operates on blocks of 128 X 128 elements.
2. Arnoldi Iteration: Arnoldi iteration reduces a square matrix A to Hessenberg form via orthogonal similarity transformation: $Q^T * A * Q = H$. We use an input size of 2048 X 2048 double precision matrix with a block size of 256 X 256elements.
3. Conjugate Gradient (CG): Parallel conjugate gradient method that iteratively solves a linear system Ax=b, where A is a symmetric positive definite matrix. We use an input size of 2048 X 2048 double precision matrix with a block size of 256 X 256 elements.
4. Matrix Multiplication(MM): Parallel implementation of dense matrix multiplication. Each input matrix is of the size 1024 X 1024 elements, with a block size of 256 X 256 elements.
5. Multisort: Parallel recursive merge-sort application where at each stage the input is split into quarters and sorted in parallel, and then merged back in pairs. Quicksort is used for the smallest chunks. We use 4K integers as input with chucks of 256 elements for each task.

Algorithm 1 LLC victim selection in a set

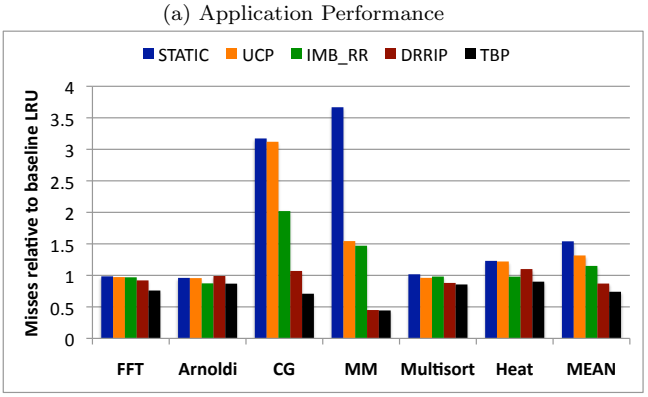
```
1: procedure SELECT_VICTIM(A, TST)
Input:
    Tag Array for Set, A
    Task Status Table, TST
Output:
    Victim Block, victim
    Task Status Table, TST
2:    victim_priority ← HIGHEST
3:    for each block i in A do
4:        priority ← GET_PRIORITY(i, TST, TASK(i))
5:        if priority < victim_priority then
6:            victim_age ← AGE(i)
7:            victim ← i
8:            victim_priority ← priority
9:        else if priority = victim_priority then
10:           if AGE(i) > victim_age then
11:               victim_age ← AGE(i)
12:               victim ← i
13:           end if
14:       else
15:           continue
16:       end if
17:   end for
18:   if victim_priority = HIGH then
19:       Downgrade priority of Task(i) to LOW by changing TST entry
20:   end if
21:   return victim
22: end procedure
23:
24:
25: procedure GET_PRIORITY(i, TST, task-id)
Input:
    Block, i
    Task Status Table, TST
    task id of block i, task-id
Output:
    Priority, pri,
    ▷ Priority from low to high: DEAD, LOW, DEFAULT, HIGH, HIGHEST
26:   if taskid = DEAD_TASK then
27:       return DEAD
28:   else if (taskid = DEFAULT) ∨ (TST[taskid] = NOT_USED) then
29:       return DEFAULT
30:   else if TST[taskid] = LOW then
31:       return LOW
32:   else if TST[taskid] = HIGH then
33:       return HIGH
34:   end if
35: end procedure
```

6. Gauss-Siedel(Heat): Iterative heat distribution solver using the 5-point Gauss-Seidel algorithm. The input matrix consists of 2048 X 2048 double precision elements.

6. RESULTS



(a) Application Performance



(b) Cache misses

Figure 10: Relative performance of STATIC, UCP, IMB_RR, DRRIP, and TDP schemes for 16 MB Cache, normalized to the baseline unpartitioned LRU cache (32-way associative, shared by 16 cores).

We compare the performance of the proposed Task-Based Partitioning technique(*TBP*) with other hardware-based shared cache partitioning techniques. The *STATIC* policy is the simplest partitioning policy that statically partitions the cache ways equally among all cores / threads. Utility-based Cache Partitioning(*UCP*) is a well-known dynamic cache partitioning technique for multiprogramming workloads. *UCP* policy allocates space to each thread based on a runtime estimate of the marginal utility of the space to each thread with the goal of maximizing total cache throughput [32]. It is expected to maximize the overall hit-rate of the cache, but not necessarily ensure balanced progress. The imbalance-based cache partitioning technique (*IMB_RR*) is a recent dynamic scheme that creates temporary imbalance of allocation among the threads of a parallel application to accelerate each thread in turn so that all threads are accelerated in the long run [27]. It uses round-robin policy of thread-prioritization to accelerate all threads. This technique also has the ability to turn off partitioning and use the baseline LRU policy, if partitioning does not provide any benefit. We also evaluate the performance of a well-known replacement-policy modification technique – the dynamic re-reference interval prediction (DRRIP) method proposed by Jaleel et al. [20]. DRRIP is a modification of NRU replacement policy, which aims to make the replacment policy both scan and thrash-resistant. A policy change from SRRIP (static RRIP) to BRRIP (Bimodal RRIP) is effected when the policy selection counter shows a bias of 1024 for one policy over another. Figure 10 shows the relative performance of the *STATIC*, *UCP*, *IMB_RR*, *DRRIP*, and the proposed *TBP* techniques relative to an unpartitioned cache (using a thread-agnostic LRU policy) for a 32-way 16 MB shared LLC. The x-axis shows the benchmarks. The Y-axes for 10a show relative performance (higher is better) and for 10b show relative miss-rates (lower is better). On average, usr the TBP technique leads to a 18% improvement in performance and 26% reduction in cache misses over the baseline. *DRRIP* achieves better performance than the baseline, with a 5% speed up and 13% reduction in misses. Other thread-based partitioning schemes actually perform worse than the baseline in average. The *STATIC* policy suffers from a 27% drop in performance with 54% increase in misses with respect to the baseline. The corresponding numbers for *UCP* and *IMB_RR* are 11% performance loss with 31% miss increase, and 2% performance loss with 15% increase in misses respectively. *IMB_RR* performs best among the competing partitioning techniques since it has the ability to turn off partitioning all-together if it determines partitioning to be harmful. As expected, *TBP* achieves very little performance gain for matrix multiplication because of the compute-intensive nature of the application. For Heat, *TBP* suffers a performance loss compared to *UCP* and *IMB_RR* despite a miss-rate reduction because the application can not recover from the temporary imbalances create in the task performance due to task-prioritization.

7. IMPLEMENTATION OVERHEAD

The major overhead of the proposed technique comes from maintaining an updated task-to-block mapping. We used 8-bit task-ids, so that the hardware has 256 task-ids that can be recycled. The core-level Task-Region Table has 16 20 byte entries, which results in a total space overhead of 5KB over 16 cores. For 256 tasks, the Task-Status Table of 256 entries has a total overhead of less than 128 bytes. We also note that the overhead of hardware-software task mapping can be obviated if the dependency resolution is implemented in hardware, as proposed by Etsion et al. [15]. The LLC tags carry task-ids, but for thread-based partitions the tags carry thread-ids which would be 4-bits for 16 cores. On the other hand the proposed scheme does not incur any overheads for creating runtime models or dynamically computing allocations unlike other thread-based partitioning techniques. For example the UMON circuits used in the UCP technique incur 2KB storage per-core, adding up to 32 KB for 16 cores. UCP also runs a greedy algorithm at pre-specified intervals of instruction to compute the partition sizes. IMB_RR scheme does not have shadow monitors, use a hardware program phase detection policy to adapt to change in program phases, and needs to repeatedly re-partition the cache space to find the best configuration.

8. RELATED WORK

8.1 Shared Cache Partitioning

The problem of partitioning a shared cache among multiple concurrently executing threads has received considerable attention in the community. The partitioning techniques can be categorized under two groups - one targeting the multi-programming environment (one application per core), the other targeting multithreaded programs (single application using all cores).

8.1.1 *Multi-programming Workloads*

The idea of minimizing the over-all miss rate for all competing threads in a multi-programming workload has been the focus of several partitioning techniques [32, 36]. A partitioning scheme that always allocates space to a thread which has maximum marginal utility for that space can minimize the over-all miss rate. If the marginal utility for each thread decreases monotonically, then the optimal partitioning can be achieved by simply continuing to allocate one unit at each iteration to the thread which has the maximum utility for it, till there are no free units left [35]. For realistic applications with no such monotonicity in their utilization behavior, this greedy algorithm has been modified to obtain near-optimal algorithms.

Qureshi and Patt propose the use of auxiliary tag directories and hit counters for each thread to track its cache-utilization behavior at runtime [32]. Cache ways are partitioned among threads using this information, in order to maximize the combined utility for all threads. The authors propose an algorithm that relaxes the requirement of allocating only one way in each iteration. At each iteration the algorithm greedily finds the thread that has maximum utility considering all free ways and the minimum number of ways that are needed to achieve that utility. The winning thread is then allocated the minimum number of ways.

Suh et al. [36] compute the marginal utility of each thread from the actual cache itself by counting the recency positions of the hits for each thread into the shared cache. The scheme does not incur the overhead of per-thread shadow tags, but suffers form inaccuracy in the marginal utility since the utility obtained for each thread is affected by the behavior of all other threads sharing the main cache. They employ another modification to Stone's algorithm where the marginal utility curve for each thread is broken into piece-wise monotonically decreasing regions, and Stone's algorithm is invoked for each combination of non-convex points in the curves, and finally the best partition is chosen from all the candidates.

Jaleel et al. proposes a shared cache management technique which partitions the cache implicitly by choosing a specific insertion policy for each of the competing applications based on its memory access behavior [19]. The key insight behind this scheme is that for a reference stream that has a working set larger than the cache capacity, cache utilization can be improved by increasing the lifetime of some cache blocks beyond that allowed by a traditional LRU scheme [31]. To this end, the authors propose the Bimodal Insertion Policy (BIP) replacement method, where most of the incoming blocks are inserted in the LRU position instead of the MRU position, the default position for traditional LRU-based schemes. As a result, any block that reaches the MRU position gets the opportunity to live in the cache for more time than it could in an aging-based scheme like LRU. For the rest, the MRU insertion position is maintained, so that the aging-based replacement is not eliminated completely. For multi-programming workloads, the BIP method would work best for the applications whose working sets could not be accommodated in the shared cache, whereas the LRU scheme would be suitable for applications with high temporal locality and small working sets. Hence, a portion of the shared cache sets is used to choose between the BIP and LRU policies for each thread in the cache at runtime and the chosen thread-aware replacement policy is enforced for the rest of the cache.

8.1.2 *Multi-threaded Workloads*

Muralidhara et al. investigate the problem of partitioning a shared L2 cache among the threads of a multi-threaded application [26]. The authors proposed a dynamic partitioning scheme that focuses on making the slowest running thread (critical path thread) faster so that the application becomes faster [26]. The proposed technique involves dividing the entire execution time into equally spaced intervals of dynamic instruction count, computing the IPC values after each interval, and allocating more cache space to the slowest thread. A record of IPC values vs. cache sizes is maintained for all past intervals. Cubic spline interpolation is used on the recored data points in order to predict the change in IPC for additional cache space allocation. Pan and Pai propose an imbalance-based partitioning scheme for symmetric multithreaded programs [27]. The authors show that the memory reuse behavior of each thread in these programs is symmetric, and, in most cases, non linear. They exploit the non-linearity by creating high levels of imbalance in thread allocations, such that one thread at a time can accelerate by securing large share of the cache at the expense of all other threads. Overall performance improvement is secured by prioritizing each thread in a round-robin fashion.

8.2 Software-assisted Cache Management

8.2.1 *Software Hints*

A key component of our cache management technique for task-based applications is the framework that allows the runtime to control the shared LLC replacement by informing the hardware about the tasks which are future consumers of the data blocks. In recent times, hardware vendors have allowed software to influence the replacement policy through appropriate hints, such as non-temporal access hints in Intel processors, or target cache-level specifications for allocating cache blocks in the Itenium architecture. Researchers have explored compiler and profile-based techniques to improve cache utilization through these hints [8, 10, 33, 34, 38, 39].

Wang et al. propose a software replacement hint in the form of an evict-me bit, which when set for a block (through extended load/store instructions) makes it the most likely candidate to be replaced. The authors develop a static compiler-level analysis to obtain a rough estimate of reuse distances of blocks for loop-based programs, and set the evict-me bit for the blocks with reuse distance greater than the cache size [38].

Beyls and D'Hollander develop profile-based and analytical techniques to generate target cache hints for loop-based applications running on the IA64 architecture [8]. Target cache hints inform the hardware about the highest(fastest) level of cache a particular memory access is expected to obtain reuse hits from. The hardware uses these hints to guide allocation and promotion decisions for the caches. First, the authors estimate the reuse distance histogram for all

accesses from each memory access instruction through profiling and extrapolation and generate static hints (hints that remain constant for all accesses due to an instruction). Second, for applications conforming to the polyhedral model, the authors develop equations to generate forward reuse distance estimations for accesses based on program parameters and use these to insert extra code which dynamically generates the hints for different instances of the same instruction. Brock et al. propose a similar profile-based method to identify accesses with high OPT distances (forward reuse distances for accesses under the optimal replacement policy) and annotate these accesses with MRU replacement hints during the compile phase [10]. The motivation of using OPT-distance over forward reuse distance stems from the fact that for cyclic access-patterns with a period larger than the cache capacity, all accesses have reuse distance greater that the capacity and hence are candidates for MRU eviction, whereas an optimal replacement policy would have retained some of them. A static memory reference inside a loop can lead to many dynamic accesses with differing OPT distances. To address this issue, the authors present an analysis to group accesses by OPT distances in loop-based programs, and performs loop splitting to enable the compiler insert appropriate hints to these separate groups. The analysis is profile-driven. First OPT-distances are measured for all dynamic accesses, and then they are grouped by static references. For each static references the authors aim to find patterns of OPT-distances across the loop iterations. Two distinct patterns are found to be dominant – for some iterations, the distances are bounded (spatial-locality), and for others they increase linearly with iteration number (temporal locality due to cyclic access patterns for loops). The authors use automatic grid-regression method to learn this patterns from training inputs. They also find that, for linear patterns, the offset depends on loop size/input size, and the slope depends on loop-shape. This enables OPT-distance prediction for different inputs sizes. Then, based on number of static references in the loop, input size and cache size, loop-splitting is performed. These profile-based or analytical estimations of reuse distance are reasonably accurate for single threaded loop-based applications, but do not work well for parallel applications because actual shared reuse distance values diverge from the predicted values due to interference from co-running threads or tasks, and the indeterminism in access interleaving.

Sandberg et al. explore the effects of identifying and eliminating non-temporal accesses in a multicore multiprogramming environment [34]. The authors develop a profile-based approach to identify memory access instructions whose data is never reused during its lifetime in the cache hierarchy as non temporal accesses through offline analysis of reuse distance profiles. These non-temporal accesses are installed only in L1 cache but not in the outer level-caches. A non-temporal access instruction is identified by the following criteria: at least one access with reuse distance greater than capacity and the number of accesses that reuse data within a distance range between L1 capacity and L2 capacity is smaller than a threshold. If an L1 data block is set as non-temporal it remains so until it is evicted from the L1 cache, even if it is reused through some other instruction, it remains non-temporal. Due to this stickiness of the non-temporal status, the above-mentioned condition must also hold for any memory access instruction that reuses the same data through the L1 cache. Confining non-temporal data to the private L1 caches allows for better utilization of the shared outer level caches. Rus et al. proposed profile-based techniques to selectively use non-temporal access instructions for string operations with poor reuse behavior to improve cache utilization for datacenter applications [33]. Yang et al. explored the benefits of using non-temporal accesses to reduce the cache pollution effects of zero-intializtions in virtual machine-managed applications [39].

8.2.2 *Software Cache Partitioning*

Lu et al. partition the cache space among heap and global objects allocated by an application in order to segregate objects with poor data locality from the rest of the objects [23]. This improves whole-program locality. Data locality signatures of individual objects are collected through profiling runs with training inputs, and are used to predict the locality behavior for the test inputs. Partitioning is done at the start of a run, using the stored locality information, run parameters, and cache configuration. Page coloring is used to partition the cache in software.

Ding et al. implement a library for user-level allocation of cache space that is implemented through the well-known page-coloring based cache-set partitioning method [11]. The library allows users to allocate private/shared space for specific data-structures and threads in the cache. Design of the allocation policy is left to the programmer.

8.3 Runtime-driven Architecture Optimizations

The runtime for task-parallel applications have been used by researchers to guide architectural optimizations for multicores. Papaefstathiou et al. develop a runtime-guided prefetch engine that can be used to prefetch data blocks to be accessed by future tasks for multicores with private caches. The authors also partition each cache between the data blocks belonging to the current and future tasks in order to prevent the prefetched data from polluting the caches [28]. Manivannan and Stenstrom propose using the runtime to guide coherence optimizations such as downgrading and self-invalidation in order to improve performance of task-parallel applications [24].

9. CONCLUSION

In this paper, we proposed and evaluated a hardware-software technique to partition a shared last-level cache among the tasks of a task-parallel application. We show that current thread-based partitioning techniques are ineffective in improving the efficiency of shared LLC for task-parallel applications, since they fail to track complex data-reuse patterns present among short-lived tasks and also to adapt to the dynamism of task-core assignments. Instead we design a scheme based on the ideas of using the runtime to map cache-resident task blocks to the tasks that are going to reuse them in future, and directing the replacement engine to preserve data blocks for as many future tasks as possible. The scheme also identifies blocks that are no longer going to be used in future and flags them for early eviction. On average,the proposed technique achieves 10% increase in application performance and 26% reduction miss-rate over a LRU-based unpartitioned cache for task-parallel applications through improved utilization of the cache space.

10. REFERENCES

[1] Barcelona Supercomputing Center bsc application repository. https://pm.bsc.es/projects/bar/wiki/Applications. Accessed: 2015-04-02.

[2] Intel Corporation intel threading builing blocks. https://www.threadingbuildingblocks.org. Accessed: 2015-03-21.

[3] OpenMP Application Programming Interface, Version 4.0, howpublished =http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, note = July 2013, Accessed: 2015-04-06.

[4] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: A dynamic dependence-based parallel execution model. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 85–96, New York, NY, USA, 2009. ACM.

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

[6] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[7] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966.

[8] K. Beyls and E. H. D'Hollander. Generating cache hints for improved program efficiency. *J. Syst. Archit.*, 51(4):223–250, Apr. 2005.

[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 *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[10] J. Brock, X. Gu, B. Bao, and C. Ding. Pacman: Program-assisted cache management. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 39–50, New York, NY, USA, 2013. ACM.

[11] X. Ding, K. Wang, and X. Zhang. Ulcc: A user-level facility for optimizing shared cache performance on multicores. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 103–112, New York, NY, USA, 2011. ACM.

[12] A. Druan, E. Aygude, 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(02):173–193, 2011.

[13] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 389–400, Washington, DC, USA, 2012. IEEE Computer Society.

[14] J. Engblom, D. Aarno, and B. Werner. Full-system simulation from embedded to high-performance systems. In R. Leupers and O. Temam, editors, *Processor and System-on-Chip Simulation*, chapter 3, pages 25–45. Springer US, 2010.

[15] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. Badia, E. Ayguade, J. Labarta, and M. Valero. Task superscalar: An out-of-order task pipeline. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 89–100, Dec 2010.

[16] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. Quality of service shared cache management in chip multiprocessor architecture. *ACM Trans. Archit. Code Optim.*, 7(3):14:1–14:33, Dec. 2010.

[17] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In *Proc. 15th Int'l Conf. Parallel Architectures and Compilation Techniques*, PACT '06, pages 13–22, New York, NY, USA, 2006. ACM.

[18] R. Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *Proc. 18th Annual Int'l Conf. Supercomputing*, ICS '04, pages 257–266, New York, NY, USA, 2004. ACM.

[19] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proc. 17th Int'l Conf. Parallel Architectures and Compilation Techniques*, PACT '08, pages 208–219. ACM, 2008.

[20] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[21] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pages 245–250, 2007.

[22] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proc. 13th Int'l Conf. Parallel Architectures and Compilation Techniques*, PACT '04, pages 111–122, Washington, DC, USA, 2004. IEEE CS.

[23] Q. Lu, J. Lin, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Soft-olp: Improving hardware cache performance through software-controlled object-level partitioning. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 246–257, Sept 2009.

[24] M. Manivannan and P. Stenstrom. Runtime-guided cache coherence optimizations in multi-core architectures. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 625–636, May 2014.

[25] 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 Comput. Archit. News*, 33:92–99, Nov. 2005.

[26] S. P. Muralidhara, M. Kandemir, and P. Raghavan. Intra-application cache partitioning. In *Proc. 2010 IEEE Int'l Symp. Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, Apr. 2010.

[27] A. Pan and V. S. Pai. Imbalanced cache partitioning for balanced data-parallel programs. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 297–309, New York, NY, USA, 2013. ACM.

[28] 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.

[29] J. Perez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151, Sept 2008.

[30] J. M. Perez, R. M. Badia, and J. Labarta. Handling task dependencies under strided and aliased references. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 263–274, New York, NY, USA, 2010. ACM.

[31] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Proc. 34th annual Int'l Symp. Computer Architecture*, ISCA '07, pages 381–391. ACM, 2007.

[32] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, MICRO 39, pages 423–432. IEEE CS, 2006.

[33] S. Rus, R. Ashok, and D. Li. Automated locality optimization based on the reuse distance of string operations. In *Code Generation and Optimization (CGO), 2011 9th Ann. IEEE/ACM Int'l Symp.*, pages 181 –190, Apr. 2011.

[34] A. Sandberg, D. Eklöv, and E. Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[35] H. Stone, J. Turek, and J. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Computers*, 41:1054–1068, 1992.

[36] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. Supercomput.*, 28:7–26, Apr. 2004.

[37] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 273–284, Washington, DC, USA, 2007. IEEE Computer Society.

[38] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 199–, Washington, DC, USA, 2002. IEEE Computer Society.

[39] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why nothing matters: The impact of zeroing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 307–324, New York, NY, USA, 2011. ACM.