Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors

Sanjeev Kumar Christopher J. Hughes Anthony Nguyen

Microprocessor Technology Labs, Intel
{sanjeev.kumar, christopher.j.hughes, anthony.d.nguyen}@intel.com

ABSTRACT

Chip multiprocessors (CMPs) are now commonplace, and the number of cores on a CMP is likely to grow steadily. However, in order to harness the additional compute resources of a CMP, applications must expose their thread-level parallelism to the hardware. One common approach to doing this is to decompose a program into parallel "tasks" and allow an underlying software layer to schedule these tasks to different threads. Software task scheduling can provide good parallel performance as long as tasks are large compared to the software overheads.

We examine a set of applications from an important emerging domain: Recognition, Mining, and Synthesis (RMS). Many RMS applications are compute-intensive and have abundant thread-level parallelism, and are therefore good targets for running on a CMP. However, a significant number have small tasks for which software task schedulers achieve only limited parallel speedups.

We propose Carbon, a hardware technique to accelerate dynamic task scheduling on scalable CMPs. Carbon has relatively simple hardware, most of which can be placed far from the cores. We compare Carbon to some highly tuned software task schedulers for a set of RMS benchmarks with small tasks. Carbon delivers significant performance improvements over the best software scheduler: on average for 64 cores, 68% faster on a set of loop-parallel benchmarks, and 109% faster on a set of task-parallel benchmarks.

Categories and Subject Descriptors: C.1.4 [Parallel Architectures]

General Terms: Design, Performance, Measurement

Keywords: CMP, loop and task parallelism, architectural support

1. INTRODUCTION

Now commonplace, chip multiprocessors (CMPs) provide applications with an opportunity to achieve much higher performance than uniprocessor systems of the recent past. Furthermore, the number of cores (processors) on CMPs is likely to continue growing, increasing the performance potential of CMPs. The most straightforward way for an application to tap this performance potential is to expose its thread-level parallelism to the underlying hardware.

Perhaps the most common approach to threading an application is to decompose each parallel section into a set of tasks. At runtime, the application, or an underlying library or run-time environment, distributes (schedules) the tasks to the software threads. To achieve maximum performance, especially in systems with many cores, it is desirable to create many more tasks than cores and to dynamically schedule the tasks. This allows for much better load balancing across the cores.

There exists a large body of work on dynamic task scheduling in software [1, 4, 5, 6, 8, 12, 14, 16, 22, 24, 26, 29, 34, 35]. For previously studied applications, overheads of software dynamic task schedulers are small compared to the size of the tasks, and therefore, enable sufficient scalability.

We examine a set of benchmarks from an important emerging application domain, Recognition, Mining, and Synthesis (RMS) [10, 32]. Many RMS applications have very high compute demands, and can therefore benefit from a large amount of acceleration. Further, they often have abundant thread-level parallelism. Thus, they are excellent targets for running on large-scale CMPs.

However, we also find that a significant number of these RMS benchmarks are dominated by parallel sections with small tasks. For these, the overheads of software dynamic task scheduling are large enough to limit parallel speedups.

In addition to emerging applications, the advent of CMPs with an increasing number of cores for mainstream computing dramatically changes how parallel programs will be written and used in a number of ways. First, CMPs reduce communication latency and increase bandwidth between cores, thus allowing parallelization of software modules that were not previously profitably parallelized. Second, unlike scientific applications, it is much more important for mainstream parallel programs to get good performance on a variety of platforms and configurations. These applications need to achieve performance benefits in a multiprogrammed environment where the number of cores can vary not only across runs but also during a single execution. These changes motivate the need to support fine-grained parallelism efficiently.

We therefore propose Carbon, a hardware technique to accelerate dynamic task scheduling on scalable CMPs. Carbon consists of two components: (1) a set of hardware queues that cache tasks and implement task scheduling policies, and (2) per-core *task prefetchers* that hide the latency of accessing these hardware queues. This hardware is relatively simple and scalable.

We compare Carbon to some highly tuned software task schedulers, and also to an idealized hardware implementation of a dynamic task scheduler (i.e., operations are instantaneous). On a set of RMS benchmarks with small tasks, Carbon provides large performance benefits over the software schedulers, and gives performance very similar to the idealized implementation.

Our contributions are as follows:

1. We make the case for efficient support for fine-grained parallelism on CMPs (Section 3).

2. We propose Carbon, which provides architectural support for fine-grained parallelism (Section 4). Our proposed solution has low hardware complexity and is fairly insensitive to access latency to the hardware queues.

3. We demonstrate that the proposed architectural support has significant performance benefits (Section 6).

- Carbon delivers much better performance than optimized software implementations: 68% and 109% faster on average for 64 cores on a set of loop-parallel and task-parallel RMS benchmarks, respectively. In addition, the proposed hardware support makes it easy to consistently get good performance. This differs from software implementations where the best heuristic depends on the algorithm, data set, and the number of cores and, therefore, requires programmers to try various knobs provided to them.

- The proposed hardware delivers performance close to an idealized task queue which incurs no communication latency (newly enqueued tasks become instantaneously available to all cores on the chip). This demonstrates that the proposed hardware is very efficient.

2. BACKGROUND

This paper uses the fork-join parallelism model that is commonly used on shared-memory machines. In this model, the program starts serially and alternates between serial and parallel sections. We will use the term *task* to denote an independent unit of work that can be executed in parallel with other tasks. In a serial section, only one thread is executing the code. Upon entering a parallel section, a group of threads are started[1] that execute the tasks in that section in parallel.

2.1 Dynamic Load Balancing

Good parallel scaling requires the load to be balanced among the participating threads. Load imbalance in a parallel section is a function of the variability of the size of the tasks as well as the number of tasks. The lower the variability, the fewer tasks are needed to obtain good load balance. However, as the number of tasks increases, parallelization overhead increases. Therefore, selecting the right task granularity involves a trade-off between parallelization overhead and load imbalance.

Task queuing is a well-known technique that is primarily designed to address the load imbalance problem. To use task queues, a programmer decomposes a parallel section into tasks. The runtime system is responsible for scheduling tasks on a set of persistent threads so as to minimize the load imbalance.

Task scheduling impacts other aspects of the program execution; therefore, using the right scheduling policy is important to scalability and yields other benefits including:

- **Improve cache locality:** Intelligent ordering and scheduling of tasks can significantly improve cache performance [4, 27, 30].

- **Minimize lock contention:** Contention on locks can be reduced (and the lock accesses can be made local) by scheduling tasks that access the same shared data on the same thread.

- **Control the amount of parallelism:** In tree-structured concurrency (where tasks form a tree and each task/node has dependencies on its parent or children), the amount of parallelism

can be controlled by changing the task execution order. Typically, a LIFO, or depth-first, order has better cache locality (and smaller working set) while a FIFO, or breadth-first, order exposes more parallelism.

- **Simplify multithreading:** A simple task queuing API can hide threading details and can allow the programmer to focus on the algorithmic aspects of the parallel program.

2.2 Types of Parallelism

In this paper, we focus on two types of parallelism—*loop-level* and *task-level*—that are commonly supported by parallel languages. Loop-level parallelism refers to situations where the iterations of a loop can be executed in parallel. Task-level parallelism refers to a more dynamic form of parallelism where a parallel task can be spawned at any point.

Loop-Level Parallelism:. Loop-level parallelism is easy to expose because this often requires little more than identifying the loops whose iterations can be executed in parallel. Numerous parallel languages including OpenMP [29], HPF [17], and NESL [3] support loop-level parallelism.

Strictly speaking, task-level parallelism is a superset of loop-level parallelism. However, we consider these separately for two reasons. First, loop-level parallelism accounts for a large fraction of kernels in parallel programs. Second, loop-level parallel sections can be supported more efficiently because they are more structured than general task-level parallel sections. A set of consecutive tasks in a parallel loop can be expressed compactly as a range of the iteration indices. In addition, a loop enqueues a set of tasks at once which allows for some optimizations.

Task-Level Parallelism:. Task-level parallelism [4, 14, 22, 34] allows a broader class of algorithms to be expressed as parallel computation. For instance, a tree-structured computation where the parent node depends on the result of the children nodes (or vice versa) is easily expressed with task-level parallelism. In general, any dependency graph where each node represents a task and the direct edges represent dependencies between tasks can be expressed.

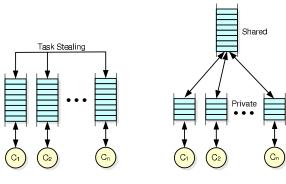### 2.3 Software Task Queuing Implementations

Static scheduling is sometimes effective in scheduling tasks when there is little variability in the task size. In static scheduling [24], tasks of an application are usually mapped to processors at the start of the parallel region. However, static scheduling has significant limitations, especially on mainstream CMPs (Section 3).

A centralized queue is the simplest way of implementing dynamic load balancing. In this case, all threads enqueue and dequeue from a single shared queue. While this is sometimes acceptable, a single queue can quickly become a bottleneck as the number of threads scales up.

Figure 1 shows two commonly used techniques to implement efficient task queues in software, namely distributed task stealing and hierarchical task queuing.

Distributed Task Stealing:. This technique [1, 5, 6, 8, 16, 26, 35] is the most popular way of implementing task queues. In this scheme, each thread has its own queue on which it primarily operates. When a thread enqueues a task, it places it in its own queue. When it finishes executing a task and needs a new task to execute, it first looks at its own queue. Both enqueue and dequeue operations by a thread on its own queue are always performed at the same end (which we will call the *head* of the queue). When a thread needs a task and there are no tasks available in its own queue, it *steals* a task

---

[1]In practice, spawning and exiting kernel threads is expensive. Implementations typically just suspend and reuse threads between parallel sections as an optimization.

(a) Distributed Task Stealing   (b) Hierarchical Task Queuing

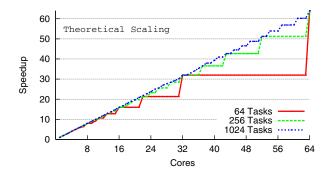**Figure 1: Standard task queue implementations**

Figure 2: Theoretical Scalability: Shows the speedup that could be achieved if the parallel program could be statically partitioned into a given number of tasks of equal size.

from one of the other queues. Task stealing is always performed at the *tail* of a queue. Note that each of the queues is shared and needs to be protected by locks.

The distributed task stealing scheme has two good properties. First, randomized distributed task stealing[2] has been shown to be provably efficient both in terms of execution time and space usage for a certain broad class of parallel applications (called fully strict) [5]. Second, distributed task stealing exploits the significant data sharing that often occurs between a task (child) and the task that spawned it (parent) by frequently executing the child on the same thread as the parent.

Hierarchical Task Queuing:. This approach is a refinement of a centralized queue. To alleviate the serialization at the shared global queue, each thread maintains a small private queue on which it primarily operates. Note that a private queue is only ever accessed by its owner thread and does not need a lock. When a private queue fills up, some of its tasks are moved to the global queue. When a private queue is empty, it gets some tasks from the global queue.

Typically, distributed task stealing results in better performance than hierarchical task queuing due to its better caching behavior. However, when the available parallelism is limited, hierarchical task queues can sometimes perform better.

3. A CASE FOR FINE-GRAINED PARALLELISM

Previous work on dynamic load balancing [1, 4, 5, 6, 8, 12, 14, 16, 22, 24, 26, 29, 34, 35] targeted coarse-grained parallelism. By this, we mean parallel sections with either large tasks, a large number of tasks, or both. They primarily targeted scientific applications for which this assumption is valid [2, 36]. For these applications, an optimized software implementation (Section 2.3) delivers good load balancing with acceptable performance overheads.

The widespread trend towards an increasing number of cores becoming available on mainstream computers—both at homes and at server farms—motivates efficient support for fine-grained parallelism. Parallel applications for the mainstream are fundamentally different from parallel scientific applications that ran on supercomputers and clusters in a number of aspects. These include:

Architecture:. CMPs dramatically reduce communication latency and increase bandwidth between cores. This allows parallelization of modules that could not previously be profitably parallelized.

Workloads:. To understand emerging applications for the multi-core architecture, we have parallelized and analyzed emerging applications (referred to as RMS [10, 32]) from a wide range of areas including physical simulation for computer games [28] as well as for movies [11], raytracing, computer vision, financial analytics [20], and image processing. These applications exhibit diverse characteristics. On one hand, a number of modules in these applications have coarse-grained parallelism and are insensitive to task queuing overheads. On the other hand, a significant number of modules have to be parallelized at a fine granularity to achieve reasonable performance scaling.

Recall that Amdahl's law dictates that the parallel scaling of an application is bounded by the serial portion. For instance, if 99% of an application is parallelized, the remaining 1% that is executed serially will limit the maximum scaling to around 39X on 64 threads. This means that even small modules need to be parallelized to ensure good overall application scaling.

Performance Robustness:. Parallel scientific computing applications are often optimized for a specific supercomputer to achieve best possible performance. However, for mainstream parallel programs, it is much more important for the application to get good performance on a variety of platforms and configurations. This has a number of implications that require exposing parallelism at a finer granularity. These include:

- The number of cores varies from platform to platform. Further, mainstream parallel applications run in a multiprogrammed environment where the number of cores can vary not only across runs but also during a single execution. To achieve performance improvements from each additional core requires finer granularity tasks. Figure 2 illustrates this with a simple example. Consider a parallel program that can be broken down into equal sized tasks. If the program is split into 64 tasks, it can theoretically achieve perfect scaling (32X) on 32 cores because each core can execute 2 tasks. However, if it executes on 33 cores, it achieves no improvement in performance over 32 cores because the performance is dictated by the thread that takes the longest. In this case, most of the threads still have to execute 2 tasks each. In fact, this program would not run any faster until 64 cores are made available to it. As the figure shows, the situation can be improved by increasing the number of tasks.

---

[2]A variant of distributed task stealing where the queue from which a task is stolen is chosen at random.

- CMPs are often asymmetric because many are composed from cores supporting simultaneous multithreading. Two threads sharing a core run at a different rate than two threads running on two different cores. In addition, CMPs may be designed with asymmetric cores [13, 15]. To ensure better load balancing in the presence of hardware asymmetry, it is best to expose parallelism at a fine grain.

Ease of Parallelization:. The use of modularity will continue to be very important for mainstream applications. Further, parallelized libraries are an effective way of delivering some of the benefits of parallelization to serial applications that use them [21]. This requires the module/library writer to expose the parallelism within each module. The result will be finer granularity tasks.

We should clarify that our claim is not that all (or even a majority) of modules in applications need to exploit fine-grained parallelism. However, for all the reasons listed above, a significant number of modules will benefit from exploiting fine-grained parallelism. The goal of this work is to provide efficient hardware support for fine-grained parallelism.

4. ARCHITECTURAL SUPPORT FOR FINE-GRAINED PARALLELISM

The overheads involved in software implementations of task queues restrict how fine-grained the tasks can be made and still achieve performance benefits with a larger number of cores. Therefore, we propose Carbon, a hardware design for scalable CMPs that accelerates task queues. Carbon provides low overhead distributed task queues, as described in Section 2.3, and is tolerant to increasing on-die latencies. Carbon achieves this by implementing key functionality of distributed task queues in hardware. In particular, we store tasks in hardware queues, implement task scheduling in hardware, and prefetch tasks from the task storage to the cores so that each thread can start a new task as soon as it finishes its current one. We use distributed task stealing as a basis for our design rather than hierarchical queues because distributed task stealing often results in significantly better cache performance (Section 6.2).

A software library provides a wrapper for Carbon so that programmers can work with an intuitive task queue API. Multiple library implementations are possible. In fact, we expect that there will be different implementations, for example, for applications dominated by loop-level parallelism versus task-level parallelism.

4.1 Task Structure

From the task queue hardware perspective, a task is simply a tuple. In the current implementation, it is a tuple of four 64-bit values. Carbon does not interpret the contents of the tuple. This provides flexibility to the software as well as the ability to optimize special cases. The software library wrapper that uses Carbon determines the meaning of each entry in a tuple. Typically, the entries will be function pointers, jump labels, pointers to shared data, pointers to task-specific data, and iteration bounds, but could be anything.

There is one instance in which Carbon assigns meaning to two fields of the tuples to provide efficient support for parallel loops (see TQ_ENQUEUE_LOOP in Section 4.2).

4.2 ISA Extension

For Carbon, after the task queue is initialized, the various threads in the program perform task enqueues (i.e., add a task to the pool of tasks stored in the task queues so that it can be executed in the future) and dequeues (i.e., remove a task from the pool of tasks stored in the task queues for execution).

The hardware task queues have only limited capacity. In order to support a virtually unbounded number of tasks for a given process, and to support a virtually unbounded number of processes, we treat the hardware queues as a cache. Therefore, we provide mechanisms to move tasks out of the hardware task queues into the memory subsystem and back. Carbon triggers user-level exceptions when the number of tasks goes above or below certain thresholds, which allows a software handler (in the wrapper library) to move tasks between the hardware task queues and memory.

We extend the ISA to use Carbon with the following instructions and exceptions:

- TQ_INIT: This instruction specifies the number of threads in the process and initializes Carbon for all the threads.
- TQ_END: This instruction signals that no new tasks will be enqueued by the process. Any threads in the process that are blocked on a TQ_DEQUEUE (see below) instruction are unblocked without returning a task. This is typically invoked at the end of the program.
- TQ_ENQUEUE: This instruction adds a task to the pool of tasks stored in the task queues.
- TQ_ENQUEUE_LOOP: This instruction is intended to greatly accelerate the enqueue process for parallel sections with loop-level parallelism. This is the sole instruction where Carbon assigns any meaning to fields of the tuples. Consider a parallel loop where the loop index starts at zero and goes to N and where we want the granularity of a task to be S iterations.[3] In this case, the TQ_ENQUEUE_LOOP instruction is invoked with the tuple: $< v1, v2, N, S >$. This results in a set of $\lceil \frac{N}{S} \rceil$ tasks being enqueued, each of the form: $< v1, v2, b, e >$ where b is the starting iteration and e is the ending iteration for that task. As before, Carbon assigns no meaning to $v1$ and $v2$ (it is up to the software library to assign their meanings).
- TQ_DEQUEUE: This instruction tries to remove a task from the pool of tasks stored in the task queues. If a task is available, the task is removed from the pool and returned by this instruction. If no tasks are available, the instruction blocks until one becomes available and then returns it. There are two situations in which this instruction completes without finding a task. First, a different thread has executed TQ_END signaling that no new tasks are going to be enqueued in the process. Second, all threads in the process are blocked on TQ_DEQUEUE. In this case, the main thread (the one that invoked TQ_INIT) will execute its TQ_DEQUEUE instruction without returning a task. This effectively works like a barrier at the end of a parallel section. The main thread can then continue executing instructions. Typically, it will continue executing the serial section following the parallel section just completed. When it reaches the next parallel section, it can enqueue new tasks.
- TQ_Overflow exception: This exception is generated if a thread tries to enqueue a task when there is no space left in the hardware queues. The exception handler can then remove some of the tasks (e.g., a third of them) and store them in some data structure in memory. This allows the process to enqueue a group of new tasks before triggering another overflow exception. After an overflow, an overflow bit is set for that process to indicate that there are tasks in memory.
- TQ_Underflow exception: If the overflow bit is set (indicating that there are some tasks that were overflown into software) and the number of tasks falls below a certain, programmable threshold, this exception is generated. This allows the software to move some of the tasks out of a software data structure back

---

[3]Loops that do not conform to this can typically be mapped into this form.
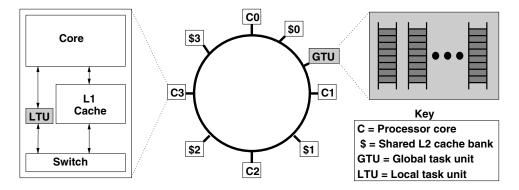
Figure 3: Example CMP with Carbon hardware support for task queues. The shaded portions are additional hardware for Carbon.

into the hardware queues. By doing so before the task queue is completely empty, this prevents the threads from waiting for tasks while the task queues are being refilled.

- TQ_SETLIMIT: This instruction specifies the threshold for triggering an underflow exception. If the threshold is 0, it clears the overflow bit, preventing the TQ_Underflow exception from being triggered.

4.3 Design

In this work we consider a CMP where the cores and last-level cache are connected by an on-die network. Carbon has two components: a centralized global task unit and a per-core local task unit. Figure 3 shows a CMP with Carbon.

4.3.1 Global Task Unit (GTU)

Carbon's global task unit holds enqueued tasks in a set of hardware queues. Carbon implements logically distributed task queues; thus, it has a hardware queue per hardware thread (hardware context) in the system. The queues support insertion and deletion for tasks at either end (but not in the middle). Thus, while we refer to these structures as queues, they are actually double-ended queues. By disallowing random access to the queues we keep the hardware complexity of the queue implementation low.

Recall that a tuple enqueued by TQ_ENQUEUE_LOOP expands into multiple tuples. The amount of buffer space used in the GTU can be greatly reduced by doing this expansion lazily.

The global task unit also implements the task scheduling policies described in Section 2.3. Since the queues are physically located close to each other, the communication latency between the queues is minimized. This allows for fast task stealing and also fast determination of when all tasks are complete. Communication between the global task unit and the cores uses the same on-die interconnect as the cache subsystem. The downside of physically centralized hardware queues is that as the number of cores scales up, the latency for a core to communicate with the global task unit increases. We address this with *task prefetchers,* discussed below.

4.3.2 Local Task Unit (LTU)

Each core has a small local task unit to interface between the core and the global task unit. Although not necessary, the local task unit also contains hardware to hide the latency of dequeuing a task from the global task unit (up to 50 cycles in the system we study). If a thread waits to send a dequeue request to the global task unit until it is done with its current task, it will stall waiting for the next task. If tasks are small enough, these stalls may be a significant fraction of execution time. Therefore, the local task unit includes a task prefetcher and a small prefetch buffer.

Task dequeues. On a dequeue, if there is a task in the prefetch buffer, that task is returned to the thread and a prefetch for the next available task is sent to the global task unit. We find that for our system and benchmarks, prefetching and buffering a single task is sufficient to hide the dequeue latency. We study sensitivity to the global task unit access latency in Section 6.

When the global task unit receives a prefetch request, it returns a task to the requester, removing it from the hardware queues. That is, a task in a prefetch buffer is not visible to the global task unit and so cannot be stolen by other threads. When there are very few tasks in the system, this can lead to load imbalance if one thread has a task sitting in its prefetch buffer and another thread has nothing to do; therefore, when the global task unit holds fewer tasks than there are threads, it rejects prefetches.

Task enqueues. Since Carbon uses a LIFO ordering of tasks for a given thread, when a thread enqueues a task, it is placed in the thread's task prefetch buffer. If the buffer is already full, the oldest task in the buffer is sent to the global task unit.

If a thread executes a TQ_ENQUEUE_LOOP, it does not buffer it, but instead sends it directly to the global task unit.

4.4 Software Issues

The contents of the hardware queues in the global task unit and the prefetch buffer in the local task unit are additional state kept for each process. This raises the questions of how Carbon handles context switching and a multi-programming environment.

Carbon handles context switches as follows. An application provides a pointer to a memory region in its application space where the tasks can be "swapped out" when the process is context-switched out. On a context switch, the OS then moves all the tasks to the specified region. The underflow mechanism moves the tasks back into the hardware queues when the process is context-switched in. Since the prefetch buffer we propose is only a single entry, it is treated as a small set of special-purpose registers (one for each entry in the tuple).

Carbon supports multiprogramming by maintaining separate pools of tasks for each process. Since there is a hardware queue per hardware thread context, individual hardware queues belong to a specific process (the one running on the corresponding hardware context). This ensures that each hardware queue only has tasks from one process, and requires only a process id per hardware queue. The task assignment logic also uses the process id to only poll the queues belonging to the process when task stealing and determining if all tasks are complete.

Processor Parameters	
# Processors	1–64
Processor width	2
Memory Hierarchy Parameters	
Private (L1) cache	32KB, 4-way, 64B line
Shared L2 cache	16MB, 16 banks, 8-way
Interconnection network	Bi-directional ring
Contentionless Memory Latencies	
L1 hit	3 cycles
L2 hit	18–58 cycles
Main memory access	298–338 cycles

Table 1: Simulated system parameters.

4.5 Alternative Designs

For loop-level parallelism, we also experimented with an alternative design similar to the Virtual Vector Architecture (ViVA) proposal [25, 33]. In this design, a virtual vector unit automatically decomposes vectorizable loops into chunks and distributes them to a group of processors. This unit decides an appropriate chunk size based on the number of iterations, loop body, and the latency to distribute chunks to the different processors. Experimental results show that the performance of this design and Carbon is comparable. However, unlike Carbon, ViVA does not address task-level parallelism.

Another design point that we implemented included a component similar to LTU on each core. However, it did not include a GTU. This design performed buffering and prefetching similar to Carbon but used a bigger local buffer. This scheme maintained the global pool of tasks in memory (similar to software distributed task stealing) which is accessed by the hardware whenever the local buffer overflows or underflows. However, Carbon yielded much better performance compared to this design.

5. EVALUATION FRAMEWORK

5.1 System Modeled

We use a cycle-accurate, execution-driven CMP simulator for our experiments. This simulator has been validated against real systems and has been extensively used by our lab. Table 1 summarizes our base system configuration.

We model a CMP where each core is in-order and has a private L1 data cache, and all processors share an L2 cache. Each L1 cache has a hardware stride prefetcher. The prefetcher adapts how far ahead it prefetches—if it detects that it is not fully covering memory access latency, it issues prefetches farther out. The processors are connected with a bi-directional ring, and the L2 cache is broken into multiple banks and distributed around the ring. Inclusion is enforced between the L1s and L2. Coherence between the L1s is maintained via a directory-based MSI protocol. Each L2 cache line also holds the directory information for that line. The ring has 41 stops, each of which can have two components connected to it (i.e., processor core, L2 cache bank, or global task unit).

For experiments with Carbon, we add hardware as described in Section 4.3 to the system. We charge eight cycles for an access (e.g., enqueue or dequeue) to the global task unit. This is in addition to the latency from the cores to the global task unit over the on-chip network. The global task unit can initiate handling for one access per cycle. In our design, all accesses to the global task unit (whether accessing a thread's own queue or stealing task from another queue) are fully pipelined and take the same number of cycles. An alternative would be to make accesses to a thread's own

queue faster and pay extra overhead for work stealing. However, the lower design complexity together with the relative insensitivity to this latency (Section 6.2) motivated our choice.

We introduce new instructions to allow software to interact with the task queue hardware, as described in Section 4.2. Each of these instructions has a latency of five cycles. This includes the time to access the local task unit. Any interaction with the global task unit incurs additional latency as explained above.

5.2 Task Queue Implementations

We compare the performance of Carbon with a number of other implementations to evaluate its benefits. As explained in Section 2.2, we consider loop-level and task-level parallelism separately.

All task queue implementations spawn worker kernel threads during initialization since spawning threads is expensive. The number of threads is the same as the number of cores in our experiments. In serial sections, all the worker threads wait in an idle loop for parallel tasks to become available.

Loop-Level Parallelism:. Loop-level implementations are different from task-level implementations in a number of ways that makes them significantly more efficient. First, they use jumps and labels to transfer control between the tasks and the idle loop. Second, they exploit the fact that a list of tasks from consecutive iterations can be expressed as a range (i.e., a pair of numbers) rather than maintaining explicit queues. They also allow dequeues to be performed more efficiently using atomic-decrement instructions instead of using locks. Finally, since all tasks are enqueued at once at the start of the loop, these implementations do not need to handle the case where new tasks become available dynamically. This allows the end of the parallel section to be implemented using an efficient tree barrier.

We evaluate the loop-level benchmarks using three implementations.

- *S/W* is an optimized version of the software implementation that uses distributed task stealing (Section 2.3) and the optimizations described above.
- *Carbon* uses our proposed architecture support from Section 4. We use a very thin interface between the application and the task queue hardware for this implementation.
- *Ideal* is an idealized version of *Carbon*. It uses the same instructions as *Carbon* to interact with the global task unit, and we charge the same latency for them. However, these instructions immediately affect the global task unit and do not require sending a message over the interconnect. This means that tasks that are enqueued are immediately visible to all threads. Also, *Ideal* does not use a local task unit since there is no latency to hide.

For these loop-level parallelism experiments, we assume that the L2 cache is warmed up, since this is the common case when these operations are used.

Task-Level Parallelism:. These implementations have to handle the more general case where a task can be enqueued at any point in the parallel section. Unlike the loop-level implementations, a task is represented here as a function pointer together with a set of arguments. This makes a task more general but incurs function call overheads for each task.

We evaluate the task-level benchmarks using four implementations. We found that there was no single software implementation that consistently performed best. Therefore, we include results from two different software implementations.

Benchmark & Description	Datasets	# of Tasks	Avg. Task Size[†]
Loop-Level Parallelism			
Gauss-Seidel (**GS**)	128x128, 2 iterations	512	1704
Red-Black Gauss-Seidel on a 2D Grid	512x512, 2 iterations	2048	6695
Dense Matrix-Matrix Multiply (**MMM**)	64x64	256	806
Both matrices are dense	256x256	4096	3067
Dense Matrix Vector Multiply (**MVM**)	128x128	128	1195
Both matrix and vector are dense	512x512	512	4679
Sparse Matrix Vector Multiply (**SMVM**)	c18: 2169 rows, 2169 columns, 8657 non-zeros	543	329
Matrix is sparse, Vector is dense	gismondi: 18262 rows, 23266 columns, 136K non-zeros	4566	588
Scaled Vector Addition (**SVA**)	4K elements	128	599
Computes $V_3 = a \times V_1 + b \times V_2$	16K elements	512	598
Task-Level Parallelism			
Game physics solver (**GPS**)	model1 : 800 bodies, 14859 constraints, 20 iterations	63436	3285
Constraint solver for physical simulation in games	model4 : 4907 bodies, 96327 constraints, 20 iterations	402754	4118
Binomial Tree (**BT**)	Tree of depth 512	595	8477
1D Binomial Tree used for option pricing	Tree of depth 2048	9453	8765
Canny edge detection (**CED**)	camera4 : 640x480 image of a room	41835	739
Detecting edges in images	costumes : 640x480 image of people	127699	335
Cholesky Factorization (**CF**)	world: 28653 columns, 1.33M non-zeros	189082	13876
Cholesky factorization on a sparse matrix	watson: 209614 columns, 3.78M non-zeros	641330	5364
Forward Solve (**FS**)	mod: 28761 columns, 1.45M non-zeros	19558	4730
Forward Triangular solve on a sparse matrix	pds: 15648 columns, 1.18M non-zeros	13855	5045
Backward Solve (**BS**)	ken: 78862 columns, 2.18M non-zeros	70956	2231
Backward Triangular solve on a sparse matrix	world: 28653 columns, 1.33M non-zeros	19372	3005

Table 2: Benchmarks. [†]**The average task size (in cycles) is from a one-thread execution with** *Carbon*.

- *S/W Distributed Stealing* is a software implementation of distributed task queues (Section 2.3) that has been heavily optimized for small tasks.
- *S/W Hierarchical* is an optimized software implementation of hierarchical task queues (Section 2.3).
- *Carbon* uses our proposed architecture support from Section 4. The application interfaces with the hardware using a library that supports the same API as the software implementations.
- *Ideal* is an idealized version of *Carbon* analogous to that described earlier for loop-level parallelism.

All implementations described here have been developed and optimized over two years. This included detailed analysis of execution traces and, when profitable, using assembly instructions directly. In addition, the software implementations employ a number of heuristics (for instance, the number of tasks that are moved between queues) that can have a big impact on performance [18, 19, 23] and benefited from careful tuning. The performance of these implementations was validated by benchmarking them against Cilk [4], TBB [22], and OpenMP [29]. These implementations have been used to parallelize over a dozen workloads each with thousands of lines of code.

5.3 Benchmarks

We evaluate our proposed hardware on benchmarks from a key emerging application domain: Recognition, Mining, and Synthesis [10, 32]. All benchmarks were parallelized within our lab. We give the benchmarks and the datasets used in Table 2.

Loop-level parallelism:. We use primitive matrix operations and Gauss-Seidel for our loop-level parallelism experiments since these are both very common in RMS applications and very useful for a wide range of problem sizes. Their wide applicability allows an optimized and parallelized library to deliver benefits of parallelization to serial programs (Section 3). Delivering good parallel speedups to smaller problem sizes increases the viability of this approach.

Most of these benchmarks are standard operations and require little explanation. The sparse matrices are encoded in compressed row format. *GS* iteratively solves a boundary value problem with finite differencing using red-black Gauss-Seidel [31]. The matrix elements are assigned red and black colors (like a checker board). Each iteration requires two passes: the first updates values for the red elements while the second updates the values for the black elements. Each update accesses values on its 4 neighbors.

These benchmarks are straightforward to parallelize; each parallel loop simply specifies a range of indices and the granularity of tasks. For instance, in *SMVM*, each task processes 4 rows of the matrix.

We evaluated each benchmark with two problem sizes to show the sensitivity of performance to problem sizes.

Task-level parallelism:. We use modules from full RMS applications for our task-level parallelism experiments. They represent a set of common modules across the RMS domain. Some of the benchmarks are based on publicly available code, and the remaining ones are based on well-known algorithms.

These benchmarks are: (1) *GPS* iteratively solves a set of force equations in a game physics constraint solver [28]. The equations are represented as a set of constraints, each of which involves two objects. Solving a constraint on an object must be atomic. Therefore, a task is to solve a single constraint, and the task dependences form a graph. The benchmark is iterative, so for maximum performance, the task graph has cycles rather than having a barrier between iterations. (2) *BT* uses a 1D binomial tree to price a single option [20]. The task dependences form a tree, where work starts at the leaves. (3) *CED* computes an edge mask for an image using the Canny edge detection algorithm [7]. The most expensive parallel region in this benchmark grows the edge mask from some seeded locations in a breadth-first manner. A task is to check if a single pixel should be classified as edge and if its neighbors should also be checked; therefore, the task dependences form a graph. (4) *CF* performs Cholesky factorization on a sparse matrix [21]. It involves bottom-up traversal of a tree where each node is a task and the edges represent dependences between tasks. (5) *BS* performs a backward triangular solve on a sparse matrix [21]. The matrix is pre-partitioned into groups of independent columns. A task is to perform the computation on the elements in a group of columns.

The task dependences form a tree, where work starts from the root. (6) *FS* performs a forward triangular solve on a sparse matrix [21], similar to *BS*. For this benchmark, the work starts from the leaves of the task dependence tree.

6. EVALUATION

6.1 Performance benefits of Carbon

Figures 4 & 5 show the relative performance of the different implementations for the loop-level and the task-level benchmarks, respectively, when running with 1, 2, 4, 8, 16, 32, and 64 cores.

Carbon vs. optimized software.. Carbon delivers significantly better performance than the software implementations (*S/W*, *S/W Distributed Stealing* and *S/W Hierarchical*). For the loop-level benchmarks in Figure 4, Carbon executes between 66% and 207% faster than the software version on 64 threads for the smaller problem sizes. Even for the larger problem sizes, the performance benefit is substantial (20% for *SMVM* and 73% for *SVA*). For the task-level benchmarks in Figure 5, Carbon executes up to 435% faster (for *GPS* on *model1*) and 109% faster on average compared to the best software version.[4] In *GPS* on *model1*, the parallelism available is limited, especially for larger numbers of cores. In the software implementations, the cores contend with each other to grab the few available tasks, which adversely impacts performance.

Carbon vs. Ideal.. Carbon delivers performance similar to *Ideal* in most cases (Carbon is 3% lower on average). For loop-level benchmarks, Carbon executes around 12% and 7% slower than *Ideal* for the smaller datasets of *SMVM* and *SVADD*, respectively. This is due to the small size of the parallel sections. For task-level benchmarks, Carbon is slower in two instances: by 10% for *FS* on *pds* and 17% for *GPS* on *model1*. In these instances, the amount of parallelism available is very limited. Consequently, the tasks that are buffered in local tasks units (we currently buffer at most one in each unit) are unavailable to idle cores. This hurts performance. Note that in a few instances (e.g., *BT* for *2048*), Carbon performs marginally better than *Ideal*. This is because of second-order effects such as caching and prefetching due to changes in the execution order of tasks.

Comparing software implementations.. Finally, comparing the two optimized software implementations for the task-level benchmarks in Figure 5 demonstrates two points. First, neither *S/W Distributed Stealing* nor *S/W Hierarchical* consistently performs better than the other. The best heuristic varies not only with the benchmark but also with the dataset and the number of cores. To allow a programmer to get the best performance, the software implementations have to provide a number of knobs (which control the heuristics) to the programmer. This makes performance tuning cumbersome. Second, the performance of software implementations can sometimes drop dramatically as the number of cores is increased (e.g. *GPS* for *model1*), as explained above.

6.2 Sensitivity Analysis

Benefit of using distributed queues.. Carbon uses distributed queues (one per hardware context) to store the tasks. An alternative would be to use a single LIFO queue for this purpose.

---

[4]For each benchmark/dataset combination, we use the execution time of the better performing software implementation.

Figure 6 (a) shows the slowdown from using a single LIFO queue instead of distributed queues. This experiment was performed with the idealized queues to avoid second-order effects due to implementation choices of Carbon. For each benchmark, the **A** and **B** bars correspond to the first and second datasets in Table 2, respectively. Overall, the slowdown from using a single queue is quite significant (35% on average). For the task-level benchmarks, the primary advantage of distributed queues is that the parent and child tasks, which usually share data, are often executed on the same core. This results in a much better L1 hit ratio. For the loop-level benchmarks, each queue is assigned contiguous iterations when using distributed queues. On the other hand, using a single queue results in a round-robin distribution of loop iterations to the cores. This results in less temporal locality when consecutive iterations share data (*GS* and *MMM*). In addition, the prefetcher efficiency is worse with a single queue for all loop-level benchmarks. This is because the stride is broken when the next task does not start its iterations right after where the previous task ended.

Sensitivity to the latency to the GTU.. Carbon uses LTUs to hide the latency to the GTU by buffering and prefetching tasks. Here, we measure the effectiveness of the latency tolerance mechanism of Carbon. The more latency tolerant Carbon is, the more flexibility a chip designer has.

Figure 6 (b) shows the slowdown that would occur if we dramatically increased the latency to access the GTU. In our current design, the GTU takes 8 cycles to process each request. We increased this latency to 280 cycles (i.e., main memory access time) and measured the performance impact.

For the task-level benchmarks, the performance degradation is minimal for all benchmarks except *CED*. While the average task sizes in our benchmarks are fairly small (Table 2), *CED* has a lot of very small tasks (around 50 cycles). The current implementation of Carbon buffers only one task in the local task unit. This is insufficient to hide the 280 cycle latency. However, this is easily addressed by increasing the buffer size. The slowdown dropped from 19% to 6% with buffer size 2, to 3% with buffer size 3, and to 0.5% with buffer size 6.

For the loop-level benchmarks, the performance degradation is larger, especially for small problem sizes. This is because the parallel sections are fairly small. The impact is mostly due to the latencies of starting and ending a parallel section. The local task unit can not help in reducing these overheads.

Note that the goal of this experiment was to measure the sensitivity by dramatically increasing the latency. Realistically, there is no reason for the latency to be this large (280 cycles). Consequently, the actual impact for reasonable latencies will be much smaller.

Performance due to LTU in Carbon.. Carbon has two hardware components: LTU and GTU. Figure 7 shows the percentage increase in execution time if the task buffering and prefetching feature of LTU is disabled. The results are shown for two Carbon configurations: (1) the baseline Carbon design, and (2) Carbon with a 280 cycle latency to the GTU. This experiment measures the contribution of LTU to the overall performance.

For the baseline Carbon design, the slowdown is up to 10.3% and 14.8% respectively for the loop-level and task-level benchmarks. For the higher-latency configuration, the slowdown is up to 39.1% and 25.8%, respectively.

These results demonstrate that, for the baseline design, task buffering and prefetching in LTU accounts for a smaller fraction of the performance benefit than the GTU. However, it makes Carbon very latency tolerant. Recall that while the buffering and prefetching
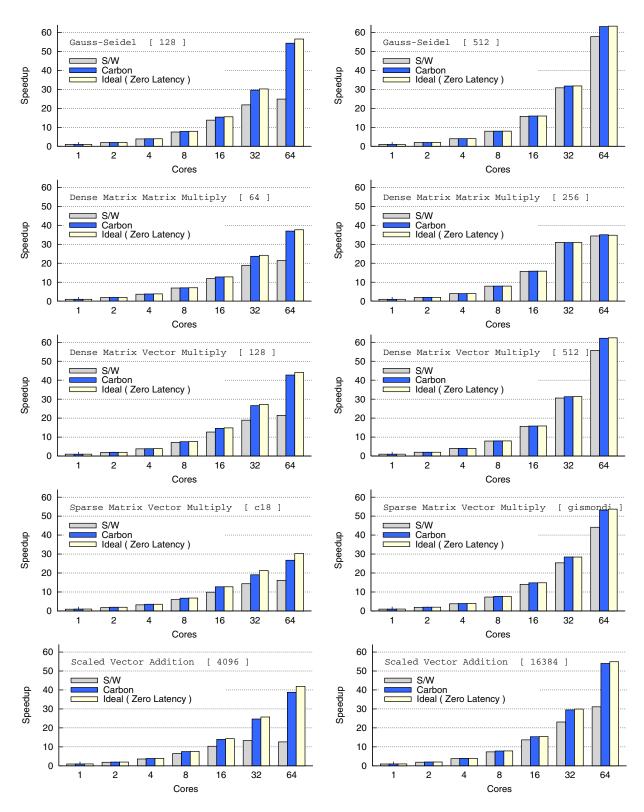
Figure 4: Loop-Level Parallelism: In each graph, all performance numbers are normalized to the one core execution time of *Ideal*. The performance scales better on the larger problem sizes (shown on the right column) than on the smaller problem sizes (on the left column) except for *MMM*. For *MMM*, the larger problem size coupled with a large number of cores saturates the on-die interconnect and prevents scaling on 64 cores.
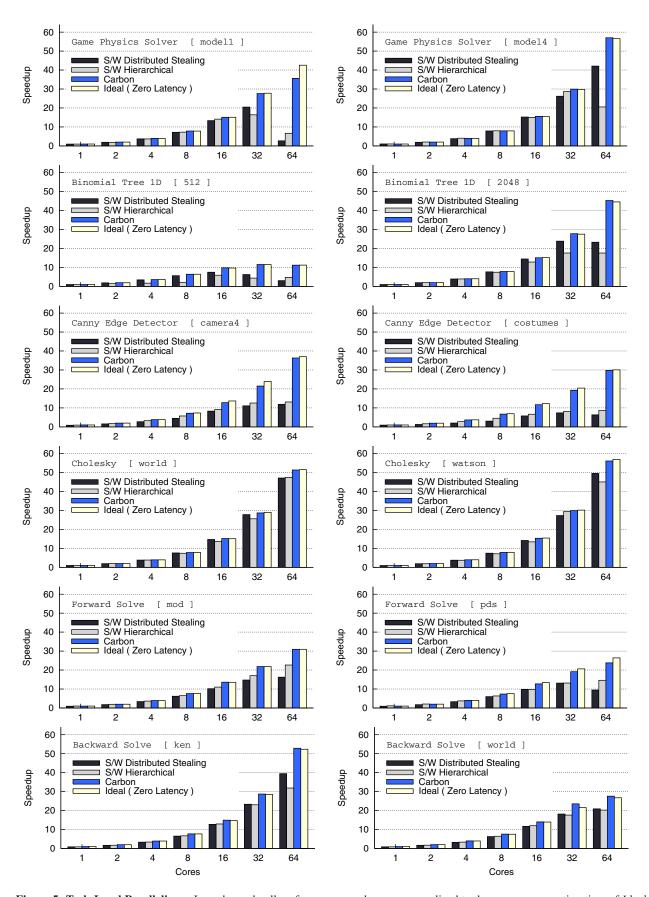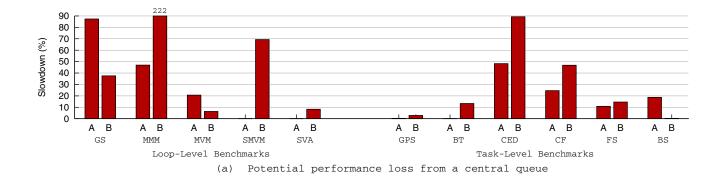
Figure 5: Task-Level Parallelism: In each graph, all performance numbers are normalized to the one core execution time of *Ideal*.

(a) Potential performance loss from a central queue

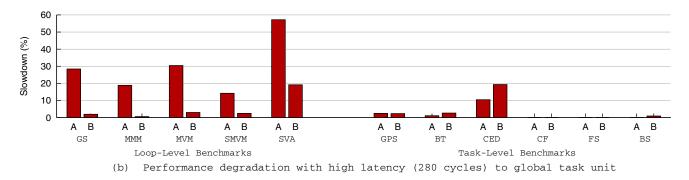(b) Performance degradation with high latency (280 cycles) to global task unit

Figure 6: Sensitivity Analysis for Carbon. For each benchmark, the A and B bars are for the first and second datasets in Table 2 respectively. Note that some bars do not show up because the value is zero at those data points.

features of LTU are optional, the portion of LTU that implements the Carbon instructions and communication with the GTU is essential to the design. The small added hardware complexity of the buffering and prefetching in LTU makes them a useful addition to Carbon.

7.   RELATED WORK

Most of the previous work on task queues is related to software implementation as already discussed. There is some previous work on using hardware to accelerate thread and task scheduling.

Korch et al. [18, 19, 23] explored different task queue implementations using a number of software as well as hardware synchronization primitives. Some of their applications are insensitive to task queue overheads. The others see benefit from hardware acceleration, but the benefit is still modest with their proposed scheme.

Hankins et al. [15] proposed Multiple Instruction Stream Processing (MISP) as a mechanism to quickly spawn and manipulate user-level threads, *shreds*, on CMP hardware contexts. Shreds are not visible to the OS and can be invoked and terminated quickly by an application thread, allowing them to efficiently execute fine-grained tasks. However, MISP leaves task scheduling to the software. Unlike MISP, Carbon uses OS managed threads and amortizes the overhead of thread creation and termination by having each thread process tasks from multiple sections. Carbon minimizes task queuing overhead by implementing task queue operations and scheduling in hardware.

Chen et al. [9] argued for a software-hardware model that allows programmers or compilers to expose parallelism while the hardware modulates the amount of parallelism exploited. They proposed Network-Driven Processor (NDP), an architecture that uses hardware to create, migrate, and schedule threads to minimize thread creation and context switching overhead. NDP's thread

scheduler is implemented in hardware to minimize scheduling delays. The hardware can quickly clone multiple threads for loop-level parallelism. In addition, NDP provides queue support in hardware for fast inter-thread communication. Despite hardware support for threading, NDP's thread overhead is still too large for small tasks. NDP shows only modest speedups for applications with a significant number of small tasks (e.g., quicksort and othello). Carbon, on the other hand, provides good scaling for parallel sections with tasks as small as hundreds of instructions.

8.   CONCLUSIONS

CMPs provide an opportunity to greatly accelerate applications. However, in order to harness the quickly growing compute resources of CMPs, applications must expose their thread-level parallelism to the hardware. We explore one common approach to doing this for large-scale multiprocessor systems: decomposing parallel sections of programs into many tasks, and letting a task scheduler dynamically assign tasks to threads.

Previous work has proposed software implementations of dynamic task schedulers, which we examine in the context of a key emerging application domain, RMS. We find that a significant number of RMS applications achieve poor parallel speedups using software dynamic task scheduling. This is because the overheads of the scheduler are large compared to the tasks for some applications.

To enable good parallel scaling even for applications with very small tasks, we propose Carbon, a hardware scheme to accelerate dynamic task scheduling. Carbon consists of relatively simple hardware and is tolerant to growing on-die latencies; therefore it is a good solution for scalable CMPs.

We compare Carbon to optimized software task schedulers and to an idealized hardware task scheduler. For the RMS benchmarks we study, Carbon gives large performance benefits over the soft-
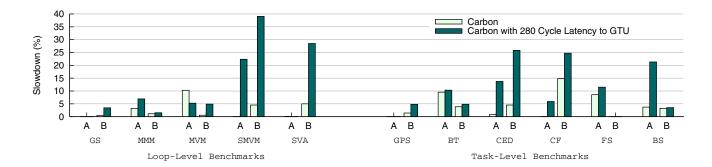
Figure 7: Performance Impact of LTU. Shows the slowdown when the task buffering and prefetching feature of LTU is disabled. For each benchmark, the A and B bars are for the first and second datasets in Table 2 respectively.

ware schedulers, and comes very close to the idealized hardware scheduler.

Acknowledgments

9. REFERENCES

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proceedings of the ACM symposium on Parallel algorithms and architectures*, 2000.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks – summary and preliminary results. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 1991.

[3] G. E. Blelloch. *NESL: A Nested Data-Parallel Language (Version 2.6)*. Pittsburgh, PA, USA, 1993.

[4] 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*, 1995.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of ACM*, 46(5):720–748, 1999.

[6] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the conference on Functional programming languages and computer architecture*, 1981.

[7] J. Canny. A Computational Approach to Edge Detection. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986.

[8] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, 2005.

[9] J. Chen, P. Juang, K. Ko, G. Contreras, D. Penry, R. Rangan, A. Stoler, L.-S. Peh, and M. Martonosi. Hardware-modulated parallelism in chip multiprocessors. *SIGARCH Comput. Archit. News*, 33(4):54–63, 2005.

[10] P. Dubey. Recognition, Mining and Synthesis Moves Computers to the Era of Tera. *Technology@Intel Magazine*, February 2005.

[11] R. Fedkiw. Physical Simulation. http://graphics.stanford.edu/ fedkiw/.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM conference on Programming language design and implementation*, 1998.

[13] M. Gschwind. Chip multiprocessing and the cell broadband engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, New York, NY, USA, 2006. ACM Press.

[14] R. H. Halstead Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the ACM Symposium on LISP and functional programming*, 1984.

[15] R. A. Hankins, G. N. Chinya, J. D. Collins, P. H. Wang, R. Rakvic, H. Wang, and J. P. Shen. Multiple instruction stream processor. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, 2006.

[16] D. Hendler and N. Shavit. Non-blocking steal-half work queues. In *Proceedings of the symposium on Principles of distributed computing*, 2002.

[17] High Performance Fortran Forum. *High Performance Fortran Language Specification, version 2.0*, 1997.

[18] R. Hoffmann, M. Korch, and T. Rauber. Performance evaluation of task pools based on hardware synchronization. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.

[19] R. Hoffmann, M. Korch, and T. Rauber. Using hardware operations to reduce the synchronization overhead of task pools. In *Proceedings of the 2004 International Conference on Parallel Processing*, pages 241–249, 2004.

[20] J. C. Hull. *Options, Futures, and Other Derivatives*, pages 346–348. Prentice Hall, third edition, 1996.

[21] *Intel®Math Kernel Library Reference Manual*.

[22] Intel Corporation. *Intel Thread Building Blocks*, 2006.

[23] M. Korch and T. Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(1), 2004.

[24] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[25] C. W. McCurdy, R. Stevens, H. Simon, W. Kramer, D. Bailey, W. Johnston, C. Catlett, R. Lusk, T. Morgan, J. Meza, M. Banda, J. Leighton, and J. Hules. Creating science-driven computer architecture: A new path to scientific leadership. *LBNL/PUB-5483*, Oct. 2002.

[26] E. Mohr, D. A. Kranz, and R. H. Halstead Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the ACM conference on LISP and functional programming*, 1990.

[27] G. J. Narlikar. Scheduling threads for low space requirement and good locality. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, 1999.

[28] *Open Dynamics Engine v0.5 User Guide*.

[29] *OpenMP Application Program Interface*. Version 2.5.

[30] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. In *Architectural Support for Programming Languages and Operating Systems*, pages 60–71, 1996.

[31] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*, pages 866–868. Cambridge University Press, second edition, 1992.

[32] J. Rattner. Cool Codes for Hot Chips: A Quantitative Basis for Multi-Core Design. HotChips keynote presentation, 2006.

[33] H. Simon, W. Kramer, W. Saphir, J. Shalf, D. Bailey, L. Oliker, M. Banda, C. W. McCurdy, J. Hules, A. Canning, M. Day, P. Colella, D. Serafini, M. Wehner, and P. Nugent. Science-driven system architecture: A new process for leadership class computing. *Journal of the Earth Simulator*, 2, Jan. 2005.

[34] E. Su, X. Tian, M. Girkar, G. Haab, S. Shah, and P. Petersen. Compiler support of the workqueuing execution model for Intel SMP architectures. In *Fourth European Workshop on OpenMP*, 2002.

[35] M. T. Vandevoorde and E. S. Roberts. Workcrews: an abstraction for controlling parallelism. *Int. J. Parallel Program.*, 17(4):347–366, 1988.

[36] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the International Symposium on Computer architecture*, 1995.