Flexible Architectural Support for Fine-Grain Scheduling

Daniel Sanchez, Richard M. Yoo, Christos Kozyrakis

Electrical Engineering Department
Stanford University
{sanchezd,rmyoo,kozyraki}@stanford.edu

Abstract

To make efficient use of CMPs with tens to hundreds of cores, it is often necessary to exploit fine-grain parallelism. However, managing tasks of a few thousand instructions is particularly challenging, as the runtime must ensure load balance without compromising locality and introducing small overheads. Software-only schedulers can implement various scheduling algorithms that match the characteristics of different applications and programming models, but suffer significant overheads as they synchronize and communicate task information over the deep cache hierarchy of a large-scale CMP. To reduce these costs, hardware-only schedulers like Carbon, which implement task queuing and scheduling in hardware, have been proposed. However, a hardware-only solution fixes the scheduling algorithm and leaves no room for other uses of the custom hardware.

This paper presents a combined hardware-software approach to build fine-grain schedulers that retain the flexibility of software schedulers while being as fast and scalable as hardware ones. We propose asynchronous direct messages (ADM), a simple architectural extension that provides direct exchange of asynchronous, short messages between threads in the CMP without going through the memory hierarchy. ADM is sufficient to implement a family of novel, software-mostly schedulers that rely on low-overhead messaging to efficiently coordinate scheduling and transfer task information. These schedulers match and often exceed the performance and scalability of Carbon when using the same scheduling algorithm. When the ADM runtime tailors its scheduling algorithm to application characteristics, it outperforms Carbon by up to 70%.

Categories and Subject Descriptors C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures (Multiprocessors); D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Design, Performance, Algorithms

1. Introduction

Chip-multiprocessors (*CMPs*) are now the mainstream approach to turn the increasing transistor budgets provided by Moore's Law into performance improvements. General-purpose CMPs with tens of cores are already available [9, 43], and chips with hundreds of cores will be available in the near future [30].

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

To use these large-scale CMPs efficiently, a program needs to explicitly divide its work in *concurrent tasks* and distribute them for execution across the available cores. A key issue is the *granularity* of this partitioning. This work focuses on programs that use *fine-grain parallelism*, with tasks as small as a thousand cycles. Fine-grain parallelism has several advantages. First, it can expose more parallelism in many applications, and for some applications parallelism is more easily expressed under this model [33]. This is particularly important for CMPs with hundreds of cores, for which parallelism becomes a precious resource [26]. Second, it gives the underlying runtime system much more freedom in distributing and reassigning work among cores in order to avoid load imbalance in irregular computations, to exploit constructive cache interference among certain tasks [18], or to adapt to environment changes such as cores becoming unavailable due to faults, thermal emergencies, or multiprogramming. On the other hand, fine-grain parallelism may introduce large overheads for representing and distributing small amounts of work, and if tasks are not assigned judiciously, locality across different tasks may be destroyed.

Fine-grain parallelism is already supported by several parallel programming models [16, 23, 29]. Their runtime systems typically implement task distribution through *work-stealing* [13]: each worker thread has a queue of ready to execute tasks, from which it can enqueue or dequeue work. When a thread runs out of tasks, it tries to steal tasks from another thread's queue. Although this technique works well in general, multiple studies have shown that many applications benefit from different algorithms in terms of the structure of queues, the order of scheduling and stealing, or the granularity of stealing [12, 21, 25]. In short, there is no single best fine-grain scheduler for all applications.

Software-only implementations of fine-grain schedulers for such programming models are flexible in terms of the algorithm used. However, they entail high overheads with fine-grain tasks, as queue operations, task stealing, and synchronization introduce communication and contention through the cache hierarchy of the CMP. The latency of a cache line transfer in CMPs with 64 or 128 cores is close to a hundred cycles, so a few such transfers can negate the benefits of parallel execution of fine-grain tasks. Such latencies will increase in large-scale CMPs, making fine-grain parallelism impractical. To mitigate this problem, Carbon [34] proposes a *hardware-only* alternative, with *specialized hardware queues* and a *custom messaging protocol* for enqueuing, dequeuing and distributing tasks across cores. Hardware implements task stealing and distribution in the background and enables applications with fine-grain parallelism to perform well on large-scale CMPs. A disadvantage of Carbon is that it introduces a non-trivial amount of custom hardware for the sole purpose of work-stealing. Ideally, we would like to minimize custom hardware structures and implement general primitives that have other uses. Moreover, Carbon fixes the scheduling algorithm in hardware, making it difficult to accelerate an application or programming model that requires a different algo-

rithm. While supporting some variations is feasible, implementing all the possible algorithms and options in hardware is prohibitively expensive in terms of design and verification complexity.

This paper advocates hardware that provides *simple and flexible support* to accelerate software schedulers for fine-grain parallelism on large-scale CMPs. We introduce *asynchronous direct messages (ADM)*, a general-purpose hardware primitive that supports sending and asynchronously receiving short messages between cores at low overhead without additional synchronization or going through the coherence protocol. ADM is tailored to integrate with cache-coherent CMPs and the shared-memory programming models for such systems, and is inspired by previous efforts in integrating message-passing in distributed shared memory machines [1, 36]. ADM provides *user-level support* for relatively infrequent messages for control purposes, while data accesses and communication occur as usual through the cache hierarchy.

ADM enables *software-mostly* schedulers that maintain the flexibility but match or exceed the performance and scalability of Carbon when using the same scheduling algorithm. Task queues are kept exclusively in software and each queue is accessed by a single thread. ADM messages are used to implement task-stealing in an asynchronous manner, without additional coherence traffic or synchronization overheads. Since software determines the scheduling algorithm, we can easily tailor it to the application characteristics. For example, we can adjust the stealing policy to improve locality, track dataflow dependencies between tasks, implement fast reduction and barrier operations, and use hierarchical scheduling approaches that scale better with the size of the CMP. Such optimizations allow ADM-based schedulers to match or outperform Carbon, sometimes by by large margins, despite using simpler hardware. Our main contributions are:

- We introduce ADM, a simple but general hardware mechanism to send messages between cores. ADM allows user-level code to send short messages (0-6 words) directly from registers, and to receive them, either synchronously or asynchronously via a user-level interrupt handler. The hardware is virtualizable, preserves message ordering, provides guaranteed delivery, and is independent of the cache hierarchy.
- We develop and present a set of novel schedulers that leverage the messaging hardware to manage fine-grain parallelism. Specifically, we use a subset of worker threads to coordinate task stealing in a distributed and scalable fashion. ADM allows threads to maintain task queues in thread-local storage, even when stealing occurs, and to overlap communication with useful computation.
- We evaluate ADM for multithreaded CMPs with up to 128 cores (256 threads) using challenging fine-grain applications. We find that our approach clearly outperforms software-only schedulers by up to a factor of 3.8× and matches or exceeds the performance of Carbon. When we tailor the ADM scheduler to the application, it outperforms Carbon by up to 70%.

2. Background & Motivation

2.1 Fine-grain Parallelism

Fine-grain parallelism is already supported by several parallel programming models [16, 21, 23, 29, 33, 46]. In a general *task-parallel* application, work is divided into tasks and dynamically assigned across worker threads for execution. Threads can dynamically generate and enqueue new tasks. The parallel phase ends when all the threads have run out of tasks to execute. To provide load balancing, the number of tasks should be significantly larger than the amount of worker threads used. Some languages restrict or adapt this model in certain ways. For example, Cilk [23] uses spawn-sync parallelism, in which tasks can generate children tasks that can only synchronize with their parents. This model works well for recursive, divide-and-conquer algorithms. OpenMP 2 [41] and HPF [28] focus on loop-level parallelism, grouping a few iterations of a parallel loop into each task. Such assumptions directly influence the design of the underlying scheduler.

The most popular fine-grain scheduling technique is work-stealing [13]. Each thread has a queue of ready tasks, to which it enqueues and dequeues work. When a thread runs out of work, it tries to steal tasks from another thread's queue. There are several algorithmic options to consider, such as the organization of the queues (distributed or hierarchical), the policy when locally enqueuing and dequeuing work (LIFO, FIFO, or priority-based), the queuing policy when stealing work, the choice of queue to steal from (from a neighbor to preserve locality or randomized stealing for more effective rebalancing), the number of tasks to steal each time, or whether a thread actively pushes work to starving threads. Several studies have shown that no single algorithm works best for all cases and that some workloads are particularly sensitive to scheduling options [21, 25, 27, 33]. Nevertheless, LIFO local enqueuing/dequeuing tends to be a default choice as it often leads to better locality and helps some models establish time and memory bounds [13].

Although work-stealing lies at the heart of many runtimes, there are other alternatives and extensions. X10 introduces the concept of places, which allow scheduling computation to a specific set of cores to improve locality [2]. Galois manages applications with hard to schedule, conflicting tasks by exposing a varied set of policies for task grouping, assignment to threads, and execution order enforcement [33]. GRAMPS supports pipeline-style workloads by exposing producer-consumer constructs that allow a multi-level scheduler to minimize the amount of on-chip storage needed for intermediate results in the pipeline [46]. Overall, there are several scheduling considerations, many of them specific to the programming model and application domain.

2.2 Current Scheduling Approaches

To implement fine-grain schedulers on a cache-coherent CMP, we can either use a software-only solution in which threads communicate through shared memory, or leverage special-purpose hardware. In large-scale CMPs, both approaches have serious disadvantages.

Software-only schedulers maintain queues in software, and threads communicate and exchange work implicitly through shared memory. Several optimized algorithms have been proposed to avoid the use of locks in most local enqueue/dequeue operations [23] or to use non-blocking stealing protocols [6, 17]. Still, stealers need to perform multiple remote cache accesses to find and obtain new work, which will take hundreds of cycles through the cache hierarchy of a large-scale CMP. If stealing is infrequent or tasks are large, stealing overheads can be amortized. However, irregular or fine-grain workloads with frequent steals suffer from large penalties even with the most optimized protocols, due to memory latency, synchronization, or contention. These overheads will only become worse as we increase the number of cores on a CMP, because 1) the latency of a remote or a shared cache access increases, and 2) the amount of work per thread decreases, leading to shorter phases and more frequent steals.

Hardware-only schedulers, such as Carbon [34], introduce specialized hardware that handles all aspects of work-stealing. Carbon uses a centralized *global task unit (GTU)*, which contains one hardware LIFO queue per thread. Software uses special instructions to enqueue and dequeue task descriptors directly to/from registers. Task descriptors have a fixed size of 4 64-bit words. A small *local task unit (LTU)* per core is used as a task buffer to hide enqueue and

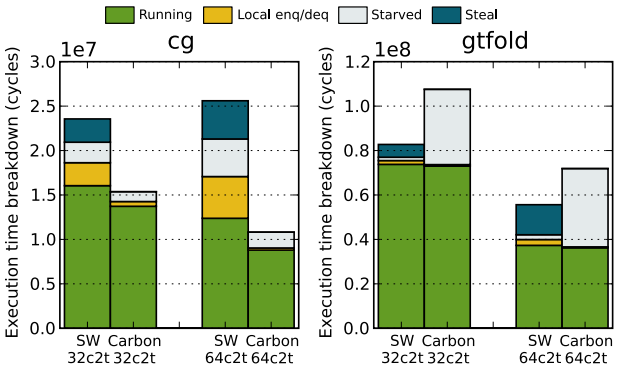


Figure 1: Execution time of cg and gtfold using Carbon and software schedulers, on CMPs with 32 and 64 dual-threaded cores (for a total of 64 and 128 threads).

dequeue latencies from the GTU. Work-stealing is implemented in hardware in the GTU by moving tasks between queues. Since the queues in the GTU are bounded, the runtime system cannot rely exclusively on Carbon for task buffering. Each worker thread maintains an unbounded task queue in software, where it can enqueue and dequeue new tasks locally. A portion of these tasks are enqueued in the corresponding hardware queue in the GTU to allow for work-stealing. Trying to dequeue a task when the GTU is empty blocks the thread. When all threads are blocked, the GTU sends a special task to every thread to signal the end of the parallel phase. The GTU generates an user-level interrupt when the capacity of a hardware queue reaches an upper or lower threshold to allow each thread to overflow to or refill tasks from the software queue.

Carbon addresses the performance issues of software-only scheduling, but hardwires the scheduling policies, such as the structure of queues and the order or granularity of stealing. Hence, it is difficult to use with applications or programming models that require alternative algorithms. Figure 1 illustrates this issue. It shows the execution time breakdown between software-only scheduling and Carbon (see Section 5 for the experimental methodology), for two applications. The first one, cg, is fine-grain, irregular, and has short phases. Carbon can do load-balancing very efficiently, thus outperforming the software scheduler by 2.2× at 128 threads. This reduction comes both from eliminating the stealing and enqueuing/dequeuing overheads of software, and performing better balancing (as threads spend less time starved, i.e. without work to execute). The second one, gtfold, is also fine-grain but can benefit from a FIFO scheduling algorithm (details in Section 6). The software FIFO scheduler, as shown, outperforms Carbon by 40% at 128 threads by reducing the starvation time. While it is possible to extend Carbon to capture a few scheduling variations, implementing all possible scheduling algorithms in hardware can be prohibitively expensive.

2.3 Fast & Flexible Fine-grain Scheduling

In this paper, we strive to find a balance between speed and flexibility for fine-grain scheduling. We also want to minimize the required hardware by introducing simple primitives that have multiple uses rather than fixed-function hardware. We note that Carbon can be deconstructed in three elements: hardware task queues, logic for stealing across these queues, and messaging hardware for fast communication of tasks. As we will see, implementing queues and policies in software is not much slower than using hardware, as long as all the scheduler state is kept *thread-local*, avoiding remote memory accesses. To allow this, we advocate keeping a fast messaging component in hardware and exposing it directly as a general-purpose mechanism. This avoids the high penalties of communicating and synchronizing through the memory hierarchy when task stealing or coordination among software schedulers is needed, and allows to overlap communication with useful computation. Moreover, scheduling in software with fast messaging for communicating control information allows us to create runtime systems tailored to the characteristics and requirements of specific applications or programming models.

3. Asynchronous Direct Messages

We now describe Asynchronous Direct Messages (ADM), the flexible messaging mechanism that we propose to accelerate fine-grain schedulers. ADM adds an extra messaging unit per core that works with any cache hierarchy or coherence protocol. To focus the discussion, we consider cache-coherent, tiled CMPs with a packet-switched interconnect, as the one shown in Figure 2a.

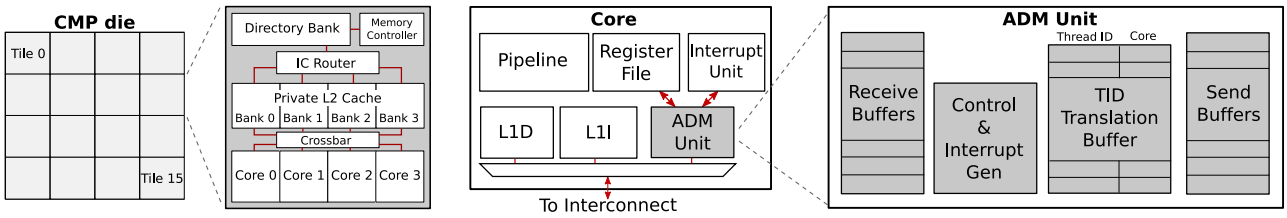
To provide low-overhead messaging with small payloads, messages are sent and received directly through registers instead of using memory-mapped buffers. Software threads initiate a send with a single instruction. Reception can be synchronous, using a receive instruction, or asynchronous, via a user-level message handler. To enforce atomicity of accesses to data structures shared by handler and non-handler code (e.g. the task queue), messaging interrupts can be disabled or enabled by non-handler code. The microarchitecture has limited message buffers, which are backed by software buffering if needed. Normal operation happens fully at user-level, but the architecture is virtualizable, and minimal OS support is required. The system preserves message ordering between sender and receiver and guarantees message delivery because these features greatly simplify writing ADM-based schedulers.

3.1 Microarchitecture and ISA

We modify a baseline CMP in two ways. First, we add an ADM unit to each core, shown in Figure 2b. This unit buffers received messages, translates thread IDs to physical cores on message sending, and interfaces to the register file (to transfer message payloads), the core's interrupt unit (to deliver message reception interrupts), and the interconnect. Second, we use an extra virtual network [20] in the packet-switched interconnect to route message packets. This only requires a moderate amount of extra buffering in the routers (e.g. 9KB of SRAM space in a 64-node CMP), and avoids deadlocks due to interference with the coherence protocol traffic.

The ADM unit includes one receive buffer per hardware thread context, implemented as a small circular buffer using SRAM memory. Thread ID to core translation is performed by the *Thread ID Translation Buffer (TTB)*, a small associative memory that caches `(TID, core)` pairs. The TTB is software-managed; if it cannot translate a destination's thread ID when sending, it triggers a privileged interrupt handler that refills it with the appropriate translation. Our hierarchical runtimes do not use all-to-all communication, but instead each thread communicates with a fixed-size subset of the threads. Thus, we only require small receive buffers (that can hold around 16 4-word messages) and TTBs (of 16∼32 entries) for full performance. Furthermore, these per-core structures do not need to grow with the size of the CMP.

Table 1 summarizes the hardware-software interface, including the extra instructions to send/receive messages and the new interrupts and exceptions introduced by ADM. The send instruction is blocking, and the instruction is considered completed when the message is copied to the send buffer. Each message is transmitted in the interconnect in a single packet, but using multiple flits [20].



(a) Target CMP, shown in a 64-core configuration with 16 tiles.

(b) The microarchitecture of the ADM unit added to each core.

Figure 2: Target CMP architecture and modifications needed to implement ADM.

Instruction	Description
`adm_send r1, r2`	Sends message of (r1) words to thread with ID (r2). The payload can have 0–6 words, taken from registers %o0-%o5
`adm_peek r1, r2`	Returns the source and message length at the head of the receive buffer, or a -1 length if the buffer is empty.
`adm_rx r1, r2`	Returns the source and message length at the head of the receive buffer, and writes its payload to registers %o0-%o5. Blocks on an empty buffer.
`adm_ei/di`	Enables or disables ADM receive handler interrupts.

Event	Type	Privileged
Receive	Interrupt	No
Receive buffer under/overflow	Interrupt	Yes
TTB miss	Exception	Yes
TTB remote invalidate	Interrupt	Yes

Table 1: ISA extensions and new events introduced by ADM, assuming a SPARCv9 ISA.

3.2 Guaranteed Delivery and Ordering

To simplify software schedulers, our hardware design preserves message ordering for each source-destination pair and provides guaranteed delivery. We first focus on how to implement guaranteed delivery. Suppose there are no send buffers in the ADM units. This implies that neither the interconnect nor the receiving buffer can drop messages and that all messages should eventually be dequeued by the receiving core. These requirements can lead to deadlock scenarios. For example, if two threads send messages to each other and neither is dequeuing them, the receive buffers will become full as well as any interconnect buffers between the two cores. Further send attempts will be blocked and the two threads will deadlock. To guarantee the absence of deadlock, we must ensure that threads will eventually dequeue any message they receive. One option is to prohibit reception handlers from blocking (e.g. by disallowing sending messages or acquiring locks) [48], but this is too restrictive for software. Instead, we include a second, privileged interrupt handler, which is triggered when the thread's receive buffer becomes full and dequeues half of the messages from the back of the buffer. To preserve ordering, the privileged handler marks the first non-dequeued message. When that message is dequeued by the thread, the privileged handler triggers again, refilling the buffer with the dequeued messages. Using the privileged handler, we provide unbounded receive buffers in software.

We structured our runtimes to almost never exceed the size of the receive buffers, avoiding the performance penalty of the second interrupt handler. However, bad-behaving user-level software (e.g. a buggy or malicious program) may cause serious interference with other programs that use ADM by quickly filling up the virtual network buffers with messages before the privileged handler can free space in the receive buffer. To avoid this, we have a send buffer per thread and use a simple ACK/NACK flow control scheme. Note that although this avoids clogging the network with messages, the privileged interrupt handler is still needed to prevent deadlock. Small send buffers suffice, since ADM is not continuously used. In our runtimes, send buffers that hold 16 messages are sufficient to cover message bursts. Send buffers also make it easier to preserve message ordering under virtualization (Section 3.3).

The flow control protocol between sender and receiver must preserve message ordering between each source-destination pair. In general, we can either use deterministic routing in the interconnect or implement a flow control protocol with reordering at the endpoints. We opt for deterministic routing in our evaluation, since networks in cache-coherent CMPs are often lightly loaded. Since the interconnect does not reorder messages, all we need to ensure is that, when the receiver R issues a NACK to sender S, all the messages in flight from S to R are discarded. To do this, every receiver keeps one bit per sender $B_{R\leftarrow S}$, and every sender keeps one bit per receiver $B_{S\rightarrow R}$. All the bits are initially 0. S stamps all its messages to R with $B_{S\rightarrow R}$. When R NACKs a message, it flips $B_{R\leftarrow S}$. When S receives the NACK, it flips $B_{S\rightarrow R}$ and tries retransmission of the messages to the receiver. Finally, R ignores all packets arriving from S with bit stamp not matching $B_{R\leftarrow S}$.

3.3 Virtualization

The OS needs to be aware of ADM, and perform some extra tasks to interact with it. First, it needs to assign a unique thread ID to each thread in the system, and maintain a mapping between thread IDs and physical cores for scheduled threads that the TTB refill handler can use. Since the TTB is software-managed, the OS can decide which thread pairs can communicate with ADM. For example, threads from multiple processes could be allowed to communicate to implement fast user-level IPC. Second, to support thread migration and descheduling, we introduce a lazy TTB invalidation mechanism. When a core receives a message intended for another thread context, it sends a NACK back to the sender, indicating that the TTB entry is stale. This NACK triggers a privileged interrupt in the sender, which either refills the TTB with the correct mapping and resends the messages for this destination in the send buffer if the thread was migrated, or saves them in a software queue and invalidates the stale TTB entry if the thread is switched out. Finally, the OS should ensure that a thread's send buffer is empty before migrating it to avoid losing message ordering under migrations.

Name	Type	Description	Format
UPDATE	↑	Inform manager of task count	<level, numTasks>
STEAL	↓	Notify victim to perform a steal	<level, tasksToTransfer, stealerId>
TASK	→	Transfer one task to stealer	<taskDescriptor, isLastTaskInSteal>
VICTIM_UPDATE	↑	Victim notifies manager of steal outcome & new task count	<level, tasksTransferred, tasksLeft>
STEALER_UPDATE	↑	Stealer notifies manager that steal is over & new task count	<level, numTasks>
UNBLOCK	↓	Notify end of phase	<level>

Table 2: Messages in the ADM runtime protocols. The type column indicates how the message flows in the hierarchy of workers and managers: down (↓), up (↑), or between workers (→). The level field indicates the tree level of the manager in the hierarchical runtime, and is not used in the centralized runtime.

4. Runtime Systems

We now present our baseline software-only runtime and the runtimes that utilize the hardware features of Carbon and ADM. In all cases, the application code is the same. The application programmer writes code for a shared-memory CMP and is oblivious of the use of Carbon or ADM. Only the low-level runtime code interacts with additional hardware features.

4.1 Task-parallel Runtimes

API: All the task-parallel runtimes implement the same simple interface, consisting of two functions:

- `void enqueue(Task t)`: Enqueues the task identifier `t` for execution in the current parallel phase. Task identifiers consist of four 64-bit words.
- `bool dequeue(Task& t)`: Tries to dequeue a task identifier from the current parallel phase. If successful, returns `true` and copies the task to `t`. Otherwise, returns `false`, signaling the end of the parallel phase.

Software-only runtime: Our baseline runtime is a highly optimized work-stealing scheduler. It uses Chase-Lev circular work-stealing deques [17], which require an atomic operation per steal, but not in local enqueues or most local dequeues. Local enqueues and dequeues are done in LIFO order. The stealing protocol is fully non-blocking: local enqueues/dequeues and steals to the same queue can happen concurrently, and a stealing thread never blocks waiting on other stealer to finish. The phase termination protocol is as in the X10 work-stealing scheduler [19]. We carefully control memory layout to avoid false sharing and maximize spatial locality.

We explored tuning the stealing policy in two dimensions: victim selection (random, round robin or nearest neighbor) and tasks to grab per steal (one or half of the queue). In our experiments, we find that trying to steal from nearest neighbors first outperforms random or round-robin stealing due to improved locality, and stealing half of the victim's queue is preferable to stealing one task to amortize software overheads and preserve inter-task locality. Therefore, the software runtime uses these policies in the evaluation.

Carbon runtime: The Carbon runtime operates as outlined in Section 2. Each worker has a private and unbounded LIFO task queue in software that is used when its hardware queue fills up. Work-stealing is enabled by maintaining at least a portion of the task queues in the hardware LIFO queues. The GTU hardware performs work-stealing in the background, and the LTUs fetch and prefetch tasks from the GTU. When the GTU needs to send a task to the LTU of a specific thread, it first attempts a dequeue from its corresponding hardware queue. If the queue is empty, it steals from a non-empty victim queue in a single cycle. As long as there are tasks in the hardware queues, the latencies of work-stealing and the communication between the GTU and the LTUs are hidden from the worker threads. Since the GTU cannot reclaim tasks from the LTUs, it does not serve prefetches when there are few tasks in the GTU to avoid load imbalance [34]. We optimized the runtime to minimize overflows and underflows of the hardware queues. As with the software runtime, we have explored different stealing policies in the GTU, and choose to steal half of the tasks from the nearest-neighbor queue, both for performance reasons and for consistency with the software runtime policy.

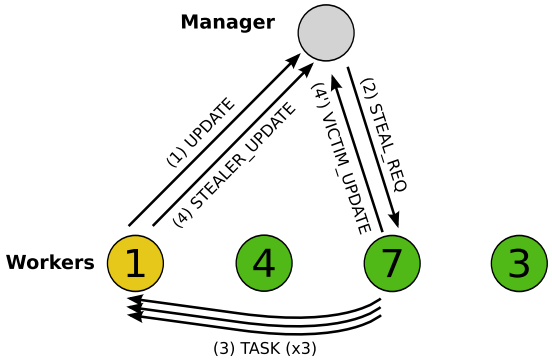
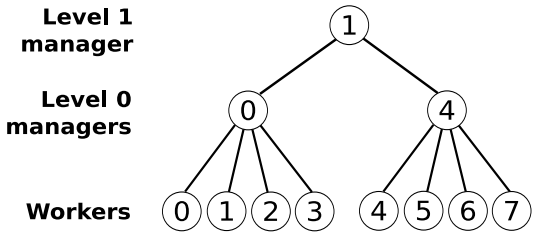


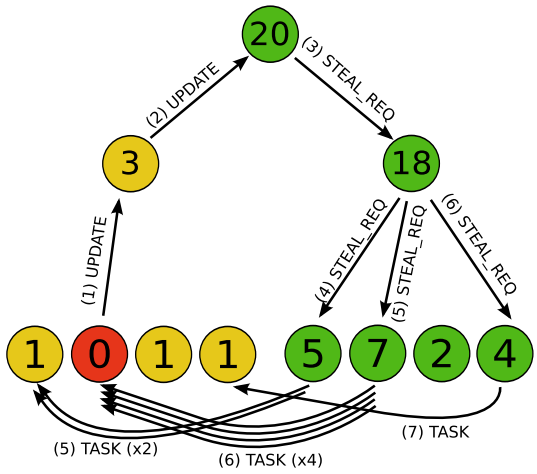
Figure 3: Messages involved in a steal in the centralized ADM runtime. The number in each worker indicates the tasks in its queue.

ADM runtimes: In ADM runtimes, threads can adopt the roles of *workers* or *managers*. A worker executes the program, enqueuing and dequeuing tasks in thread-local software queues. A manager handles task distribution, load balancing, and parallel phase termination by exchanging messages with workers. Managers do not maintain task queues themselves. A thread can act as a dedicated worker, as a dedicated manager, or as both worker and manager, switching between the two roles as messages arrive and the interrupt handler is triggered.

A simple ADM-based runtime can be *centralized*, using a single manager to coordinate all the threads, or *distributed* if it uses multiple managers. Our centralized runtime operates with four kinds of messages: updates, steal requests, task transfers, and unblocks. The details of each message are shown in Table 2. Workers send *update* messages to notify the manager about changes in the number of locally queued tasks, using the `adm_send` instruction in the functions for task enqueuing and dequeuing. To avoid saturating the manager, workers send updates only when their queues exceed exponentially varying thresholds. When an update message arrives, the interrupt handler invokes the manager code. The manager initiates and coordinates steals among workers based on its approximate knowledge of the number of tasks in each worker, as shown in Figure 3. If the number of tasks of a worker S (stealer) goes below an *underflow threshold*, the manager sends a *steal request* message to notify the worker V (victim) with the most tasks that it should send a portion of its queue to worker S. Worker V sends tasks to S, using one *task* message per task descriptor. When the steal is finished, S and



(a) Workers and managers



(b) Level 1 steal

Figure 4: Organization and operation of the hierarchical ADM runtime. An 8-thread, radix-4 runtime is shown. (a) indicates which threads perform the roles of workers and managers. Note how the manager threads are also workers. (b) shows the messages involved in a multi-level steal. In (b), the numbers indicate tasks left in each worker, or task aggregates in managers. The update messages sent after the steal are omitted for clarity.

V send updates to the manager. These updates may trigger further rebalances. A worker that tries to dequeue from an empty queue blocks and sends an update to its manager. When all the workers block, the manager sends an *unblock* message to every worker to signal the end of the parallel phase.

The centralized runtime is simple, but does not scale to large thread counts (e.g. 64 or 128), even when using a dedicated thread for the manager. If frequent stealing is needed, the manager quickly saturates when matching stealers and victims. Additionally, if the program has short phases, the single manager takes a long time to detect and signal phase termination.

To improve scalability, we implemented a *hierarchical* ADM-based runtime, with multiple levels of managers organized in a tree, as shown in Figure 4a. A level-0 manager directly coordinates a sub-group of workers, while managers at higher levels coordinate groups of managers down below. The hierarchical scheduler uses the same set of messages as in the centralized implementation, detailed in Table 2: updates flow up the tree, steal requests and unblocks flow down the tree, and task transfers happen between workers. Each manager keeps an approximate count of the aggregate number of tasks in its child partitions, and initiates steals within its partition when needed to counter imbalance. Hence, there can be steals spanning multiple levels, as Figure 4b shows. In these multi-level steals, managers distribute the steal request among its children, so a single steal request can rebalance two whole partitions. The i^{th} worker of the victim partition always transfers tasks to the i^{th} stealer worker. This may require additional rebalances in the stealer partition, but enables managers to keep only aggregate task counts. This implementation provides global load balancing, but at the same time improves locality by first solving local imbalances with steals from nearby threads. Compared to a centralized runtime, this approach amplifies stealing bandwidth, allowing for frequent steals. We observe that radix-4 to radix-16 configurations perform best for our workloads, with marginal performance differences between them. Larger radices exhibit reduced performance due to manager saturation. Our evaluation uses a radix-8 tree.

Finally, the scheduler needs to be aware of the limitations of ADM. As explained in Section 3, ADM operates very efficiently as long as the TTB and receive buffer capacities are not exceeded. TTB overflow is avoided with a relatively small radix, since the number of threads that a thread can communicate with grows logarithmically with system size (e.g. to avoid overflows completely, a radix-8 tree of 512 threads needs at most 31 TTB entries). Receive buffer overflow is avoided by limiting the number of tasks that a victim can send in a burst. Setting this limit to half of the receive buffer size makes overflows rare.

4.2 Loop-parallel runtimes

We adapt the task-parallel runtimes to improve performance with loop-parallel applications. The API is slightly different: Instead of enqueuing and dequeuing tasks, the application enqueues a whole loop and dequeues its iterations. The Carbon runtime uses special support for loop tasks [34]: a loop is enqueued to the GTU in a single enqueue, and loop tasks are partitioned in the GTU. In the software and ADM runtimes, a single task represents a range of loop iterations. Thus, a task can be efficiently split when stealing. Additionally, the runtimes support loops with reductions. The software and Carbon runtimes implement tree-based reductions through shared memory, and in ADM reductions are piggybacked on top of the unblock messages used to signal phase termination, incurring practically zero overhead.

4.3 Discussion

Even though our exploration was not exhaustive and better ADM-based managers may be possible, we draw some important insights about software scheduling for fine-grain parallelism. First, for large-scale CMPs, distributed runtimes are necessary even when ADM is available. Second, the availability of ADM allows all task queues to be kept thread-local. Each queue is accessed by only one thread, either to retrieve its own work or to serve steals. Hence, there is no need for locks or a few expensive misses when stealing occurs between remote threads. For reference, a single remote L2 miss takes around 90 cycles on average in the large-scale CMPs we explored. Third, the overhead of exchanging tasks or other scheduling information through ADM is significantly lower. The latency through the interconnect (25 cycles on average) is typically hidden as the messages are asynchronous. The message handlers in our runtimes typically run in about 50 cycles (including interrupt overhead). Finally, asynchronous messages allow us to overlap scheduling with useful computation.

Since ADM is a general messaging primitive, it could be used to implement other synchronization or communication mechanisms, such as barriers, locks or fast IPC. We leave these further uses to future work.

5. Experimental Methodology

5.1 Simulation Setup

We perform execution-driven simulation of large-scale CMPs using the M5 simulator [11] coupled with the Wisconsin GEMS toolset

Cores	32–128 cores, 1/2/4 threads per core, in-order, 2-way issue SMT, SPARCv9 ISA, 2 GHz
Coherence	Directory-based, MOESI among L1s-L2s and L2-directories, sequential consistency
L1 caches	32 KB, 4-way set associative, split D/I, 1-cycle latency, private per-core
L2 caches	1 MB per bank, 4 banks/tile, 16-way set associative, non-inclusive, 5-cycle tag / 10-cycle data latencies pipelined, shared by the L1s of the 4 cores in a tile, crossbar interconnect to L1s
L3 cache	3D-stacked, 16 MB per bank, 1 bank/tile, 16-way set associative, shared across the whole CMP, acts as victim cache for L2s, 10-cycle tag / 21-cycle data latencies, pipelined
Directory	1 bank/tile, idealized 6-cycle latency
MCU	1 memory controller/tile, single DDR-3 channel
Interconnect	2D flattened butterfly, connects tiles of 4 cores
Routers	3-stage pipeline (look-ahead RC and VA, SA, ST [20]), 4 VCs/virtual network, buffering of 8 flits/VC 3 virtual networks (1 for coherence requests, 1 for ADM/Carbon requests, 1 for replies)
Links	18B flits, repeated and pipelined; 1 cycle latency in local interconnect, 2–11 cycles in global interconnect
Carbon	4-task LTUs, 32 tasks per GTU queue, pipelined GTU
ADM	64-word receive and send buffers (16 4-word messages), 32-entry TTBs

Table 3: Main characteristics of the simulated CMPs. The latencies assume a 32 nm process at 2GHz.

	Input set	Type	Instrs	Task length	Phase length	Phases	Stolen tasks	L1D hits	L1 misses served by			
									Loc L2	Rem L2	L3	Mem
canneal	native	Loop	2.1 B	4.9 K	657 K	100	2.4 %	94.3 %	14.8 %	34.4 %	36.5 %	14.3 %
mergesort	1M keys	Task	346 M	3.9 K	4.6 M	1	9.8 %	97.3 %	18.5 %	66.1 %	0.0 %	15.4 %
maxflow	4K RLG	Task	1.5 B	674	99 M	1	8.5 %	90.0 %	22.3 %	77.5 %	0.0 %	0.2 %
ced	nyc	Task	381 M	419	3.7 M	2	13.1 %	96.0 %	37.6 %	23.4 %	6.5 %	32.6 %
cg	bcsstk16	Loop	465 M	976	21.8 K	601	7.5 %	90.9 %	77.9 %	20.0 %	0.0 %	2.1 %
gtfold	x54252	Loop	4.3 B	14.8 K	143 K	693	22.2 %	98.3 %	87.1 %	12.3 %	0.0 %	0.6 %
hashjoin	100td3	Task	166 M	1.6 K	3.8 M	1	75.3 %	95.8 %	24.1 %	49.8 %	0.0 %	26.1 %

Table 4: Main workload characteristics, using a 64-core CMP with Carbon. The type column indicates whether the application is task or loop-parallel. The average task and parallel phase lengths are in cycles. In loop-parallel applications, task length means iteration length.

for memory hierarchy modeling [37]. We simulate user-level application and library code, using detailed microarchitectural models for both the memory hierarchy and the interconnect. All the simulations are performed with warmed-up caches, and we introduce a small random perturbation in the main memory latency and do multiple runs per workload to obtain stable averages [4].

We model tiled CMPs with directory-based cache coherence, focusing on large-scale designs with 32 to 128 cores and a 3-level cache hierarchy, with the parameters shown in Table 3. The cores are 2-way in-order similar to the Niagara-2 pipeline [24]. They are also multithreaded to reduce the effect of memory latency. All components are sized to fit under reasonable area and power budgets at 32 nm for the 64-core configuration ($360mm^2$ and 55 W). Area, latency and power of caches and interconnect are estimated using CACTI 5.3 [47], ITRS 2007 predictions, and the models in [8]. The L2s are sized to take 40% of the chip area. We include a 3D-stacked L3, implemented in a 32 nm DRAM process.

Our Carbon model follows the original ISA and microarchitecture [34]. The local task units (LTUs) buffer up to 4 tasks per thread, and can have one task prefetched from the global task unit (GTU). The GTU is located in the center of the CMP, can serve one request per cycle, and holds 32 4-word tasks per thread queue (2KB per thread). The per-core ADM unit uses 64-word send and receive buffers per thread (same size as a Carbon queue) and a 32-entry TTB. All other queues for ADM-based runtimes are in software.

5.2 Applications

Our evaluation has three main goals. For balanced applications or codes with sufficiently large tasks, we want to show that the ADM runtime does not introduce any overheads and performs as well as an optimized software-only scheduler. For irregular applications with small tasks that match the scheduling algorithm of Carbon, we want to show that ADM performs and scales as well as Carbon despite using less hardware (queue management and algorithm control in software). Finally, for applications that perform best with other scheduling algorithms, we want to show that the software-mostly nature of ADM runtimes allows us to match the application characteristics and significantly outperform Carbon.

We have selected seven parallel workloads, summarized in Table 4. They cover a wide set of domains, use programming models, and exhibit a varied behavior in terms of miss rates, task granularities, available parallelism, and imbalance. They are:

canneal: A loop-parallel circuit routing algorithm using simulated annealing, refactored from the PARSEC suite [10].

mergesort: A parallel implementation of the mergesort algorithm using spawn-sync Cilk-style parallelism. It applies a divide-and-conquer strategy, resorting to serial mergesort when the subarray fits in the L1 cache. Merging two subarrays is parallelized as well.

maxflow: Computes the maximum flow of a graph using the push-relabel algorithm. This graph problem is computationally intensive and has many applications in networking, computer vision, etc., but is hard to scale [7, 33]. It has very short tasks and cores often exhaust their task queues, resulting in frequent stealing.

ced: Performs canny edge detection, a widely used algorithm in image processing and computer vision [15]. Refactored from the OpenCV library.

cg: An iterative solver for sparse linear systems using the conjugate gradient method. Includes different types of phases: sparse matrix-vector multiplications (long but irregular), scaled vector additions (short and regular), and dot products with frequent reductions.

gtfold: A bioinformatics application that predicts the secondary structure of large RNA molecules [38]. It has dependencies between loop iterations and an irregular iteration length, which results in short, imbalanced phases. Tuning the scheduler to the characteristics of this application can yield large benefits.

hashjoin: A hash-join algorithm implementation, common in database workloads [18].

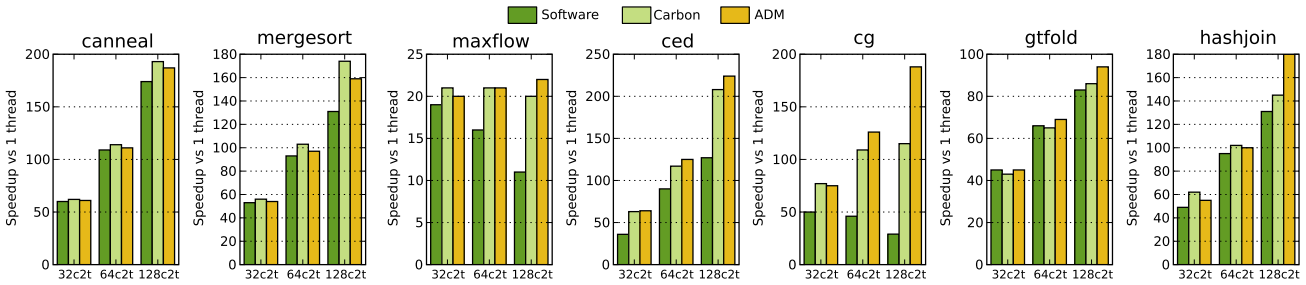


Figure 5: Performance of the different fine-grain schedulers, software, Carbon, and ADM, using CMPs with 32, 64 and 128 dual-threaded cores (64–256 threads). Speedups are normalized to the single-thread software version. Higher numbers are better.

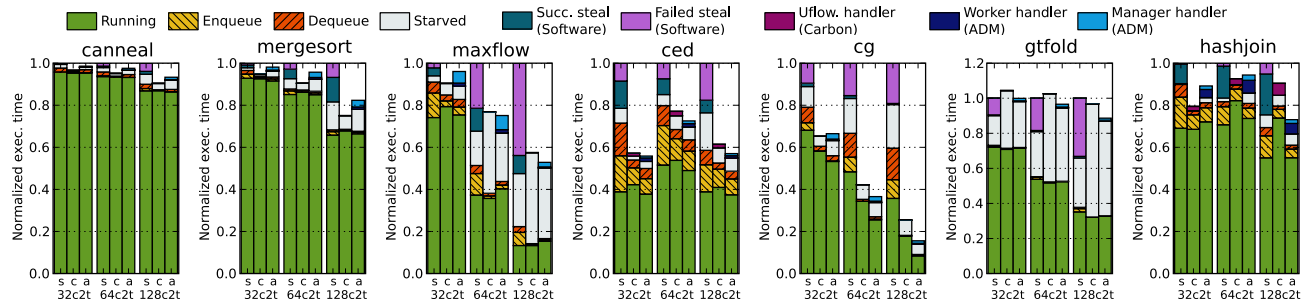


Figure 6: Execution time breakdown for software (s) Carbon (c) and ADM (a), using CMPs with 32, 64 and 128 dual-threaded cores (64–256 threads). Each result is normalized to the execution time of the software version *with the same number of cores*. Lower numbers are better.

6. Evaluation

6.1 Software, Carbon and ADM Schedulers

Figure 5 shows the performance of software, Carbon and ADM schedulers. Each graph shows the speedup of a single application on CMPs with 32, 64, and 128 dual-threaded cores (64–256 threads). Speedups are normalized to the single-thread software version. This experiment uses the regular software and ADM schedulers explained in Section 4 (we only alter the scheduling algorithms in Section 6.4). There are several things to observe. First, all applications except maxflow can scale reasonably well. Second, five of the seven benchmarks show large performance differences across the schedulers. Third, there is no single best scheduler across all applications, but ADM performs best on average, being slower than Carbon only on mergesort.

To gain further insight into these results, Figure 6 breaks down execution time into different components on 32–128 dual-threaded cores. Execution time is normalized to the total running time of the software-scheduled version *with the same number of cores*. There are four components common to all runtimes: running (executing non-scheduler application code), enqueuing and dequeuing (from the thread's own queue), and starved (waiting for tasks to become available). For the software scheduler, we also provide the time spent in steals, classified into successful steals (i.e. yielding one or more tasks) and failed steals. For Carbon, we show the amount of time spent in the underflow handler. For ADM, we show the time spent in interrupt handlers, broken down in the worker and manager portions. We can see some general trends: for Carbon and ADM, most of the scheduling overhead comes from starvation. The ADM overheads from handler code and local enqueues/dequeues are comparatively small, and ADM often beats Carbon by reducing starvation time. For the software scheduler, however, local enqueue/dequeue and stealing overheads can be major, due to loss of locality in the task queues (the software scheduler is non-blocking, so there is no lock contention). We now explain the behavior of each application in detail:

canneal is fairly balanced and coarse-grained, and shows minimal differences between the runtimes.

mergesort, due to its tree-style parallelization, has regions with scarce parallelism, with only a few threads generating new tasks that need to be redistributed as quickly as possible. Thus, mergesort is latency-sensitive, i.e. it significantly benefits from reducing the time that the scheduler takes to distribute work. ADM's fast directed stealing is able to perform within 5% of Carbon in these critical portions, while the software runtime is up to 24% slower.

maxflow does not scale beyond 64 threads, but it is a good example of how efficient scheduling can aid the performance of parallelism-constrained codes. Its tasks are very small (a graph node traversal, ∼600 cycles) and steals are common. ADM and Carbon achieve the same performance. The software runtime is 90% slower at 256 threads, due to enqueue/dequeue and stealing overheads, which become large due to cache misses on queue accesses. Maxflow shows that software queues are inexpensive if kept thread-local: even with 600-cycle tasks, at 64 threads the enqueue/dequeue overheads are 5% in ADM and 4% in Carbon, but 20% in software.

ced has very small tasks (400 cycles), long phases with deep queues and a mild imbalance. ADM and Carbon have similar overheads, both for queuing (because Carbon often reverts to software queues) and load-balancing.

cg combines long (65 K cycles at 128 threads), imbalanced phases, followed by short (4 K cycles at 128 threads), balanced

phases, and reductions. ADM matches the performance of Carbon in the short and long phases, but provides tree reductions that are an order of magnitude faster (390 vs 4 K cycles for 128 threads). Reductions become more important as we increase the number of cores (they are the portion of running time over that of ADM). Additionally, the GTU saturates in the short phases with small tasks, as we will see in Section 6.2. These issues cause ADM to outperform Carbon by 70% at 256 threads.

gtfold has relatively large loop iterations: 15 K cycles on average, with a bimodal distribution (either 1 K or 40 K cycles), and has many short, imbalanced parallel phases. Carbon performs sensibly worse than ADM because, even with the GTU disallowing prefetches when there are few tasks, an LTU sometimes prefetches a long task while executing another long task, leading to imbalance since tasks cannot be reclaimed from LTUs. The software and ADM schedulers do not have this problem, but ADM scales better than software.

hashjoin has a large load imbalance (about half of the threads enqueue most of the tasks), causing frequent steals from the empty threads. This frequent stealing saturates the GTU at 256 threads (note the increased overheads due to starvation and overflow handler). Hashjoin also has significant inter-task locality, and since the GTU only steals a small number of tasks per steal (half of the hardware queue size), it produces larger fragmentation, degrading locality and increasing the time spent per task by up to 20%.

To conclude, we discuss the effects of having different stealing policies. In theory, since we use nearest-neighbor stealing in Carbon and software, but hierarchical stealing in ADM, there could be differences in the application locality seen with each runtime. However, excluding hashjoin, the stealing policy has a negligible effect on the execution time of our applications. The only other cases in Figure 6 with significant differences in running (non-scheduler) time are maxflow and cg. For maxflow, this is due to algorithmic effects as the amount of work depends on the amount of parallelism. Carbon and ADM can keep more threads busy and end up with more running time on average. For cg, it is due to the faster reductions with ADM. In the remaining applications, the maximum difference between non-scheduler times is 4% (ced), where ADM is slightly faster.

6.2 Sensitivity to Hardware Parameters

Multithreading: Figure 7 shows the performance of mergesort, maxflow, and cg using a 64-core CMP with 1, 2 or 4 threads/core (other applications behave similarly). In general, the relative performance differences between software, Carbon, and ADM schedulers remain the same, despite the better latency tolerance with more threads. In mergesort, we note a slight performance reduction for ADM with 4 threads per core. This happens because mergesort is particularly latency-sensitive. With 4 threads, interrupt handlers take more time to execute due to contention in the core's pipeline. This could be addressed by scheduling less threads on cores with ADM managers, or prioritizing interrupt handler execution.

Idealized interconnect: Figure 8 shows the performance of Carbon and ADM when their traffic is not routed through the conventional interconnect, but through an idealized interconnect with a fixed 25-cycle latency between any source-destination pair and infinite bandwidth. Coherence traffic still uses the conventional interconnect. Additionally, the ideal Carbon GTU serves any number of requests in a single cycle. This allows us to evaluate the effect of *contention* in the GTU. Differences between ideal and non-ideal configurations are marginal on all applications except hashjoin and cg, where Carbon improves its speed by up to 25% and 55%, respectively, at 256 threads. hashjoin requires very frequent stealing, and the GTU cannot keep up distributing tasks at the required rate. cg has short phases, and the termination messages that the GTU

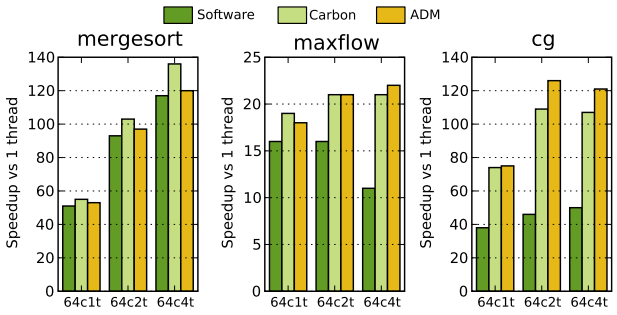


Figure 7: Performance of software, Carbon and ADM schedulers on a 64-core CMP with 1, 2 and 4 threads/core.

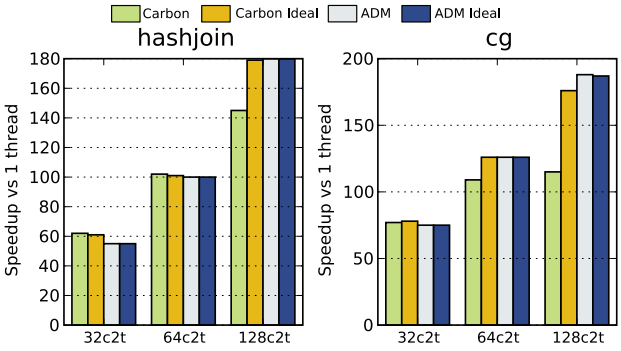


Figure 8: Performance of software, Carbon and ADM schedulers when the Carbon/ADM traffic is routed through an idealized network, eliminating contention in the GTU.

sends at the end of each phase become the bottleneck. Due to its distributed nature, the ADM scheduler is insensitive to these issues. These results indicate that, even with fast hardware, having centralized queues leads to contention with small tasks.

6.3 Analysis of ADM Benefits

Since the software and ADM schedulers are fundamentally different, it is hard to understand whether the benefits of ADM come from being able to implement a better scheduling algorithm or from bypassing the memory hierarchy. However, the software runtime cannot be structured in an asynchronous worker/manager organization without some hardware support. The minimal hardware required is to have *user-level interrupts* (ULI), supported in several recent proposals by monitoring updates to specific cache lines [14, 39, 45]. ULI allows an asynchronous worker/manager scheduler, where one thread can cheaply interrupt another to indicate the availability of a message that includes a task or information on the load of a worker. However, the actual message payload goes through the cache hierarchy. ADM provides *both* asynchronous user-level interrupts and register-to-register messaging.

To understand the benefits of ADM, we implement the same ADM task-parallel scheduler using ULI: threads communicate scheduling events through previously-known cache lines, and notify each other that a message is available using an ULI. Figure 9 shows the speedups and execution time breakdowns of the software, ULI and ADM schedulers for maxflow, mergesort, and hashjoin. ADM always outperforms ULI, since communicating

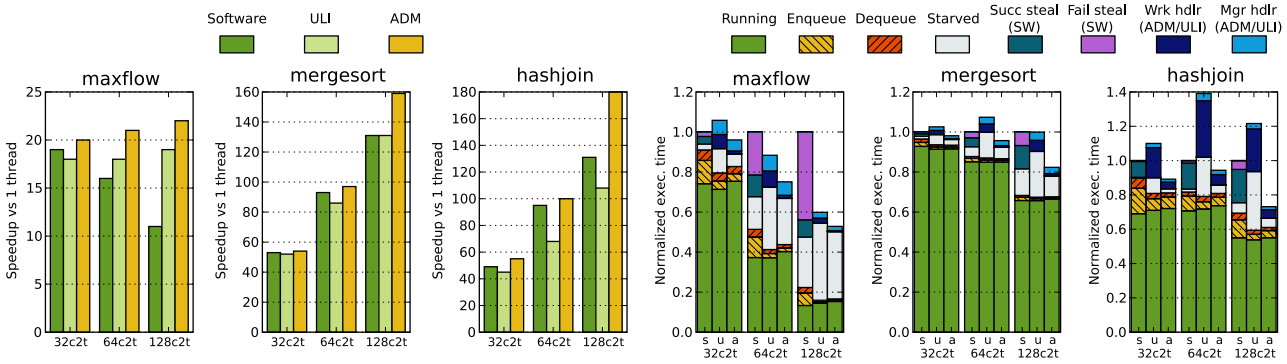


Figure 9: Speedup and execution time breakdowns of software (s), ULI (u) and ADM (a) runtimes, with 32–128 cores (64–256 threads).

Operation	ULI			ADM		
	Best	Avg	Worst	Best	Avg	Worst
Enqueue	41	66	84	34	57	72
Dequeue	39	60	94	32	37	44
Recv. update (mgr)	240	263	278	49	52	55
Steal match (mgr)	589	630	703	262	310	365
Send task (victim)	256	403	549	50	76	96
Recv. task (stealer)	259	300	326	40	47	56

Table 5: Cost breakdown for ULI and ADM schedulers using a 64-core, 128-thread CMP. The cost for each scheduler operation is given in cycles. Each cost is the average of a specific workload, and includes interrupt overheads for operations triggered by a ULI/message reception. The table includes the best and worst application's costs, and the average cost across all workloads.

through registers entails lower overheads than through the cache hierarchy. Table 5 provides further detail with a cost breakdown of ULI and ADM schedulers. We observe that in ADM schedulers all common operations are fast, taking 37 to 72 cycles on average (matching two threads for a steal takes longer, but occurs relatively rarely). With ULI, the costs for enqueuing and dequeuing are only slightly higher than in ADM, since task queues are thread-local and update messages are sent only at specific thresholds. However, costs for the other operations, which always involve sending or receiving messages, are two to six times larger than with ADM. Sending or receiving a message with ULIs requires at least two cache-to-cache transfers (one for the data and one for the ULI), taking up to 200 cycles (the penalty is lower if sender and receiver share the L2). Moreover, if the application is sensitive to scheduler latency, these higher overheads will increase starvation. The results in Figure 9 illustrate how these issues affect each application. In maxflow, which is somewhat latency-sensitive, ULI is only 18% slower than ADM. In mergesort, ADM achieves higher scalability than ULI because it can perform steals faster, reducing starvation. Finally, in hashjoin the ADM/ULI schedulers typically steal several tasks at once, causing multiple messages to be sent per handler execution. As a result, the handler overhead increases by up to 4× in ULI, being even slower than the software scheduler.

In conclusion, while ULI can be a useful mechanism for schedulers, ADM outperforms ULI by large margins. Moreover, the benefits of ADM versus a normal, synchronous software scheduler come from *both* being able to implement an asynchronous scheduler and bypassing the cache hierarchy, with the relative importance of each cause being application-dependent.

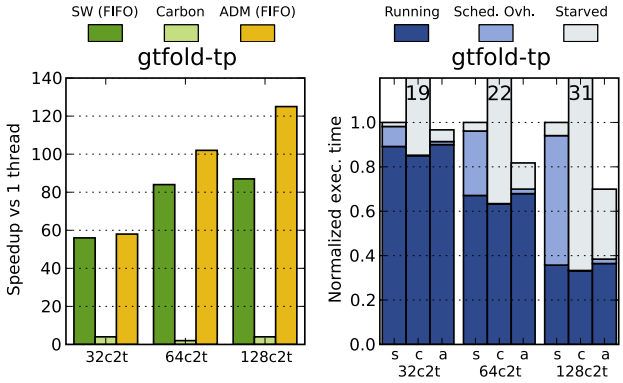


Figure 10: Speedup and execution time breakdowns of software (s), Carbon (c) and ADM (a) runtimes for the task-parallel version of gtfold, with 32–128 dual-thread CPUs (64–256 threads). Software and ADM schedulers are modified to use FIFO queues.

6.4 Using Custom Scheduling Algorithms

We now use gtfold to illustrate the potential of adapting the scheduling algorithm to the application. This application operates over an upper triangular matrix, and has non-trivial dependencies: task (i, j) depends on $(i, j-1)$ and $(i+1, j)$. The original application is loop-parallel, scheduling one diagonal per phase to avoid these dependencies. Since tasks also have a bimodal distribution (being either 1 K or 38 K cycles), this leads to short, imbalanced phases. However, we can avoid having multiple parallel phases by refactoring gtfold as a task-parallel application: each task works on a single element of the matrix, and every time a task completes, it checks if it is the last dependent task to complete for each of its successors, and enqueues them if so. This can be done at low overhead with counters and atomic operations through shared memory, since tasks are 15 K cycles on average. Ideally, we want to execute the tasks that have a longer dependence chain first, which roughly corresponds to a FIFO enqueuing policy. LIFO will do poorly, since it keeps older tasks (with higher potential to clear critical dependencies) at the bottom of the queue.

Figure 10 shows the speedups and execution time breakdowns for the refactored gtfold. The software and ADM schedulers are modified to perform FIFO enqueues and dequeues, while Carbon retains its LIFO policy. Software and ADM achieve significant per-

formance improvements over the loop-parallel versions in Figure 5 (35% for ADM at 256 threads). ADM still outperforms software by 40%. In contrast, Carbon achieves a maximum speedup of only 3× over the sequential version, being 40× slower than ADM. With 256 threads, the task-parallel version of gtfold on ADM outperforms the loop-parallel version on Carbon by 50%.

This example demonstrates the benefits of being able to implement scheduling algorithms in software. While Carbon could also implement FIFO queues in hardware, this would increase its design and verification complexity. Given the large number of scheduling algorithms that can be useful, supporting them all in hardware is infeasible. For instance, hashjoin benefits from directed hierarchical stealing, cg requires fast reductions, and other applications need more complex queuing policies (e.g. a priority queue). Our results show that supporting a simple, yet flexible primitive like ADM and leaving algorithmic decisions to software is more practical and leads to better performance than implementing them in hardware.

7. Related work

ADM is inspired by previous efforts in architectures that integrate shared memory and message passing. UDM [36], the messaging system in Alewife/FUGU [1], implements a model similar to ADM in some aspects. UDM supports low-overhead short messages, which can be received synchronously or asynchronously via user-level interrupts (with around a 100-cycle interrupt overhead). User-level code can disable these user-level interrupts, and threads transparently buffer received messages in memory with a privileged interrupt that triggers when a handler or atomic section takes too long. Like UDM, ADM includes asynchronous message receive through user-level interrupts, but at a lower overhead since no privileged interrupt handling code needs to run. ADM uses a similar mechanism to UDM to back the limited receive-side buffers with unbounded queues in software, but it is based on receiver-side buffer occupancy, not on timeouts. Finally, in ADM messages are sent to virtual threads, not physical thread contexts, allowing the OS to perform thread migration and more flexible scheduling. StarT-Voyager [5] implements user-level message passing by exposing memory-mapped send and receive queues that can overflow to main memory, but these memory-mapped queues entail additional overheads. The J-Machine [40] and M-Machine [35] also include a set of flexible messaging mechanisms suitable for fine-grain asynchronous communication, but unlike these message-driven architectures, for which messaging is the main means of communication, we advocate introducing messaging support in a shared-memory CMP.

Several recent architecture proposals target scheduling issues. Apart from Carbon, researchers have proposed several hardware schedulers that are tailored to specific applications or hardware [3, 44]. Rigel [32] is a large-scale accelerator CMP design with incoherent shared memory that includes a globally shared cache and special support for atomic operations to improve the efficiency of task-parallel software runtimes. However, Rigel targets task sizes one to two orders of magnitude larger than we do (100 K cycles per task). Pangaea [14, 49] is a tightly integrated small scale CPU-GPU design in which a CPU core dispatches work to GPU cores using user-level interrupts (ULIs). As we have shown, ULI-based schedulers can suffer large performance penalties because communication still happens through shared memory.

In the context of shared memory multiprocessors and CMPs, there have been several proposals to accelerate synchronization primitives using message-like constructs. Decoupled software pipelining [42] uses synchronous producer-consumer queues between processors for fine-grain parallelization of sequential programs. QOLB [31] focuses on reducing locking overhead, with hardware that maintains a distributed queue and performs direct node-to-node lock transfers. Active Memory Operations [22] use messages between cores and memory controllers, which are augmented with some extra logic, to implement fast locks and barriers. We note that ADM could also be used to implement these primitives, and leave the detailed evaluation to future work.

8. Conclusions

This paper has presented a hardware-software approach to build efficient fine-grain schedulers on large-scale CMPs. We propose asynchronous direct messages (ADM), a flexible and practical messaging mechanism that allows threads to communicate scheduling information without going through the memory hierarchy. Using ADM, we develop scalable schedulers that keep all scheduling data structures, such as task queues, thread-local, even when stealing occurs, and overlap most communication with useful computation. We show that ADM-based schedulers clearly outperform software-only schedulers and match or exceed the performance of hardware-only Carbon. When the ADM scheduler tailors its scheduling algorithm to the application characteristics, it exceeds Carbon's performance by up to 70%. Our results show that supporting a simple, yet flexible primitive like ADM and leaving the algorithmic decisions to software is more practical and leads to better performance than implementing scheduling in hardware.

Acknowledgements

We sincerely thank Woongki Baek, Jacob Leverich, Anthony Romano and the anonymous reviewers for their useful feedback on earlier versions of this manuscript; the development teams of the M5 and GEMS simulators for their collaboration in integrating them; and Shimin Chen for providing the mergesort and hashjoin workloads. This work was supported in part by the Stanford Pervasive Parallelism Lab and the Gigascale Systems Research Center (FCRP/GSRC). Daniel Sanchez was supported by a Fundacion Caja Madrid Fellowship and a Hewlett-Packard Stanford School of Engineering Fellowship.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: architecture and performance. *Proc. of the 22nd annual International Symposium on Computer Architecture*, 1995.

[2] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick. Deadlock-free scheduling of X10 computations with bounded resources. In *Proc. of the 19th annual ACM Symposium on Parallel Algorithms and Architectures*, 2007.

[3] G. Al-Kadi and A. S. Terechko. A hardware task scheduler for embedded video processing. In *Proc. of the 4th International Conference on High Performance and Embedded Architectures and Compilers*, 2009.

[4] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proc. of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[5] B. Ang, D. Chiou, L. Rudolf, and Arvind. Message passing support on StarT-Voyager. In *Proc. of the 5th International Conference on High Performance Computing*, 1998.

[6] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proc. of the 10th annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[7] D. A. Bader and V. Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In *Proc. 18th International Conference on Parallel and Distributed Computing Systems*, 2005.

[8] J. Balfour and W. J. Dally. Design tradeoffs for tiled CMP on-chip networks. In *Proc. of the 20th annual International Conference on Supercomputing*, 2006.

[9] S. Bell et al. TILE64 processor: A 64-core SoC with mesh interconnect. In *International Solid-State Circuits Conference*, 2008.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, 2008.

[11] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4), 2006.

[12] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2), 1999.

[13] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. of the 35th Annual Symposium on Foundations of Computer Science*, 1994.

[14] A. Bracy, K. Doshi, and Q. Jacobson. Disintermediated active communication. *IEEE Comput. Archit. Lett.*, 5(2), 2006.

[15] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6), 1986.

[16] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[17] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proc. of the 17th annual ACM Symposium on Parallel Algorithms and Architectures*, 2005.

[18] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *Proc. of the 19th annual ACM Symposium on Parallel Algorithms and Architectures*, 2007.

[19] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *Proc. of the 37th International Conference on Parallel Processing*, 2008.

[20] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

[21] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP task scheduling strategies. In *4th International Workshop in OpenMP*, 2008.

[22] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim, and M. A. Parker. Active memory operations. In *Proc. of the 21st annual International Conference on Supercomputing*, 2007.

[23] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, 1998.

[24] G. Grohoski. Niagara2: A highly-threaded server-on-a-chip. In *18th Hot Chips Symposium*, 2006.

[25] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for terminally strict parallel programs. In *Proc. of the 23rd IEEE International Parallel and Distributed Processing Symposium*, 2009.

[26] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7), 2008.

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

[28] HPF Language Specification. Version 2.0, 1997.

[29] Intel TBB. `http://www.threadingbuildingblocks.org`.

[30] Intel Tera-scale Computing Research Program. `http://www.intel.com/research/platform/terascale`.

[31] A. Kägi, D. Burger, and J. R. Goodman. Efficient synchronization: let them eat QOLB. In *Proc. of the 24th annual International Symposium on Computer Architecture*, 1997.

[32] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, B. Tuohy, A. Mahesri, S. Lumetta, M. Frank, and S. J. Patel. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *Proc. of the 36th International Symposium on Computer Architecture*, 2009.

[33] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *Proc. of the 20th annual Symposium on Parallelism in Algorithms and Architectures*, 2008.

[34] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *Proc. of the 34th annual International Symposium on Computer Architecture*, 2007.

[35] W. S. Lee, W. Dally, S. Keckler, N. Carter, and A. Chang. An efficient, protected message interface. *IEEE Computer*, 31(11), 1998.

[36] K. Mackenzie, J. Kubiatowicz, M. Frank, W. Lee, V. Lee, A. Agarwal, and M. Kaashoek. Exploiting two-case delivery for fast protected messaging. In *Proc. of the 4th International Symposium on High-Performance Computer Architecture*, 1998.

[37] M. M. Martin et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 2005.

[38] A. Mathuriya, D. A. Bader, C. E. Heitsch, and S. C. Harvey. GTfold: a scalable multicore code for RNA secondary structure prediction. In *Proc. of the 2009 ACM Symposium on Applied Computing*, 2009.

[39] V. Nagarajan and R. Gupta. ECMon: exposing cache events for monitoring. In *Proc. of the 36th annual International Symposium on Computer Architecture*, 2009.

[40] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-machine multicomputer: an architectural evaluation. In *Proc. of the 20th annual International Symposium on Computer Architecture*, 1993.

[41] OpenMP Application Program Interface. Version 2.5, 2005.

[42] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[43] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3), 2008.

[44] M. Själander, A. Terechko, and M. Duranton. A look-ahead task management unit for embedded multi-core architectures. In *Proc. of the 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, 2008.

[45] M. F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, and M. L. Scott. Alert-on-update: a communication aid for shared memory multiprocessors. In *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.

[46] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan. GRAMPS: A programming model for graphics pipelines. *ACM Trans. Graph.*, 28(1), 2009.

[47] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Labs, 2008.

[48] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. of the 19th annual International Symposium on Computer Architecture*, 1992.

[49] H. Wong, A. Bracy, E. Schuchman, T. M. Aamodt, J. D. Collins, P. H. Wang, G. Chinya, A. K. Groen, H. Jiang, and H. Wang. Pangaea: a tightly-coupled IA32 heterogeneous chip multiprocessor. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.