Two Academic Papers attached for use with Q2 and Q3

Two and a Half hours

UNIVERSITY OF MANCHESTER
SCHOOL OF COMPUTER SCIENCE

M.Sc. in Advanced Computer Science

Future Multi-Core Computing

Date: Wednesday 3rd June 2009
Time: 14:00 – 16:30

Please answer Question ONE and ONE other question

Two academic papers are attached for use with Question 2 and Question 3 otherwise, this is a CLOSED book examination

The use of electronic calculators is NOT permitted
1. **Compulsory**

a) Explain the need for caches in a modern processor. (2 marks)

b) A processor has 1st and 2nd level caches. The first level cache has an access time of 1nS, the second level cache access time is 4nS and the main memory access time is 50 nS. If the 1st level cache hit rate is 96% and the 2nd level cache hit rate is 75%, calculate the average access time of the memory system. (4 marks)

c) Explain the need for cache coherence in a multi-core processor. (2 marks)

d) What is the difference between an ‘invalidate’ and an ‘update’ cache coherence protocol. What are the relative merits of the two approaches? (4 marks)

e) What is an ‘atomic section’ in a shared memory program? Why is it necessary in a program with parallel threads which update a shared data structure? (4 marks)

f) Describe the difference between coarse and fine grain locking. What are the relative merits of the two approaches? (4 marks)

2. The attached paper “LogTM-SE: Decoupling Hardware Transactional Memory from Caches” describes a hardware system which takes a different approach to hardware implementation of Transactional Memory. Provide a short précis of the paper and analyse its strengths and weaknesses. (30 marks)

3. The attached paper “Cilk: An Efficient Multithreaded Runtime System” describes a software run-time system to support parallel programming. Provide a short précis of the paper and analyse its strengths and weaknesses. (30 marks)

END OF EXAMINATION

Papers associated with Q2 and Q3 are attached to this exam
LogTM-SE: Decoupling Hardware Transactional Memory from Caches


Department of Computer Sciences, University of Wisconsin–Madison

Abstract

This paper proposes a hardware transactional memory (HTM) system called LogTM Signature Edition (LogTM-SE). LogTM-SE uses signatures to summarize a transaction's read- and write-sets and detects conflicts on coherence requests (eager conflict detection). Transactions update memory "in place" after saving the old value in a per-thread memory log (eager version management). Finally, a transaction commits locally by clearing its signature, resetting the log pointer, etc., while aborts must undo the log.

LogTM-SE achieves two key benefits. First, signatures and logs can be implemented without changes to highly-optimized cache arrays because LogTM-SE never moves cached data, changes a block's cache state, or flash clears bits in the cache. Second, transactions are more easily virtualized because signatures and logs are software accessible, allowing the operating system and runtime to save and restore this state. In particular, LogTM-SE allows cache victimization, unbounded nesting (both open and closed), thread context switching and migration, and paging.

1 Introduction

Transactional memory (TM) [15] is a promising programming approach for effectively using the threads offered by future chips with multiple (often multi-threaded) cores. A TM system lets a programmer invoke a transaction and rely on the system to make its execution appear atomic and isolated. A successful transaction commits, while an unsuccessful one that conflicts with a concurrent transaction aborts. While some TM systems operate completely in software (STMs) [12, 14, 27], this paper concentrates on those implemented with hardware support (HTMs).

Hardware accelerates transactional memory with two key capabilities. First, hardware provides conflict detection among transactions by recording the read-set (addresses read) and write-set (addresses written) of a transaction. A conflict occurs when an address appears in the write-set of two transactions or the read-set of one and the read-set of another. Second, hardware provides version management by storing both the new and old values of memory written by a transaction. Most HTMs achieve their good performance in part by making demands on critical L1 cache structures. These demands include read/write (R/W) bits for read- and write-set tracking [3, 11, 19, 25], flash clear operations at commits/aborts [3, 11, 19, 25], and write buffers for speculative data [7, 11]. In addition, some depend on broadcast coherence protocols, precluding implementation on directory-based systems [7].

We see three reasons future HTMs may wish to decouple version management and conflict detection from the L1 cache tags and arrays. First, these are critical structures in the design of high performance processors that are better left untouched by an emerging idea like transactional memory. Second, the desire to support both T-way multi-threaded processors and L-level nested transactions leads to T x L copies of the state. Third, having transactional state integrated with the L1 cache makes it more difficult to save and restore, a necessary step to virtualize transactional memory—i.e., support cache victimization, unbounded nesting, thread suspension/migration, and paging [3, 25].

Fortunately, two HTMs provide complementary partial solutions to decoupling HTM demands from L1 caches.

LogTM [19] decouples version management from L1 cache tags and arrays. With LogTM, a transactional thread saves the old value of a block in a per-thread log and writes the new value in place (eager version management). LogTM’s version management uses cacheable virtual memory that is not tied to a processor or cache. It never forces writebacks to cache speculative data, because it does not exploit cache incoherence, e.g., where the L1 holds new transactional values and the L2 holds the old versions [11, 3, 7]. Instead, caches are free to replace or writeback blocks at any time. No data moves on commit, because new versions are in place, but on abort a handler walks the log to restore old versions. LogTM, however, fails to decouple conflict detection, because it maintains R/W bits in the L1 cache.

Bulk [7] decouples conflict detection by recording read- and write-sets in a hashed signature separate from L1 cache tags and arrays. A simple 1K-bit signature might logically OR the decoded 10 least-significant bits of block addresses. On transaction commit, Bulk broadcasts the write signature and all other active transactions compare it against their own read and write signatures. A non-null intersection indi-
icates a conflict, triggering an abort. Due to aliasing, non-null signature intersection may occur even when no actual conflict exists (a false positive) but no conflicts are missed (no false negatives). Moreover, Bulk's signatures make it easier to support multi-threading and/or nested transactions since replicating signatures doesn't impact critical L1 structures. Bulk's version management, however, is still tied to the L1 cache: the cache must (i) writeback committed, but modified blocks before making speculative updates, (ii) save speculatively modified blocks in a special buffer on cache overflow, and (iii) only allow a single thread of a multi-threaded processor to have speculative blocks in any one L1 cache set. In addition, it depends on broadcast coherence for strong atomicity [5] and requires global synchronization for ordering commit operations.

**LogTM-SE.** In this paper, we propose LogTM Signature Edition (LogTM-SE), which decouples both conflict detection and version management from L1 tags and arrays. LogTM-SE combines Bulk's signatures and LogTM's log, but adapts both to reap synergistic benefits. With LogTM-SE, transactional threads record conflicts with signatures and detect conflicts on coherence requests. Transactional threads update memory in place after saving the old value in a per-thread memory log. Like LogTM, LogTM-SE does not depend on broadcast coherence protocols. Finally, a transaction commits locally by clearing its signature and resetting its log pointer—there are no commit tokens, data writebacks, or broadcast—while aborts locally undo the log.

Transactions in LogTM-SE are virtualizable, meaning that they may be arbitrarily long and can survive OS activities such as context switching and paging, because the structures that hold their state are software accessible. Both old and new versions of memory can be victimized transparently because the cache holds no inaccessible transactional state. Similarly, the ability to save and restore signatures allows unbounded nesting. LogTM-SE achieves this using an additional summary signature per thread context to summarize descheduled threads. Finally, LogTM-SE supports paging by updating signatures using the new physical address after relocating a page.

Using Simics [17] and GEMS [18] to evaluate a simulated transactional CMP, we show that LogTM-SE performs comparably with the less-virtualizable, original LogTM. Furthermore, for our workloads even very small (e.g., 64-bit) signatures perform comparably better than locking.

In our view, LogTM-SE contributes an HTM design that (1) leaves L1 cache state, tag, and data arrays unchanged (no in-cache R/W bits or transactional write buffers), (2) has no dependence on a broadcast coherence protocol, (3) effectively supports systems with multi-threaded cores (replicating small signatures) on one or more chips (with local commit), and (4) supports virtualization extensions for victimization, nesting, paging, and context switching because signatures are easily copied. In Section 8 we detail how LogTM-SE differs from existing HTMs.

**2 LogTM-SE Architecture**

This section describes the LogTM-SE architecture, while Section 5 develops a specific example LogTM-SE system.

**Tracking Read- and Write-Sets with Signatures.** LogTM-SE tracks read- (R) and write- (W) sets with conservative signatures inspired by Bulk, as well as others who conservatively encode sets [4, 21, 24, 26]. A signature implements several operations. Let O be a read or a write and A be a block-aligned physical address. INSERT(O, A) adds A to the signature's O-set. Every load instruction invokes INSERT(read, A) and every store invokes INSERT(write, A). CONFLICT(read, A) returns whether A may be in a signature's write set (thereby conflicting with a read to A). CONFLICT(write, A) returns whether A may be in a signature's read- or write-sets. Both tests may return false positives (report a conflict when none existed), but may not have false negatives (fail to report a conflict). Finally, CLEAR(O) clears a signature's O-set. Section 5 discusses specific signature implementations.

**Eager Conflict Detection.** LogTM-SE performs eager conflict detection like LogTM, except that LogTM-SE uses signatures (not read/write bits in the L1 caches) and handles multi-threaded cores. Consider conflict detection with single-threaded cores first. A load (store) that misses to block A generates a GETS(A) (GETM(A)) coherence request. A core that receives a GETS (GETM) request checks its write (read) signatures (not read/write bits in the L1 caches) and handles conflict detection like LogTM, except that LogTM-SE uses signatures inspired by Bulk, as well as others who conservatively encode sets [4, 21, 24, 26]. A signature implements several operations. Let O be a read or a write and A be a block-aligned physical address. INSERT(O, A) adds A to the signature's O-set. Every load instruction invokes INSERT(read, A) and every store invokes INSERT(write, A). CONFLICT(read, A) returns whether A may be in a signature's write set (thereby conflicting with a read to A). CONFLICT(write, A) returns whether A may be in a signature's read- or write-sets. Both tests may return false positives (report a conflict when none existed), but may not have false negatives (fail to report a conflict). Finally, CLEAR(O) clears a signature's O-set. Section 5 discusses specific signature implementations.

**Tracking Read- and Write-Sets with Signatures.** LogTM-SE tracks read- (R) and write- (W) sets with conservative signatures inspired by Bulk, as well as others who conservatively encode sets [4, 21, 24, 26]. A signature implements several operations. Let O be a read or a write and A be a block-aligned physical address. INSERT(O, A) adds A to the signature's O-set. Every load instruction invokes INSERT(read, A) and every store invokes INSERT(write, A). CONFLICT(read, A) returns whether A may be in a signature's write set (thereby conflicting with a read to A). CONFLICT(write, A) returns whether A may be in a signature's read- or write-sets. Both tests may return false positives (report a conflict when none existed), but may not have false negatives (fail to report a conflict). Finally, CLEAR(O) clears a signature's O-set. Section 5 discusses specific signature implementations.

**Eager Conflict Detection.** LogTM-SE performs eager conflict detection like LogTM, except that LogTM-SE uses signatures (not read/write bits in the L1 caches) and handles multi-threaded cores. Consider conflict detection with single-threaded cores first. A load (store) that misses to block A generates a GETS(A) (GETM(A)) coherence request. A core that receives a GETS (GETM) request checks its write (read) signatures (not read/write bits in the L1 caches) and handles conflict detection like LogTM, except that LogTM-SE uses signatures inspired by Bulk, as well as others who conservatively encode sets [4, 21, 24, 26]. A signature implements several operations. Let O be a read or a write and A be a block-aligned physical address. INSERT(O, A) adds A to the signature's O-set. Every load instruction invokes INSERT(read, A) and every store invokes INSERT(write, A). CONFLICT(read, A) returns whether A may be in a signature's write set (thereby conflicting with a read to A). CONFLICT(write, A) returns whether A may be in a signature's read- or write-sets. Both tests may return false positives (report a conflict when none existed), but may not have false negatives (fail to report a conflict). Finally, CLEAR(O) clears a signature's O-set. Section 5 discusses specific signature implementations.

**Summary Signature.** LogTM-SE adds a summary signature to each coherence request, which indicates whether a block is modified and therefore requires a coherence response. This allows LogTM-SE to quickly identify conflicted blocks and avoid unnecessary coherence operations.

**LogTM-SE System.** We develop a specific example LogTM-SE system. This system is based on the LogTM-SE architecture described above and includes additional features such as support for speculative transactions and support for virtualization extensions. The LogTM-SE system is implemented on a real-world transactional CMP, and we evaluate its performance using a variety of workloads.

**Conclusion.** In conclusion, LogTM-SE provides an effective solution for implementing HTM systems on real-world transactional CMPs. It combines the advantages of LogTM and Bulk while addressing their limitations. We believe that LogTM-SE will be an important contribution to the field of HTM systems.
Signatures have the potential to cause interference between memory references in different processes. If thread $t_3$ in process A running on core C1 accesses a memory block residing on core C2, which is running $t_6$ from process B, a signature on C2 may signal a false conflict. While not affecting correctness, this interference could allow one process to prevent all other processes from making progress. LogTM-SE prevents this problem by adding an address space identifier to all coherence requests. Requests are only NACKed if the signature signals a potential conflict and the address space identifiers match, preventing false conflicts between processes.

Multi-threaded cores require additional mechanisms to detect conflicts among threads on the same core. Each thread context maintains its own read and write signatures. Loads or stores to blocks in M (and stores to E) must query the signatures of other threads on the same core. This check should not impact performance because conflicts need only be detected before the memory instruction commits.

**Eager Version Management.** LogTM-SE adopts LogTM's per-thread log, but adds a new mechanism to suppress redundant logging. Like a Pthread's stack, the log is allocated in thread-private memory. Before a memory block is first written in a transaction, its virtual address and previous contents must be written to the log. It is correct, but wasteful, to write the same block to the log more than once within a transaction. LogTM reuses the W bit in the L1 cache, which records whether a block has been written by the active transaction, to suppress redundant logging. However, this optimization does not extend to LogTM-SE because signatures permit false positives. If the hardware fails to log a block due to a false positive in the write-set signature, it would be impossible to correctly undo the effects of a transaction.

Instead, LogTM-SE uses an array of recently logged blocks for each thread context as a simple but effective log filter. When a thread stores to a block not found in its log filter, LogTM-SE logs the block and adds its address to the log filter. Stores to addresses in the log filter are not logged. Much like a TLB, the array can be fully associative, set associative, or direct mapped and use any replacement algorithm. As with write buffers in multi-threaded cores, the filters are logically per-thread, but can be implemented in a tagged shared structure. Because the filter contains virtual addresses and is a performance optimization not required for correctness, it is always safe to clear the log filter (e.g., on context switch).

**Local Commit & Abort.** LogTM-SE's transactional commit is a fast, local operation that also avoids LogTM's flash-clear of L1 cache read/write bits. To commit, a thread must only clear its local signatures to release isolation on its read- and write-sets and reset its log pointer. Since eager version management updates data in place, no data movement is necessary. Thus, commit, which should be much more common than abort, is a fast, thread-local operation requiring no communication or synchronization with other threads or cores. Like LogTM, LogTM-SE permits multiple non-conflicting transactions to commit in the same cycle.

LogTM-SE implements abort, the uncommon case, using a software handler. A thread aborts a transaction by trapping to an abort handler, which first walks the log in LIFO order to restore transactionally modified blocks. Once memory is restored to pre-transaction values, the handler releases isolation by clearing the thread's signature. Although abort takes time proportional to the number of blocks written by a transaction, it does not require any global resources.

**Summary.** The circled items in Figure 1 illustrate what LogTM-SE adds to each thread context to support TM. Like LogTM, LogTM-SE adds a register checkpoint and registers to store the log address, nesting depth, and abort handler address. LogTM-SE also adds two signatures, a log filter, and a summary signature (described in Section 4.1), but makes no changes to the critical L1 and L2 caches and has no structures that explicitly limit transaction size.

### 3 Virtualizing LogTM-SE

Application programmers reason about threads and virtual memory, while hardware implements multi-threaded cores, caches, and physical memory. Operating systems (OSs) provide programmers with a higher-level abstraction by virtualizing physical resource constraints, such as memory size and processor speed, using mechanisms such as paging and context switching. To present application programmers a suitable abstraction of transactional memory,
the OS must virtualize the HTM's physical resource limits, using hardware and low-level software mechanisms that are fast in common cases, correct in all cases, and, if possible, simple [3, 25].

This section discusses how LogTM-SE efficiently executes transactions unbounded in size and nesting depth using limited hardware. The following section discusses context switching and paging. LogTM-SE has two key advantages with regard to virtualization. First, LogTM-SE's version management is naturally unbounded, since logs are mapped into per-thread virtual memory. Second, LogTM-SE's signatures and logs are software accessible, allowing software to save and restore signatures to/from the log.

### 3.1 Cache Victimization

Caches may need to evict transactional blocks when a transaction's data size exceeds cache capacity or associativity. Multi-threaded cores make this more likely and unpredictable, due to interference between threads sharing the same L1 cache. Furthermore, after eviction, an HTM must continue to efficiently handle both version management and conflict detection. This is important, since cache victimization is likely to be more common than other virtualization events (e.g., thread switching and paging).

Notably, cache victimization has no effect on LogTM-SE's version management. Like LogTM, both new values (in place) and old values (in the log) may be victimized without resorting to special buffers, etc.

LogTM-SE's mechanism for conflict detection depends upon the underlying cache coherence protocol. Like all HTMs with eager conflict detection, LogTM-SE relies on the coherence protocol to direct requests to all caches that might represent a conflict. With broadcast coherence, cache victimization has no effect on conflict detection, because LogTM-SE can check all signatures on every broadcast.

With a naive directory protocol, cache victimization could lead LogTM-SE to miss some signature checks and hence miss some conflicts. LogTM-SE avoids this case by extending the directory protocol to use LogTM's sticky states [19]. As in many MOESI protocols, LogTM-SE's caches silently replace blocks in states E and S and write back blocks in states M and O. When evicting a cache block (e.g., core C1 replaces block B), however, LogTM-SE does not change the directory state, so that the directory continues to forward conflicting requests to the evicting core (e.g., a conflicting operation by C2 is still forwarded to C1, which checks its signature). Thus, LogTM-SE allows transactions to overflow the cache without a loss in performance.

### 3.2 Transactional Nesting

To facilitate software composition, HTMs must allow transactional nesting; invoking a transaction within a transaction [22]. This is trivially done by flattening: only committing transactional state when the outer-most transaction commits. Unfortunately with flat nesting, a conflict with the inner-most transaction forces a complete abort all its ancestors as well. An improvement is closed nesting with partial aborts that, for the above case, would allow an abort of just the inner-most transaction. To increase concurrency, some also argue for open nesting [30] which allows an inner transaction to commit its changes and release isolation before the outer transactions commit. In addition, some proposed language extensions for transactional memory, such as retry and orelse, depend on arbitrarily deep nesting [13]. Ideally, HTMs should provide unbounded nesting to fully support these language features. Otherwise, some composed software may fail when transactions nest too deeply.

LogTM-SE supports unbounded transactional nesting with no additional hardware by virtualizing the state of the parent's transaction while a child transaction is executing. Following Nested LogTM [20], LogTM-SE segments a thread's log into a stack of frames, each consisting of a fixed-sized header (e.g., register checkpoint) and a variable-sized body of undo records. LogTM-SE augments the header with a fixed-sized signature-save area.

A nested transaction begins by saving the current thread state: LogTM-SE copies the signature to the current transaction's log frame header and allocates a new header with a register checkpoint. To ensure the child correctly logs all blocks, it clears the log filter. Loads and stores within the child transaction behave normally, adding to the signature and log as necessary. On commit of a closed transaction, LogTM-SE merges the inner transaction with its parent by discarding the inner transaction's header and restoring the parent's log frame. An open commit behaves similarly, except that it first restores the signature from the parent's header into the (hardware) signature to release isolation on blocks only accessed by the committing open transaction.

On an abort, LogTM-SE's software handler first unrolls the child transaction's log frame and restores the parent's signature. If this resolves the conflict, the partial abort is done and a retry can begin. If a conflict remains with the parent's signature, the handler repeats this process until the conflict disappears or it aborts the outer-most transaction.

LogTM-SE supports unbounded transactional nesting with a per-thread hardware signature, saved to the log on nested begins. To reduce overhead, each thread context could provide one or more extra signatures to avoid synchronously saving and restoring signatures. On a nested begin, for example, hardware can copy the current signature S to S_{backup}. Inner commit of a closed transaction discards S_{backup} while inner commit of an open transaction and all inner aborts restore S_{backup} to S. Like register windows, the benefit depends on program behavior.
4 OS Resource Management

While OS resource management events, such as context switches and paging, may be infrequent relative to the duration of a transaction, they must still be handled correctly. This section discusses how LogTM-SE allows threads executing in transactions to be suspended and rescheduled on other thread contexts and how pages accessed within a transaction can be relocated in memory.

4.1 Thread Suspension/Migration

Operating systems (OSs) increase processing efficiency and responsiveness by suspending threads and rescheduling them on any thread context in the system. To support thread context switch and migration, the OS must remove all of a thread's state from its thread context, store it in memory, and load it back, possibly on a different thread context on the same or a different core. For HTMs that rely on the cache for their thread state, this is difficult because the transactional state of a thread is not visible to the operating system. One simple approach is to abort transactions when a context switch occurs. This is difficult for eager version management HTMs, though, because aborting is not instantaneous. In addition, some long-running transactions may never complete if they are forced to abort when preempted. A better approach allows thread preemption, but ensures that transactional state is saved and restored with the thread’s other state.

In LogTM-SE, all of a thread's transactional state—its version management and conflict detection state—is accessible to the OS. Both old and new versions of transactional data reside in virtual memory and require no special OS support. The log filter is purely an optimization and can be cleared when a thread is descheduled.

A thread's conflict detection state can be saved by copying the read/write signatures to the log's current header. However, the hardware must continue to track conflicts with the suspended thread's signatures to prevent other threads from accessing uncommitted data. For example, another thread in the same process may begin a transaction on the same thread context and try to access a block in its local cache. The system must check this access to ensure that the block is not in the write-set of a descheduled transaction. The challenge is to ensure that all active threads check the signatures of descheduled threads in their process on every memory reference.

LogTM-SE achieves this goal using an additional summary signature, which represents the union of the suspended transactions' read- and write-sets. The OS maintains the following invariant for each active/summary signature pair: If thread $t$ of process $P$ is scheduled to use an active signature, the corresponding summary signature holds the union of the saved signatures from all descheduled threads from its process $P$. On every memory reference, including hits in the local cache (both transactional and non-transactional), LogTM-SE checks the summary signature to ensure that the request does not conflict with a descheduled transaction. Multi-threaded cores, where each thread on a core may belong to a separate process, require a summary signature per thread context.

The OS maintains, in software, a summary signature for the entire process. When descheduling a thread, the OS merges the thread's saved signatures into its process summary signature. It then interrupts all other thread contexts running threads from the process and installs the new summary signature. In contrast to the normal signature, the summary signature is checked on memory references but not on coherence requests (because it is present on all thread contexts running in the same process). Any memory request that conflicts with a saved signature immediately traps to a conflict handler, since stalling is not sufficient to resolve a conflict with a descheduled thread.

When the OS reschedules a thread, it copies the thread's saved signatures from its log into the hardware read/write signatures. However, the summary signature is not recomputed until the thread commits its transaction, to ensure that blocks in sticky states remain isolated after thread migration. The thread executes with a summary signature that does not include its own signatures, to prevent conflicts with its own read- and write-sets. On transaction commit, LogTM-SE traps to the OS, which pushes an updated summary signature to active threads.1

Thus, with a single additional signature per thread and small changes to the operating system, LogTM-SE supports both context switching and thread migration. The cost of context switching within a transaction is relatively high, and for that reason we expect operating systems to support preemption control mechanisms [29] that defer context switches occurring within a transaction if possible. In addition, aborting short transactions may be preferable to incurring the overhead of propagating new summary signatures.

4.2 Virtual Memory Paging

HTMs must support paging of transactional data for several reasons. First, an OS may page out data in the read- and write-sets of active transactions and page it back in at a different physical address. If transactions are short, swapping of transactional data to disk is unlikely, because the memory was touched recently. However, paging may be required because one or more transactions' read- or write-sets exceed the physical memory size (but we hope this case is uncommon). Second, OS techniques, such as copy-on-write, may also cause a page that was read to subsequently be relocated.

---

1. To efficiently compute summary signatures, the OS could maintain a counting signature data structure to track the number of suspended threads setting each summary signature bit, similar to VTM’s XF data structure [25].
when it is written. HTMs should therefore work correctly in the presence of paging and should not cause an automatic abort (to handle large transactions).

LogTM-SE's version management operates on virtual addresses and is not tied to cores or caches. Thus, both new (in place) and old (in log) versions can be transparently paged. Moreover, eager version management allows a transaction to commit without restoring paged-out pages, since the new version is already in place. In contrast, lazy version management, in which memory is updated on commit, would require restoring paged-out pages at commit time, removing any benefit of paging them out in the first place.

LogTM-SE's signatures do not lose any information when a page is removed from memory, so transactional data remains isolated. However, because signatures operate on physical addresses, false conflicts may arise if the page is remapped to a different virtual address within the same address space. As with other false positives, this is acceptable if it is infrequent (as it should be).

More important are false negatives, indicating loss of isolation, that can arise when all of the following hold: (a) a page was transactional, (b) was paged out, (c) was paged back in at a different physical address (d) while the original transaction was still active. Since paging transactional data should be very rare, we propose a correct solution and leave optimization to future work.

When bringing a page back into a process at a different physical address, LogTM-SE notifies all threads to update their signatures with the new physical address for the page. For active threads, this requires interrupting each thread and, for those executing a transaction, walking the signature and testing whether it contains any blocks from the old address of the page. If so, the same blocks are inserted in the signature using their new physical address. The OS queues a signal for descheduled transactions to update their summary signatures (as well as signatures in the log from nesting) before they resume execution. Thus, the updated signatures contain both the old and new physical addresses for read- and write-set elements on the page.

This simple mechanism requires no additional hardware support and will incur little overhead if paging within a transaction is rare. If paging proves more frequent (i.e. if large transactions become the norm), additional mechanisms can detect whether a page has been touched during a transaction to avoid unnecessary signature updates.

5 A LogTM-SE Implementation

This section presents a specific LogTM-SE implementation for a CMP with non-broadcast coherence, which will be important for future larger-scale CMPs. Base CMP. Figure 2 illustrates the baseline 16-core LogTM-SE system and Table 1 summarizes the system parameters. Each of the 16 cores executes instructions out-of-order and supports 2-way multi-threading, providing 32 thread contexts on chip. The cores are 4-way-issue superscalar, use a 15-stage pipeline, 64-entry issue window, 128 entry reorder buffer, YAGS branch predictor, and have abundant fully-pipelined functional units (2 integer ALU, 2 integer divide, 2 branch, 4 FP ALU, 2 FP multipliers, and 2 FP divide/square root). Each core has 32 KB private L1 I & D caches, with the latter using writeback. All cores share an 8 MB L2 cache consisting of sixteen banks interleaved by block address. A packet-switched interconnect connects the cores and cache banks in a 4x3 grid topology using 64-byte links and adaptive routing. On-chip memory controllers connect to standard DRAM banks.

A MESI directory protocol provides cache coherence with less bandwidth demand than a broadcast protocol. The protocol enforces inclusion and each L2 tag contains a bit-vector of the L1 sharers and a pointer to the exclusive copy, if it exists. To eliminate a potential race, an E replacement from an L1 cache sends a control message to update the exclusive pointer, but S replacements are completely silent. An alternative implementation of the on-chip directory could use shadow tags instead of an inclusive L2 cache.

<table>
<thead>
<tr>
<th>Table 1: System Model Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System Model Settings</strong></td>
</tr>
<tr>
<td>Processor Cores</td>
</tr>
<tr>
<td>L1 Cache</td>
</tr>
<tr>
<td>L2 Cache</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>L2-Directory</td>
</tr>
<tr>
<td>Interconnection Network</td>
</tr>
</tbody>
</table>
The signature compactly represents the read- and write-sets of a transaction. A perfect filter, which precisely records the addresses read and written, can be implemented as bit vector with a bit for each block in the address space. However, this is unnecessary and inefficient, as false conflicts represent a performance, rather than a correctness issue. The key goals for this practical signature mechanism are (1) size, (2) accuracy, and (3) simplicity. We focus on signatures that can be computed from simple binary operations, such as shifting, ORing, and decoding.

Figure 3 shows three signature implementations, where an actual signature needs two copies of the illustrated hardware for read- and write-sets, respectively. Part (a) illustrates inserting a block address A into a simple bit-select (BS) signature implementation of size \( N = 2^n \) bits. The insert merely decodes the \( n \) least-significant bits of A’s block address and logically ORs the result with the current signature. While not illustrated, a CONFLICT(O, A) operation simply tests the appropriate bit, while a CLEAR(O) zeros the signature. Part (b) illustrates double-bit-select (DBS) that decodes two fields, setting both on an INSERT(O,A) and signaling a conflict only when both are set. DBS is similar to Bulk’s default signature mechanism, which permutes the address and then decodes two 10 bit fields. Finally, part (c) illustrates coarse-bit-select (CBS) that tracks conflicts at a coarser granularity than blocks (e.g., pages). CBS targets large transactions whose read- or write-sets at the block granularity would fill a small signature.

The next section shows that these simple signatures perform well for current transactional workloads. More creative signatures may prove necessary if larger transactions and deep nesting become the norm.

6 Evaluation

This section evaluates the LogTM-SE implementation described in Section 5. Results show that signature-based transactional memory generally performs comparably to lock-based synchronization, small, simple signature implementations suffice, and cache victimization occurs rarely for most workloads.

6.1 Methodology

We evaluate LogTM-SE using full-system execution-driven simulation based on the Wisconsin GEMS toolset [18, 31] in conjunction with Virtutech Simics [17]. The GEMS toolset includes detailed timing models for the processor pipeline and memory system. Simics provides functional correctness for the SPARC ISA and unmodified Solaris 9. Each simulation was pseudo-randomly perturbed to produce 95% confidence intervals [2].

6.2 Workloads

In order to observe a range of program behavior, we converted a variety of multi-threaded workloads to use transactions. These include a database storage library [28] and four SPLASH benchmarks [32]. In each case, we converted the original lock-based multi-threaded program to use transactions in place of lock-protected critical sections. Transaction begin and commit were implemented via Simics “magic”
instructions, which are special no-ops passed directly to the memory model.

**BerkeleyDB.** BerkeleyDB is an open-source database storage manager library that is commonly used for server applications (such as OpenLDAP), database systems (MySQL) and many other applications. We based our workload on the open-source version distributed by Sleepycat software [28].

We converted the mutex-based critical sections in BerkeleyDB to transactions. The resulting transactions contain non-transactional pieces of code such as system calls, I/O operations, and memory allocation, which are handled using non-transactional escape actions [20]. A simple multi-threaded driver program initializes a database with 1000 words and then creates a group of worker threads that randomly read from the database. This driver stresses the BerkeleyDB lock subsystem due to repeated requests for locks on database objects.

**Cholesky, Radiosity, Raytrace and Mp3d.** These scientific programs are taken from the SPLASH benchmark suite [32]. We replace the critical sections with transactions while retaining barriers and other synchronization mechanisms. Raytrace was modified to eliminate false sharing between transactions [19].

To reduce simulation times, we do not measure the entire parallel segment of the program. Instead we take representative execution samples and measure throughput in terms of well-defined units of work [1]. For example, in the BerkeleyDB workload, each database read comprises a unit of work. Table 2 lists our workloads, their input parameters, and their units of work.

### 6.3 Results

**Performance with Perfect Signatures.** We begin by showing that LogTM-SE with idealized signatures generally performs at least comparably to lock-based programs. For each benchmark, Figure 4 presents the execution time speedups for different TM variants relative to the left-most bar which represents the lock-based programs (Lock). The second bar, P, displays the performance of LogTM-SE using perfect signatures—idealized signatures that record exact read- and write-sets, regardless of their size.

**Result 1:** LogTM-SE with unimplementable perfect signatures performs comparable to locks or better. BerkeleyDB and Raytrace perform 20-50% better using transactions, while the differences for Cholesky, Mp3d, and Radiosity are not statistically significant (note the 95% confidence intervals denoted by the error bars).

**Implication 1:** LogTM-SE’s eager version management and local commit allows programmers to use the easier TM programming model without sacrificing performance, provided that realistic signature implementations do not degrade performance.

**Performance with Realistic Signatures.** To evaluate realistic signature implementations, Figure 4 presents the speedups for LogTM-SE with 2 Kb signatures using bit-select (BS), coarse-bit-select (CBS), and double-bit-select (DBS). BS decodes the least-significant 11 bits of the block address. CBS decodes the least-significant 11 bits of a 1 KB macro-block (sixteen 64-byte blocks). DBS separately decodes the 10 least-significant bits of a block address and the next 10 address bits, setting and checking two signature bits.

**Result 2:** LogTM-SE with the CBS and DBS signatures performs comparably to LogTM-SE with perfect signatures, while the simplest scheme, BS, degrades performance modestly for Radiosity and Raytrace.

**Implication 2:** If these results generalize to future TM workloads, LogTM-SE can use simple signatures to approximate perfect signatures and perform well.

**Signature Sizing.** Smaller signatures reduce implementation cost, but increase the probability of false positives. Given the well-known birthday paradox, one might expect small signatures to perform poorly. The last bar in Figure 4 presents the speedup for a 64 bit BS signature (BS_64).

**Result 3:** The 64 bit BS signature performs comparably to perfect signatures for 3 of the 5 benchmarks, but performs up to 20% slower for Radiosity and Raytrace. Small signatures suffice because most transactions have small read and write sets (Table 2) and spend most of their time executing non-transactional code (not shown).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input</th>
<th>Unit of Work</th>
<th>Units Measured</th>
<th>Transactions</th>
<th>Read Avg</th>
<th>Read Max</th>
<th>Write Avg</th>
<th>Write Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>BerkeleyDB</td>
<td>1000 words</td>
<td>1 database read</td>
<td>128</td>
<td>1,120</td>
<td>8.1</td>
<td>30</td>
<td>6.8</td>
<td>28</td>
</tr>
<tr>
<td>Cholesky</td>
<td>tk14.O</td>
<td>Factorization</td>
<td>1</td>
<td>261</td>
<td>4.0</td>
<td>4</td>
<td>2.0</td>
<td>2</td>
</tr>
<tr>
<td>Radiosity</td>
<td>batch</td>
<td>1 task</td>
<td>512</td>
<td>11,172</td>
<td>2.0</td>
<td>25</td>
<td>1.5</td>
<td>45</td>
</tr>
<tr>
<td>Raytrace</td>
<td>small image</td>
<td>parallel phase</td>
<td>1</td>
<td>47,781</td>
<td>5.8</td>
<td>550</td>
<td>2.0</td>
<td>3</td>
</tr>
<tr>
<td>Mp3d</td>
<td>128 molecules</td>
<td>1 step</td>
<td>512</td>
<td>17,733</td>
<td>2.2</td>
<td>18</td>
<td>1.7</td>
<td>10</td>
</tr>
</tbody>
</table>

---

1. We use the term “unit of work” in place of “transaction” (used by Alameldeen et al.) to avoid confusion with TM transactions.
Implication 3: These results suggest that small signatures may allow initial HTM implementations to use modest resources until the nature and importance of TM applications becomes clear.

Importance of Victimization. We also studied how often these benchmarks victimize transactional data from L1 or L2 caches. Only Raytrace had more than 20 transactions that evicted transactional data from its caches.

Result 4: Raytrace victimized transactional L1 or L2 blocks 481 times in 48K transactions, while other benchmarks victimized transactional blocks less than 20 times.

Implication 4: If these results generalize to future TM workloads, HTM should handle victimization, but do so with minimal complexity and resources.

In More Detail. To gain further insight, Table 3 presents additional information on Raytrace and BerkeleyDB. For both benchmarks, Table 3 presents the number of transaction commits, transaction stalls (i.e., the number of times transactions have a request NACKed), and transaction aborts for both perfect and practical signatures. It also presents the fraction of conflicts that arise from false positives.

For 2Kb signatures, for example, false positives account for 0-60% of all conflicts. This increases to 40-82% of all conflicts as the signature size shrinks to 64 bits. While false positives increase stalls for both benchmarks, the impact on aborts differs. For BerkeleyDB with all signature schemes and Raytrace with CBS and DBS, the number of aborts is comparable for 2 Kb and perfect signatures. Raytrace with 2 Kb BS signatures incurs roughly 21% more aborts. Furthermore, while reducing the signature size to 64 bits has little discernible effect on BerkeleyDB’s abort frequency, it increases the number of aborts for Raytrace by 18% for CBS and DBS, but decreases them by a third for BS. This illustrates a complex interaction: false positives may lead to false cycles (and thus aborts) or to serializing transactions (and thus no aborts). To see why, consider a single bit signature, which effectively acts as a global lock, eliminating the need to ever abort a transaction.

The large number of stalls relative to aborts indicates that given time, many conflicts will resolve themselves. Thus stalling a transaction may be preferable to aborting it and discarding otherwise useful work. While the stall to abort ratio is highest for small signatures, even with a perfect signature there are more stalls than aborts. BerkeleyDB has many more stalls than transactions, which occurs because a transaction may retry a coherence operation multiple times before the conflict clears and it makes progress.

The false positive rate roughly correlates to the size of transactional read- and write-sets. Table 2 shows the average and maximum number of cache lines in each workload’s read- and write-sets using perfect signatures. Since read-sets average 2 to 8 blocks and write-sets 1 to 7 blocks, few signature bits are set on average. However, the read- and write-set distribution can be highly skewed, resulting in some transactions that set many signature bits and create many false conflicts. Raytrace’s 550-block maximum read-set size represents the worst case, which helps explain why Raytrace’s performance falls off with the 64 bit BS signature.

7 Alternative LogTM-SE Implementations

The LogTM-SE approach should work well with other shared-memory systems, including a single CMP with snooping coherence and a multiple-CMP system.

A Snooping CMP. Consider a single CMP as described in Section 5—per-core writeback L1 caches, multi-banked shared L2 cache, standard off-chip DRAM—but change the MESI coherence protocol to use broadcast snooping. As is common, assume that L1 and L2 banks determine whether a coherence request has an L1 owner (one or more L1 sharers) via a logically-ORed owner (shared) signal.

Adding LogTM-SE to this snooping system requires the same additions to the core as in Section 5, but different coherence changes. With snooping, LogTM-SE requires a third logically-ORed signal, called nack, that cores use to NACK coherence requests when their signatures detect a conflict. Because snooping protocols broadcast all coherence requests, they eliminate the need for sticky states or other
special mechanisms to reach all necessary signatures. Because directories provide a first-level filter, broadcast snooping systems may need larger signatures to achieve comparable false positive rates.

**Multiple CMPs.** Consider a system with four CMPs (attached to standard DRAM) interconnected with a reliable point-to-point network. Assume that intra-chip coherence is maintained with the L2 directory of Section 5. Assume that inter-chip coherence is maintained with full-map directory protocol requiring a few state bits and 4 sharer bits per memory block. Directory state can be stored in memory bits freed by calculating SECDED ECC on 256 bits rather than the standard 64 bits [23]. For speed, directory state can be cached in a structure beside the home CMP's L2 cache.

LogTM-SE extends this multiple CMP system by adding the on-chip changes of Section 5 and altering the inter-chip directory coherence protocol to support NACKs on transaction conflicts and sticky states to handle victimization. An L2 cache that wishes to victimize a transactionally-modified block, for example, does a writeback to the directory at memory, so the directory can store the block and enter "sticky M". While these changes are conceptually straightforward, a full paper may be required to address the details.

**8 Related Work**

**HTMs.** LogTM-SE builds on the large body of research on HTM systems [3, 7, 8, 9, 11, 15, 19, 25]. LogTM-SE derives most directly from LogTM [19] and Bulk [7]. LogTM-SE improves upon LogTM by removing flash-cleared R and W bits from L1 caches and by improving virtualization. The R and W bits in LogTM do not scale easily with multi-threaded cores (requiring T copies for T hardware thread contexts) or nesting levels (requiring L copies for L levels of nesting support). In addition, LogTM's R and W bits pose a challenge for virtualizing transaction support as R and W bits can not be easily saved or restored. As a result, LogTM-SE supports thread suspension and migration while LogTM does not.

LogTM-SE differs from Bulk by making commit a local operation, supporting non-broadcast coherence protocols and allowing arbitrary signatures. Bulk's commit operation broadcasts the write signature of the committing transaction to all cores and possibly restores victimized transactional data to their original locations in memory. LogTM-SE's commit, by contrast, simply clears the committing transaction's signatures and resets its log pointer. In order to maintain strong atomicity, all Bulk cores must check their read signatures to see if it might contain the address of any non-transactional stores executed by any other core in the system even if that core is not currently caching the block. LogTM-SE, on the other hand, leverages LogTM's sticky states to ensure that coherence requests are sent to all necessary signatures without relying on broadcast. Finally, because LogTM-SE's version management is independent of caching, it eliminates Bulk's requirement that each signature precisely identify (no false negatives or positives) the cache sets of all
addresses it represents (e.g., using 1K bits for a cache with 1K sets).

Virtualization. LogTM-SE, similar to UTM [3], VTM [25], UnrestrictedTM [6], PTM [8] and XTM [9], supports the virtualization of transactions. Compared to other systems, LogTM-SE adds less hardware, uses its virtualization mechanism less frequently, and requires less work to process cache misses and transaction commits after virtualization events.

UTM virtualizes transactions using state (including a pointer) added to each memory block and an additional level of address translation. VTM supports virtualization with a combination of software and firmware, which stores transactional data and read- and write-sets in software tables when transactional data are evicted from the cache or when a transactional thread is suspended. UnrestrictedTM virtualizes transactions by allowing only one unrestricted transaction at a time to execute after cache victimization (but allowing the execution of multiple restricted transactions). XTM and PTM leverage paging and address translation mechanisms to virtualize transactions. Both provide software solutions and propose hardware mechanisms to accelerate common operations (XTM-g and PTM-Select).

Table 4 presents a rough comparison of the different systems’ efficiencies by displaying the actions they take on various system and cache events. As indicated by the “Before Virtualization” columns (left), all of the previous systems handle the common case of non-virtualized small transactions using simple hardware mechanisms. All these systems have a conceptual virtualization mode, which they switch to after evicting transactional data from the cache, or a paging operation or context switch during a transaction. As indicated by the “After virtualization” columns, all these systems either restrict concurrency or require complex hardware or slow software for at least one common case operation. UnrestrictedTM blocks all other transactions until the virtualized transaction commits. VTM and PTM-Copy require slow software-based conflict detection on cache misses. UTM and PTM-Select perform similarly complex operations in hardware on cache misses. XTM and XTM-g require expensive page-based validation of transactions’ read-sets at commit.

Like these systems, LogTM-SE requires little hardware overhead to support virtualization—one summary signature per thread context. In LogTM-SE, however, virtualization does not force the use of software for conflict detection, nor restrict the concurrency of transactions. LogTM-SE requires the least effort and expense to handle cache misses and commits—the most frequent events—after virtualization. Most importantly, in LogTM-SE, cache victimization of transactional data does not require virtualization.

Hybrid transactional memory systems [10, 16] provide virtualization by integrating an HTM with an STM. Small transactions, in the absence of virtualization events, execute as hardware transactions, while transactions that require virtualization execute as software transactions. HyTM [10] requires the least amount of hardware support of any of the virtualization schemes (it can run purely in software). However, hybrid schemes add overhead to hardware transactions in order to detect conflicts with concurrent software transactions. Initial results with HyTM indicate that a virtualized HTM will perform better in the presence of cache victimization [10].

9 Conclusions

This paper proposes a hardware transactional memory (HTM) system called LogTM Signature Edition (LogTM-SE) that combines features of prior HTM systems—especially LogTM, Nested LogTM, and Bulk. LogTM-SE stores principal transactional state in two structure types—signature and log—to achieve two key benefits. First, signatures and logs can be implemented without changes to highly-optimized cache arrays. Leaving critical cache arrays untouched may facilitate HTM adoption by reducing risk. Second, signatures

<table>
<thead>
<tr>
<th></th>
<th>Before Virtualization</th>
<th>After Virtualization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$Miss$</td>
<td>Commit</td>
</tr>
<tr>
<td>UTM [3]</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VTM [25]</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>XTM [9]</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>XTM-g [9]</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PTM-Copy [8]</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PTM-Select [8]</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LogTM-SE</td>
<td>-</td>
<td>SC</td>
</tr>
</tbody>
</table>

Legend
- = handled in simple hardware
H = complex hardware
S = handled in software
A = abort transaction
C = copy values
W = walk cache
V = validate read set
B = block other transactions
and logs are software accessible to allow OS and runtime software to manipulate them for virtualization. With little extra hardware, LogTM-SE handles cache victimization, unbounded nesting (both open and closed), thread context switching and migration, and paging.

10 Acknowledgements

This work is supported in part by the National Science Foundation (NSF), with grants CCF-0085949, CCR-0105721, EIA/CNS-0205286, CCR-0324878, as well as donations from Intel and Sun Microsystems. Hill and Wood have significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of the NSF, Intel, or Sun Microsystems.

We thank Virtutech, the Wisconsin Condor group, and the Wisconsin Computer Systems Lab for their help and support. We thank Dan Gibson and Simha Sethumadhavan for paper comments.

11 References


**Abstract**

Cilk (pronounced "silk") is a C-based runtime system for multithreaded parallel programming. In this paper, we document the efficiency of the Cilk work-stealing scheduler, both empirically and analytically. We show that on real and synthetic applications, the "work" and "critical path" of a Cilk computation can be used to accurately model performance. Consequently, a Cilk programmer can focus on reducing the work and critical path of his computation, insulated from load balancing and other runtime scheduling issues. We also prove that for the class of "fully strict" (well-structured) programs, the Cilk scheduler achieves space, time, and communication bounds all within a constant factor of optimal.

The Cilk runtime system currently runs on the Connection Machine CM5 MPP, the Intel Paragon MPP, the Silicon Graphics Power Challenge SMP, and the MIT Phish network of workstations. Applications written in Cilk include protein folding, graphic rendering, backtracking search, and the Socrates chess program, which won third prize in the 1994 ACM International Computer Chess Championship.

**1 Introduction**

Multithreading has become an increasingly popular way to implement dynamic, highly asynchronous, concurrent programs [1, 8, 9, 10, 11, 12, 15, 19, 21, 22, 24, 25, 28, 33, 34, 36, 39, 40]. A multithreaded system provides the programmer with a means to create, synchronize, and schedule threads. Although the schedulers in many of these runtime systems seem to perform well in practice, none provide users with a guarantee of application performance. Cilk is a runtime system whose work-stealing scheduler is efficient in theory as well as in practice. Moreover, it gives the user an algorithmic model of application performance based on the measures of "work" and "critical path" which can be used to predict the runtime of a Cilk program accurately.

A Cilk multithreaded computation can be viewed as a directed acyclic graph (dag) that unfolds dynamically, as is shown schematically in Figure 1. A Cilk program consists of a collection of Cilk procedures, each of which is broken into a sequence of threads, which form the vertices of the dag. Each thread is a nonblocking C function, which means that it can run to completion without waiting or suspending once it has been invoked. As one of the threads from a Cilk procedure runs, it can spawn a child thread which begins a new child procedure. In the figure, downward edges connect threads and their procedures with the children they have spawned. A spawn is like a subroutine call, except that the calling thread may execute concurrently with its child, possibly spawning additional children. Since threads cannot block in the Cilk model, a thread cannot spawn children and then wait for values to be returned. Rather, the thread must additionally spawn a successor thread to receive the children's return values when they are produced. A thread and its successors are considered to be parts of the same Cilk procedure. In the figure, sequences of successor threads that form Cilk procedures are connected by horizontal edges. Return values, and other values sent from one thread to another, induce data dependencies among the threads, where a thread receiving a value cannot begin until another thread sends the value. Data dependencies are shown as upward, curved edges in the figure. Thus, a Cilk computation unfolds as a spawn tree composed of procedures and the spawn edges that connect them to their children, but the execution is constrained to follow the precedence relation determined by the dag of threads.

The execution time of any Cilk program on a parallel computer with \( P \) processors is constrained by two parameters of the computation: the work and the critical path. The work, denoted \( T_w \), is the time used by a one-processor execution of the program, which corresponds to the sum of the execution times of all the threads. The critical path length, denoted \( T_{\infty} \), is the total amount of time required by an infinite-processor execution, which corresponds to the largest sum of thread execution times along any path. With \( P \) processors, the
execution time cannot be less than $T_1 / P$ or less than $T_{int}$. The Cilk scheduler uses “work stealing” [3, 7, 13, 14, 15, 19, 27, 28, 29, 34, 40] to achieve execution time very near to the sum of these two measures. Off-line techniques for computing such efficient schedules have been known for a long time [5, 16, 17], but this efficiency has been difficult to achieve on-line in a distributed environment while simultaneously using small amounts of space and communication.

We demonstrate the efficiency of the Cilk scheduler both empirically and analytically. Empirically, we have been able to document that Cilk works well for dynamic, asynchronous, tree-like, MIMD-style computations. To date, the applications we have programmed include protein folding, graphic rendering, backtrack search, and the Socrates chess program, which won third prize in the 1994 ACM International Computer Chess Championship. Many of these applications pose problems for more traditional parallel environments, such as message passing [38] and data parallel [2, 20], because of the unpredictability of the dynamic workloads on processors. Analytically, we prove that for “fully strict” (well-structured) programs, Cilk’s work-stealing scheduler achieves execution space, time, and communication bounds all within a constant factor of optimal. To date, all of the applications that we have coded are fully strict.

The Cilk language is an extension to C that provides an abstraction of threads in explicit continuation-passing style. A Cilk program is preprocessed to C and then linked with a runtime library to run on the Connection Machine CM5 MPP, the Intel Paragon MPP, the Silicon Graphics Power Challenge SMP, or the MIT Phish [4] network of workstations. In this paper, we focus on the Connection Machine CM5 implementation of Cilk. The Cilk scheduler on the CM5 is written in about 30 pages of C, and it performs communication among processors using the Strata [6] active-message library.

The remainder of this paper is organized as follows. Section 2 describes Cilk’s runtime data structures and the C language extensions that are used for programming. Section 3 describes the work-stealing scheduler. Section 4 documents the performance of several Cilk applications. Section 5 shows how the work and critical path of a Cilk computation can be used to model performance. Section 6 shows analytically that the scheduler works well. Finally, Section 7 offers some concluding remarks and describes our plans for the future.

2 The Cilk Programming Environment and Implementation

In this section we describe a C language extension that we have developed to ease the task of coding Cilk programs. We also explain the basic runtime data structures that Cilk uses.

In the Cilk language, a thread $T$ is defined in a manner similar to a C function definition:

$$\text{thread } T \ (\text{arg-decls} \ldots) \ {\text{stmts} \ldots}$$

The Cilk preprocessor translates $T$ into a C function of one argument and void return type. The one argument is a pointer to a closure data structure, illustrated in Figure 2, which holds the arguments for $T$. A closure consists of a pointer to the C function for $T$, a slot for each of the specified arguments, and a join counter indicating the number of missing arguments that need to be supplied before $T$ is ready to run. A closure is ready if it has obtained all of its arguments, and it is waiting if some arguments are missing. To run a ready closure, the Cilk scheduler invokes the thread as a procedure using the closure itself as its sole argument. Within the code for the thread, the arguments are copied out of the closure data structure into local variables. The closure is allocated from a simple runtime heap when it is created, and it is returned to the heap when the thread terminates.

The Cilk language supports a data type called a continuation, which is specified by the type modifier keyword cont. A continuation is essentially a global reference to an empty argument slot of a closure, implemented as a compound data structure containing a pointer to a closure and an offset that designates one of the closure’s argument slots. Continuations can be created and passed among threads, which enables threads to communicate and synchronize with each other. Continuations are typed with the C data type of the slot in the closure.

At runtime, a thread can spawn a child thread by creating a closure for the child. Spawning is specified in the Cilk language as follows:

$$\text{spawn } T \ (\text{args} \ldots)$$

This statement creates a child closure, fills in all available arguments, and initializes the join counter to the number of missing arguments. Available arguments are specified as in C. To specify a missing argument, the user specifies a continuation variable (type cont) preceded by a question mark. For example, if the second argument is $?k$, then Cilk sets the variable $k$ to a continuation that refers to the second argument slot of the created closure. If the closure is ready, that is, it has no missing arguments, then spawn causes the closure to be immediately posted to the scheduler for execution. In typical applications, child closures are usually created with no missing arguments.

To create a successor thread, a thread executes the following statement:

$$\text{spawnx } T \ (\text{args} \ldots)$$

This statement is semantically identical to spawn, but it informs the scheduler that the new closure should be treated as a successor, as opposed to a child. Successor closures are usually created with some missing arguments, which are filled in by values produced by the children.

A Cilk procedure does not ever return values in the normal way to a parent procedure. Instead, the programmer must code the parent procedure as two threads. The first thread spawns the child procedure, passing it a continuation pointing to the successor thread’s closure. The child sends its “return” value explicitly as an argument to the waiting successor. This strategy of communicating between threads is called explicit continuation passing. Cilk provides primitives of the following form to send values from one closure to another:

$$\text{send_argument } (k, \text{value})$$

This statement sends the value $\text{value}$ to the argument slot of a waiting closure specified by the continuation $k$. The types of the continuation and the value must be compatible. The join counter of the waiting
thread fib (cont int k, int n) 
  { if (n<2) 
    send_argument (k,n) 
  else 
    { cont int x, y; 
      spawn_next sum (k, ?x, ?y); 
      spawn fib (x, n-1); 
      spawn fib (y, n-2); 
    } 
  }
thread sum (cont int k, int x, int y) 
  { send_argument (k, x+y); 
  }

Figure 3: A Cilk procedure, consisting of two threads, to compute the nth Fibonacci number.

closure is decremented, and if it becomes zero, then the closure is ready and is posted to the scheduler.

Figure 3 shows the familiar recursive Fibonacci procedure written in Cilk. It consists of two threads, fib and its successor sum. Reflecting the explicit continuation passing style that Cilk supports, the first argument to each thread is the continuation specifying where the “return” value should be placed.

When the fib function is invoked, it first checks to see if the boundary case has been reached, in which case it uses send_argument to “return” the value of n to the slot specified by continuation k. Otherwise, it spawns the successor thread sum, as well as two children to compute the two subcases. Each of these two children is given a continuation specifying to which argument in the sum thread it should send its result. The sum thread simply adds the two arguments when they arrive and sends this result to the slot designated by k.

Although writing in explicit continuation passing style is somewhat onerous for the programmer, the decision to break procedures into separate nonblocking threads simplifies the Cilk runtime system. Each Cilk thread leaves the C runtime stack empty when it completes. Thus, Cilk can run on top of a vanilla C runtime system. A common alternative [19, 25, 32, 34] is to support a programming style in which a thread suspends whenever it discovers that required values have not yet been computed, resuming when the values become available. When a thread suspends, however, it may leave temporary values on the runtime stack which must be saved, or each thread must have its own runtime stack. Consequently, this alternative strategy requires changes to the runtime system that depend on the C calling stack layout and register usage conventions. Another advantage of Cilk’s strategy is that it allows multiple children to be spawned from a single nonblocking thread, which saves on context switching. In Cilk, r children can be spawned and executed with only r + 1 context switches, whereas the alternative of suspending whenever a thread is spawned causes 2r context switches. Since our primary interest is in understanding how to build efficient multithreaded runtime systems, but without redesigning the basic C runtime system, we chose the alternative of burdening the programmer with a requirement which is perhaps less elegant linguistically, but which yields a simple and portable runtime implementation.

Cilk supports a variety of features that give the programmer greater control over runtime performance. For example, when the last action of a thread is to spawn a ready thread, the programmer can use the keyword call instead of spawn that produces a “tail call” to run the new thread immediately without invoking the scheduler. Cilk also allows arrays and subarrays to be passed as arguments to closures. Other features include various abilities to override the scheduler’s decisions, including on which processor a thread should be placed and how to pack and unpack data when a closure is migrated from one processor to another.

3 The Cilk work-stealing scheduler

Cilk’s scheduler uses the technique of work-stealing [3, 7, 13, 14, 15, 19, 27, 28, 34, 40] in which a processor (the thief) who runs out of work selects another processor (the victim) from whom to steal work, and then steals the shallowest ready thread in the victim’s spawn tree. Cilk’s strategy is for thieves to choose victims at random [3, 27, 37].

At runtime, each processor maintains a local ready queue to hold ready closures. Each closure has an associated level, which corresponds to the number of spawn’s (but not spawn’s) on the path from the root of the spawn tree. The ready queue is an array in which the i-th element contains a linked list of all ready closures having level i.

Cilk begins executing the user program by initializing all ready queues to be empty, placing the root thread into the level-0 list of Processor 0’s queue, and then starting a scheduling loop on each processor. Within a scheduling loop, a processor first checks to see whether its ready queue is empty. If it is, the processor commences “work stealing,” which will be described shortly. Otherwise, the processor performs the following steps:

1. Remove the thread at the head of the list of the deepest nonempty level in the ready queue.
2. Extract the thread from the closure, and invoke it.

As a thread executes, it may spawn or send arguments to other threads. When the thread terminates, control returns to the scheduling loop.

When a thread at level L spawns a child thread T, the scheduler executes the following operations:

1. Allocate and initialize a closure for T.
2. Copy the available arguments into the closure, initialize any continuations to point to missing arguments, and initialize the join counter to the number of missing arguments.
3. Label the closure with level L + 1.
4. If there are no missing arguments, post the closure to the ready queue by inserting it at the head of the level-(L + 1) list.

Execution of spawn’s is similar, except that the closure is labeled with level L and, if it is ready, posted to the level-L list.

A processor that executes send_argument(k, value) performs the following steps:

1. Find the closure and argument slot referenced by the continuation k.
2. Place value in the argument slot, and decrement the join counter of the closure.
3. If the join counter goes to zero, post the closure to the ready queue at the appropriate level.

When the continuation k refers to a closure on a remote processor, network communication ensues. The processor that initiated the send_argument function sends a message to the remote processor to perform the operations. The only subtlety occurs in step 3. If the closure must be posted, it is posted to the ready queue of the initiating processor, rather than to that of the remote processor. This policy is necessary for the scheduler to be provably good, but as a practical matter, we have also had success with posting the closure to the remote processor’s queue, which can sometimes save a few percent in overhead.

If the scheduler attempts to remove a thread from an empty ready queue, the processor becomes a thief and commences work stealing as follows:

1. Select a victim processor uniformly at random.
2. If the victim’s ready queue is empty, go to step 1.
3. If the victim’s ready queue is nonempty, extract a thread from the tail of the list in the shallowest nonempty level of the ready queue, and invoke it.

Work stealing is implemented with a simple request-reply communication protocol between the thief and victim. Why steal work from the shallowest level of the ready queue? The reason is two-fold. First, we would like to steal large amounts of work, and shallow closures are likely to execute for longer than deep ones. Stealing large amounts of work tends to lower the communication cost of the program, because fewer steals are necessary. Second, the closures at the shallowest level of the ready queue are also the ones that are shallowest in the dag, a key fact proven in Section 6. Consequently, if processors are idle, the work they steal tends to make progress along the critical path.

4 Performance of Cilk applications

This section presents several applications that we have used to benchmark the Cilk scheduler. We also present empirical evidence from experiments run on a CMS supercomputer to document the efficiency of our work-stealing scheduler. The CMS is a massively parallel computer based on 32MHz SPARC processors with a fat-tree interconnection network [30].

The applications are described below:

- **fib** is the same as was presented in Section 2, except that the second recursive spawn is replaced by a “tail call” that avoids the scheduler. This program is a good measure of Cilk overhead, because the thread length is so small.
- **queens** is a backtrack search program that solves the problem of placing $N$ queens on an $N \times N$ chessboard so that no two queens attack each other. The Cilk program is based on serial code by R. Sargent of the MIT Media Laboratory. Thread length was enhanced by serializing the bottom 7 levels of the search tree.
- **pfold** is a protein-folding program [35] written in conjunction with V. Pande of MIT’s Center for Material Sciences and Engineering. This program finds hamiltonian paths in a three-dimensional grid using backtrack search. It was the first program to enumerate all hamiltonian paths in a $3 \times 4 \times 4$ grid. We timed the enumeration of all paths starting with a certain sequence.
- **ray** is a parallel program for graphics rendering based on the serial POV-Ray program, which uses a ray-tracing algorithm. The entire POV-Ray system contains over 20,000 lines of C code, but the core of POV-Ray is a simple doubly nested loop that iterates over each pixel in a two-dimensional image. For ray we converted the nested loops into a 4-ary divide-and-conquer control structure using spawns.\(^1\) Our measurements do not include the approximately 50 microseconds and have efficiency greater than 90 percent.
- **knary** is a synthetic benchmark whose parameters can be set to produce a variety of workloads for value and critical paths. It generates a tree of branching factor $k$ and depth $n$ in which the first $r$ children at every level are executed serially and the remainder are executed in parallel. At each node of the tree, the program runs an empty “for” loop for 400 iterations.

\(^1\) Initially, the serial POV-Ray program was about 5 percent slower than the Cilk version running on one processor. The reason was that the divide-and-conquer decomposition performed by the Cilk code provides better locality than the doubly nested loop of the serial code. Modifying the serial code to imitate the Cilk decomposition improved its performance. Timings for the improved version are given in the table.

- **Socrates** is a parallel chess program that uses the Jamboree search algorithm [23, 29] to parallelize a minmax tree search. The work of the algorithm varies with the number of processors, because it does speculative work that may be aborted during runtime. Socrates is a production-quality program that won third prize in the 1994 ACM International Computer Chess Championship running on the 512-node CM5 in the National Center for Supercomputing Applications at the University of Illinois, Urbana-Champaign.

Table 4 shows typical performance measures for these Cilk applications. Each column presents data from a single run of a benchmark application. We adopt the following notations, which are used in the table. For each application, we have an efficient serial C implementation, compiled using gcc -O2, whose measured runtime is denoted $T_{\text{serial}}$. The work $T_{W}$ is the measured execution time for the Cilk program running on a single node of the CM5.\(^2\) The critical path length $T_{\text{crit}}$ of the Cilk computation is measured by timestamping each thread and does not include scheduling or communication costs. The measured $P$-processor execution time of the Cilk program running on the CM5 is given by $T_{P}$, which includes all scheduling and communication costs. The row labeled “threads executed,” and “thread length” is the average thread length (work divided by the number of threads).

Certain derived parameters are also displayed in the table. The ratio $T_{\text{serial}}/T_{1}$ is the efficiency of the Cilk program relative to the serial program. The ratio $T_{1}/T_{\text{crit}}$ is the average parallelism. The value $T_{1}/P + T_{\text{crit}}$ is a simple model of the runtime, which will be discussed in the next section. The speedup is $T_{1}/T_{P}$, and the parallel efficiency is $T_{1}/(P T_{P})$. The row labeled “space/proc” indicates the maximum number of closures allocated at any time on any processor. The row labeled “requests/proc” indicates the average number of steals requested made by a processor during the execution, and “steals/proc” gives the average number of steals actually stolen.

The data in Table 4 shows two important relationships: one between efficiency and thread length, and another between speedup and average parallelism.

Considering the relationship between efficiency $T_{\text{serial}}/T_{1}$ and thread length, we see that for programs with moderately long threads, the Cilk scheduler induces very little overhead. The queens, pfold, ray, and knary programs have threads with average length greater than 50 microseconds and have efficiency greater than 90 percent. On the other hand, the fib program has low efficiency, because the threads are so short: fib does almost nothing besides spawn and send/argument.

Despite its long threads, the Socrates program has low efficiency, because its parallel Jamboree search algorithm [29] is based on speculatively searching subtrees that are not searched by a serial algorithm. Consequently, as we increase the number of processors, the program executes more threads and, hence, does more work. For example, the 256-processor execution did 7023 seconds of work whereas the 32-processor execution did only 3644 seconds of work. Both of these executions did considerably more work than the serial program’s 1665 seconds of work. Thus, although we observe low efficiency, it is due to the parallel algorithm and not to Cilk overhead.

Looking at the speedup $T_{1}/T_{P}$ measured on 32 and 256 processors, we see that when the average parallelism $T_{1}/T_{\text{crit}}$ is large compared with the number $P$ of processors, Cilk programs achieve nearly perfect linear speedup, but when the average parallelism is small, the speedup is much less. The fib, queens, pfold, and ray programs, for the Socrates program, $T_{1}$ is not the measured execution time, but rather it is an estimate of the work obtained by summing the execution times of all threads, which yields a slight underestimate. Socrates is an unusually complicated application, because its speculative execution yields unpredictable work and critical path. Consequently, the measured runtime on one processor does not accurately reflect the work on $P > 1$ processors.
for example, have in excess of 7000-fold parallelism and achieve more than 99 percent of perfect linear speedup on 32 processors and more than 95 percent of perfect linear speedup on 256 processors.\footnote{In fact, the \texttt{ray} program achieves superlinear speedup even when comparing to the efficient serial implementation. We suspect that cache effects cause this phenomenon.} The \texttt{socrates} program exhibits somewhat less parallelism and some-what less speedup. On 32 processors the \texttt{socrates} program has 1163-fold parallelism, yielding 90 percent of perfect linear speedup, while on 256 processors it has 2168-fold parallelism yielding 80 percent of perfect linear speedup. With even less parallelism, as exhibited in the \texttt{knary} benchmarks, less speedup is obtained. For example, the \texttt{knary}(10,5,2) benchmark exhibits only 70-fold parallelism, and it realizes barely more than 20-fold speedup on 32 processors (less than 65 percent of perfect linear speedup). With 178-fold parallelism, \texttt{knary}(10,4,1) achieves 27-fold speedup on 32 processors (87 percent of perfect linear speedup), but only 98-fold speedup on 256 processors (38 percent of perfect linear speedup).

Although these speedup measures reflect the Cilk scheduler’s ability to exploit parallelism, to obtain application speedup, we must factor in the efficiency of the Cilk program compared with the serial C program. Specifically, the application speedup $T_{\text{serial}} / T_1$ is the product of efficiency $T_{\text{serial}} / T_1$ and speedup $T_1 / T_F$. For example, applications such as \texttt{fib} and \texttt{socrates} with low efficiency generate correspondingly low application speedup. The \texttt{socrates} program, with efficiency 0.2371 and speedup 204.6 on 256 processors, exhibits application speedup of 0.2371 \times 204.6 = 48.51. For the purpose of performance prediction, we prefer to decouple the efficiency of the application from the efficiency of the scheduler.

Looking more carefully at the cost of a \texttt{spawn} in Cilk, we find that it takes a fixed overhead of about 50 cycles to allocate and initialize a closure, plus about 8 cycles for each word argument. In comparison, a C function call on a CM5 processor takes 2 cycles of fixed overhead (assuming no register window overflow) plus 1 cycle for each word argument (assuming all arguments are transferred in registers). Thus, a \texttt{spawn} in Cilk is roughly an order of magnitude more expensive than a C function call. This Cilk overhead is quite apparent in the \texttt{fib} program, which does almost nothing besides \texttt{spawn} and \texttt{send} arguments. Based on \texttt{fib}’s measured efficiency of 0.116, we can conclude that the aggregate average cost of a \texttt{spawn} and \texttt{send} argument in Cilk is between 8 and 9 times the cost of a function call/return in C.

Efficient execution of programs with short threads requires a low-overhead spawn operation. As can be observed from Table 4, the vast majority of threads execute on the same processor on which they are spawned. For example, the \texttt{fib} program executed over 17 million threads but migrated only 6170 (24.10 per processor) when run with 256 processors. Taking advantage of this property, other researchers [25, 32] have developed techniques for implementing spawns such that when the child thread executes on the same processor as its parent, the cost of the spawn operation is roughly equal the cost of a C function call. We hope to incorporate such techniques into future implementations of Cilk.

Finally, we make two observations about the space and communication measures in Table 4.

Looking at the “space/proc.” rows, we observe that the space per processor is generally quite small and does not grow with the number of processors. For example, \texttt{socrates} on 32 processors executes over 26 million threads, yet no processor ever has more than 386 allocated closures. On 256 processors, the number of executed threads nearly doubles to over 51 million, but the space per processors barely changes. In Section 6 we show formally that for Cilk programs, the space per processor does not grow as we add processors.

Looking at the “requests/proc.” and “steals/proc.” rows in Table 4, we observe that the amount of communication grows with the critical path but does not grow with the work. For example, \texttt{fib}, \texttt{queens},

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline

& \text{fib} & \text{queens} & \text{pfold} & \text{ray} & \text{knary} & \text{knary} & \text{socrates} & \text{socrates} \\
& (33) & (15) & (3,3,4) & (500,500) & (10,5,2) & (10,4,1) & (depth 10) & (depth 10) \\
\hline
$T_{\text{serial}}$ & 8.487 & 252.1 & 615.15 & 729.2 & 288.6 & 40.993 & 1665 & 1665 \\
$T_1$ & 73.16 & 254.6 & 647.8 & 732.5 & 314.6 & 45.43 & 3644 & 7023 \\
$T_{\text{serial}} / T_1$ & 0.116 & 0.9902 & 0.9496 & 0.9955 & 0.9714 & 0.9023 & 0.4569 & 0.2371 \\
$T_{\text{cyclic}}$ & 0.000326 & 0.0345 & 0.04354 & 0.0415 & 4.458 & 0.255 & 3.134 & 3.24 \\
\hline
threads & 17,108,660 & 210,740 & 9,515,098 & 424,475 & 5,859,374 & 873,812 & 26,151,774 & 51,685,823 \\
thread length & 4.276 µs & 1208 µs & 68.08 µs & 1726 µs & 53.69 µs & 51.99 µs & 139.3 µs & 135.9 µs \\
\hline
\end{tabular}
\caption{Performance of Cilk on various applications. All times are in seconds, except where noted.}
\end{table}
pfold, and ray all have critical paths under a tenth of a second long and perform fewer than 220 requests and 80 steals per processor, whereas knary(10,5,2) and Socrates have critical paths more than 3 seconds long and perform more than 20,000 requests and 1500 steals per processor. The table does not show any clear correlation between work and either requests or steals. For example, ray does more than twice as much work as knary(10,5,2), yet it performs two orders of magnitude fewer requests. In Section 6, we show that for “fully strict” Cilk programs, the communication per processor grows linearly with the critical path length and does not grow as function of the work.

5 Modeling performance

In this section, we further document the effectiveness of the Cilk scheduler by showing empirically that it schedules applications in a near-optimal fashion. Specifically, we use the knary synthetic benchmark to show that the runtime of an application on P processors can be accurately modeled as 

\[ T_P \approx T_1 / P + c_\infty T_\infty, \]

where \( c_\infty \approx 1.5 \). This result shows that we obtain nearly perfect linear speedup when the critical path is short compared with the average amount of work per processor. We also show that a model of this kind is accurate even for Socrates, which is our most complex application programmed to date and which does not obey all the assumptions assumed by the theoretical analyses in Section 6.

A good scheduler should run an application with \( T_1 \) work in \( T_1 / P \) time on \( P \) processors. Such perfect linear speedup cannot be obtained whenever \( T_\infty > T_1 / P \), since we always have \( T_P \geq T_\infty \), or more generally, \( T_P \geq \max \{ T_1 / P, T_\infty \} \). The critical path \( T_\infty \) is the stronger lower bound on \( T_P \) whenever \( P \) exceeds the average parallelism \( T_1 / T_\infty \), and \( T_1 / P \) is the stronger bound otherwise. A good scheduler should meet each of these bounds as closely as possible.

In order to investigate how well the Cilk scheduler meets these two lower bounds, we used our knary benchmark (described in Section 4), which can exhibit a range of values for work and critical path.

Figure 5 shows the outcome of many experiments of running knary with various values for \( k, n, r, \) and \( P \). The figure plots the speedup \( T_1 / T_P \) for each run against the machine size \( P \) for that run. In order to compare the outcomes for runs with different parameters, we have normalized the data by dividing the plotted values by the average parallelism \( T_1 / T_\infty \). Thus, the horizontal position of each datum is \( P / (T_1 / T_\infty) \), and the vertical position of each datum is \( (T_1 / T_P) / (T_1 / T_\infty) = T_\infty / T_P \). Consequently, on the horizontal axis, the normalized machine-size is 1.0 when the average available parallelism is equal to the machine size. On the vertical axis, the normalized speedup is 1.0 when the runtime equals the critical path, and it is 0.1 when the runtime is 10 times the critical path. We can draw the two lower bounds on time as upper bounds on speedup. The horizontal line at 1.0 is the upper bound on speedup obtained from the critical path, and the 45-degree line is the upper bound on speedup obtained from the work per processor. As can be seen from the figure, on the knary runs for which the average parallelism exceeds the number of processors (normalized machine size < 1), the Cilk scheduler obtains nearly perfect linear speedup. In the region where the number of processors is large compared to the average parallelism (normalized machine size > 1), the data is more scattered, but the speedup is always within a factor of 4 of the critical-path upper bound.

The theoretical results from Section 6 show that the expected running time of an application on \( P \) processors is 

\[ T_P = O(T_1 / P + T_\infty). \]

Thus, it makes sense to try to fit the data to a curve of the form 

\[ T_P = c_1 (T_1 / P) + c_\infty (T_\infty). \]

A least-squares fit to the data to minimize the relative error yields 

\[ c_1 = 0.9543 \pm 0.1775 \] and 

\[ c_\infty = 1.54 \pm 0.3888 \] with 95 percent confidence. The \( R^2 \) correlation coefficient of the fit is 0.989101, and the mean relative error is 13.07 percent. The curve fit is shown in Figure 5, which also plots the simpler curves 

\[ T_P = T_1 / P + T_\infty \] and 

\[ T_P = T_1 / P + 2 \cdot T_\infty \]

for comparison. As can be seen from the figure, little is lost in the linear speedup range of the curve by assuming that \( c_1 = 1. \) Indeed, a fit to 

\[ T_P = T_1 / P + c_\infty (T_\infty) \]

yields 

\[ c_\infty = 1.509 \pm 0.3727 \] with 

\[ R^2 = 0.983592 \] and a mean relative error of 4.04 percent, which is in some ways better than the fit that includes a \( c_1 \) term. (The \( R^2 \) measure is a little worse, but the mean relative error is much better.)

It makes sense that the data points become more scattered when \( P \) is close to or exceeds the average parallelism. In this range, the amount of time spent in work stealing becomes a significant fraction of the overall execution time. The real measure of the quality of a scheduler is how much larger \( T_1 / T_\infty \) must be than \( P \) before \( T_P \) shows substantial influence from the critical path. One can see from Figure 5 that if the average parallelism exceeds \( P \) by a factor of 10, the critical path has almost no impact on the running time.

To confirm our simple model of the Cilk scheduler’s performance on a real application, we ran Socrates on a variety of chess positions. Figure 6 shows the results of our study, which confirm the results from the knary synthetic benchmarks. The curve shown is the best fit to 

\[ T_P = c_1 (T_1 / P) + c_\infty (T_\infty), \]

where 

\[ c_1 = 1.067 \pm 0.0141 \] and 

\[ c_\infty = 1.042 \pm 0.0467 \] with 95 percent confidence. The \( R^2 \) correlation coefficient of the fit is 0.99941, and the mean relative error is 4.05 percent.

Indeed, as some of us were developing and tuning heuristics to increase the performance of Socrates, we used work and critical path as our measures of progress. This methodology let us avoid being trapped by the following interesting anomaly. We made an “improvement” that sped up the program on 32 processors. From our measurements, however, we discovered that it was faster only because it saved on work at the expense of a much longer critical path. Using the simple model 

\[ T_P = T_1 / P + T_\infty, \]

we concluded that on a 512- processor machine, which was our platform for tournaments, the “improvement” would yield a loss of performance, a fact that we later verified. Measuring work and critical path enabled us to use experiments on a 32-processor machine to improve our program for the 512-processor machine, but without using the 512-processor machine, on which computer time was scarce.

6 A theoretical analysis of the Cilk scheduler

In this section we use algorithmic analysis techniques to prove that for the class of “fully strict” Cilk programs, Cilk’s work-stealing scheduling algorithm is efficient with respect to space, time, and communication. A fully strict program is one for which each thread sends arguments only to its parent’s successor threads. For this class of programs, we prove the following three bounds on space, time, and communication.

Space The space used by a \( P \)-processor execution is bounded by 

\[ S_P \leq S_1 P, \]

where \( S_1 \) denotes the space used by the serial execution of the Cilk program. This bound is existentially optimal to within a constant factor [3].

Time With \( P \) processors, the expected execution time, including scheduling overhead, is bounded by 

\[ T_P = O(T_1 / P + T_\infty). \]

Since both \( T_1 / P \) and \( T_\infty \) are lower bounds for any \( P \)-processor execution, our expected time bound is within a constant factor of optimal.

Communication The expected number of bytes communicated during a \( P \)-processor execution is 

\[ O(T_\infty P S_{\max}), \]

where \( S_{\max} \) denotes the largest size of any closure. This bound is existentially optimal to within a constant factor [41].

The expected time bound and the expected communication bound can be converted into high-probability bounds at the cost of only a small
Figure 5: Normalized speedups for the knary synthetic benchmark using from 1 to 256 processors. The horizontal axis is $P$ and the vertical axis is the speedup $T_1/T_P$, but each data point has been normalized by dividing the these parameters by $T_1/T_\infty$.

Figure 6: Normalized speedups for the *Socrates chess program.
We consider the three possible ways that a primary-leaf closure can be freed, if that closure has a right sibling and that sibling created. First, when a thread spawns children, the leftmost of these children is a primary leaf if, in addition, it has no left-sibling closures allocated. In Figure 7, which shows the allocated closures at some time during an execution, closure \( a \) is the only primary leaf. Closure \( b \) is a leaf, but it is not primary, since it has left siblings and closure \( c \) is not a leaf, because \( a \) and its two siblings are counted as children of \( c \). The busy-leaves property states that every primary leaf closure has a processor working on it. To prove the space bound, we show that Cilk’s scheduler maintains the busy-leaves property, and then show that the busy-leaves property implies the space bound.

**Theorem 1** For any fully strict Cilk program, if \( S_1 \) is the space used to execute the program on 1 processor, then with any number \( P \) of processors, Cilk’s work-stealing scheduler uses at most \( S_1 P \) space.

**Proof:** We first show by induction on execution time that Cilk’s work-stealing scheduler maintains the busy-leaves property. We then show that the busy-leaves property implies the space bound.

To see that Cilk’s scheduler maintains the busy-leaves property, we consider the three possible ways that a primary-leaf closure can be created. First, when a thread spawns children, the leftmost of these children is a primary leaf. Second, when a thread completes and its closure is freed, if that closure has a right sibling and that sibling has no children, then the right-sibling closure becomes a primary leaf. And third, when a thread completes and its closure is freed, if that closure has no allocated siblings, then the leftmost closure of its parent’s successor threads is a primary leaf. The induction follows by observing that in all three of these cases, Cilk’s scheduler guarantees that a processor works on the new primary leaf. In the third case we use the fact that a newly activated closure is posted on the processor that activated it and not on the processor on which it was residing.

The space bound \( S_P \leq S_1 P \) is obtained by showing that every allocated closure can be associated with a primary leaf and that the total space of all closures assigned to a given primary leaf is at most \( S_1 \). Since Cilk’s scheduler keeps all primary leaves busy, with \( P \) processors we are guaranteed that at every time during the execution, at most \( P \) primary-leaf closures can be allocated, and hence the total amount of space is at most \( S_1 P \).

We associate each allocated closure with a primary leaf as follows. If the closure is a primary leaf, it is assigned to itself. Otherwise, if the closure has any allocated children, then it is assigned to the same primary leaf as its leftmost child. If the closure is a leaf but has some left siblings, then the closure is assigned to the same primary leaf as its leftmost sibling. In this recursive fashion, we assign every allocated closure to a primary leaf. Now, we consider the set of closures assigned to a given primary leaf. The total space of these closures is at most \( S_1 \), because this set of closures is a subset of the closures that are allocated during a 1-processor execution when the processor is executing this primary leaf, which completes the proof.

We now give the theorems bounding execution time and communication cost. Proofs for these theorems generalize the results of [3] for a more restricted model of multithreaded computation. As in [3], these proofs assume a communication model in which messages are delayed only by contention at destination processors, but no assumptions are made about the order in which contending messages are delivered [31]. The bounds given by these theorems assume that no thread has more than one successor thread.

The proofs of these theorems are analogous to the proofs of Theorems 12 and 13 in [3]. We show that certain “critical” threads are likely to be executed after only a modest number of steal requests, and that executing a critical thread guarantees progress on the critical path of the dag.

We first construct an augmented dag \( D' \) that will be used to define the critical threads. The dag \( D' \) is constructed by adding edges to the original dag \( D \) of the computation. For each child procedure \( v \) of a thread \( t \), we add an edge to \( D' \) from the first thread of \( v \) to the first thread of the next child procedure spawned by \( t \) after \( v \) is spawned. We make the technical assumption that the first thread of each procedure executes in zero time since we can add a zero-time thread to the beginning of each procedure without affecting work or depth. An example of the dag \( D' \) is given in Figure 7, where the additional edges are shown gray and curved. We draw the children spawned by a node in right-to-left order in the figure, because the execution order by the local processor is left to right, corresponding to LIFO execution. The dag \( D' \) is constructed for analytic purposes only and has no effect on the scheduling of the threads. An important property of \( D' \) is that its critical path is the same as the critical path of the original dag \( D \).

We next define the notion of a critical thread formally. We have already defined a ready thread as a thread all of whose predecessors in \( D \) have been executed. Similarly, a critical thread is a thread all of whose predecessors in \( D' \) have been executed. A critical thread must be ready, but a ready thread may or may not be critical. We now state a lemma which shows that a critical thread must be the shallowest thread in a ready queue.

**Lemma 2** During the execution of any fully strict Cilk program for which no thread has more than one successor thread, any critical thread must be the shallowest thread in a ready queue. Moreover, the critical thread is also first in the steal order.

**Proof:** For a thread \( t \) to be critical, the following conditions must hold for the ready queue on the processor in which \( t \) is enqueued:

1. No right siblings of \( t \) are in the ready queue. If a right sibling procedure \( v \) of \( t \) were in the ready queue, then the first thread of \( v \) would not have been executed, and because the first thread of \( v \) is a predecessor of \( t \) in \( D' \), \( t \) would not be critical.
2. No right siblings of any of \( t \)’s ancestors are in the ready queue. This fact follows from the same reasoning as above.
3. No left siblings of any of \( t \)’s ancestors are in the ready queue. This condition must hold because all of these siblings occur
before \( t \)'s parent in the local execution order, and \( t \)'s parent must have been executed for \( t \) to be critical.

4. No successor threads of \( t \)'s ancestors are enabled. This condition must be true, because any successor thread must wait for all children to complete before it is enabled. Since \( t \) has not completed, no successor threads of \( t \)'s ancestors are enabled. This condition makes use of the fact that the computation is fully strict, which implies that the only thread to which \( t \) can send its result is \( t \)'s parent’s unique successor.

A consequence of these conditions is that no thread could possibly be above \( t \) in the ready queue, because all threads above \( t \) are either already executed, stolen, or not enabled. In \( t \)'s level, \( t \) is first in the work-stealing order, because it is the rightmost thread at that level.

**Theorem 3** For any number \( P \) of processors and any fully strict Cilk program in which each thread has at most one successor, if the program has work \( T_1 \) and critical path length \( T_\infty \), then Cilk’s work-stealing scheduler executes the program in expected time \( E[T_F] = O(T_1^2 P + T_\infty) \). Furthermore, for any \( \epsilon > 0 \), the execution time is \( T_F = O(T_1^+ P + T_\infty + 1g P + 1g(1/\epsilon)) \) with probability at least \( 1 - \epsilon \).

**Proof:** This proof is just a straightforward application of the techniques in [3], using our Lemma 2 as a substitute for Lemma 9 in [3]. Because the critical threads are first in the work-stealing order, they are likely to be stolen (or executed locally) after a modest number of steal requests. This fact can be shown formally using a delay sequence argument.

**Theorem 4** For any number \( P \) of processors and any fully strict Cilk program in which each thread has at most one successor, if the program has critical path length \( T_\infty \) and maximum closure size \( S_{\text{max}} \), then Cilk’s work-stealing scheduler incurs expected communication \( O(T_\infty P S_{\text{max}}) \). Furthermore, for any \( \epsilon > 0 \), the communication cost is \( O((T_\infty + 1g(1/\epsilon)) P S_{\text{max}}) \) with probability at least \( 1 - \epsilon \).

**Proof:** This proof is exactly analogous to the proof of Theorem 13 in [3]. We observe that at most \( O(T_\infty P) \) steal attempts occur in an execution, and all communication costs can be associated with one of these steal requests such that at most \( O(S_{\text{max}}) \) communication is associated with each steal request. The high-probability bound is analogous.

**7 Conclusion**

To produce high-performance parallel applications, programmers often focus on communication costs and execution time, quantities that are dependent on specific machine configurations. We argue that a programmer should think instead about work and critical path, abstractions that can be used to characterize the performance of an algorithm independent of the machine configuration. Cilk provides a programming model in which work and critical path are observable quantities, and it delivers guaranteed performance as a function of these quantities. Work and critical path have been used in the theory community for years to analyze parallel algorithms [26]. Blelloch [2] has developed a performance model for data-parallel computations based on these same two abstract measures. He cites many advantages to such a model over machine-based models. Cilk provides a similar performance model for the domain of asynchronous, multithreaded computation.

Although Cilk offers performance guarantees, its current capabilities are limited, and programmers find its explicit continuation-passing style to be onerous. Cilk is good at expressing and executing dynamic, asynchronous, tree-like, MIMD computations, but it is not yet ideal for more traditional parallel applications that can be programmed effectively in, for example, a message-passing, data-parallel, or single-threaded, shared-memory style. We are currently working on extending Cilk’s capabilities to broaden its applicability. A major constraint is that we do not want new features to destroy Cilk’s guarantees of performance. Our current research focuses on implementing “dag-consistent” shared memory, which allows programs to operate on shared memory without costly communication or hardware support; on providing a linguistic interface that produces continuation-passing code for our runtime system from a more traditional call-return specification of spawns; and on incorporating persistent threads and less strict semantics in ways that do not destroy the guaranteed performance of our scheduler. Recent information about Cilk is maintained on the World Wide Web in page http://theory.lcs.mit.edu/~cilk.

**Acknowledgments**

We gratefully acknowledge the inspiration of Michael Halbherr, now of the Boston Consulting Group in Zurich, Switzerland. Mike’s PCM runtime system [18] developed at MIT was the precursor of Cilk, and many of the design decisions in Cilk are owed to him. We thank Shail Aditya and Sivan Toledo of MIT and Larry Rudolph of Hebrew University for helpful discussions. Ximin Tian of McGill University provided helpful suggestions for improving the paper. Rolf Riesen of Sandia National Laboratories ported Cilk to the Intel Paragon MPP running under the SUNMOS operating system, John Litvin and Mike Stupak ported Cilk to the Paragon running under OSF, and Andy Shaw of MIT ported Cilk to the Silicon Graphics Power Challenge SMP. Thanks to Matteo Frigo and Rob Miller of MIT for their many contributions to the Cilk system. Thanks to the Scout project at MIT and the National Center for Supercomputing Applications at University of Illinois, Urbana-Champaign for access to their CM5 supercomputers for running our experiments. Finally, we acknowledge the influence of Arvind and his dataflow research group at MIT. Their pioneering work attracted us to this path, and their vision continues to draw us forward.

**References**


Robert H. Halstead, Jr. Multilisp: A language for concurrent
W. Hillis and G. Steele. Data parallel algorithms.
Proceedings of the Twelfth ACM Symposium on Operating Systems
Parallel programming on a network of multiprocessors. In
Rainer Feldmann, Peter Mysliwietz, and Burkhard Monien. Studying overheads in massively parallel min/max-tree evaluation.