Using Memory Mapping to Support Cactus Stacks in Work-Stealing Runtime Systems

I-Ting Angelina Lee   Silas Boyd-Wickizer   Zhiyi Huang[*]   Charles E. Leiserson

MIT CSAIL
32 Vassar Street
Cambridge, MA 02139

ABSTRACT

Many multithreaded concurrency platforms that use a work-stealing runtime system incorporate a "cactus stack," wherein a function's accesses to stack variables properly respect the function's calling ancestry, even when many of the functions operate in parallel. Unfortunately, such existing concurrency platforms fail to satisfy at least one of the following three desirable criteria:

- full interoperability with legacy or third-party serial binaries that have been compiled to use an ordinary linear stack,
- a scheduler that provides near-perfect linear speedup on applications with sufficient parallelism, and
- bounded and efficient use of memory for the cactus stack.

We have addressed this ***cactus-stack problem*** by modifying the Linux operating system kernel to provide support for ***thread-local memory mapping (TLMM)***. We have used TLMM to reimplement the cactus stack in the open-source Cilk-5 runtime system. The Cilk-M runtime system removes the linguistic distinction imposed by Cilk-5 between serial code and parallel code, erases Cilk-5's limitation that serial code cannot call parallel code, and provides full compatibility with existing serial calling conventions. The Cilk-M runtime system provides strong guarantees on scheduler performance and stack space. Benchmark results indicate that the performance of the prototype Cilk-M 1.0 is comparable to the Cilk 5.4.6 system, and the consumption of stack space is modest.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Virtual memory*; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*.

General Terms

Experimentation, Languages, Performance.

[*]Zhiyi Huang is Senior Lecturer in the Department of Computer Science at University of Otago in New Zealand and was a Visiting Scientist at MIT during the course of this research.

This work was supported in part by the National Science Foundation under Grants 0540248 and 0615215.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PACT'10, September 11–15, 2010, Vienna, Austria.
Copyright 2010 ACM 978-1-4503-0178-7/10/09 ...$10.00.

Keywords

Work stealing, memory mapping, cactus stack, Cilk, interoperability, serial-parallel reciprocity.

1. INTRODUCTION

Work stealing [2, 6–9, 11–13, 16, 18, 21, 23, 24, 33, 38] is fast becoming a standard way to load-balance dynamic multithreaded computations on multicore hardware. Concurrency platforms that support work stealing include Cilk-1 [6],[1] Cilk-5 [16], Cilk++ [28], Fortress [1], Hood [8], Java Fork/Join Framework [26], Task Parallel Library (TPL) [27], Threading Building Blocks (TBB) [34], and X10 [10]. Work stealing admits an efficient implementation that guarantees bounds on both time and stack space [7,16], but existing implementations that meet these bounds — including Cilk-1, Cilk-5, and Cilk++ — suffer from interoperability problems with legacy (and third-party) serial binary executables that have been compiled to use a linear stack.[2] This paper addresses the question of how an operating system can support algorithmically sound work-stealing concurrency platforms that interoperate seamlessly with legacy serial binaries.

An execution of a serial Algol-like language, such as C [22] or C++ [36], can be viewed as a "walk" of an ***invocation tree***, which dynamically unfolds during execution and relates function instances by the "calls" relation: if function instance *A* calls function instance *B*, then *A* is a ***parent*** of the ***child*** ***B*** in the invocation tree. Such serial languages admit a simple array-based stack for allocating function activation frames. When a function is called, the stack pointer is advanced, and when the function returns, the original stack pointer is restored. This style of execution is space efficient, because all the children of a given function can use and reuse the same region of the stack. The compact linear-stack representation is possible only because in a serial language, a function has at most one extant child function at any time.

In a multithreaded language, such as Cilk-5 [16] or Cilk++ [28], a parent function can also ***spawn*** a child — invoke the child without suspending the parent — thereby creating parallelism. The notion of an invocation tree can be extended to include spawns, as well as calls, but unlike the serial walk of an invocation tree, a parallel execution unfolds the invocation tree more haphazardly and in parallel. Since multiple children of a function may exist simultaneously, a linear-stack data structure no longer suffices for storing activation frames. Instead, the tree of extant activation frames forms a ***cactus stack*** [19], as shown in Figure 1. The implementation of cactus

[1]Called "Cilk" in [6], but renamed "Cilk-1" in [16] and other MIT documentation.

[2]Although Fortress, Java Fork/Join Framework, TPL, and X10 employ work stealing, they do not suffer from the same interoperability problems, because they are byte-code interpreted by a virtual-machine environment.

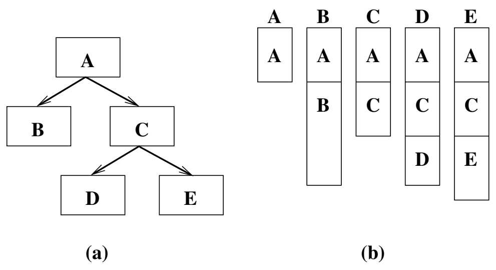

Figure 1: A cactus stack. **(a)** The invocation tree, where function `A` invokes `B` and `C`, and `C` invokes `D` and `E`. **(b)** The view of the stack by each of the five functions. In a serial execution, only one view is active at any given time, because only one function executes at a time. In a parallel execution, however, if some of the invocations are spawns, then multiple views may be active simultaneously.

stacks is a well-understood problem for which low-overhead implementations exist [16, 17].

In all known software implementations, however, transitioning from serial code (using a linear stack) to parallel code (using a cactus stack) is problematic, because the type of stack impacts the calling conventions used to allocate activation frames and pass arguments. We call the property of allowing arbitrary calling between parallel and serial code — including especially legacy (and third-party) serial binaries — ***serial-parallel reciprocity***, or ***SP-reciprocity*** for short.

SP-reciprocity is especially important if one wishes to multicore-enable legacy object-oriented environments by parallelizing an object's member functions. For example, suppose that a function `A` allocates a new object `x` whose type has a member function `foo()`, which we parallelize. Now, suppose that `A` is linked with a legacy binary containing a function `B`, and `A` passes `x` to `B`, which proceeds to invoke `x.foo()`. Without SP-reciprocity, this simple callback does not work.

Existing work-stealing concurrency platforms that support SP-reciprocity fail to provide provable bounds on either scheduling time or consumption of stack space. These bounds typically follow those of Blumofe and Leiserson [7]. Let T_1 be the ***work*** of a deterministic computation — its serial running time — and let T_∞ be the ***span***[3] of the computation — its theoretical running time on an infinite number of processors. Then, a work-stealing scheduler can execute the computation on P processors in time

$$T_P \le T_1/P + c_\infty T_\infty \ , \tag{1}$$

where $c_\infty > 0$ is a constant representing the ***span overhead***. This formula guarantees linear speedup when $P \ll T_1/T_\infty$, that is, the number P of processors is much less than the computation's ***parallelism*** T_1/T_∞. Moreover, if S_1 is the stack space of a serial execution, then the (cactus) stack space S_P consumed during a P-processor execution satisfies

$$S_P \le PS_1 \ . \tag{2}$$

Generally, we shall measure stack space in hardware pages, where we leave the page size unspecified. Many systems set an upper bound on S_1 of 256 4-KByte pages.

We refer to the problem of simultaneously achieving the three criteria of SP-reciprocity, a good time bound, and a good space bound, as the ***cactus-stack problem***. In this paper, we show how operating-system support for ***thread-local memory mapping (TLMM)*** allows a work-stealing runtime system to support full SP-reciprocity, so that a cactus stack interoperates seamlessly with the linear stack of legacy binaries, while simultaneously providing bounds on scheduling time and stack space. Whereas thread-local storage [35] gives each thread its own local memory at different virtual addresses within shared memory, TLMM allows a portion of the virtual-memory address space to be mapped independently by the various threads. We implemented a prototype TLMM-Linux operating system by modifying the page table maintained by the open-source Linux 2.6.29 kernel.

We also implemented a prototype Cilk-M concurrency platform, called Cilk-M 1.0, by modifying the open-source Cilk-5 runtime system[4] to use a TLMM-based cactus stack. The Cilk-M ***worker*** threads, which comprise the distributed scheduler, allow the user code to operate using traditional linear stacks, while the runtime system implements a cactus stack behind the scenes using TLMM support. Since TLMM allows the various worker stacks to be aligned, pointers to ancestor locations on the cactus stack are dereferenced correctly no matter which worker executes the user code.

Our prototype TLMM-Linux operating system and prototype Cilk-M 1.0 runtime system solve the cactus-stack problem. In Cilk-M, we define a ***Cilk function*** to be a function that spawns, and the ***Cilk depth*** of an application to be the maximum number of Cilk functions nested on the stack during a serial execution. Suppose that an application has work T_1, span T_∞, consumes stack space S_1 on one processor, and has a Cilk depth D. Then, analogously to Inequalities (1) and (2), the Cilk-M scheduler executes the computation on P processors in time

$$T_P \le T_1/P + c_\infty T_\infty \ , \tag{3}$$

where $c_\infty = O(S_1 + D)$, and it consumes stack space

$$S_P \le P(S_1 + D) \ . \tag{4}$$

Inequality (3) guarantees linear speedup when $P \ll T_1/(S_1+D)T_\infty$.

We have compared Cilk-M 1.0 to the original Cilk 5.4.6 on a variety of benchmarks in the Cilk-5 distribution. These studies indicate that the time overhead for managing the cactus stack with TLMM is generally as good or better than Cilk-5. The per-worker consumption of stack space in Cilk-M 1.0 on these benchmarks was no more than 2.5 times the serial space requirement. Moreover, the overall space consumption (including both stack and heap) of Cilk-M 1.0 is comparable to or better than that of Cilk-5.

The remainder of this paper is organized as follows. Section 2 provides background on work-stealing schedulers and cactus stacks using Cilk-5 as a model. Section 3 describes a range of conventional approaches that fail to solve the cactus-stack problem. Section 4 describes how Cilk-M leverages TLMM support to solve the cactus-stack problem. Section 5 describes the implementation of the prototype TLMM-Linux operating system. Section 6 analyzes the performance and space usage of the Cilk-M 1.0 prototype both theoretically and empirically. Section 7 sketches an alternative scheme to TLMM which also uses memory mapping but does not require operating-system support. Section 8 provides some concluding remarks.

2. CACTUS STACKS

This section describes how a work-stealing scheduler manipulates its cactus stack using Cilk-5 [16] as an exemplar. We cover in some detail how the Cilk-5 runtime system operates, because its runtime system forms the basis of Cilk-M 1.0 . We briefly review the theoretical bounds on space and time provided by Cilk-5.

[3]"Span" is sometimes called "critical-path length" and "computation depth" in the literature.

[4]The open-source Cilk-5 system is available at `http://supertech.csail.mit.edu/cilk/cilk-5.4.6.tar.gz`.

Before we dive into how the Cilk-5 work-stealing scheduler works, we first overview its linguistic model. Cilk-5 is a fork-join programming language, which permits dynamic creation of parallelism. More specifically, Cilk-5's linguistic constructs allow the programmer to denote the logical parallelism of the program rather than the actual parallelism at execution time. The Cilk-5 work-stealing scheduler [4, 7] respects the logical parallelism specified by the programmer while guaranteeing that programs take full advantage of the processors available at runtime.

Cilk-5 extends C with two main keywords: `spawn` and `sync`.[5] Parallelism is created using the keyword `spawn`. When a function call is preceded by the keyword `spawn`, the function is ***spawned*** and the scheduler may continue to execute the continuation of the caller in parallel with the spawned subroutine without waiting for it to return. The complement of `spawn` is the keyword `sync`, which acts as a local barrier and joins together the parallelism forked by `spawn`. The Cilk-5 runtime system ensures that statements after a `sync` are not executed until all functions spawned before the `sync` statement have completed and returned.

Cilk-5's work-stealing scheduler load-balances parallel execution across the available worker threads. Cilk-5 follows the "lazy task creation" strategy of Kranz, Halstead, and Mohr [23], where the worker suspends the parent when a child is spawned and begins work on the child.[6] Operationally, when the user code running on a worker encounters a `spawn`, it invokes the child function and suspends the parent, just as with an ordinary subroutine call, but it also places the parent frame on the bottom of a ***deque*** (double-ended queue). When the child returns, it pops the bottom of the deque and resumes the parent frame. Pushing and popping frames from the bottom of the deque is the common case, and it mirrors precisely the behavior of C or other Algol-like languages in their use of a stack.

The worker's behavior departs from ordinary serial stack execution if it runs out of work. This situation can arise if the code executed by the worker encounters a `sync`. In this case the worker becomes a ***thief***, and it attempts to steal the topmost (oldest) frame from a ***victim*** worker. Cilk-5's strategy is to choose the victim randomly, which can be shown [7, 16] to yield provably good performance. If the steal is ***successful***, the worker resumes the stolen frame.

Another situation where a worker runs out of work occurs if it returns from a spawned child to discover that its deque is empty. In this case, it first checks whether the parent is stalled at a `sync` and if this child is the last child to return. If so, it performs a ***joining steal*** and resumes the parent function, passing the `sync` at which the parent was stalled. Otherwise, the worker engages in random work-stealing as in the case when a `sync` was encountered.

The analysis of the Cilk-5 scheduler's performance is complicated (see [7]), but at a basic level, the reason it achieves the bound in Inequality (1) is that every worker is either working, in which case it is chipping away at the T_1/P term in the bound, or work-stealing, in which case it has a good probability of making progress on the T_∞ term. If the scheduler were to wait, engage in bookkeeping, or perform any action that cannot be amortized against one of these two terms, the performance bound would cease to hold, and in the worst case, result in much less than linear speedup on a program that has ample parallelism.

[5]Cilk-5 includes three additional keywords, `cilk`, `inlet`, and `abort`. The `cilk` keyword is a modifier so that the type system can preclude calls from C functions to Cilk functions. The `inlet` and `abort` keywords provide advanced features, including support for programs with speculative computation, which are orthogonal to the stack issues we are investigating.

[6]An alternative strategy is for the worker to continue working on the parent, and have thieves steal spawned children. Cilk-1 [6], TBB [34], and TPL [27] employ this strategy, but it can require unbounded bookkeeping space even on a single processor.

Strategy	*SP-Reciprocity*	*Time Bound*	*Space Bound*
1. Recompile everything	no	very strong	very strong
2. One stack per worker	yes	very strong	no
3. Depth-restricted stealing	yes	no	very strong
4. Limited-depth stacks	yes	no	very strong
5. New stack when needed	yes	very strong	weak
6. Recycle ancestor stacks	yes	strong	weak
7. TLMM cactus stacks	yes	strong	strong

Figure 2: Attributes of different strategies for implementing cactus stacks.

The analysis of the Cilk-5 scheduler's space usage is more straightforward. The scheduler maintains the so-called ***busy-leaves property*** [7], which says that at every moment during the execution, every ***extant*** — allocated but not yet deallocated — leaf of the spawn tree has a worker executing it. The bound on stack space given in Inequality (2) follows directly from this property. Observe that any path in the spawn tree from a leaf to the root corresponds to a path in the cactus stack, and the path in the cactus stack contains no more than S_1 space. Since there are P workers, PS_1 is an upper bound on stack space (although it may overcount). Tighter bounds on stack space have been derived for specific applications [5] using the Cilk-5 scheduler and for other schedulers [3].

3. THE CACTUS-STACK PROBLEM SEEMS HARD

This section overviews challenges in supporting SP-reciprocity while maintaining bounds on space and time, illustrating the difficulties that various traditional strategies encounter. Figure 2 categorizes attributes of the strategies of which we are aware. This list of strategies is not exhaustive but is meant to illustrate the challenges in supporting SP-reciprocity while maintaining bounds on space and time, and to motivate why naive solutions to the cactus-stack problem do not work. We now overview these strategies.

The main constraint on any strategy is that once a frame has been allocated, its location in virtual memory cannot be changed, because generally, there may be a pointer to a variable in the frame elsewhere in the system. Moreover, the strategies must respect the fact that a legacy binary can act as an adversary, allocating storage on the stack at whatever position the stack pointer happens to lie. Thus, when a legacy function is invoked, the runtime system has only one "knob" to dial — namely, choosing the location in virtual memory where the stack pointer points — and there better be enough empty storage beneath that location for all the stack allocations that the binary may choose to do. (Many systems assume that a stack can be as large as 1 MByte.) A strategy does have the flexibility to choose how it allocates memory in parallel code, that is, code that spawns, since that is not legacy code, and it can change the stack pointer. It must ensure, however, that when it invokes legacy serial code, there is sufficient unallocated storage on the stack for whatever the legacy serial code's needs might be.

Strategy 1: Recompile everything

This approach allocates frames off the heap and "eats the whole elephant" by recompiling all legacy serial functions to use a calling convention that directly supports a cactus stack. Very strong time and space bounds can be obtained by Strategy 1, and it allows serial code to call back to parallel code, but it does not provide true SP-reciprocity, since serial functions in legacy (and third-party) binary executables, which were compiled assuming a linear stack, cannot call back to parallel code. Cilk++ [20] employs this strategy. An interesting alternative is to use binary-rewriting technology [25, 31, 32] to rewrite the legacy binaries so that they use a heap-allocated cactus stack. This approach may not be feasible due to the difficulty of extracting stack references in optimized code. Moreover, it may have trouble obtaining good performance because transformations must err on the side of safety, and dynamically linked libraries might need to be rewritten on the fly, which would preclude extensive analysis.

Strategy 2: One stack per worker

This strategy gives each worker an ordinary linear stack. Whenever a worker steals work, it uses its stack to execute the work. For example, imagine that a worker W_1 runs parallel function `foo`, which spawns `A`. While W_1 executes `A`, another worker W_2 steals `foo` and resumes the continuation of `foo` by setting its base pointer to the top of `foo`, which resides on W_1's stack, and setting its stack pointer to the next available space in its own stack, so that the frames of any function called or spawned by `foo` next is allocated on W_2's stack.

With Strategy 2, the busy-leaves property no longer holds, and the stacks can grow much larger than S_1. In particular, W_1 must steal work if `foo` is not yet ready to sync when W_1 returns from `A`. Since `foo` is not ready to be resumed and cannot be popped off the stack, W_1 can only push the next stolen frame below `foo`. If `foo` is already deep in the stack and W_1 happens to steal a frame shallow in the stack, then W_1's stack could grow almost as large as $2S_1$. That is not so bad if it only happens once, but unfortunately, this scenario could occur recursively, yielding impractically large stack space consumption.

Strategy 3: Depth-restricted stealing

This approach is another modification of Strategy 2, where a worker is restricted from stealing any frame shallower than the bottommost frame on its stack. Thus, stacks cannot grow deeper than S_1. The problem with Strategy 3 is that a worker may be unable to steal even though there is work to be done, sacrificing the time bound. Indeed, Sukha [37] has shown that there exist computations for which depth-restricted work-stealing exhibits at most constant speedup on P workers, where ordinary work-stealing achieves nearly perfect linear speedup. TBB [34] employs a heuristic similar to depth-restricted work-stealing to limit stack space.

Strategy 4: Limited-depth stacks

This approach is similar to Strategy 2, except that a limit is put on the depth a stack can grow. If a worker reaches its maximum depth, it waits until frames are freed before stealing. The problem with Strategy 4 is that the cycles spent waiting cannot be amortized against either work or span, and thus the time bound is sacrificed, precluding linear speedup on codes with ample parallelism.

Strategy 5: New stack when needed

This strategy, which is similar to Strategy 2, allocates a new stack on every steal. In the scenario described in Strategy 2, when W_1 goes off to steal work, Strategy 5 switches to a new stack to execute the stolen work. Thus, nothing is allocated below `foo`, which avoids the unbounded space blowup incurred by Strategy 2.

Since Strategy 5 maintains the busy-leaves property, the total physical memory used for extant frames at any given moment is bounded by PS_1. The extant frames are distributed across stacks, however, where each stack may contain as little as a single extant frame. Since each stack may individually grow as large as S_1 over time and the stacks cannot be recycled until they contain no extant frames, the virtual-address space consumed by stacks may grow up to DPS_1, where D is the Cilk depth (defined in Section 1), a weak bound. Moreover, Strategy 5 may incur correspondingly high swap-space usage. Swap space could be reduced by directing the operating system to unmap unused stack frames when they are popped so that they are no longer backed up in the swap space on disk, but this scheme seems to laden with overhead. It may be possible to implement the reclamation of stack space lazily, however.

Strategy 6: Reuse ancestor stacks

This scheme is like Strategy 5, but before allocating a new stack after stealing a frame, it checks whether an ancestor of the frame is suspended at a `sync` and that the ancestor is the bottommost frame on the stack. If so, it uses the ancestor's stack rather than a new one. Strategy 6 is safe, because the ancestor cannot use the stack until all its descendants have completed, which includes the stolen frame. Although Strategy 6 may cut down dramatically on space compared with Strategy 5, it has been shown [14] to still require at least $\Omega(P^2 S_1)$ stack space for some computations. As with Strategy 7, the time bound obtained with this strategy exhibits some additional steal overhead compared to Inequality (2), which results from the traversal of ancestors' frames when searching for a reusable stack.

Strategy 7: TLMM cactus stacks

The strategy employed by Cilk-M and explored in this paper. In particular, we obtain the strong bounds given by Inequalities (3) and (4).

4. Cilk-M

Cilk-M leverages TLMM to solve the cactus-stack problem by modifying the Cilk-5 runtime system in two key ways. First, whereas Cilk-5 uses a heap-allocated cactus stack, Cilk-M uses a linear stack in each worker, fusing them into a cactus stack using TLMM. Second, whereas Cilk-5 uses a special calling convention for parallel functions and forbids transitions from serial code to parallel code, Cilk-M uses standard C subroutine linkage for serial code and compatible linkage for parallel code. This section describes how the Cilk-M runtime system implements these two modifications.

The Cilk-M cactus stack

A traditional operating system provides each process with its own virtual-address space. No two processes share the same virtual-address space, and all threads within a given process share the process's entire virtual-address space. TLMM, however, designates a region of the process's virtual-address space as "local" to each thread. This special ***TLMM region*** occupies the same virtual-address range for each thread, but each thread may map different physical pages to the TLMM region. The rest of the virtual-address space outside of the TLMM region remains shared among all threads within the process.

Recall that any strategy for solving the cactus-stack problem must obey the constraint that once allocated, a stack frame's location in virtual memory cannot be moved. Cilk-M respects this constraint by causing each worker thread to execute user code on a stack that resides in its own TLMM region. Whenever a successful steal occurs, the thief memory-maps the stolen frame and the ancestor frames in the invocation tree — the ***stolen stack prefix*** — so that these frames are shared between the thief and victim. The sharing is achieved by mapping the physical pages corresponding to the stolen stack prefix into the thief's stack, with the frames occupying the same virtual addresses at which they were initially allocated. Since the physical pages corresponding to the stack prefix are mapped to the same virtual addresses, a pointer to a local

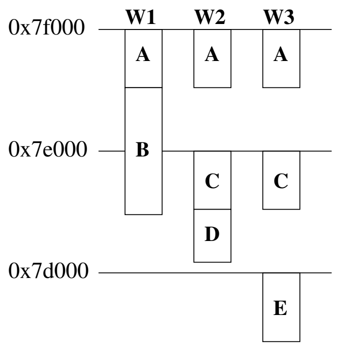

Figure 3: The view of stacks mapped in the TLMM region of each worker. The stack layout corresponds to the execution of the invocation tree shown in Figure 1. The horizontal lines indicate page boundaries, and the hexadecimal values on the left correspond to the virtual-memory addresses.

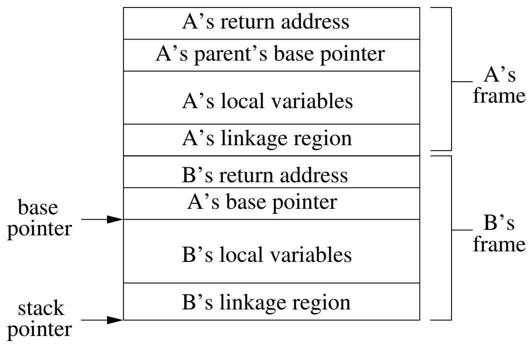

Figure 4: The layout of a linear stack with two frames. The figure shows a snapshot of a linear stack during execution, where `A` has called the currently executing function `B`. The figure labels the stack frame for each function on the right and marks the current base and stack pointers on the left.

variable in a stack frame references the same physical location no matter whether the thief or the victim dereferences the pointer.

Consider the invocation tree shown in Figure 1(a) as an example. Imagine three workers working on the three extant leaves `B`, `D`, and `E`. Figure 3 illustrates the corresponding TLMM region for each worker. Upon a successful steal, Cilk-M must prevent multiple extant children frames from colliding with each other. For instance, worker W_1 starts executing `A`, which spawns `B` and worker W_2 steals `A` from W_1, maps the stack prefix (i.e., the page where `A` resides) into its stack, resumes `A`, and subsequently spawns `C`. In this case, W_2 cannot use the portion of the page below frame `A`, because W_1 is using it for `B`. Thus, the thief, W_2 in this example, advances its stack pointer to the next page boundary upon a successful steal.

Continuing with the example, W_2 executes `C`, which spawns `D`. Worker W_3 may steal `A` from W_2 but, failing to make much progress on `A` due to a `sync`, be forced to steal again. In this case, W_3 happens to steal from W_2 again, this time stealing `C`. Thus, W_3 maps into its stack the pages where `A` and `C` reside, aligns its stack pointer to the next page boundary to avoid conflicting with `D`, resumes `C`, and spawns `E`.[7] In this example, W_1 and W_2 each map 2 pages in their respective TLMM regions, and W_3 maps 3. The workers use a total of 4 physical pages: 1 page for each of `A`, `C`, and `E`, and an additional page for `B`. Function `D` is able to share a page with `C`.

Upon a successful steal, the thief always advances its stack pointer to the next page boundary before resuming the stolen parent frame to avoid conflicting with the parallel child executing on the victim. Advancing the stack pointer causes the thief's stack to be fragmented.[8] Cilk-M mitigates fragmentation by employing a ***space-reclaiming policy*** in which the stack pointer is reset to the bottom of the frame upon a joining steal or a successful sync. This space-reclaiming policy is safe, because all other parallel subcomputations previously spawned by the frame have returned, and so the executing worker is not sharing this portion of the stack with any other worker.

Since a worker's TLMM region is not addressable by other workers, one deficiency of the TLMM strategy for implementing cactus stacks is that it does not support legacy serial binaries where the stack must be visible externally to other threads. For instance, an application that uses MCS locks [30] might allocate nodes for the lock queues on the local stack, rather than out of the heap. This code would generally not work properly under Cilk-M, because the needed nodes might not be visible to other threads. This issue seems to be more theoretical than practical, however, because we are unaware of any legacy applications that use MCS locks in this fashion or otherwise need to see another worker's stack. Nevertheless, the limitation is worth noting.

[7]Actually, this depiction omits some details. We shall elaborate more fully later in this section.

[8]An alternative strategy to prevent collision is to have workers to always spawn at a page boundary. This strategy, however, would cause more fragmentation of the stack space and potentially use more physical memory.

Cilk-M's calling convention

TLMM allows Cilk-M to support a cactus stack in which a frame can pass pointers to local variables to its descendants, but additional care must be taken to ensure that transitions between serial and parallel code are seamless. Specifically, the parallel code must use calling conventions compatible with those used by serial code.

Before we present Cilk-M's calling convention, we digress for a moment to outline the calling convention used by ordinary C functions. The calling convention described here is based on the x86 64-bit architecture [29], the platform on which we implemented the Cilk-M 1.0 system.

Figure 4 illustrates the stack-frame layout for a linear stack, assuming that the stack grows downward, where function `A` calls function `B`. The execution begins with frame `A` on the stack, where the frame contains (from top to bottom) `A`'s return address, `A`'s caller's base pointer, and some space for storing `A`'s local variables and passing arguments. Typically, arguments are passed via registers. If the argument size is too large, or when there are more arguments than the available registers, some arguments are passed via memory. We refer to these arguments as ***memory arguments*** and the region of frame where memory arguments are passed as the ***linkage region***.

Modern compilers generate code in the function prologue to reserve enough space in the frame for the function's local variables, as well as a linkage region large enough to pass memory arguments to any potential child function that the function may invoke, which takes an extra compiler pass to compute. Thus, in this example, when `A` calls `B`, the execution simply moves values to `A`'s linkage region. Even though this linkage region is reserved by `A`'s prologue and is considered part of `A`'s frame, it is accessed and shared by both `A` and `A`'s callee (e.g., `B`). Function `A` may access the area via either positive offset from `A`'s stack pointer or, if the exact frame size is known at compile time, negative offset from its base pointer. On the other hand, `A`'s callee typically accesses this area to retrieve values for its parameters via positive offset from its base pointer, but it could also access this area via its stack pointer if the frame size is known at compile time.

This calling convention assumes a linear stack where a parent's frame lies directly above its child's frame and the shared linkage region is sandwiched between the two frames. All children of a given function access the same linkage region to retrieve memory arguments, since the calling convention assumes that during an ordinary serial execution, at most one child function exists at a time. While these assumptions are convenient for serial code, it is problematic for parallel code employing work stealing, because multiple extant children cannot share the same linkage region. Furthermore, a gap may exist between the parent frame and the child frame in the TLMM-based cactus stack if the child frame is allocated immediately after a successful steal.

To circumvent these issues while still obeying the calling convention, workers in Cilk-M allocate a fresh linkage region immediately upon a successful steal by advancing the stack pointer a littler further beyond the next page boundary.[9] This strategy allocates the linkage region immediately above the child frame and allows additional linkage region to be created only when parallel execution occurs. Since multiple linkage regions may exist for multiple extant children, some care must be taken so that the parent passes the memory arguments via the appropriate linkage region, which we elaborate next.

Implementation of Cilk-M 1.0

We implemented Cilk-M 1.0 by modifying the Cilk 5.4.6 runtime system to use a TLMM-based cactus stack. The Cilk-M 1.0 runtime system also differs from the Cilk-5 runtime system in that it supports SP-reciprocity: a Cilk function may be called as well as spawned. If a Cilk function is spawned, it may execute in parallel with the continuation of its parent. If it is called, while it may execute in parallel with its children, the continuation of its parent cannot be resumed until it returns. Therefore, the runtime system must be aware of how a function is invoked. Maintaining the correct call/spawn semantics during execution is mainly a matter of handling the runtime data structure differently. Many of the implementation details of the Cilk-M 1.0 runtime resemble those of the Cilk++ runtime system, and we refer interested readers to [15].

To ensure execution correctness and to obey the Cilk-M calling convention, all the compiled Cilk functions maintain the following invariants:

1. All memory arguments are passed by stack pointer with positive offset.
2. All local variables are referenced by base pointer with negative offset.
3. Before each `spawn` statement, all live registers are flushed onto the stack.
4. If `sync` fails, all live registers are flushed onto the stack.
5. When resuming a stolen function after a `spawn` or `sync` statement, restore live register values from the stack.
6. When a call or spawn returns, flush the return value from the register onto the stack.
7. The frame size is fixed before any `spawn` statement.

Invariants 1 and 2 ensure correct execution in the event where a gap exists between the frames of the caller and the callee. Using the stack pointer to pass arguments to the child frame ensures that the arguments are stored right above the child frame. Similarly, the locals need to be referenced by the base pointer with negative offset, since the stack pointer may have changed.

[9]For simplicity, Cilk-M 1.0 reserves a fixed amount, 128 bytes, for each linkage region. Had we built a Cilk compiler, it would calculate the space required for each linkage region and pass that information to the runtime.

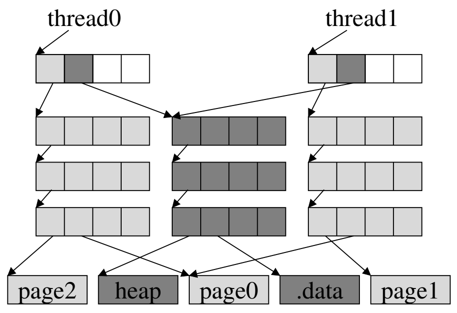

Figure 5: Example of a x86 64-bit page-table configuration for two threads on TLMM-Linux. The portion of the data structure dealing with the TLMM region is shaded light grey, and the remainder corresponding to the shared region is shaded dark grey. In the TLMM region, thread0 maps page2 first and then page0, whereas thread1 maps page1 first and then page0. The pages associated with the heap and the data segments are shared between the two threads.

Invariants 3–6 ensure that a thief resuming the stolen function accesses the most up-to-date local variable values, including return values from subroutines. This method is analogous to Cilk-5's strategy of saving execution states in heap-allocated frames [16]. Cilk-M 1.0 adapts the strategy to store live values directly on the stack, which is more efficient.

Finally, although Invariant 7 is not strictly necessary, it is a convenient simplification, because it ensures that a frame is allocated in contiguous virtual-address space. Since a frame may be stolen many times throughout the computation, if we were to allow a thief to allocate more stack space upon a successful steal, the frame allocation would end up fragmented and allocated in noncontiguous virtual-address space.

We did not build a Cilk-M compiler. To evaluate the Cilk-M 1.0 runtime system, we manually hand-compiled the set of benchmarks using gcc's inline-assembly feature to force the compiler to generate the desired assembly code. Ideally, a Cilk-M compiler would produce code satisfying the above invariants which is more optimized than our hand-compiled benchmarks.

5. SUPPORT FOR TLMM

TLMM provides the memory abstraction to allow workers to reserve a part of the virtual address range to be mapped independently while keeping the rest shared. We modified the Linux kernel to implement TLMM, referred as TLMM-Linux, which provides a low-level virtual-memory interface organized around allocating and mapping physical pages. The design attempts to impose as low overhead as possible while allowing the Cilk-M 1.0 runtime system to implement its work-stealing protocol efficiently. In addition, the design tries to be as general as possible so that the API can be used by other user-level utilities, applications, and runtime systems besides Cilk-M. This section describes the implementation of TLMM-Linux and the TLMM interface.

TLMM Implementation

We implemented TLMM for Linux 2.6.29 running on x86 64-bit CPU's, such as AMD Opterons and Intel Xeons. We added about 600 lines of C code to manage TLMM virtual-memory mappings and modified several lines of the context-switch and memory-management code to be compatible with TLMM.

Figure 5 illustrates the design. TLMM-Linux assigns a unique root page directory to each thread in a process. The x86 64-bit page tables have four levels, and the page directories at each level contain 512 entries. One entry of the root-page directory is re-

```
addr_t sys_reserve(size_t n):
    Reserve n bytes for the TLMM region, and return the start address.
pd_t sys_palloc(void):
    Allocate a physical page, and return its descriptor.
sys_pfree(pd_t p):
    Free the page descriptor p.
sys_pmap(pd_t p[], addr_t a):
    Map the pages represented by the descriptors in p starting at virtual
    address a.
```

Figure 6: System-call API for TLMM.

served for the TLMM region, which corresponds to 512-GByte of virtual address space, and the rest of the entries correspond to the shared region. Threads in TLMM-Linux share page directories that correspond to shared region. Therefore, the TLMM-Linux virtual-memory manager needs to synchronize the entries in each thread's root page directory and populate the shared lower-level page directories only once.

TLMM interface

Figure 6 summarizes the TLMM system call interface. `sys_reserve` marks `n` bytes of the calling thread's process address space as the TLMM region and returns the starting address of the region. `sys_palloc` allocates a physical page and returns its page descriptor. A page descriptor is analogous to a file descriptor and can be accessed by any thread in the process.[10] `sys_pfree` frees a page descriptor and its associated physical page.

To control the physical-page mappings in a thread's TLMM region, the thread calls `sys_pmap`, specifying an array of page descriptors to map, as well as a base address in the TLMM region at which to begin mapping the descriptors. `sys_pmap` steps through the array of page descriptors, mapping physical pages for each descriptor to subsequent page-aligned virtual addresses, to produce a continuous virtual-address mapping that starts at the base address. A special page-descriptor value `PD_NULL` indicates that a virtual-address mapping should be removed. Thus, a thief in Cilk-M that finishes executing a series of functions that used a deep stack can map a shorter stolen stack prefix with a single system call.

This low-level design for the TLMM-Linux interface affords a scalable kernel implementation. One downside, however, is that the kernel and the runtime system must both manage page descriptors. The kernel tracks at which virtual addresses the page descriptors are mapped. The runtime tracks the mapping between page descriptors and stack pages so that a thief can remap its stack with pages corresponding to the stolen prefix upon a successful steal. We considered an alternative interface design where the TLMM-Linux provides the cactus abstraction, rather than the Cilk-M runtime system, and Cilk-M relies on the kernel to remap a thief's stack upon a successful steal. In this design, the runtime no longer needs to keep track of the page mappings on the stacks, but a thief must acquire a lock on the victim's deque while the remapping takes place, which may prevent other thieves from stealing from the same victim for a longer period of time. Because of this performance consideration and the fact that is perhaps overly specific to work-stealing systems, we chose the low-level interface over the cactus-abstraction interface.

The most unfortunate aspect of the TLMM scheme for solving the cactus-stack problem is that it requires a change to the operating system. Section 7 sketches an alternative "workers-as-processes" scheme, which, although it does not require operating-system support, has other deficiencies. Most of our theoretical and empirical analysis for the TLMM scheme applies to the workers-as-processes scheme as well.

[10]To save some coding effort, our prototype TLMM-Linux implementation supports a maximum of 500 page descriptors per process, because it employs a static map to keep track of the mappings between page descriptors and physical pages. To relax this limitation, one could employ a dynamic map, which would add little overhead, because the overhead of growing the map can be amortized over many page allocations.

6. EVALUATION

This section evaluates the Cilk-M system. We provide theoretical bounds on stack space and running time, which, although not as strong as those of Cilk-5, nevertheless provide reasonable guarantees. We compare Cilk-M 1.0's empirical performance to that of the original Cilk-5 system. The results indicate that Cilk-M 1.0 performs similarly to Cilk-5 and that the overhead for remapping stacks is modest. Cilk-M 1.0's consumption of stack space appears to be well within the range of practicality, and its overall space consumption (including stack and heap space) is comparable to that of Cilk-5.

Theoretical bounds

We first analyze the consumption of stack space for an application run under Cilk-M. Let S_1 be the number of pages in a serial execution of the program, let S_P be the number of pages that Cilk-M consumes when run on P workers, and let D be the Cilk depth of the application. The bound $S_P \le P(S_1 + D)$ given in Inequality (4) holds, because the worst-case stack depth of a worker is $S_1 + D$ pages. This worst case occurs when every Cilk function on a stack that realizes the Cilk depth D is stolen. The stack pointer is advanced to a page boundary for each of these D stolen frames, contributing an extra D to the normal number S_1 of pages in the stack. Since there are P workers, the bound follows.

As we shall see from the benchmark studies, this upper bound is loose in terms of actual number of pages. First, since different stack prefixes are shared among workers, we are double-counting these shared pages. Second, we should not expect, which the benchmark studies bear out, that every frame on a stack is stolen. Moreover, the space-reclaiming policy also saves space in practice. Nevertheless, the theoretical bound provides confidence that space utilization cannot go drastically awry.

Cilk-M achieves the time bound $T_p \le T_1/P + c_\infty T_\infty$ given in Inequality (3), where T_1 is the work of the program, let T_∞ be its span, and $c_\infty = O(S_1 + D)$. The proof of this bound requires theoretical arguments beyond the scope of this paper. The bound reflects the increased cost of a steal compared to the constant-time cost in Cilk-5. In the worst case, every steal might need to map a nearly worst-case stack of depth $S_1 + D$, which costs $O(S_1 + D)$ time. The actual bound can be proved using the techniques of [2] and [7], adapted to consider the extra cost of stealing in Cilk-M .

As with the space bound, the time bound is loose, because the worst-case behavior used in the proof is unlikely. One would not normally expect to map an entire nearly worst-case stack on every steal. Nevertheless, the bound provides confidence, because applications with sufficient parallelism are guaranteed to achieve near-perfect linear speedup on an ideal parallel computer, as is assumed by prior theoretical studies.

Empirical studies

Theoretical bounds alone, especially those based on asymptotic analysis, do not suffice to predict whether a technology works in practice, where the actual values of constants matter. In particular, we had two main concerns when we started this work. The first concern was whether the cost of entering and exiting the kernel would be too onerous to allow a memory-mapping solution to the

Application	*Input*	*Description*
`cholesky`	4000/40000	Cholesky factorization
`cilksort`	10^8	Parallel merge sort
`fft`	2^{26}	Fast Fourier transform
`fib`	42	Recursive Fibonacci
`fibx`	280	Synthetic benchmark with deep stack
`heat`	2048×500	Jacobi heat diffusion
`knapsack`	32	Recursive knapsack
`lu`	4096	LU-decomposition
`matmul`	2048	Matrix multiply
`nqueens`	14	Count ways to place N queens
`rectmul`	4096	Rectangular matrix multiply
`strassen`	4096	Strassen matrix multiply

Figure 7: The 12 benchmark applications.

cactus-stack problem. The second concern was whether the fragmentation of the stack would consume too much space, rendering the solution impractical. To address the first concern, we compared the performance of Cilk-M 1.0 and Cilk-5 empirically on 12 applications. The benchmark results indicate that the two systems perform similarly, with Cilk-M 1.0 sometimes outperforming Cilk-5 despite the additional overhead for remapping the stacks. To address the second concern, we profiled the stack space of the applications running on Cilk-M 1.0 with 16 cores. The data from this experiment indicate that the per-worker consumption of stack space on these benchmarks was at most a factor of 2.5 more than the serial space requirement, which is modest. Due to the fragmentation of the stack, Cilk-M 1.0 indeed has higher stack space overhead than Cilk-5; as a trade-off, however, Cilk-5 tends to consume more heap space than Cilk-M 1.0 due to the use of a heap-allocated cactus stack. To better understand the trade-offs made between the two runtime systems, we profiled the stack and heap space consumption of each system running the applications with 16 cores. The benchmark results indicate that the additional stack space overhead in Cilk-M 1.0 is inexpensive when one considers the overall space consumption.

General setup. We ran all experiments on an AMD Opteron system with 4 quad-core 2 GHz CPU's having a total of 8 GBytes of memory. Each core on a chip has a 64-KByte private L1-data-cache and a 512-KByte private L2-cache, but all cores on a chip share a 2-MByte L3-cache.

We evaluated the system with 12 benchmark applications, all of which are included in the Cilk-5 distribution except `fibx`, which is a synthetic benchmark we devised to generate large stacks. Figure 7 provides a brief description of each application. Whereas applications for Cilk-M 1.0 were hand-compiled as described in Section 4, applications for Cilk-5 were compiled with the source-to-source translator included in the Cilk-5 distribution to produce C postsource. The postsources for both systems were compiled with gcc 4.3.2 using `-O2` optimization.

Relative performance. Figure 8 compares the performance of the applications run on Cilk-M 1.0 and Cilk-5. For each application we measured the mean of 10 runs on each of Cilk-M 1.0 and Cilk-5, and the mean on each has standard deviation less than 3%. We normalized the mean for Cilk-M 1.0 by the mean for Cilk-5. Cilk-M 1.0 performs similarly to Cilk-5 for most of the applications and is sometimes faster. These results indicate that the additional overhead in Cilk-M 1.0 for remapping the stacks is modest and does not impact application performance in general. Moreover, the good performance on `fib`, which involves mostly spawning and function calls and little computation *per se*, indicates that the Cilk-M linear-stack-based calling convention is generally superior to the Cilk-5 heap-based one.

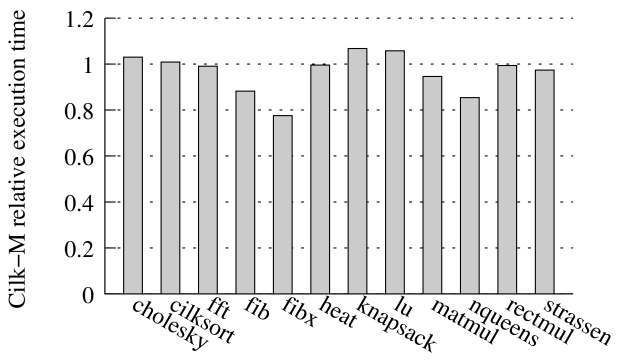

Figure 8: The relative execution time of Cilk-M 1.0 compared to Cilk-5 for 12 Cilk applications on 16 cores. Each value is calculated by normalizing the execution time of the application on Cilk-M 1.0 with the execution time of the application on Cilk-5.

Application	D	S_1	$S_{16}/16$	$S_1 + D$
`cholesky`	12	2	3.19	14
`cilksort`	18	2	3.19	20
`fft`	22	4	3.75	26
`fib`	43	2	3.69	45
`fibx`	281	8	8.44	289
`heat`	10	2	2.38	12
`knapsack`	34	2	5.00	36
`lu`	10	2	3.19	12
`matmul`	22	2	3.31	24
`nqueens`	16	2	3.25	18
`rectmul`	27	2	4.06	29
`strassen`	8	2	3.13	10

Figure 9: Consumption of stack space per worker for 12 Cilk applications running on Cilk-M 1.0 as measured in 4-KByte pages. The value D is the Cilk depth of the application. The serial space S_1 was obtained by running the application on one processor. The value S_{16} was measured by taking the maximum of 10 runs on 16 cores. Shown is the average space per worker $S_{16}/16$. The rightmost column shows the theoretical upper bound for consumption of stack space per worker from Inequality (4).

Space utilization. Figure 9 shows the stack space consumption of the benchmark applications running on Cilk-M 1.0 with 16 cores. Since the consumption of stack pages depends on scheduling, it varies from run to run. We ran each application 10 times and recorded the maximum number of pages used. Overall, the applications used less space than predicted by the theoretical bound, and sometimes much less, confirming the observation that the upper bound given in Inequality (4) is loose. Indeed, none of the applications used more than 2.5 times the stack space per worker of the serial stack space.

Figure 10 shows the stack and heap space consumptions of the benchmark applications running on Cilk-M 1.0 and on Cilk-5 with 16 workers. Both runtime systems employ an internal memory allocator that maintains local memory pools for workers to minimize contention and a single global pool to rebalance the memory distribution between local pools. The heap consumption is measured by the total number of physical pages requested by the memory allocator from the operating system at the end of the execution.[11] Again, we ran each application 10 times and recorded the maximum number of pages used.

[11]This measurement does not include space for the runtime data structures allocated at the system startup, which is relatively small, comparable between the two systems, and always fixed with respect to the number of workers.

Application	*Cilk-M* S_{16}	*Cilk-5* S_{16}	*Cilk-M* H_{16}	*Cilk-5* H_{16}	*Cilk-M Sum*	*Cilk-5 Sum*
`cholesky`	51	16	193	345	244	361
`cilksort`	51	16	193	265	244	281
`fft`	60	48	169	1017	229	1065
`fib`	59	16	169	185	228	201
`fibx`	135	64	217	217	353	281
`heat`	38	16	209	273	247	289
`knapsack`	80	16	169	361	249	377
`lu`	51	16	185	265	236	281
`matmul`	53	16	169	257	222	273
`nqueens`	52	16	161	249	213	265
`rectmul`	65	32	169	240	234	272
`strassen`	50	16	161	417	211	433

Figure 10: Comparison of the overall stack and heap consumptions between Cilk-M 1.0 and Cilk-5 for 12 Cilk applications running with 16 workers. The values were measured by taking the maximum of 10 runs on 16 cores, and measured in 4-KByte pages. The last two columns show the sum of the stack and heap space consumptions for the two systems.

Across all applications, Cilk-M 1.0 uses about 2–4 times, and in one case (i.e., `knapsack`) 5 times more pages on the stack, than that of Cilk-5 due to fragmentation resulted from successful steals. The additional space overhead caused by fragmentation is never referenced by the runtime or the user code, however, and thus the additional stack space usage does not cause memory latency. On the other hand, Cilk-5 tends to use comparable or slightly more the heap space used by Cilk-M 1.0 (less than 3 times more), except for one application, `fft`. Since `fft` contains some machine generated code for the base cases, the Cilk functions in `fft` contain large number of temporary local variables that are used within the functions but not across `spawn` statements. The `cilk2c` compiler used by Cilk-5 faithfully generates space for these variables on the heap-allocated cactus stack, resulting in large heap space usage. With the same program, Cilk-M 1.0 uses the same amount of stack space for these temporary local variables as however much space a C compiler would allocate for them. Finally, when comparing the overall space consumption, Cilk-M 1.0 tends to use less space than Cilk-5, except for `fib` and `fibx`. The Cilk functions in these two applications have very few local variables, and therefore their heap-allocated cactus stack in Cilk-5 consumes relatively little space. Furthermore, `fibx` is a synthetic benchmark that we devised to generate large stacks (i.e., with large Cilk depth), so Cilk-M 1.0 ends up having a deep stack for `fibx`.

7. AN ALTERNATIVE TO TLMM

Some may view TLMM as too radical a solution to the cactus-stack problem, because it involves modifying the operating system. This section considers another possible memory-mapping solution to the cactus-stack problem which does not require operating-system support. The idea of the ***workers-as-processes*** scheme is to implement workers as processes, rather than threads, but still use memory mapping to support the cactus stack. This section sketches a design for this alternative scheme and discusses its ramifications.

During the start-up of the workers-as-processes scheme, each worker uses memory-mapping to share the heap and data segments across the workers' address spaces by invoking `mmap` with a designated file descriptor on the virtual-address range of where the heap and data segments reside. Since processes by default do not share memory, this strategy provides the illusion of a fully shared address space for these segments. Since a thief needs access to the stolen stack prefix of its victim, the runtime system also must memory-map all the workers' stacks to the file, recording the file offsets for all pages mapped in the stacks so that they can be manipulated. In addition, other resources — such as the file system, file descriptors, signal-handler tables, and so on — must be shared, although at least in Linux, this sharing can be accomplished straightforwardly using the `clone` system call.

Although this workers-as-processes approach appears well worth investigating, there are few complications that one needs to deal with if this approach is taken. Here is a summary of challenges.

First, the runtime system would incur some start-up overhead to set up the shared memory among workers. A particular complication would occur if the runtime system is initialized in the middle of a callback from C to Cilk for the first time. In this case, the runtime system must first unmap the existing heap segment used by the C computation, remap the heap segment with new pages so that the mapping is backed up by a file (so as to allow sharing), and copy over the existing data from the old mapping to the new mapping.

Second, it seems that the overhead for stealing would increase. If m is the number of pages that a thief must map to install its victim's stack prefix, the thief might need to invoke `mmap` m times, once for each address range, rather than making a single call as with our TLMM implementation, because it is unlikely that consecutive pages in the stolen prefix reside contiguously in the designated file. These m calls would result in $2m$ kernel crossings, and thus increase the steal overhead. One might imagine an `mmap` interface to that would support mapping of multiple physical pages residing in a noncontiguous address range, but such an enhancement would involve a change to the operating system, exactly what the workers-as-processes scheme tries to avoid.

Finally, and perhaps most importantly, workers-as-processes makes it complicated to support system calls that change the address space, such as `mmap` and `brk`. When one worker invokes `mmap` to map a file into shared memory, for example, the other workers must do the same. Thus, one must implement a protocol to synchronize all the workers to perform the mapping before allowing the worker that performed the `mmap` to resume. Otherwise, a race might occur, especially if the application code communicates between workers through memory. This protocol would likely be slow because of the communication it entails. Furthermore, in some existing implementation of system calls library such as `glibc`, calling `malloc` with size larger than 64 KBytes results in invoking `mmap` to allocate a big chunk of memory. Therefore, with this scheme, one would need to rewrite the `glibc` library to intercept the `mmap` call and perform the synchronization protocol among workers for the newly allocated memory as well.

Despite these challenges, the workers-as-processes "solution" appears to be an interesting research direction. It may be that hybrid schemes exist which modify the operating system in a less intrusive manner than what TLMM does, for example, by allowing noncontiguous address ranges in `mmap`, by supporting `mmap` calls across processes, etc. We adopted TLMM's strategy of sharing portions of the page table, because we could explore a memory-mapping solution with relatively little engineering effort. Our work focuses more on such solution's implication on the runtime system, however, and not as much on how the memory-mapping should be supported. Most of the work described in this paper, including the design of the runtime system and the theoretical bounds, applies to the workers-as-processes approach as well. Cilk-M 1.0 seems to perform well, which may motivate the exploration of other, possibly more complex strategies that have different systems ramifications.

8. CONCLUSION

From an engineering perspective, we have laid out some choices for implementers of work-stealing environments. There seem to be four options for solving the cactus-stack problem: sacrificing interoperability with binaries that assume a linear-stack calling convention, sacrificing a time bound, sacrificing a space bound, and coping with a memory-mapping solution similar to those laid out in this paper.

Sacrificing interoperability limits the ability of the work-stealing environment to leverage past investments in software. An engineering team may be willing to sacrifice interoperability if it is developing a brand-new product, but they may be more cautious if they are trying to upgrade a large codebase to use multicore technology.

Sacrificing the time or space bound may be fine for a product where good performance and resource utilization are merely desirable. It may be unreasonable, however, for a product hoping to meet a hard or soft real-time constraint. Moreover, even for everyday software where fast performance is essential for good response times, time and space bounds provide a measure of predictability.

Coping with memory mapping by modifying the operating system may not be possible for those working on closed operating systems which they cannot change, but it may be fine for applications running on an open-source platform. Moreover, as multicore platforms grow in importance, future operating systems may indeed provide TLMM-like facilities to meet the challenges. In the shorter term, if it is not possible to modify the operating system, it may still be possible to implement a workers-as-processes scheme in order.

The particular engineering context will shape which option is the most reasonable, and in developing the case for a memory-mapped solution to the cactus-stack problem, we have placed an important new option on the table.

9. ACKNOWLEDGMENTS

Thanks for Matteo Frigo of Cilk Arts for tips and insights on the code sequences necessary to fool gcc into doing what we wanted, as well as for many helpful discussions. Thanks to Bradley Kuszmaul of MIT CSAIL and Tim Huang of the University of Otago for helpful discussions.

10. REFERENCES

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification, Version 1.0*. Sun Microsystems, Inc., Mar. 2008.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98*, pages 119–129, June 1998.

[3] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *SPAA '95*, pages 1–12, July 1995.

[4] R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, MIT Department of EECS, Sept. 1995.

[5] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *SPAA '96*, pages 297–308, June 1996.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *JPDC*, 37(1):55–69, August 1996.

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

[8] R. D. Blumofe and D. Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical Report, University of Texas at Austin, 1999.

[9] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81*, pages 187–194, Oct. 1981.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538. ACM, 2005.

[11] R. Feldmann, P. Mysliwietz, and B. Monien. Studying overheads in massively parallel min/max-tree evaluation. In *SPAA '94*, pages 94–103, June 1994.

[12] R. Finkel and U. Manber. DIB — A distributed implementation of backtracking. *ACM TOPLAS*, 9(2):235–256, Apr. 1987.

[13] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *OSDI '94*, pages 201–213, Nov. 1994.

[14] M. Frigo, 2009. Private communication.

[15] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *SPAA '09*, pages 79–90, Calgary, Canada, Aug. 2009. ACM.

[16] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98*, pages 212–223, 1998.

[17] S. C. Goldstein, K. E. Schauser, and D. Culler. Enabling primitives for compiling parallel languages. In *LCR '95*, May 1995.

[18] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, Oct. 1985.

[19] E. A. Hauck and B. A. Dent. Burroughs' B6500/B7500 stack mechanism. *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 245–251, 1968.

[20] Intel Corporation. *Intel Cilk++ SDK Programmer's Guide*, October 2009. Document Number: 322581-001US.

[21] R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, July 1993.

[22] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., second edition, 1988.

[23] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: A high-performance parallel Lisp. In *PLDI '89*, pages 81–90, June 1989.

[24] B. C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, MIT Department of EECS, May 1994.

[25] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Softw. Pract. Exper.*, 24(2):197–218, 1994.

[26] D. Lea. A Java fork/join framework. In *Java Grande Conference*, pages 36–43, 2000.

[27] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *OOPSLA '09*, pages 227–242, 2009.

[28] C. E. Leiserson. The Cilk++ concurrency platform. In *46th Design Automation Conference*. ACM, July 2009.

[29] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99. Available at www.x86-64.org/documentation/abi.pdf, May 2009.

[30] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PPOPP '91*, pages 106–113, 1991.

[31] MIPS Computer Systems, Inc. *RISCompiler Languages Programmer's Guide*, December 1988.

[32] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07*, pages 89–100, 2007.

[33] R. S. Nikhil. Cid: A parallel, shared-memory C for distributed-memory machines. In *LCPC '94*, Aug. 1994.

[34] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[35] D. Stein and D. Shah. Implementing lightweight threads. In *USENIX '92*, pages 1–9, 1992.

[36] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, third edition, 2000.

[37] J. Sukha. Brief announcement: A lower bound for depth-restricted work stealing. In *SPAA '09*, Aug. 2009.

[38] M. T. Vandevoorde and E. S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, Aug. 1988.