A Practical Solution to the Cactus Stack Problem

Chaoran Yang
Department of Computer Science
Rice University
chaoran@rice.edu

John Mellor-Crummey
Department of Computer Science
Rice University
johnmc@rice.edu

ABSTRACT

Work-stealing is a popular method for load-balancing dynamic multithreaded computations on shared-memory systems. In theory, a randomized work-stealing scheduler can achieve near linear speedup when the computation has sufficient parallelism and requires stack space that is linear in the number of processors. In practice, however, work-stealing runtimes sacrifice interoperability with serial code to achieve these bounds. For example, both Cilk and Cilk++ prohibit a C function from calling a Cilk function. Other work-stealing runtime systems that do not have this restriction either lack a strong time bound, which might cause them to deliver little or no speedup in the worst case, or lack a strong space bound, which might lead to an excessive memory footprint. This problem was previously described as the *cactus stack problem*.

In this paper, we present *Fibril*, a new multithreading library that supports a fork-join programming model using work-stealing. Fibril solves the cactus stack problem by (1) implementing on a cactus stack that conforms to the calling conventions of serial code and (2) returning unused memory pages of suspended stacks to the operating system to bound consumption of physical memory. Theoretically, Fibril achieves strong bounds on both time and memory usage without sacrificing interoperability with serial code. Empirically, Fibril achieves up to 3× the performance of Intel Cilk Plus and up to 8× the performance of Intel Threading Building Blocks for the 12 benchmarks we evaluated.

Keywords

Work-stealing; cactus stack; interoperability.

1. INTRODUCTION

Work-stealing is a standard technique for load balancing parallel tasks in multicore systems. Many parallel programming models employ work-stealing to support dynamic task parallelism, including Cilk [8], OpenMP 3.0 [3], Threading Building Blocks (TBB) [15], Java Fork/Join Framework [9], Task Parallel Library [11], and X10 [5]. Work-stealing enables an efficient implementation that guarantees bounds on both time and stack space. For a class of computations, Blumofe and Leiserson [4] prove strong time and space bounds when stealing is unrestricted. Let T_1 be the running time of a deterministic parallel computation on one processor, i.e., its *work*, and let T_∞ be its ideal running time on an infinite number of processors, i.e., its *span*. Then a randomized work-stealing scheduler can perform the computation on P processors in expected time

$$T_p \leq T_1/P + c_\infty T_\infty, \tag{1}$$

where c_∞ is a constant value called *span overhead*. This bound guarantees *linear speedup* when $P \ll T_1/T_\infty$, that is, the number of processors P is much less than the computation's average parallelism T_1/T_∞. Moreover, if S_1 is the stack space of a serial execution, a randomized work-stealing scheduler can execute the computation on P processors using stack space

$$S_p \leq PS_1. \tag{2}$$

This bound guarantees that the stack space usage in a parallel execution is at worst *linear* in the number of processors.

For a program written in an Algol-like programming language, such as C, its execution can be thought of as a depth-first traversal of its *invocation tree*. If a function instance A calls a function instance B,[1] A is the parent of B in the invocation tree. When a processor executes a function, it allocates an *activation frame* that stores local data, including arguments and local variables for that function. A processor frees an activation frame when it returns from that function. For a serial execution of a program, activation frames can be allocated using a simple *linear stack*. In a linear stack, a function call advances the stack pointer to allocate a new frame on the top of the stack; when the function returns, it restores the stack pointer to its value for the parent frame, freeing the child frame.

When a program executes in parallel, however, a linear stack is insufficient. In a parallel execution, multiple children of a function may exist simultaneously; their activation frames form a tree of stacks, namely a *cactus stack*. A cactus stack is essentially an N-ary tree in which each node has a pointer to its parent node. Each node in a cactus stack is a linear stack that allocates and frees activation frames by manipulating a stack pointer. If a frame creates a new child while another child still exists, it allocates a new stack to

[1] In the rest of the paper, we use the term *function* instead of *function instance* where there is no room for ambiguity.

SPAA'16, July 11–13, 2016, Pacific Grove, California, USA.

ISBN 978-1-4503-4210-0/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2935764.2935787

store the new child's activation frame with a parent pointer that points to itself.

Cactus stacks are a key building block in parallel runtime systems. Ideally, a cactus stack implementation should:

- support full interoperability with serial binaries compiled to use a linear stack,
- enable a scheduler to achieve nearly perfect linear speedup on computations with ample parallelism, and
- consume a bounded amount of stack space that is practical for general-purpose systems.

It is difficult for work-stealing runtimes to achieve all three criteria at the same time. For example, Cilk [8] guarantees the time bound and space bound, however it forbids a C function from calling Cilk functions. Interoperability with serial code, aka *serial-parallel reciprocity*, is important for building reusable software. Common design patterns, such as *visitor* and *observer* patterns, use callback functions to communicate between objects. One cannot use these patterns if the visitor or observer's callback invokes a Cilk function. TBB [15] is interoperable with serial code. However, to avoid consuming potentially unbounded stack space, TBB restricts a worker to steal only frames that are deeper on the stack than its own suspended frame; this may lead to sub-linear speedup for some computations. Lee et al. [10] refer to the problem of achieving the three aforementioned criteria simultaneously as the *cactus stack problem*. They presented a solution that leverages a *thread-local memory mapping* (TLMM) mechanism. TLMM enables threads in a process to map different physical memory pages for the same region of the process's virtual address space. Sadly, no existing operating systems supports TLMM, which makes Lee et al.'s approach impractical on systems running conventional kernels.

This paper presents a practical solution to the cactus stack problem. We designed a library-based work-stealing system, *Fibril*, that supports a fork-join programming model. Fibril uses a cactus stack that is fully interoperable with serial code that uses a linear stack. Fibril achieves a strong time bound $T_p \leq T_1/P + c_\infty T_\infty$, where $c_\infty = O(S_1 + D)$, and a strong space bound $S_p \leq P(S_1 + D)$. In both bounds, D is the *Fibril depth* (defined in Section 4.4) of a Fibril program.

The contribution of Fibril is two-fold. First, we present a cactus stack implementation that is more efficient than existing work-stealing systems. Second, we present a stack management technique that maintains a strong space bound without sacrificing a strong time bound. Fibril's cactus stack may use up to DPS_1 pages of virtual address space, which is practical for systems that use a 64-bit address space.

Section 2 explains the cactus stack problem in work-stealing runtime systems. Section 3 describes existing approaches for managing cactus stacks in work-stealing systems. Section 4 describes the implementation of Fibril in detail. Section 5 presents an evaluation of Fibril. Section 6 presents our conclusions.

2. BACKGROUND

We investigate the cactus stack problem in the context of a fork-join programming model. In this model, a programmer expresses logical parallelism using a `fork` construct. When a function A initiates a `fork` to perform an asynchronous call to a function B, A may continue to execute without waiting for B to complete. We refer to the callee B at a `fork` as a child *task* and the caller A as its *parent*. A `join` construct synchronizes between a parent and its child tasks, suspending execution of the parent until all of its children complete.

We consider the cactus stack problem in the context of a work-stealing scheduler, which executes a computation expressed in a fork-join programming model on a finite set of hardware threads, called *workers*. In particular, we consider the randomized work-stealing scheduler proposed by Blumofe and Leiserson [4]. Cilk and its descendants, Cilk++ and Intel Cilk Plus, use such a scheduler in their runtime system to support their fork-join programming models.

In Cilk's work-stealing scheduler, each worker maintains a double-ended queue, known as a *deque*. A worker pushes the parent frame onto its deque at a `fork` (`cilk_spawn`) and pops it when the task completes.[2] When a worker runs out of work, it becomes a *thief* and steals from the top of a randomly selected worker's deque. We refer to the worker that has its parent frame stolen as a *victim*. When a worker steals a frame, it executes the frame until it encounters a `join` (`cilk_sync`). At a `join`, the current frame is *suspended* if any of its child tasks have not yet completed; otherwise, the worker continues with the frame. If a thief steals a frame, the victim will fail when attempting to pop the frame from its deque. Then the victim will attempt to *resume* the stolen frame, expecting that the thief has completed the frame. If the resumption succeeds, the victim executes the stolen frame starting at the `join`; otherwise, it becomes a thief and performs randomized stealing.

Blumofe and Leiserson [4] have shown that Cilk's work-stealing scheduler can execute a computation on P workers in expected time $T_p \leq T_1/P + c_\infty T_\infty$. Their proof of the time bound indicates that, to achieve near-linear speedup, each worker needs to be either working and making progress on the T_1/P term, or randomly stealing and making progress on the T_∞ term with high probability. If any worker were to wait, or candidate frames for stealing are restricted, the time bound would cease to hold, resulting in sublinear speedup on a program with sufficient parallelism.

Cilk's work-stealing scheduler maintains the so-called *busy-leaves* property [4], which states that during the execution of the computation, every extant leaf of the invocation tree has a worker executing it. The busy-leaves property leads to the space bound of PS_1, since any path from a leaf to the root of the invocation tree corresponds to a path in the cactus stack, no path can require more than S_1 space, and there can be at most P leaves active at a time.

We analyze the cactus stack problem under the assumption that a work-stealing scheduler cannot move a frame to a different address in memory once it is allocated as moving a frame will invalidate all pointers to variables in the frame.[3] Under this restriction, if a worker fails to pop from its deque and finds the frame on top of its stack suspended, it cannot move the frame out of its stack. The worker also cannot

[2]Some work-stealing schedulers push a child frame created by a `fork` onto a worker's local deque and continue executing the parent frame after the `fork`. We do not consider such schedulers in this paper because they may require an unbounded amount of space for bookkeeping.

[3] Some work-stealing programming models forbid users from creating a pointer to a local variable and hence do not have this restriction. Other work-stealing systems leverage the help of a managed environment that can update pointers when frames are moved. However we assume a native runtime environment in this paper.

use its stack space to execute a stolen task because the suspended frame may be resumed, allocating more frames on the stack before the stolen task completes. A naive approach to construct a cactus stack would be to allocate a new stack to execute the stolen task; however, this approach may lead to an impractically large number of stacks since each stack may contain as little as a single frame in the worst case. Cilk and Cilk++ allocate frames of Cilk functions in heap to avoid having a suspended frame tie up a stack. However, this requires Cilk functions to use a custom calling convention that is not compatible with the standard calling conventions. Therefore a Cilk program is not fully interoperable with serial code.

3. RELATED WORK

In this section, we describe various approaches to the cactus stack problem used by existing work-stealing runtimes. To summarize, these work-stealing runtimes address the cactus stack problem by either sacrificing one of the three criteria of a cactus stack or relying on custom operating system extensions. Therefore, we do not consider them as practical solutions to the cactus stack problem.

Cilk [8] and Cilk++ [12] use heap-allocated activation frames for Cilk functions. For normal C functions, Cilk stack allocates their activation frames. Cilk forbids a C function from calling a Cilk function to prevent stack frames from being suspended. This restriction helps Cilk to maintain a strong space bound, however sacrifices serial-parallel reciprocity.

Intel Cilk Plus [16] does not forbid a C function from calling a Cilk function. In Cilk Plus, a worker executes stolen frames on empty stacks acquired from a pool of stacks. The stack pool contains a bounded number of stacks. When the stack pool is empty, a thief has to refrain from stealing until some worker returns a stack to the pool. This implementation choice sacrifices the theoretical time bound of randomized work stealing to provide a tight space bound. Lee et al. [10] state that the theoretical space bound of using a new stack for each stolen frame is DPS_1, where D is the depth of a Cilk application. Therefore, to avoid staling due to a shortage of stacks, executing an application in Cilk Plus on a large number of workers may require an impractically large number of stacks.

A different approach is to allocate activation frames, both called or forked, on a linear stack. In particular, if a worker has a suspended frame on top of its stack, it steals a frame and executes it using its remaining stack space. In this case, the scheduler cannot resume the suspended frame until the stolen frame completes since the suspended frame may allocate new frames on the stack when resumed. This restriction violates the busy-leaves property because a suspended frame that is ready to resume is an extant leaf of the invocation tree and no worker is working on it. Violating the busy-leaves property can cause each worker's stack to grow impractically large. To prevent a worker's stack from growing beyond S_1, one can restrict a worker to only steal frames with a depth greater than the depth of the current suspended frame in the invocation tree. This form of stealing, called *depth-restricted stealing*, sacrifices the time bound. Sukha [17] has shown that there exist computations for which a depth-restricted stealing scheduler cannot execute faster than a sequential execution. TBB [15] uses a strategy that is similar to depth-restricted stealing.

A technique called *leapfrogging* can be applied to conserve stack space when implementing a cactus stack. Wagner and Calder proposed the leapfrogging technique to implement *futures* [18]. The logic behind leapfrogging is that if a worker has a suspended frame on its stack, it can steal a frame that is a descendant of the suspended frame and execute it using its remaining stack space. It is safe to reuse the stack in this way because the suspended frame will not resume until all its descendants complete, including the stolen frame. Because leapfrogging restricts a thief to steal only descendant tasks of the suspended frames on its stack, it is more restrictive than depth-restricted stealing. Therefore, Sukha's lower bound [17] applies to leapfrogging as well. Agrawal et al. [1] endorsed this idea for Cilk, but they did not show an empirical evaluation of the approach.

Lee et al. [10] addressed the cactus stack problem by modifying the operating system kernel to provide support for thread-local memory mapping (TLMM). TTLM designates a region of a process's virtual address space as "local" to each thread, i.e. each worker may map different memory pages to the TLMM region. Lee et al. created a work-stealing runtime system, Cilk-M, by using TLMM to implement the cactus stack in Cilk's runtime system. In Cilk-M, each worker executes on a stack that resides in the TLMM region. Whenever a thief steals a frame, it moves the memory pages that contain the stolen frame and the ancestor frames — the stack prefix — to its stack. Because stacks reside in the TLMM region, a thief can place the stolen stack prefix at the same virtual address as it was allocated. Therefore, pointers to local variables in the stack prefix are still valid. There are two major drawbacks of the TLMM approach. First, TLMM is currently not supported by any existing operating system. Second, since a worker's stack resides in the TLMM region, variables allocated on a worker's stack are not addressable by other workers. This restriction means shared data structures that allocate storage on the stack, such as MCS locks [14], do not work in Cilk-M.

4. FIBRIL

We present *Fibril*, a lightweight multithreading library to support a fork-join programming model. The design of Fibril closely follows that of Cilk. Fibril's randomized work-stealing scheduler behaves similarly to Cilk's scheduler as described in Section 2. Nevertheless, there are two major differences between Cilk and Fibril. First, Fibril is a pure C library that does not require a custom compiler front-end. Instead, Fibril exposes its interface as C macros, leveraging the C preprocessor to generate parallel code for a Fibril program. Second, unlike Cilk, which forbids a C function from calling a Cilk function, Fibril functions are fully interoperable with C functions. Fibril realizes our solution to the cactus stack problem by maintaining a strong time bound, a strong space bound, and serial-parallel reciprocity.

Section 4.1 introduces Fibril's API. Section 4.2 describes how Fibril exploits the standard x86-64 calling conventions to construct a cactus stack for parallel execution. Section 4.3 presents the implementation of Fibril's work-stealing scheduler. Section 4.4 analyzes the theoretical time and space bounds of Fibril's scheduler.

4.1 Fibril's API

We use an example to show how to use Fibril to parallelize a serial program using a fork-join programming model.

```
1  int fib(int n) {
2    if (n < 2) return n;
3    int x, y;
4    x = fib(n - 1);
5    y = fib(n - 2);
6    return x + y;
7  }
8  fibril int parfib(int n) {
9    if (n < 2) return n;
10   fibril_t fr;
11   fibril_init(&fr);
12   int x, y;
13   fibril_fork(&fr, x, parfib, (n - 1));
14   y = parfib(n - 2);
15   fibril_join(&fr);
16   return x + y;
17 }
```

Listing 1: A function `fib` and its parallel version `parfib` implemented in Fibril. Fibril macros are shown in *italic* font.

Listing 1 shows a serial function, `fib`, which computes a Fibonacci number recursively, and a Fibril function, `parfib`, which is a parallel implementation of `fib` using Fibril.

Fibril requires that one label a *Fibril function* — a functions that forks or joins tasks — with the `fibril` keyword (line 8). The `fibril` keyword instructs compilers to generate code for the function that conforms to a calling convention that permits parallel execution (described in Section 4.2). Unlike Cilk, which restricts that a Cilk function can only be spawned, one can either fork a Fibril function (line 13) or call the function directly (line 14).

Before forking or joining a Fibril function, one should declare (line 10) and initialize (line 11) a variable of type `fibril_t`, namely a *Fibril frame*. A Fibril frame synchronizes child tasks and stores the execution state of their parent task. To fork a function, one invokes the `fibril_fork` macro with (1) an initialized Fibril frame, (2) a variable to receive the return value, unless the function returns `void`, (3) the name of the function to fork, and (4) the arguments to the function wrapped within parenthesis (line 13). Logically, `fibril_fork` creates a child task that executes in parallel with its parent. One can use the same Fibril frame to fork multiple functions. To join the forked functions on a Fibril frame, one invokes the `fibril_join` macro with the Fibril frame as an argument (line 15). Logically, `fibril_join` blocks the execution of its caller until all functions forked with the Fibril frame as their parent complete.

Like Cilk, the design of Fibril's API follows the *C elision* rule proposed by Frigo et al. [8]. If we expand `fibril_init` and `fibril_join` to empty statements, and `fibril_fork` to a function call, we will have a syntactically and semantically correct serial program.

Unlike other library based multithreading systems, such as Posix threads, Fibril's API does not require Fibril functions to use a uniform function signature. Using a uniform function signature forces the parent function to pass arguments to a forked function via memory, which is more costly than passing arguments via registers. Therefore, Fibril's API is not only flexible to use, but also permits a low overhead implementation of `fibril_fork`, which can be used to create fine-grained parallelism.

Position	Contents	Frame
`8n+16(%rbp)`	B's memory argument b_n	linkage region
	...	
`16(%rbp)`	B's memory argument b_0	
`8(%rbp)`	B's return address	B's frame
`0(%rbp)`	A's `%rbp` value	
`-8(%rbp)`	B's local variables	
	...	
`0(%rsp)`	(unspecified variable size)	
`-8(%rsp)`	C's memory argument c_n	linkage region
	...	
`-8n-8(%rsp)`	C's memory argument c_0	

Figure 1: Stack organization on x86-64 systems. `%rbp`: frame pointer register; `%rsp`: stack pointer register. High address is at top of the figure.

4.2 Fibril's calling conventions

A calling convention defines how a function receives arguments from its caller, how it returns a result, and how a function's state is preserved across function calls. Different compilation units are interoperable only when they conform to the same calling convention. An architecture's Application Binary Interface (ABI) defines the calling conventions that a binary compiled for the architecture should use. We designed Fibril to work on the x86-64, aka AMD64, architecture. In this section, we describe x86-64 architecture's calling conventions [13]. We then explain the implementation of Fibril's cactus stack, which supports x86-64 calling conventions to enable parallel execution of a Fibril function. In the end, we describe Fibril's context switch scheme which leverages x86-64 calling conventions to reduce overhead. Fibril can be ported to non x86-64 systems with minor changes.

4.2.1 x86-64's linear stack and Fibril's cactus stack

Figure 1 shows a snapshot of the stack organization on x86-64 systems. The snapshot is taken when a thread is executing function B. B was called by function A, and is calling function C. A thread's run-time stack contains the stack frames of live function instances. A function's stack frame contains its return address, the value of its parent's frame pointer, and storage for its local variables. Typically, when a function makes a function call, it passes the callee's arguments in registers. For arguments that cannot be passed in a register, it allocates a *linkage region* between the caller and the callee to pass these arguments. When a thread executes a function, it accesses its stack using two registers: `%rbp` and `%rsp`. In Figure 1, the thread uses `%rbp`, aka the frame pointer, with positive offsets to access memory arguments and the stack frame of the current function B. When B calls C, the thread decrements `%rsp`, aka the stack pointer, pushing memory arguments to be passed to its callee C onto the stack, then decrements `%rsp` again to allocate C's stack frame.

Using both a frame pointer and a stack pointer allows a stack frame to be of an unspecified variable size. Using two pointers to manipulate the stack frames enables us to use this calling convention to construct a cactus stack. In Fib-

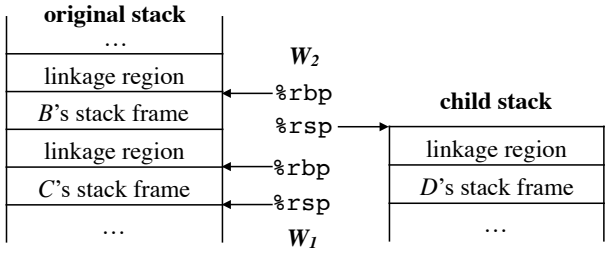

Figure 2: Fibril's cactus stack organization.

ril, when a thread W_1 forks a function C and a thief W_2 steals C's parent B, W_2 resumes execution of B on B's original stack using a new stack pointer (`%rsp`) that points to the top of a new stack. Figure 2 illustrates this situation. In this case, W_2 can access B's stack frame and memory arguments via using its frame pointer (`%rbp`). When W_2 calls or forks another child function C, C's memory arguments and stack frame will be allocated on the new stack, avoiding a collision with C's frame on the original stack. In this way, we construct a cactus stack that allows multiple children of a function to exist simultaneously. When W_2 returns from B, we restore its stack pointer to its original value pointing to the original stack.

When the size of a function's frame is known at compile time, the x86-64 ABI allows the function to avoid using `%rbp` as a frame pointer for the fixed-size frame. This optimization saves the instructions to push and pop `%rbp` in a function's prologue and epilogue; it also allows `%rbp` to be used as a general-purpose register. Nevertheless, omitting a function's frame pointer disallows it to be executed on a non-linear stack. Therefore, Fibril assumes a Fibril function always retains a frame pointer. Some compilers, such as GCC, provide a function attribute that instructs the compiler to use a frame pointer when generating code for a function. Fibril expands the `fibril` macro into such an attribute. For compilers that do not provide this feature, one can define a dummy variable that is a variable-length array in the Fibril function to achieve the same effect.

4.2.2 Enabling context switch in Fibril

The x86-64 ABI classifies all registers in an x86-64 system into two categories: a *caller-saved* register, whose value must be preserved by the caller, and a *callee-saved* register, whose value must be preserved by the callee. When a function uses a callee-saved register, it must save the register's value in its stack frame in its prologue and restore the value in its epilogue. There are six callee-saved registers on x86-64: `%rbp`, `%rbx`, and `%r12` through `%r15`; all other registers are caller-saved registers.

Fibril exploits these rules to implement a low-overhead context switch scheme to save and resume a Fibril frame's execution state. Listing 2 shows the code for the `parfib` function in Listing 1 with Fibril's macros expanded. In Fibril, a context switch may only happen at a `fibril_fork` or a `fibril_join`. Specifically, if a worker W_1 forks B and W_2 steals the parent A, it resumes A's execution starting from the instruction immediately after the `fibril_fork`; if a worker resumes A after A's children complete, it executes A starting from the instruction immediately after the `fibril_join` that corresponds to the fork.

```
18 typedef struct {
19   int lock;
20   int count;
21   struct { void * rbp, rsp, rip; } state;
22   void * stack;
23 } fibril_t;

24 int parfib(int n) {
25   if (n < 2) return n;

26   fibril_t fr;
27   { // fibril_init(&fr);
28     fr.lock = 0;
29     fr.count = 0;
30   }

31   int x, y;
32   { // fibril_fork(&fr, x, fib, (n - 1));
33     fr.state.rbp = get_frame_pointer();
34     fr.state.rsp = get_stack_pointer();
35     void fork_parfib(int a1, int * ret, fibril_t * f) {
36       f->state.rip = get_return_address();
37       _fibril_push(f);
38       *ret = parfib(a1);
39       if (!_fibril_pop()) _fibril_resume(f);
40     }
41     fork_parfib(n - 1, &x, &fr); clobber_regs();
42   }
43   y = fib(n - 2);

44   if (fr.count > 0) { // fibril_join(&fr);
45     fr.state.rbp = get_frame_pointer();
46     fr.state.rsp = get_stack_pointer();
47     _fibril_join(&fr); clobber_regs();
48   }
49   return x + y;
50 }

51 void _fibril_join(fibril_t * f) {
52   f->state.rip = get_return_address();
53   _fibril_resume(f);
54 }
```

Listing 2: `parfib` with Fibril macros expanded. Names with `_fibril_` prefix are Fibril's runtime routines. Functions shown in *italic* font are implemented using compiler extensions.

To preserve a frame's execution context at a fork or a join, we need to save the values of all registers. For caller-saved registers, since the calling convention requires the compiler to save them before a function call, we rely on the compiler to save them by implementing both `fibril_fork` and `fibril_join` as function calls. In particular, Fibril expands `fibril_fork` into a call to a task-specific nested function (line 41) and expands a `fibril_join` into a call to a Fibril runtime routine `_fibril_join` inside an `if`-block (line 47). GCC implements nested functions using several strategies: (1) when a nested function can be inlined, GCC inlines the function; (2) if the nested function cannot be inlined and it references a variable of its enclosing function, GCC generates a trampoline for the nested function and the nested function uses a general purpose register to reference its enclosing function's variables; (3) if the nested function cannot be inlined and does not reference variables in its enclosing function, GCC treats the nested function as a normal function. Fibril uses a `noinline` attribute to mark the nested function to prevent the compiler from inlining it, therefore GCC will generate code that follows the standard calling conventions for our task-specific nested function, which saves the caller-saved registers before a fork.

For callee-saved registers, we add a special instruction `clobber_regs` to accompany the calls to `fibril_fork` and `fibril_join`. `clobber_regs` is an inline assembly instruction that tells the compiler that the prior call may modify callee-saved registers, causing the compiler to generate code that saves these registers before the call.

Fibril only explicitly saves a frame's frame pointer, `%rbp`, its stack pointer, `%rsp`, and its return address, `%rip`. These values are accessed using GNU builtin functions. Since Fibril does not use a custom compiler frontend, it requires these compiler extensions to generate nested functions, clobber registers, and read the value of the frame pointer, stack pointer, and return address of a frame. However, these extensions are fairly standard; both the GNU C Compiler and the Intel C Compiler support these features.

4.3 Fibril's work-stealing scheduler

We divide the address space of a Fibril program into two spaces: the *application space* and the *scheduler space*. A Fibril program executes its *application code* — user-defined functions and code generated by Fibril's macros — in the application space. For routines of Fibril's work-stealing scheduler, shown in Listing 3, Fibril executes them in the scheduler space. In particular, each worker uses an alternative stack to execute the work-stealing scheduler to avoid collision with threads executing the application code. The `_fibril_resume` function, called when a `pop` fails (line 39) or at a `_fibril_join` (line 47), is the entry point to the scheduler space; it switches a worker's context to the scheduler space and invokes `schedule`. We omit the code for `_fibril_resume` from this paper.

In Fibril, each worker is uniquely identified using a thread-local variable `_wid`. Fibril represents a worker's state using a pair (`stack`, `deque`). A worker's `stack` is a pointer to the stack the worker should use to execute application code. A worker uses its `deque` to `push` and `pop` tasks. We omit the implementation of Fibril's deque, including `push`, `pop`, and `steal`, since it is the same as Cilk's deque [8], which employs Dijkstra's protocol for mutual exclusion [7].

A Fibril frame (defined in line 18) contains a `count` that records the number of pending child tasks of the frame. A frame's `count` is zero initially. When a frame is stolen for the first time, the thief increments the frame's count twice (line 88 and 89) to account for both the parent frame and the child task that was forked earlier.

When a worker completes a child task and attempts to resume its parent by calling `_fibril_resume`, it enters the scheduler space and calls `schedule`. `schedule` first locks the frame the worker is attempting to resume, then decrements the frame's `count`.

If the frame still has pending children, i.e., `count` is larger than zero, the scheduler has to perform randomized stealing to find a task for the worker to execute. Before the worker starts stealing, however, it should check whether the suspended frame resides on top of its current stack (line 62). If the frame is on top of the worker's stack, the worker `unmap`s the unused portion of its current stack and clears the stack from its worker state (line 64). The unused space starts at the stack's address and ends at the page boundary that is just above the top of the stack. It is safe to unmap unused memory pages of the stack because no worker will access the unused space until one resumes the suspended frame. When

```
55  typedef struct { deque_t * deque; void * stack; } worker_t;
56  __thread int _wid;
57  worker_t * _workers;

58  void schedule(fibril_t * f) {
59    worker_t * w = _workers[_wid];
60    lock(f);
61    if (--f->count > 0) {
62      if (f->stack == w->stack) {
63        unmap(f->stack, PAGE_ALIGN(f->state.rsp));
64        w->stack = NULL;
65      }
66      unlock(f);
67      random_steal(w);
68    } else {
69      unlock(f);
70      if (f->stack != w->stack) {
71        put_stack_into_pool(w, w->stack);
72        remap(f->stack, PAGE_ALIGN(f->state.rsp));
73        w->stack = f->stack;
74      }
75      execute(f, f->state.rsp);
76    }
77  }

78  void random_steal(worker_t * w) {
79    worker_t * victim; fibril_t * f;
80    while (1) {
81      victim = _workers[rand() % NPROCS];
82      lock(v->deque);
83      f = steal(victim->deque);
84      if (f == NULL) unlock(victim->deque);
85      else break;
86    }
87    lock(f); unlock(victim->deque);
88    if (f->count++ == 0) {
89      f->count += 1;
90      f->stack = victim->stack;
91    }
92    unlock(f);
93    if (!w->stack) w->stack = take_stack_from_pool(w);
94    execute(f, w->stack + STACK_SIZE);
95  }

96  void execute(fibril_t * f, void * rsp) {
97    set_stack_pointer(rsp);
98    set_frame_pointer(f->state.rbp);
99    set_return_address(f->state.rip);
100 }
```

Listing 3: The implementation of Fibril's scheduler.

a worker steals a frame, it will take a stack from a stack pool to execute the stolen frame (line 93).

If all children of the frame that the scheduler is trying to resume have completed, i.e., `count` becomes zero, the worker can resume the frame. If the stack the frame resides on is not the stack the worker is using (line 70), the worker puts its current stack into the stack pool and *remap*s the memory pages of the suspended stack. Then it uses the frame's stack to execute the frame.

Eventually, the scheduler will call `execute` on a frame to transfer the control back to the application space and execute the frame. Note that if the frame is a stolen frame, the worker will execute the frame using a stack pointer that points to the top of a new stack (line 94).

Implementation of unmap/remap.

There are two ways to implement the `unmap`/`remap` routine in Listing 3. Many operating systems provides an interface to change the memory mapping of a virtual address range. For example, Linux provides an `mmap` system call that changes the memory mapping of a virtual address

range. To `unmap`, one can call `mmap` to change the mappings of the unused address space to an empty dummy file. Since the dummy file will not be accessed, these mappings will not consuem physical memory or incur I/O activity. Note that we cannot implement `unmap` by calling `munmap` to free the unused memory pages because the operating system (OS) may give the unused virtual address space back to the memory allocator. Changing the mappings of the unused stack space to a dummy file preserves the address space but frees the physical memory pages. In `remap`, we can call `mmap` again to obtain anonymous pages from the OS for the unused portion of the stack.

This approach assumes that a multithreaded process's underlying virtual memory subsystem allows operations that change non-overlapping memory regions of a shared address space to execute concurrently. Unfortunately, most operating systems today serialize these operations. Specifically, widely used OS kernels, such as Linux and FreeBSD, use a single lock per shared address space. System calls that change the shared address space, such as `mmap`, need to acquire the lock before applying changes.[4] Therefore, `mmap` will become a bottleneck for Fibril, since multiple workers may call `unmap`/`remap` at the same time on different stacks.

On Unix-like systems, one can implement `unmap` using the `madvise` system call with the `MADV_DONTNEED` flag. `madvise(..., MADV_DONTNEED)` tells the OS that the memory pages of a specified address range are no longer needed and the OS may choose to free the pages. This approach gives the OS the flexibility to free the memory pages lazily, which may yield better performance if the OS has memory to spare and does not need to reclaim the memory pages.[5] More importantly, calling `madvise` with `MADV_DONTNEED` does not acquire the address space lock to free the memory pages, which permits concurrent `unmap` calls. When `unmap` is implemented using `madvise`, `remap` becomes an no-op. Accessing the unmapped address range will trigger a page fault then the OS will allocate a memory page for the accessed address.

4.4 Theoretical bounds

Because Fibril changes the memory mapping to conserve memory footprint, there is a gap between a Fibril program's physical memory and virtual address usage. We analyze bounds on the physical and virtual space separately. We measure space in memory pages, the size of which we leave unspecified. We define the **stack depth** of a Fibril program to be the number of memory pages on the stack in a serial execution of the program. We define the **Fibril depth** of a Fibril program to be the maximum number of Fibril frames on any path from the root to a leaf of the program's invocation tree. We show the following theorems regarding the space and time requirements of a Fibril program.

THEOREM 4.1. *For a Fibril program with stack depth S_1 and Fibril depth D, the Fibril scheduler can execute the program on P processors using DPS_1 pages of virtual address space.*

PROOF. Fibril maintains the busy-leaves property. This implies that, at any given time, the number of leaves in the invocation tree is bounded by P. The stack frames along a path from the root of the invocation tree to a leaf may reside on different stacks. Since a path changes to a different stack only when resuming a stolen Fibril frame, each path may span at most D stacks. Moreover, because each of these stacks may grow to be as large as S_1, each path may use DS_1 virtual address space. Therefore, the bound on the virtual address space is DPS_1. □

Although the bound on virtual address space can be very large for an execution with a deep Fibril depth, it is practical on 64-bit systems because the virtual address space is significantly larger than any realistic value of DPS_1.

THEOREM 4.2. *For a Fibril program with stack depth S_1 and Fibril depth D, the Fibril scheduler can execute the program on P processors using $P(S_1 + D)$ pages of physical memory.*

PROOF. Because the Fibril scheduler maintains the busy-leaves property, the number of leaves in the program's invocation tree is bounded by P. For a path from the root of the invocation tree to a leaf, the stack frames on the path may span multiple stacks. Since a path can only change to a different stack at a Fibril frame, each path may span at most D stacks. Because Fibril unmaps the unused pages of a stack, each dormant stack only contains memory pages that hold stack frames. Because stack frames may not align with page boundaries, there may be a partially unused page at the top of each stack. Since there are at most D stacks, there can be at most D such partially unused pages on a path. Since the stack frames on each path may consume up to S_1 pages, the total number of pages a path may use is $S_1 + D$. Therefore, the program uses at most $P(S_1 + D)$ memory pages. □

Fibril's physical memory bound is a strong bound. Comparing with Blumofe and Leiserson's space bound, it only adds a constant overhead D per processor, staying well below the border of practicality. Fibril's physical memory bound is also a loose bound since it counts every non-leaf frame in the invocation tree more than once. Moreover, we should not expect that every Fibril frame will be stolen. For a program with sufficient parallelism, the number of steals is usually much less than the number of forks.

THEOREM 4.3. *For a Fibril program with work T_1 and span T_∞, the Fibril scheduler can execute the program on P processors in expected time $T_p \leq T_1/P + c_\infty T_\infty$, where $c_\infty = O(S_1 + D)$.*

The proof of Fibril's time bound needs complex theoretical analysis that is beyond the scope of this paper. Fibril's time bound trades the constant span overhead c_∞ on the span term of Cilk's time bound for an overhead that is linear in the program's depth $S_1 + D$. This increased overhead reflects the cost of unmapping and remapping a stack's unused memory pages. We consider the cost of changing the mappings of an address space to be linear in the number of mappings it changes. In the worse case, every steal might

[4] In principle, if two `mmap`s operate on non-overlapping memory regions, these operations should be perfectly parallelizable. Clements et al. applied this principle in the design of a scalable address space for multithreaded applications, called *RadixVM* [6].

[5] Currently, the Linux implementation of `madvise(..., MADV_DONTNEED)` always frees the memory pages. But this may change in the future.

Table 1: The description of the 12 benchmarks.

Benchmark	Input	Description
cholesky	4000/40000*	Cholesky decomposition
fft	2^{26}	Fast Fourier transformation
fib	42	Recursive Fibonacci
heat	2048×500	Jacobi heat diffusion
integrate	$10^4(\epsilon = 10^{-9})$	Quadrature adaptive integration
knapsack	32	Recursive knapsack
lu	4096	LU decomposition
matmul	2048	Matrix multiply
nqueens	14	Count ways to place N queens
quicksort	10^8	Parallel quicksort
rectmul	4096	Rectangular matrix multiply
strassen	4096	Strassen matrix multiply

*: 40000 is the number of non-zero values in the matrix.

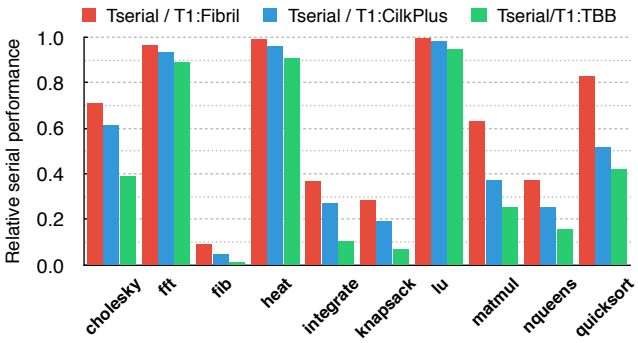

Figure 3: The relative performance on one thread.

cause the scheduler to change the mappings of a worst-case stack of depth S_1+D. One may prove Fibril's time bound by applying the proof techniques of Blumofe and Leiserson [4] or those of Arora et al. [2].

Like the space bound, Fibril's time bound is also a loose bound because not every steal will cause an `unmap` action. Fibril does not unmap a stack if the worker that executes on the stack completes later than the thief who steals the frame on top of the stack. The bound is also a strong bound; it guarantees that a program with sufficient parallelism achieves near-perfect linear speedup. We consider a program has sufficient parallelism if $T_1/T_\infty \gg O(S_1 + D)P$.

5. EVALUATION

To evaluate Fibril's performance, we implemented Fibril on an x86-64 architecture. There are three concerns that we investigate in our empirical study:

1. how Fibril's performance compares with other work-stealing runtime systems,
2. whether Fibril's `unmap/remap` routine incurs significant overhead, and
3. how effective is Fibril's `unmap/remap` in reducing the memory footprint of a parallel program.

To address these concerns, we compare Fibril with Intel Cilk Plus [16] and Thread Building Blocks (TBB) [15]. Because Cilk Plus's work-stealing scheduler is similar to that of Fibril, it serves as a fair point of comparison. Intel TBB is similar to Fibril because it is also a library-based implementation of a work-stealing scheduler. We evaluate Fibril using 12 benchmarks. Table 1 briefly describes each benchmark and lists the input size we used for each of them in our experiments. These benchmarks are a representative subset of the benchmarks used in previous studies of work-stealing. We evaluate two variants of Fibril: **Fibril** and **Fibril (w/o unmap)**, where Fibril implements `unmap` using `madvise(..., MADV_DONTNEED)` and Fibril (w/o unmap)'s `unmap` is an no-op. In both versions, `remap` is an no-op.

General Setup.

We ran all experiments on an Intel Haswell system with two 18-core 2.3 GHz CPUs with 2-way simultaneous multithreading on each core, with a total of 72 hardware threads. The total memory available on the system is 128 GBytes. The operating system ran on the machine is Red Hat Enterprise Linux Server 7.0 (Maipo) with kernel version 3.16.7. All code, both Fibril and the benchmarks, are compiled with GCC 5.2.0. Since GCC 5.2.0 comes with native support for Cilk Plus, we also used it to compile the Cilk Plus versions of the benchmarks. We used Intel TBB 4.4 Update 4 in our experiments. We compiled all code with `-O2` optimization. We performed all experiments ten times and recorded the mean execution time and the maximum stack usage of all ten runs. The standard deviation of our results is negligible. All experiments use 4KB memory pages and 1M stacks. Cilk Plus by default limits the maximum number of stacks its workers can use to 2400. We do not pin threads to physical cores to allow the OS to migrate workers among cores when it sees an oppertunity.

Serial Performance.

Figure 3 shows the relative performance of Fibril, Cilk Plus, and TBB versions of the benchmark on a single thread. We normalize each performance number ($1/T_1$:Fibril, $1/T_1$:CilkPlus, and $1/T_1$:TBB) with the performance of the serial version of the benchmark ($1/T_{serial}$). A number below one indicates that the benchmark's parallel version is slower than its serial version. For instance, the performance of `matmul` using Fibril on one thread is approximately 40% slower than the performance of the serial `matmul`. For all benchmarks we tested, Fibril outperforms both Intel Cilk Plus and TBB. In particular, The Fibril versions of `fib`, `integrate`, `knapsack`, `matmul`, `nqueens`, and `quicksort` outperform those of Cilk Plus by **1.9**×, **1.3**×, **1.5**×, **1.7**×, **1.4**×, and **1.6**×, respectively, and those of TBB by **6.0**×, **3.6**×, **4.2**×, **2.5**×, **2.3**×, and **2.0**×, respectively. These results indicate that Fibril's calling conventions have lower overhead than those of Cilk Plus. Since `fft`, `heat`, `lu`, `rectmul`, and `strassen` perform a large amount of floating point computation within each task, the serial overhead of fork and join operations becomes insignificant. Therefore, the performance of different versions of these benchmarks is almost the same.

Parallel Performance.

Figure 4 shows the performance of the benchmarks on 1–72 threads. We assume the ideal speedup a benchmark achieves is $P\times$ on P threads. Fibril greatly outperforms both Cilk Plus and TBB for most of the benchmarks. In particular, the Fibril versions of `choleskey`, `fib`, `integrate`, `knapsack`, `matmul`, and `nqueens` outperform those of Cilk Plus by **1.4**×, **2.8**×, **1.8**×, **2.2**×, **2.3**×, and **2.1**×, respectively, and those of TBB by **3.0**×, **7.9**×, **4.5**×, **4.8**×, **4.3**×, and **2.8**×, respectively. For other

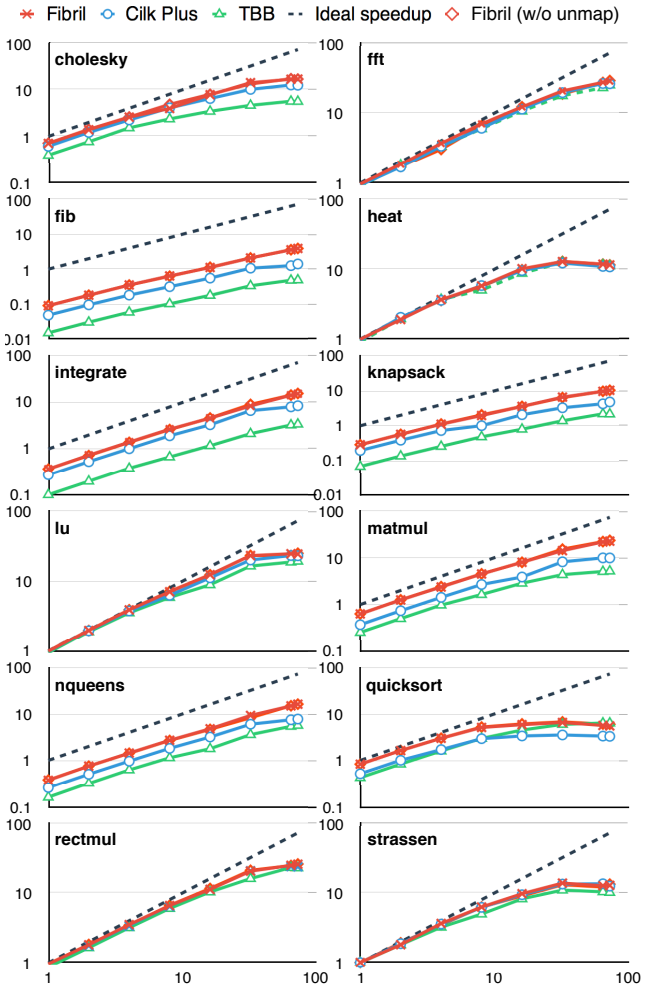

Figure 4: The speedup on 1–72 threads. The X axis are the number of threads, and Y axis values are T_{serial}/T_P, where T_P is the execution time on P threads.

benchmarks, their Fibril versions perform better or similar to their Cilk Plus and TBB versions. Since Cilk Plus's limitation on the total number of stacks a program can use is never reached in our experiments, Fibril's speedup over Cilk Plus is a result of Fibril's better serial performance. Figure 4 also shows that the performance difference between Fibril and Fibril (w/o unmap) is negligible. This indicates that our approach to bound the memory usage is of minimal overhead.

Table 2 lists the average number of key actions performed on 72 threads. Table 2 shows that not every successful steal in Fibril will cause an `unmap` operation. The percentage of steals that cause an `unmap` in Fibril varies from 33% to 92% among the 12 benchmarks. Table 2 also shows that Fibril's `unmap` results in an increase in the number of page faults. This is because a page that is returned to the OS may be accessed again when the suspended frame on the stack is resumed. However, the increase in the number of page faults does not incur significant overhead because the Fibril (w/o unmap) version performs the same as Fibril as shown in Figure 4.

Table 2: Profile of key operations on 72 threads.

	steals	unmaps	page faults		
	Fibril	Fibril	Fibril	Cilk Plus	TBB
cholesky	6568K	2795K	89K	31K	193K
fft	81K	63K	2392K	1475K	346K
fib	20K	12K	6K	3K	9K
heat	579K	452K	10K	10K	9K
integrate	22K	19K	5K	3K	5K
knapsack	13K	12K	1K	2K	1K
lu	2440K	1384K	41K	2K	5K
matmul	92K	62K	13K	10K	69K
nqueens	37K	12K	19K	3K	4K
quicksort	7263K	4005K	1299K	693K	66K
rectmul	43K	33K	3012K	2712K	6472K
strassen	25K	9K	9336K	9494K	23926K

Table 3: Stack space usage in Fibril.

Application	D	S_1	$S_1 + D$	$S_{72}/72$
cholesky	10	2	12	4.82
fft	19	5	24	13.90
fib	41	3	44	8.46
heat	12	1	13	4.49
integrate	32	3	35	8.76
knapsack	32	3	35	5.71
lu	8	2	10	4.92
matmul	10	2	12	4.90
nqueens	14	3	17	6.19
quicksort	69	4	73	9.89
rectmul	24	3	27	9.31
strassen	6	2	8	4.79

Stack Space Utilization.

Table 3 shows the stack footprint of the benchmarks using Fibril on 72 threads. For comparison, we also list the theoretical space bound per thread ($S_1 + D$) in the table. Overall, Fibril's actual stack usage is well below its theoretical bound. Specifically, all of the benchmarks use less than 60% of the stack space of their theoretical bound. This confirms that the space bound in Theorem 4.2 is loose.

Table 4 shows the resident set size (RSS) and stack usage of the benchmarks on 72 threads. The RSSes of Fibril benchmarks are similar to those of Cilk Plus. This may be because Fibril uses more stacks than Cilk Plus to execute these computations. Comparing with TBB, Fibril uses much less memory to execute the same computations.

6. CONCLUSIONS

It is very important for a work-stealing framework to implement a cactus stack that provides interoperability and strong bounds in both time and space. Sacrificing any one of the three criteria can compromise a work-stealing framework's usability or performance. The TLMM approach by Lee et al. [10] is not a viable solution for today's commodity operating systems because it requires custom OS support that is unavailable. Our cactus stack implementation is the first solution to the cactus stack problem that is practical and efficient for use in common systems. We have demonstrated its practicability and efficiency both theoretically and empirically in this paper. Our theoretical bounds ensure a Fibril program's memory usage stays within the range of practicability and guarantee near-linear speedup given sufficient parallelism. Our empirical evaluation shows that Fib-

Table 4: The RSS and stack usage on 72 threads.

	ΔRSS / Max RSS			# of stacks	
	Fibril	C.P.	TBB	Fibril	C.P.
cholesky	84/95	81/91	173/184	346	333
fft	1540/2055	1546/2060	1639/2154	334	321
fib	3/6	3/5	108/111	315	297
heat	36/65	31/31	139/175	323	314
integrate	3/6	3/5	97/100	326	310
knapsack	3/6	3/5	106/109	316	326
lu	3/134	3/133	92/224	356	348
matmul	11/54	14/57	123/167	356	348
nqueens	5/5	3/3	113/116	352	314
quicksort	385/769	391/775	883/883	340	323
rectmul	513/772	680/938	1026/1285	335	333
strassen	2434/2694	2762/3020	2603/2862	343	329

C.P.=Cilk Plus. The RSS are reported in MBs. Δ RSS=the incremental RSS during the computation; this excludes the physical memory used for input data and libraries.

ril's approach to bound memory usage is of minimal overhead and Fibril outperforms the-state-of-art work-stealing frameworks on most benchmarks.

7. ACKNOWLEDGMENTS

This work was supported in part by the DOE Office of Science's Advanced Scientific Computing Research Program through Cooperative Agreement number DE-SC0008883. Experiments were performed on a dual-socket Intel Haswell system on loan from Intel.

8. REFERENCES

[1] K. Agrawal, I.-T. A. Lee, and J. Sukha. Brief announcement: Serial-parallel reciprocity in dynamic multithreaded languages. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 186–188, New York, NY, USA, 2010. ACM.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

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

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

[5] 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 *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[6] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, New York, NY, USA, 2013. ACM.

[7] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, Sept. 1965.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[9] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[10] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 411–420, New York, NY, USA, 2010. ACM.

[11] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM.

[12] C. E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM.

[13] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. System V application binary interface AMD64 architecture processor supplement. http://www.x86-64.org/documentation/abi.pdf, October 2013.

[14] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991.

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

[16] A. D. Robison. Composable parallel patterns with Intel Cilk Plus. *Computing in Science and Engg.*, 15(2):66–71, Mar. 2013.

[17] J. Sukha. Brief announcement: A lower bound for depth-restricted work stealing. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 124–126, New York, NY, USA, 2009. ACM.

[18] D. B. Wagner and B. G. Calder. Leapfrogging: A portable technique for implementing efficient futures. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 208–217, New York, NY, USA, 1993. ACM.