Combining Task Parallelism and Multithreaded Concurrency

by

Sai Sameer Pusapaty

B.S. Computer Science and Engineering, Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2022

Author .....
Department of Electrical Engineering and Computer Science
January 24th, 2022

Certified by .....
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Combining Task Parallelism and Multithreaded Concurrency

by

Sai Sameer Pusapaty

Submitted to the Department of Electrical Engineering and Computer Science
on January 24th, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I present Multicilk, a threads library based on C11 threads and the OpenCilk runtime for enabling task parallelism within multiple concurrent threads. With Multicilk, a programmer can parallelize threads in a multithreaded application simply by using Cilk independently within each thread. Without Multicilk, doing so violates the semantics that we expect in concurrent thread programming, leading to catastrophic failure of the application. No other existing combination of task-parallel system — including OpenMP, TBB, and TPL, and the various other implementations of Cilk — and threads library — including C11, C++11, Pthreads, and WinAPI threads — can parallelize multithreaded applications transparently and modularly.

The key insight behind Multicilk recognizes that integrating task-parallel systems with multithreaded concurrency requires two layers of thread abstraction that are conflated in previous systems. ***Service*** threads implement the workers in the Cilk runtime system. But the Cilk computation itself, called a ***cilk***, though implemented by many service threads, itself provides the abstraction of a single ***application*** thread to other threads within the multithreaded application, regardless of whether they are cilks. Multicilk employs a technique called ***impersonation*** to individual workers to act on behalf of the entire cilk, providing the same interface to the outside world as it would it were an ordinary thread. This powerful "two-layer-cake" abstraction enables ordinary multithreaded applications to be ported to a Cilk environment and parallelized in a straightforward and modular fashion.

My Multicilk implementation for OpenCilk provides two thread libraries corresponding to the layers in the two-layer cake, as well as some modifications to the OpenCilk runtime system. The service-thread library is the standard glibc. The application-thread library was created by modifying eight glibc functions using 63 lines of source code and writing wrappers for four glibc functions using 115 lines of code, amounting to about half of the 25 functions in the C11 sublibrary of glibc. I wrote 180 lines of utility functions for threading and synchronization. The changes to OpenCilk amounted to 62 lines, also for threading and synchronization. In sum, I added or modified just over 400 lines of source code to implement Multicilk.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Acknowledgments

I would like to thank Charles E. Leiserson for taking me as an MEng and for his invaluable guidance and advice during his supervision of my research.

I would also like to extend my gratitude to Tao B. Schardl, Tim Kaler, and Alexandros-Stavros Iliopoulos for sharing their knowledge and providing constant support during our weekly meetings.

I want to thank Nikos P. Pitsianis, Xiaobai Sun, and Dimitris Floros for meeting with me to discuss potential use-cases for Multicilk. The discussions were helpful to determine possible ways programmers might try to use Multicilk and helped elucidate the conditions for when Multicilk might be appropriate to use.

I would like to thank my parents for all of their unconditional love and support.

This research was supported in part by NSF grant 1533644. This research was also sponsored in part by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Contents

Chapter 1

Introduction

My thesis aims to fuse a well-structured alliance between task parallelism [41] and multithreaded concurrency [8]. Task parallelism parallelizes code by distributing independent work across many processors to be executed in parallel. On the other hand, multithreaded concurrency employs the use of multiple threads to control the interaction between different units of a program running at the same time. Existing systems do not provide the abstraction of a task-parallel subsystem as a single thread. Consequently, parallelizing a thread that operates within a multithreaded setting is fraught with difficulties. My thesis offers a first step toward remedying this longstanding problem.

In my thesis I explore how to build a framework that enables task parallelism within individual threads of a multithreaded program. This introduction begins by first familiarizing the reader with the concepts of task parallelism and concurrency and introduces OpenCilk [34], a parallel programming framework that enables task parallelism in serial programs. After describing the challenges of composing task parallelism with multithreaded concurrency and demonstrating the issues with using platforms like OpenCilk directly in multithreaded code, I present Multicilk and briefly discuss its solution to the problem. Finally I conclude this chapter by discussing my contributions and the outline for the rest of the thesis.

Concurrency aims to organize the behavior and address conflicts [36] among many running units of a computing system. Threaded concurrency uses the shared-memory

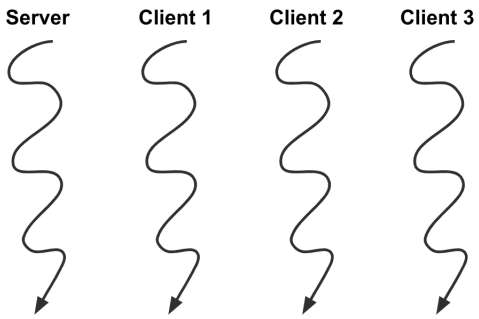

Figure 1-1: Concurrent threads of a network server.

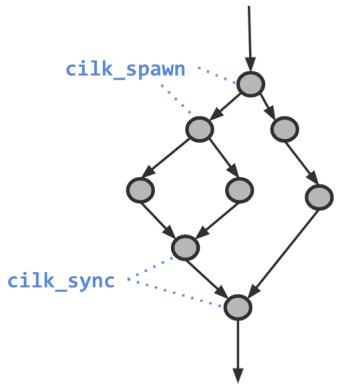

Figure 1-2: The trace of a serial program parallelized with Cilk.

abstraction of threads to manage different units of a system, and a concurrent program's logic describes the interaction between these threads. For example, consider a network server with multiple clients. An implementation using threaded concurrency might involve running the server logic and each client handler on individual threads as shown in Figure 1-1. Each client thread can communicate with the server thread using shared memory to send messages back and forth.

The prevalence of multicore architectures today allows programs to utilize parallelism easily. One form of parallelization is task parallelism which enables programs to execute logically independent segments of computational work in parallel by distributing each independent task to separate processors. Unlike concurrency, which aims to organize an interacting system of running units, task parallelism improves a program's algorithmic nature by taking advantage of a machine's multiple processors to run largely independent instances of work simultaneously.

```
// Returns the nth fibonacci number
int fib(int n) {
    if (n < 2) return n;
    // spawn off call to run in parallel
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);
    // wait for all cilk_spawns in frame to finish
    cilk_sync;
    return x + y;
}
```

Figure 1-3: Cilk program to find nth Fibonacci number

Programmers can exploit task parallelism using OpenCilk, a framework designed for parallel programming, as well as many other platforms such as OpenMP [10], TBB [37], and TPL [22]. Cilk [17], the programming language of OpenCilk, employs a fork-join parallel-programming model. It uses the keywords `cilk_spawn` to create tasks in parallel and `cilk_sync` to wait for incomplete tasks to finish. Figure 1-2 illustrates the trace of a program using `cilk_spawn`s and `cilk_sync`s. The `cilk_spawn`s split the program's control flow to run separate strands [9, Ch. 26], sequences of serial instructions with no procedural control, in parallel, and `cilk_sync`s act as synchronization points, waiting for outstanding strands to finish executing. The simple syntax enables programmers to parallelize their serial programs by identifying independent tasks without worrying about how the tasks are scheduled. Figure 1-3 shows an example of a simple function parallelized with Cilk, that finds the nth Fibonacci number. We refer to the basic framework of Cilk and its language constructs as the ***Pure Cilk Model***. Understanding this model is the extent necessary for most application programmers to use Cilk effectively.

Composing task parallelism with threaded concurrency is a seemingly simple idea. With the concise framework of OpenCilk, adding task parallelism to the serial logic of a thread using Cilk seems trivial on the surface. Simply use Cilk within the logic of a thread, treating the thread as a serial program. Thus the trace of a thread using Cilk would look akin to Figure 1-4. With the network server described earlier, we can use Cilk to parallelize the logic of the server and the client threads (assuming there

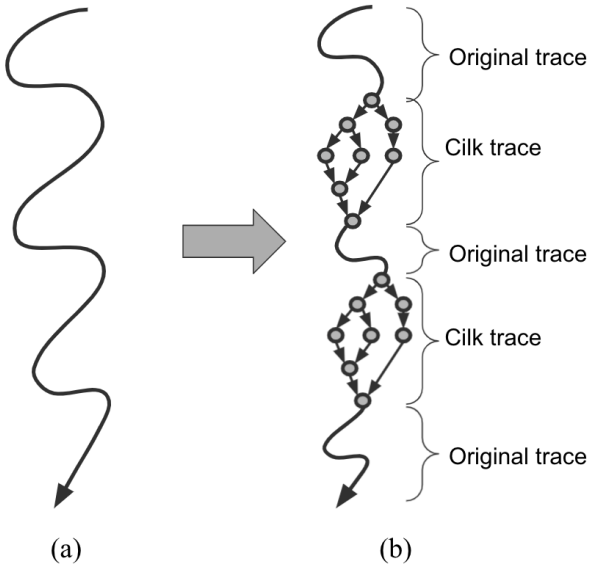

Figure 1-4: Pure Cilk view of a thread's logic parallelized with Cilk. (a) The original thread. (b) The thread parallelized with Cilk.

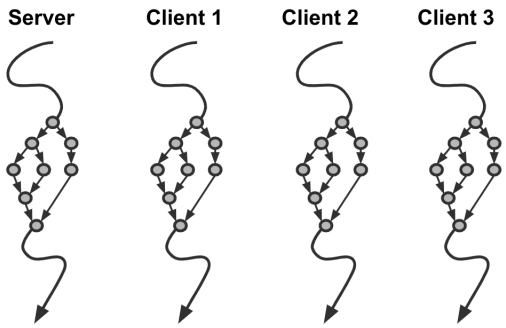

Figure 1-5: Concurrent threads of a network server that have all been parallelized with Cilk.

is task parallelism to exploit). With Cilk, the serial threads we have in Figure 1-1 become the ***cilks*** (threads that do Cilk computation) seen in Figure 1-5.

We can visualize the composition of task parallelism and threaded concurrency as a ***two-layer cake***, illustrated in Figure 1-6, where task parallelism is the bottom layer, and threaded concurrency is the top layer. By using Cilk to parallelize the serial logic of multiple concurrent threads, we see that the two-layer cake provides a clear distinction between programming done in the task-parallel layer versus the threaded concurrency layer. The parallelization of an individual thread's logic via Cilk

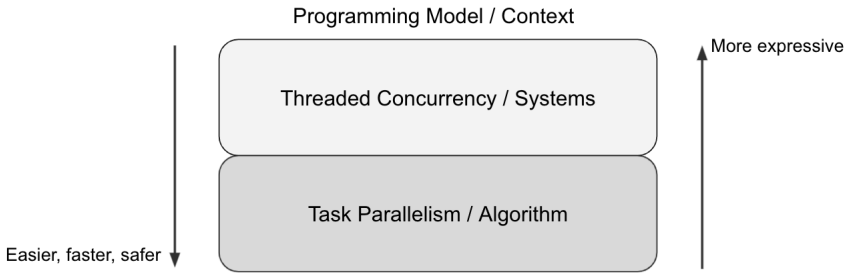

Figure 1-6: An illustration of the two-layer cake.

is always done in the task-parallel layer oblivious to the existence of other threads. The concurrent logic of threads in the threaded program remains in the top layer of the cake and is oblivious to which threads are cilks. This idea of a two-layer cake fits directly as the top two layers of Arch Robison's three-layer cake model for shared-memory programming [39]. Robison's three-layer cake describes its bottom layer as the SIMD programming model, the middle layer as the fork-join model, and the top layer as the message-passing model, another form of concurrency.

OpenCilk, like OpenMP, TBB, and TPL, utilizes a set of threads under the hood to execute parallel logic. OpenCilk implements a single runtime [1, 35] that creates and schedules a pool of threads called ***Cilk workers*** to execute work. The same set of Cilk workers executes any Cilk computation in the program. The ***Cilk Worker Model*** describes the scope of Cilk with the runtime and Cilk workers. Typically, the implementer of Cilk uses the Cilk Worker Model and although an API allows application programmers to access some aspects of the worker model, such as the number of workers, its use is discouraged. Figure 1-7 depicts the relationship between the Pure Cilk Model view and the Cilk Worker Model view of a serial program parallelized with Cilk. Cilk worker threads, seen in the Cilk Worker Model view, execute the strands of the trace in the Pure Cilk Model view.

Under the Pure Cilk Model, a programmer can view a cilk as a single thread that executes some of its logic in parallel, but the Cilk Worker Model gives an accurate view of what happens under the hood when Cilk computation runs as shown in Figure 1-8. The reality is that multiple Cilk worker threads and a boss thread execute the logic, not the cilk itself. In other words, a cilk simply takes on the appearance of a

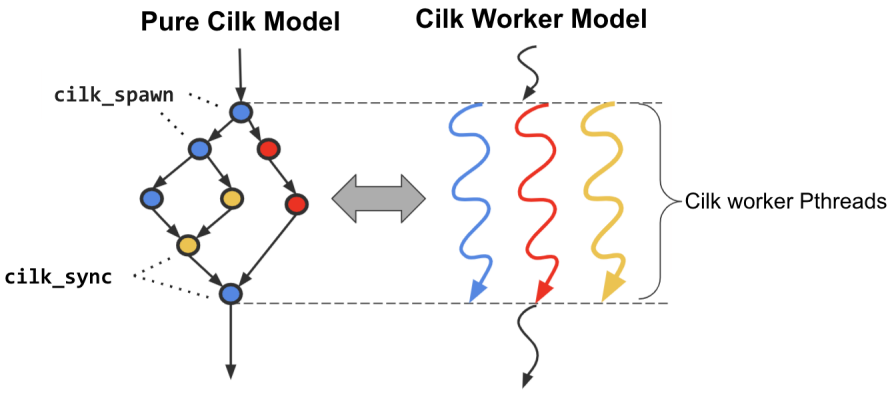

Figure 1-7: The relationship between the Pure Cilk Model view and Cilk Worker Model view of a Cilk program.

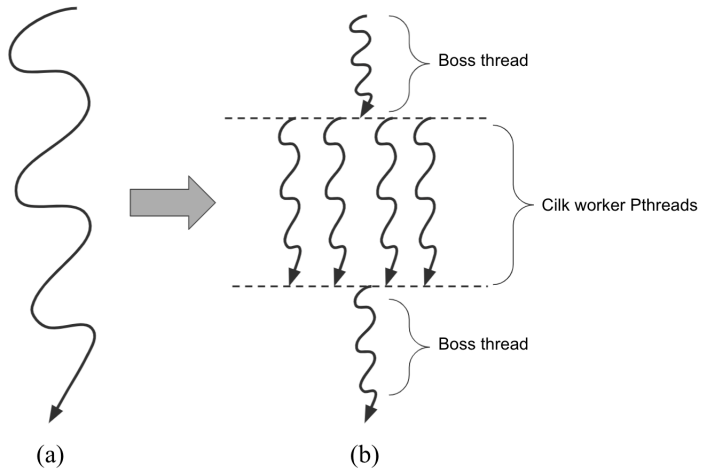

Figure 1-8: Cilk Worker view of a thread's logic parallelized with Cilk. (a) The cilk. (b) The worker threads and boss thread that make the cilk.

thread in the threaded program, but is an abstraction of the interaction of several underlying threads in the Cilk Worker Model.

Despite the apparent simplicity of using Cilk language constructs to parallelize the logic of threads in a concurrent environment, having Cilk workers execute a thread's logic naively is potentially catastrophic. If the thread's logic depends on the thread's identity, then the program uses the identity of the Cilk worker thread instead of the identity of the boss thread (which is used to define the identifier of the cilk) that was semantically intended. For example, consider Figure 1-9. The code first prints the thread's ID (via the call to `thrd_current()` which returns the ID of the currently

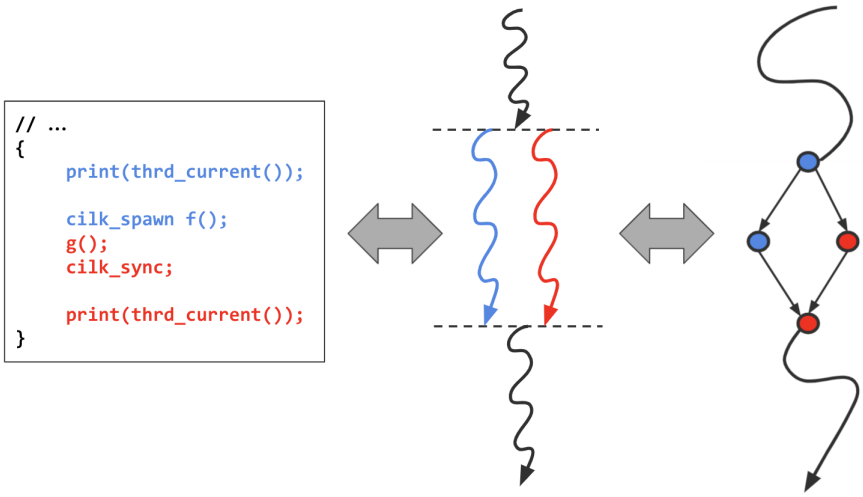

Figure 1-9: Cilk code being executed in parallel by two different Cilk worker Pthreads.

executing thread), executes some code in parallel with Cilk, and prints the thread's ID again. By the Pure Cilk Model, we should expect identical printed IDs and for the ID to belong to the boss thread. Instead, one Cilk worker prints the first ID, however, and a different worker prints the second ID. Not only do the printed ids fail to be equivalent, since the call to `thrd_current()` is executed by different Pthreads, neither matches the desired ID of the boss thread. Such correctness issues also extend beyond typical threading-library functions to synchronization mechanisms such as acquiring mutexes [28] or using thread-local storage [12], both of which depend on the identity of the calling thread.

To understand Cilk within threaded programs let us distinguish two classes of threads using Figure 1-10. The ***application threads*** are the threads used for threaded concurrency. These threads include cilks, which to other application threads in the program, appear as standard threads. From the perspective of the Pure Cilk Model, application threads are the only threads in threaded program and lie strictly in the top layer of the two-layer cake model. ***Service threads*** interface with the OS and implement the application threads above. Non-cilk threads are both application threads and service threads as they are always visible to the programmer but also interface directly with the OS. Cilks on the hand are an abstraction of multiple service threads interacting, but never touch the OS themselves. Rather, a cilk's respective

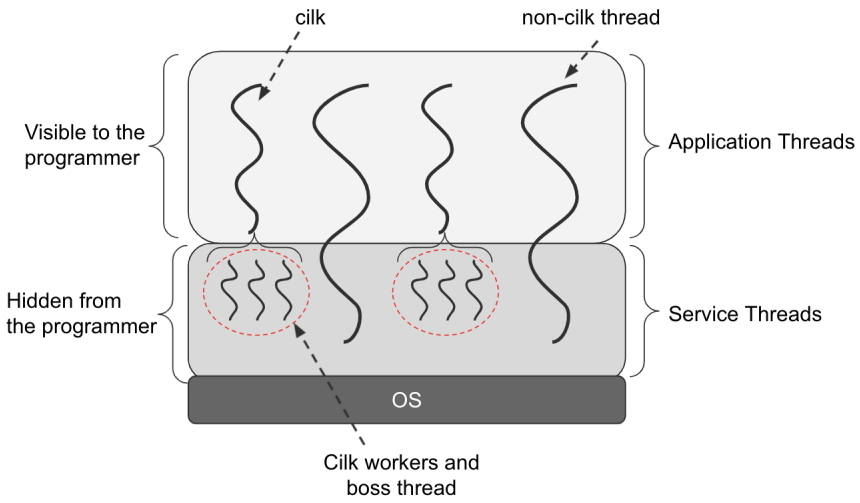

Figure 1-10: The distinction between application threads and service threads.

service threads, consisting of Cilk workers and a boss thread, are hidden away from the programmer and contribute to the work done in the task-parallel layer of the two-layer cake via Cilk computation.

The fundamental issue with current threading libraries is that they do not make this distinction between these two types of threads. How should a function such as `thrd_current()` behave if the threads of an application are parallelized? Existing task-parallel systems always return the ID of the service thread, but this violates the assumption made by other threads in the system that the parallelized application thread is indeed a single thread. This inherent limitation makes composing task parallelism and threaded concurrency difficult.

When using Cilk within threaded programs, application threads and service threads, which belong to entirely separate layers of the two-layer cake model, semantically should be independent of each other. I coin this notion as the ***Separation-of-Layers Principle***. Application threads of the concurrent program should be oblivious to the existence of the Cilk workers and the boss thread. Similarly, these service threads should be oblivious to the existence of other application threads outside of their given cilk.

In this thesis, I address the following two questions:

- How can runtime systems be organized to enable task parallelism within mul-

tiple concurrent threads?

- How can a threading interface be designed to best achieve the Separation-of-Layers Principle?

I present Multicilk, a framework that enables programmers to introduce task parallelism via OpenCilk into programs that use threaded concurrency. Multicilk upholds the Separation-of-Layers Principle by actively distinguishing between application threads and service threads in order to maintain their respective semantics. It employs a technique called ***cilk impersonation*** that allows Cilk worker threads to act on behalf of the cilk that they are in. Multicilk provides an API based on the C11 threads interface, which allows Multicilk to integrate well with existing threaded applications.

Understanding how a programmer uses Multicilk is best showcased through a brief example. Consider the producer-consumer programs in Figure 1-11. At a high level, both programs create two threads. The first thread represents a producer that generates some data with the expensive function `long_computation1()` and sends that data over for the consumer to process. When possible, the consumer thread retrieves data from the producer and processes it using the expensive function `long_computation2()`. Assume there exists a natural way to exploit task parallelism within `long_computation1()` and `long_computation2()`. Multicilk allows users to safely parallelize both of these functions using Cilk as described in the Multicilk program in the figure.

The structure of the producer-consumer logic remains mostly the same even as Multicilk is introduced into the program. The `produce()` and `consume()` routines remain identical. All the modifications occur in `main()` (apart from the modifications in `long_computation1()` and `long_computation2()` that are now parallelized with Cilk). As implied by the minimal modifications made to the program, Multicilk's interface is a drop-in-replacement for many existing threading library functions.

Before Multicilk

```
int produce( void* args ) {
    while ( keep_producing ) {
        data = long_computation1(args);

        send_to_consumer(data);
    }
    return 0;
}

int consume() {
    while ( keep_consuming ) {
        data = read_from_producer();

        long_computation2(data);
    }
    return 0;
}

int main(int argc, char** argv) {
    thrd_t p,c;









    thrd_create(&p, produce, &args);
    thrd_create(&c, consume, NULL);

    int res1, res2;
    thrd_join(p, &res1);
    thrd_join(c, &res2);
}
```

With Multicilk

```
int produce( void* args ) {
    while ( keep_producing ) {
        // long_computation1 is parallelized with Cilk
        data = long_computation1(args);
        send_to_consumer(data);
    }
    return 0;
}

int consume() {
    while ( keep_consuming ) {
        data = read_from_producer();
        // long_computation2 is parallelized with Cilk
        long_computation2(data);
    }
    return 0;
}

int main(int argc, char** argv) {
    thrd_t p,c;

    /* Create Cilk runtime configurations to be used
       when creating the respective cilks */
    cilk_config_t cfg1 = cilk_cfg_from_env("PROD_CFG");
    cilk_config_t cfg2 = cilk_cfg_from_env("CONS_CFG");

    // Creating two cilks
    cilk_thrd_create(cfg1, &p, produce, &args);
    cilk_thrd_create(cfg2, &c, consume, NULL);

    int res1, res2;
    thrd_join(p, &res1);
    thrd_join(c, &res2);
}
```

Figure 1-11: Producer-consumer programs without (left) and with (right) Multicilk.

Thesis overview

This thesis makes the following contributions:

- A design for programmers to explicitly create cilks with unique Cilk runtimes.
- An explanation of how common threading-library functions can be supported to achieve cilk impersonation.
- An exploration how the glibc implementations of synchronization primitives (mutexes, condition variables, and thread local storage) can be modified to support cilk impersonation.
- Mechanisms for programmers to partition CPU resources between cilks and threads.

- An API based off of the C11 `<threads.h>` interface that supports using OpenCilk with a threaded environment.

- A discussion of open questions and opportunities for future research to extend the capabilities of Multicilk.

The rest of this thesis is organized as follows. Chapter 2 introduces important terminology, discusses past attempts at composing parallelism and concurrency, and more thoroughly explains requisite background knowledge about OpenCilk and general threading libraries that Multicilk is based on. Chapter 3 explains how Multicilk's underlying infrastructure enables task parallelism within multiple concurrent threads and explains other properties that Multicilk provides. Chapter 4 discusses the modifications to standard threading-library functions from the C11 threads interface to support cilk impersonation while maintaining their original semantics. Chapter 5 discusses changes necessary to glibc implementations of common synchronization mechanisms to support cilk impersonation. Chapter 6 evaluates Multicilk's understandability and simplicity while also addressing its limitations. Finally, chapter 7 concludes the thesis and suggests possible directions for future work.

Chapter 2

Background

This chapter covers background knowledge useful for understanding the rest of the thesis. Section 2.1 begins by discussing relevant terminology. Section 2.2 describes past work on systems that employ both concurrency and parallelism. Section 2.3 gives an overview of common threading interfaces used in the C that the Multicilk API is based on. Section 2.4 gives an overview of the Cilk programming language and its runtime system. Section 2.5 discusses the details of the system on which the Multicilk framework was designed and implemented.

2.1 Terminology

This section provides a brief glossary of commonly used terms in this thesis.

In this thesis, ***thread***, mentioned without any modifier, refers to application threads strictly used in the threaded concurrency layer of the two-layer cake model and not service threads unless otherwise noted.

Threaded concurrency refers to using threads as the mechanism for concurrent computing. *Multithreaded concurrency* has an equivalent definition, but this thesis uses "threaded" for convenience.

Task parallelism refers to distributing different logically independent tasks across multiple nodes/threads which execute the tasks in parallel. Past papers sometimes use the term *multithreading* for this definition [17], but this thesis strictly uses "task

parallelism" to prevent any ambiguity when also discussing threaded concurrency or threaded programs.

Data parallelism refers to distributing data across multiple nodes/threads, which operate on elements of the data in parallel.

A ***Pthread*** describes a POSIX standard thread.

A ***threaded program*** describes a program that utilizes threading for concurrency.

A ***Cilk function*** describes a function that contains Cilk logic in its immediate scope.

2.2 Related work

The idea of a hybrid concurrent-parallel system has been studied before with MPI+X [21]. MPI (Message Passing Interface) is a standard that utilizes concurrency in the form of message-passing to distribute computation across nodes in a computing cluster, each running an MPI process, and uses a framework X, which can range from standard Pthreads to OpenMP or OpenCilk, to parallelize the computation across the cores of a single node [19, 21, 38]. Typically X uses a shared memory/thread model. MPI by itself has limited scalability due to the lack of proper load management and overhead by the MPI library [21]. With a hybrid model, X can be used to deal with proper load management and avoid communication overhead within a single node to account for MPI's deficiency [21, 38].

While this hybrid approach works in practice and has been effective, it has its disadvantages. The first is that under the MPI+X approach, concurrent applications that wish to utilize Cilk require refactoring to use MPI. If the program was previously using a shared memory/threading model, changing to MPI implies drastic changes to program structure and added burden to the programmer. Implementing a hybrid MPI model, MPI + OpenMP is noted to be quite cumbersome, and required expertise [19]. There is also no guaranteed speedup with the hybrid approach, as implementing the hybrid system might introduce overheads that outweigh the advantages of both MPI and framework X [19].

To the best of my knowledge, no system that attempts to compose parallel programming frameworks like OpenCilk with *threaded* concurrency existed before Multicilk.

2.3 Common threading interfaces

POSIX threads

A standard execution model for implementing shared-memory concurrency in C and C++ is by using the POSIX thread, or ***Pthread*** interface. The Pthread interface provides a platform-independent API that adheres to the POSIX standard. The Native POSIX thread library, ***NPTL***, is the implementation of the Pthread interface used in Linux-based operating systems with newer versions of glibc [29].

The Pthread API is very low-level and requires programmers to create and manage threads explicitly. The API also allows programmers to manage necessary synchronization mechanisms like mutexes to achieve mutual exclusion and condition variables to signal events. This low-level control aims to maximize the ability of the programmer to get good performance.

C11 threads

The C language specification standardized threading support in C11 [7], by providing the C11 threads interface, accessible via the `<threads.h>` header. Like the POSIX threads interface, this interface is also platform-independent and provides similar functionality. NPTL also provides an implementation for this interface. `<threads.h>` is a compact library consisting of a few preprocessor macros and the 25 functions listed in Figure 2-1. The general categories of functions, separated by the horizontal lines in the figure, are threading functions, functions for mutual exclusion, functions for condition variables, and functions for thread-specific data.

thrd_create()	thrd_exit()	thrd_join()	thrd_detach()
thrd_sleep()	thrd_yield()	thrd_equal()	thrd_current()
mtx_init()	mtx_lock()	mtx_unlock()	mtx_timedlock()
mtx_trylock()	mtx_destroy()	call_once()	
cnd_init()	cnd_signal()	cnd_broadcast()	cnd_wait()
cnd_timedwait()	cnd_destroy()		
tss_create()	tss_delete()	tss_get()	tss_set()

Figure 2-1: Functions in the C11 threads API.

2.4 OpenCilk

The Cilk programming language

Cilk is a programming language based on C and C++ designed for parallel programming. Cilk is based on the fork-join model and allows programmers to structure their serial programs to exploit task parallelism. The language consists of the keywords `cilk_spawn` and `cilk_sync`. `cilk_spawn` is used to allow functions to run in parallel, while `cilk_sync` is used as a blocking method to wait for all spawned tasks in the current scope to finish [17]. The simple and intuitive syntax lets programmers easily parallelize their serial programs. An example of a Cilk program, taking the function from Figure 1-3, is shown in Figure 2-2. Calls to `fib(n-1)` and `fib(n-2)` are logically independent and are thus are allowed to run in parallel. After both calls have completed (marked by the `cilk_sync`), their results are summed and returned.

The trace of a Cilk program illustrates how its parallel logic is executed. Figure 2-3 shows the trace of a call to `fib(4)` using the code from Figure 2-2. In the trace, a fork appears at the `cilk_spawn` of `fib(n-1)` indicating that the call to `fib(n-2)` and `fib(n-1)` are completed in parallel. The `cilk_sync` forces both calls to finish before continuing to successive strands of work. The joining of strands denotes a `cilk_sync` in the trace.

The Cilk runtime system

The Cilk runtime system defines the core behavior of how Cilk works. The Cilk runtime maintains a pool of ***workers***. It employs a provably-good work-stealing

```
1
2  int fib(int n) {
3      if (n < 2) return n;
4
5      // spawn off call to run in parallel
6      int x = cilk_spawn fib(n-1);
7      int y = fib(n-2);
8
9      // wait for all cilk_spawns in frame to finish
10     cilk_sync;
11     return x + y;
12 }
13
14 int main(int argc, char** argv) {
15     int n = atoi(argv[1]);
16     int result = fib(n);
17     printf("%d\n", result);
18 }
```

Figure 2-2: Cilk program to find n'th Fibonacci number

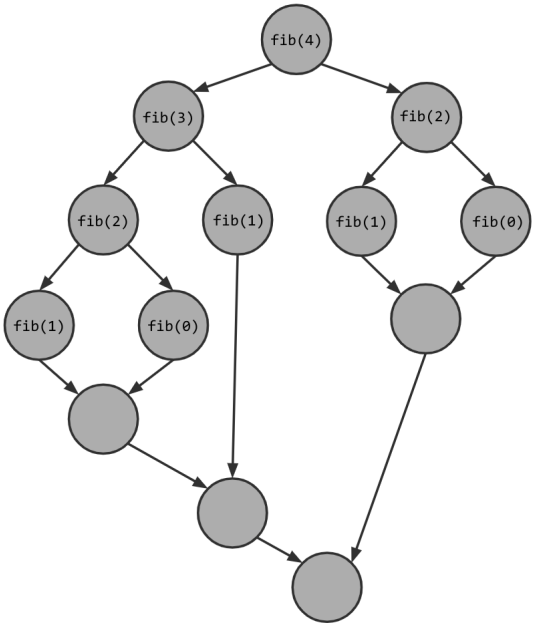

Figure 2-3: The trace of a call to `fib(4)`.

scheduler to load balance computation amongst these workers [1]. Each worker has a deque that maintains their work to be executed. Each worker operates on the bottom of its deque, treating it as a stack. The core idea of the work-stealing scheduler is that when a worker runs out of outstanding work on its deque, it can steal work from

```
1
2  def cilkify():
3      if Cilk worker Pthreads do not exist:
4          Create worker Pthreads
5      Transfer execution control to workers
6
7  def uncilkify():
8      Transfer execution control back to main thread
9
10 def Cilk_function():
11     if current thread is not a Cilk worker:
12         cilkify()
13     Logic of Cilk function
14     if current thread is the last executing Cilk worker:
15         uncilkify()
```

Figure 2-4: Steps for how Cilk workers are enabled to execute a Cilk function

the top of the deque of another worker. This process of stealing keeps the workers busy and effectively load-balances the computation.

In OpenCilk v1.0, a single global Cilk runtime object is implicitly created before `main()` by a compiler-generated function call to `__cilkrts_startup()`. Memory for the workers is allocated at this time, along with other data structures used by the Cilk runtime, such as the worker deques.

Execution control of the program is swapped from the main program thread to Cilk workers at the beginning of **Cilk functions**, functions that use any Cilk language constructs in their immediate scope. Whenever the main thread enters a Cilk function, it will invoke a call to `cilkify()`. At this point, two main events happen. If this is the first time the main thread invokes `cilkify()`, then the routine creates a Pthread for each Cilk worker. Secondly, the routine transfers execution control of the program from the main thread to the Cilk worker Pthreads. The main thread falls asleep when the workers begin executing work. When the workers finish executing the Cilk function, one of them calls `uncilkify()` which wakes up the main thread and transfers control of program execution back to it. If a *Cilk worker* makes a call to another Cilk function, it will not invoke `cilkify()`. Since the "would-be" caller is a Cilk worker, that must mean `cilkify()` was already invoked before, and the Cilk

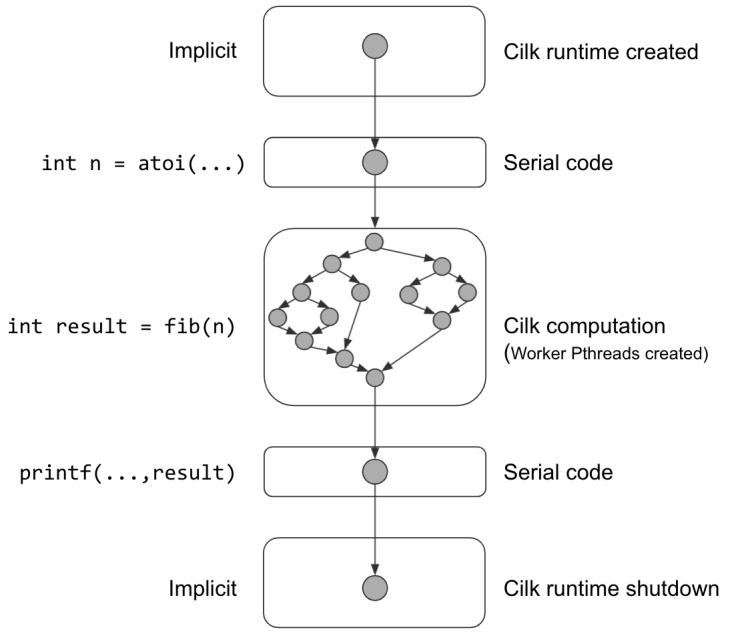

Figure 2-5: The steps the thread of execution takes to run the program defined by Figure 2-2.

workers already have control of the program execution. If the *main thread* enters *another* Cilk function, it will call `cilkify()`, but if a previous invocation of `cilkify()` already created the workers' Pthreads then no new Pthreads are created, and only the control of program execution is transferred over to the Cilk workers. Figure 2-4 summarizes the behavior of `cilkify()` and `uncilkify()` with pseudocode.

Finally, at the program's exit, an implicit call to `__cilkrts_shutdown()` shuts down the Cilk runtime, freeing any allocated memory and joining the worker Pthreads.

Figure 2-5 is a visual of the lifetime of the Cilk runtime during the execution of Figure 2-2. The Cilk runtime is created at the start of the main program thread and is shut down at exit.

For any serial programs using Cilk, there is only a single Cilk runtime and all Cilk computation in the serial program relies on using the same set of workers from this default global Cilk runtime.

Tapir/LLVM

The Tapir/LLVM compiler [40] compiles and optimizes Cilk code to run parallel execution efficiently on shared-memory machines. Specifically, Tapir extends LLVM

by providing an intermediate representation for logically parallel tasks, allowing for optimizations across calls like `cilk_spawn`, `cilk_for`, and `cilk_sync` rather than just simply treating them as syntactic sugar to invoke the Cilk runtime. In addition, Tapir enables standard compiler optimizations for serial code to also work with Cilk language constructs while utilizing parallel optimizations such as loop scheduling.

The Pure Cilk Model

The application programmer generally does not need to be aware of the details of the Cilk runtime system, including the existence of Cilk workers. Instead, a programmer using Cilk only needs to be aware of the Cilk language constructs and understand how to use Cilk to exploit task parallelism in their code. This simple notion represents the scope of the Pure Cilk Model.

The Cilk Worker Model

The Cilk Worker Model exposes details of the Cilk runtime and the Cilk worker Pthreads. Specifically, with this model we can understand how and when workers are scheduled to execute Cilk computation. Generally only the Cilk implementer uses this model.

2.5 The development environment

The Linux operating system is the basis for the design of Multicilk.

I implemented Multicilk's interface on a Ubuntu 20.04 machine on an x86 architecture using the ELF (Executable and Linkable Format) file format standard. I based my implementation off of the glibc 2.31 and 2.34 NPTL implementation of the mutexes/condition variables and thread-specific storage respectively.

Chapter 3

A framework with multiple Cilk runtimes

In this chapter I layout the properties of Multicilk and discuss how Multicilk is able to achieve each one. I first explain the problems with using a single Cilk runtime and justify the necessity of using multiple Cilk runtimes. I discuss how multiple runtimes can be utilized to properly partition CPU resources between application threads doing Cilk computation. I also reintroduce the Separation-of-Layers Principle as a way to guide the design of Multicilk's interface. Finally I justify why Multicilk's API is based on the C11 threads interface.

3.1 Properties of Multicilk

Multicilk enables programmers to introduce task parallelism within each application thread of their threaded program while maintaining the original semantics of their threaded program. The properties that Multicilk provides are as follows:

1. **The ability to express task parallelism within application threads using Cilk**

2. **The ability to partition of CPU resources between any application threads, including cilks, for performance.**

3. **A simple programming interface for threading functions that can be integrated easily with legacy code.**.

3.2 Expressing task parallelism in threads

Using Cilk with multiple threads

We cannot rely on using a single Cilk runtime in a program with multiple concurrent threads executing Cilk logic. There are several issues if multiple threads share a single Cilk runtime with a single set of Cilk workers. First, we lose any liveness and performance guarantees since it is entirely possible for one thread to utilize all of the Cilk workers at once and starve the other threads. We also have a determinacy race on the initialization of runtime itself. There might be concurrent calls to `cilkify()`, which is not thread-safe, leading to potentially terrible consequences.

Multicilk solves this issue by providing the ability of each thread to have its own unique Cilk runtime, thereby enabling each thread to have its own set of Cilk workers.

There two types of application threads in Multicilk. ***Normal threads***, which don't have a Cilk runtime, and ***cilks***, which do have their own Cilk runtime. To all other application threads in the program, a cilk appears as a normal thread. The only programmatic difference is that cilks can safely execute Cilk computation while running concurrently with any thread or cilk, whereas concurrent normal threads cannot.

The internals of a cilk

While to the outside world, a cilk appears as a single application thread, under the hood a cilk is an abstraction of the interaction between several underlying service threads. The different service threads are as follows:

- The ***boss thread*** always starts and ends the execution of a cilk's logic. It is also responsible for executing any serial code that is not executed by a Cilk worker. There is a single boss thread per cilk.

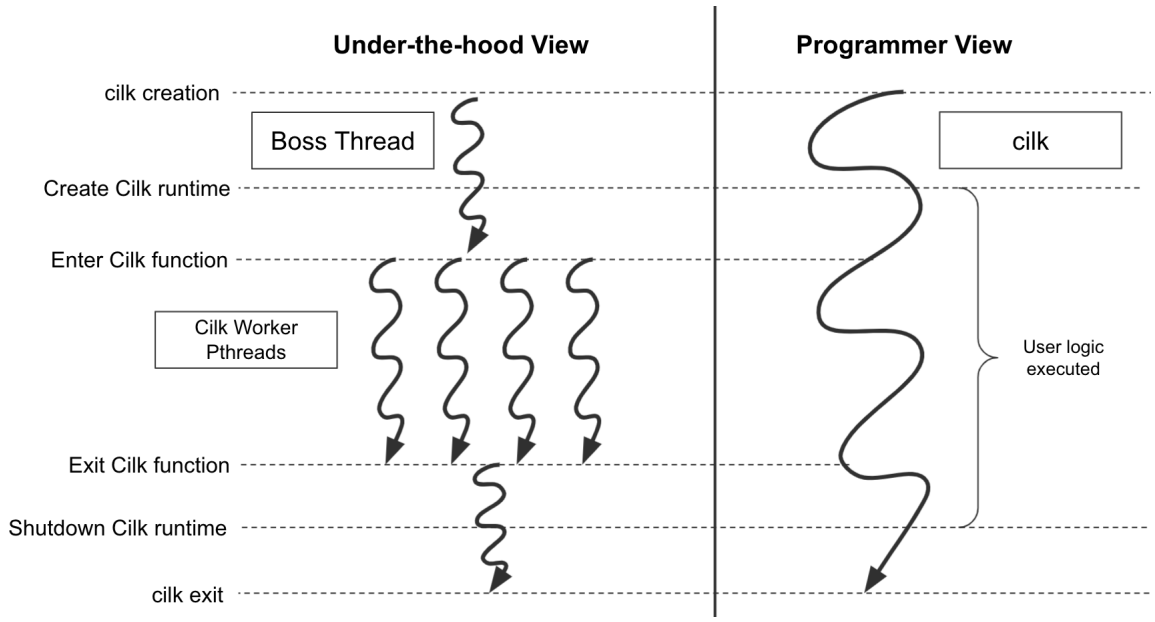

Figure 3-1: Anatomy and lifetime of a cilk from the programmers perspective and from under the hood.

- The Cilk worker Pthreads which are responsible for executing any Cilk computation.

Figure 3-1 contrasts the programmer's view of a cilk with the details of the cilk internals. A boss thread is created and starts running when a cilk is created and eventually starts a Cilk runtime. After encountering Cilk computation, the boss thread creates Cilk worker Pthreads and transfers program execution control. After the Cilk computation is done, the Cilk workers return control to the boss thread and wait in a scheduling loop in the runtime until more work exists for them to do. Finally, at the end of the cilk's logic, the boss thread shuts down the Cilk runtime, thus joining the Cilk worker Pthreads, and finally exits.

3.3 Partitioning CPU resources

It is common in multithreaded programs to partition CPU resources among multiple concurrent threads such that threads avoid any overheads from resource sharing. One method of partitioning is to "pin" threads to disjoint sets of cores.

In OpenCilk, the global Cilk runtime automatically decides how to allocate CPU

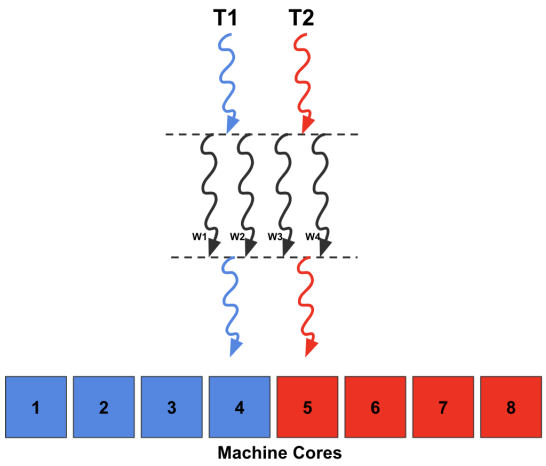

Figure 3-2: Two threads, each parallelized with Cilk, pinned to disjoint sets of cores.

cores to Cilk workers depending on how the number of available cores in the machine compares with the number of Cilk workers.

Using a single Cilk runtime, however, in a setting with multiple threads is problematic if application designers want to isolate different threads to different cores or even machine sockets. With a single Cilk runtime, the Cilk worker Pthreads' core affinities are decided based on the number of available cores. Consider Figure 3-2, where thread T1 is pinned to cores 1-4 of an eight-core machine, and thread T2 is pinned to cores 5-8. The single Cilk runtime created before T1 and T2 may pin each of the 4 Cilk workers to 2 cores (W1 to 1-2, W2 to 3-4, ...). It is entirely possible for the Cilk runtime to schedule W1 to execute Cilk computation in T2. But doing so violates the desired of core affinity of T1 since W2 will execute on cores 1 and 2 which both lie outside the range of T2's core affinity.

While an application maintainer can pin threads to cores however they want, the core affinity of the pinned threads is not reflected on the core affinity of the Cilk workers. A programmer may set core affinities dynamically within Cilk functions, but it is cumbersome to do properly and can be potentially expensive due to added context switching.

In Multicilk, since each cilk has its own Cilk runtime, there is an explicit and natural separation between different application threads in the program. It becomes simpler to think about how to partition CPU resources between cilks since the cilks

don't share any Cilk runtime resources such as workers. Application designers can specify the core affinity for each cilk separately in the program. Upon the creation of a cilk, the boss thread is set to the specified core affinity, which is inherited by the cilk's workers. Because core affinity is explicitly tied to each individual cilk, the cores that Cilk workers execute work on are strictly restricted by the core affinity of the workers' cilk.

3.4 Designing a simple interface

The Separation-of-Layers Principle

I now introduce the ***Separation-of-Layers Principle***. To the programmer using Multicilk under the Pure Cilk Model, the Separation-of-Layers Principle can be understood with the following two points.

- We can view the serial logic within an application thread as a single serial instruction stream and can use Cilk to parallelize the logic.
- The concurrent logic of the program that describes the interaction of different application threads is independent of which threads are actually cilks.

The design of Multicilk aims to maintain this principle as much as possible. With this principle, a programmer can explicitly separate the parts of the program logic that describe concurrency and task parallelism to utilize both techniques simultaneously without the risk of blurring the boundary between the two or adding unnecessary complexity to the code. Going back to the two-layer cake model, Multicilk can be thought of as the icing between the top and bottom layer. It allows both layers to coexist in their intended forms while separating them logically.

Under the Cilk Worker Model, we must carefully consider the interaction of cilks and threads with Cilk workers of the cilks. With the Cilk Worker Model, the Separation-of-Layers Principle can be reinterpreted as the following three points.

- The Cilk workers within a cilk are oblivious to the existence of application threads besides the cilk that they reside in or any application threads that they create themselves.
- All normal threads in the program are oblivious to the existence of any Cilk workers.
- All cilks in the program are oblivious to the existence of any Cilk workers of any cilk besides their own.

Programming with Multicilk

A common way of expressing concurrency through threading in C programs is by using the Pthread/C11 interface to create and manage threads explicitly. Therefore it makes sense for Multicilk to adopt a similar programming interface for users to create and manage cilks explicitly. Using an API based on existing threading libraries makes Multicilk ideal for adoption in legacy code.

Multicilk's API is based on the C11 threads interface, which is significantly simpler and more compact than the Pthread interface, while still sharing the same core functionality. Using the C11 threads interface, allows the design of Multicilk to utilize another layer of indirection when describing the differences between application and service threads as we can associate cilks with C11 threads and workers with Pthreads.

Chapter 4

Designing the Multicilk Interface

This chapter presents the design of the threadling functions in Multicilk's API and discusses the their semantics when working with cilks. I first introduce, cilk impersonation, the technique Multicilk employs for cilks to safely use threading functions. I then describe the two levels of functions Multicilk provides to distinguish between application and service threads. Section 4.2 shows how Cilk runtime configurations can be configured individually. Section 4.3 describes how Multicilk creates Cilk runtimes from a given configuration. 4.4 discusses the semantics of each function in the Multicilk interface, and how its behavior changes in the presence of cilks. For each function, a brief description of its implementation is also presented. Finally, I discuss how I interpose glibc's C11 threading function definitions with Multicilk's own definitions and my survey of the implications of interposing.

4.1 cilk impersonation

Multicilk employs a technique called ***cilk impersonation*** where Cilk workers act on behalf of the cilk and assume the cilk's identity. By my definition, I use the boss thread's ID to represent the ID of the overall cilk. Cilk impersonation alters the behavior of caller-dependent functions so that they work on the level of the application thread, and not the service thread. Cilk impersonation upholds the Separation-of-Layers Principle by maintaining the abstraction that the multithreaded program

Function	Scope	Caller thread behavior
application-level	Operates on the overall thread or cilk as a whole.	Normal threads use their own identifiers. A service thread uses the identity of its cilk.
service-level	Operates on the calling thread only.	The calling service thread's identity is used.

Figure 4-1: Differences between application-level and service-level functions.

consists only of application threads.

Given cilk impersonation as a mechanism to allow Cilk workers act on behalf of their cilk, a natural question to ask is when should it be used?

To answer this question, we began by reviewing the behavior of each type of thread in a Multicilk program. Assume we have a function, F, that depends on the identity of the calling thread. If F's calling thread is a normal thread, it is obvious that the calling thread's identity should be used. If the calling thread is a boss thread or a Cilk worker, then the thread identity used depends on the desired semantics of F itself and the context in which the call to F was made. If semantically the cilk's identity is correct, then cilk impersonation should be used. However, if F semantics are correct on the scope of the service thread, then the calling thread's identity should be used. Note, semantically the boss thread should use the identity of the cilk, but by definition the boss thread's identity is equivalent to the cilk's identity so it is correct for a boss thread always use its own identity when calling F.

We thus have two possible levels of thread-identity-dependent functions: ***application-level*** and ***service-level***. Their distinction is explained in Figure 4-1.

When designing Multicilk to support certain functions in its threading interface, a necessary point to determine is for which functions the application-level definition is appropriate and for which a service-level definition is suitable. This decision depends entirely on the desired semantics of the function.

The general strategy for implementing an application-level function, using cilk impersonation, and service-level function can be understood with the pseudocode in Figure 4-2. For an application-level function, if the calling thread is not a Cilk worker Pthread, we fall back to using the original standard glibc implementation; otherwise,

```
def application_level_func_f():
    if calling_thread is not Cilk_worker:
        return normal_f() # standard definition
    else:
        return modified_f() # assumes boss thread's identity

def service_level_func_f():
    return normal_f() # assumes its own identity
```

Figure 4-2: General strategy for implementing an application-level and service-level function.

we use a modified implementation that uses the cilk's identity instead. The service-level definition is always the standard glibc implementation of the function, and requires no additional modification.

4.2 Configuring a Cilk runtime

In OpenCilk, specific Cilk runtime parameters can be defined as environment variables. Most notably, the number of workers in the Cilk runtime can be defined via `CILK_NWORKERS` [42]. By default, the number of workers equals the number of available cores in the machine.

Since a program that uses Multicilk may have multiple Cilk runtimes, each runtime's parameters can be configured differently. The current implementation of Multicilk provides two parameters for programmers to configure. One is the number of Cilk workers in a runtime. The other is the core affinity of the cilk, which defines the set of processors that the Cilk workers of that cilk are allowed to operate on.

My implementation of Multicilk allows users to set each runtime configuration as an environment variable. To specify both parameters of a given configuration, users can define an environment variable and set it to `"nworkers=#;cpuset=#,#,#..."`, where # represents a whole number. For example, `CFG="nworkers=4;cpuset=0,1,2,3"` creates a configuration that specifies a Cilk runtime with four Cilk workers, which the runtime distributes on CPU cores 0 through 3.

The Multicilk API defines a struct, `cilk_config_t` that stores the state of a Cilk runtime configuration, and it can be created with the function

```
cilk_config_t cilk_cfg_from_env(const char* env_name);
```

given the name of the defined environment variable.

4.3 Creating a Cilk runtime

The Multicilk API provides the following function

```
void cilk_thrd_init(cilk_config_t config);
```

which takes in a `cilk_config_t` object and creates a new Cilk runtime based on the object's parameters. This object records the number of workers of the runtime and the set's core affinity of the thread that uses this runtime. This function also sets the Cilk runtime to be shut down when the calling thread exits.

`cilk_thrd_init()` is used by `cilk_thrd_create()` to create a Cilk runtime object at the beginning of a cilk's lifetime.

4.4 Supporting the C11 threads interface

Identifying a cilk

In the C11 threading library, threads are identified by a unique `thrd_t` value [3, 7] that is associated with them. A thread may use the following function to retrieve its thread identifier.

```
thrd_t thrd_current()
```

The semantics for this function in a Multicilk program are straightforward. As thread identifier describes the thread's identity as a whole, `thrd_current()` is an application-level function. Therefore, the semantics are when a thread calls the function, and that thread is not a Cilk worker, the thread's own `thrd_t` value is returned.

However, when a Cilk worker calls the function, it will retrieve its boss thread's `thrd_t` value instead of its own as a way to impersonate its cilk.

To allow Cilk workers to retrieve their boss' identifier, I store the value of the identifier in the cilk's runtime object which every worker has access to.

Creating cilks

C11 provides `thrd_create()` to create new threads. Instead of using the C11 function name, Multicilk provides its own function to create cilks.

```
int cilk_thrd_create( cilk_config_t cfg, thrd_t* thr,
                      int (*start_routine)(void *), void* arg )
```

This function creates a cilk with its own unique Cilk runtime configured based on `cfg`. After creation, the cilk begins executing the function at `start_routine()` with arguments `arg`. Afterward, the cilk shuts down the Cilk runtime and exits. The created cilk inherits its default attributes from the underlying C11 thread. This guarantees that the newly created cilk is joinable, which means that upon finishing execution, the thread will not relinquish its resources until it is joined or detached by another thread (done with `thrd_join()` or `thrd_detach()`).

The reason for having `thrd_create()` and `cilk_thrd_create()` as separate functions is to make cilk creation explicitly different from creating normal threads. I also needed to provide a different function signature so that Cilk runtime configurations could be passed in as parameters.

In this function's implementation, the newly created thread begins executing a wrapper function that first calls `cilk_thrd_init()` with the passed in `cilk_config_t` argument to setup the Cilk runtime, before invoking the call to `start_routine()`.

Creating threads within a cilk

Although we provide a function to create cilks, `thrd_create()` can still be used to create normal threads, this includes from within a cilk.

By the C11 specification, `thrd_create()` creates a new C11 thread independent of the calling thread's identity. However, in NPTL, `thrd_create()` is implemented as a wrapper around `pthread_create()` whose behavior does depend on the calling thread's identity. `pthread_create()` allows the newly created thread to inherit the attributes of the caller thread. If a Cilk worker calls `thrd_create()`, the attributes of the worker Pthread is used instead of the Cilk.

Fortunately, worker Pthreads inherit all of the attributes, except one, from the boss thread that creates them. That exception is core affinity.

My Multicilk implementation solves this trivially. If the calling thread is not a Cilk worker, then the standard definition of `thrd_create()` is used. Otherwise, the thread is first created, then the core affinity is retrieved from the boss thread and set as the core affinity of the newly created thread using the functions `pthread_getaffinity_np()` and `pthread_setaffinity_np()` [30] respectively.

Joining or detaching cilks

`thrd_join(thrd_t thrd, int* res)` and `thrd_detach(thrd_t)` are functions that control the exiting behavior of the thread that is being joined or detached [4,5]. Their behavior is best understood through an example. Assume we have two concurrent threads A and B. If A calls `thrd_join()` on B, then thread A blocks until thread B exits. By default, all created threads are joinable, so if thread B completes its routine before thread A calls `thrd_join()`, thread B will maintain its state and not free its resources until it is joined. On the other hand, if thread A calls `thrd_detach()` on B, as the name implies, that prevents any dependence between the threads. Thread A doesn't wait until the completion of the thread B, and thread B does not wait to relinquish its resources when it completes its routine.

Under the NPTL implementation, `thrd_join()` requires minor modification. Under the NPTL implementation, if a thread tries to join with itself, `thrd_join()` returns an error. If a Cilk worker calls NPTL's `thrd_join()` with Multicilk's `thrd_current()`, the cilk's ID is compared with the worker's identity and does not result in a deadlock error, which is incorrect behavior. To fix this inconsistency, my

Multicilk implementation does a check to make sure that if the calling thread is a Cilk worker and the passed in thread identifier matches the identity of the worker's cilk then the function returns an error, otherwise the functions calls NPTL's `thrd_join()`. `thrd_join()` uses an application-level definition because its behavior changes if the calling thread is a Cilk worker.

Under the NPTL implementation, `thrd_detach()` requires no modification for Multicilk support as its behavior does not depend on the identity of the calling thread.

Exiting from a cilk

At a high level, `thrd_exit(int res)` causes the calling thread to terminate [2]. While the behavior of `thrd_exit()` is apparent in a normal thread, its semantics within a cilk are hazier.

The first question is, if a Cilk worker encounters a call to `thrd_exit()`, should just the worker exit or should the entire cilk exit? The semantics are vague because it is unclear what the potential use-case of calling `thrd_exit()` is within Cilk computation.

The other challenge lies in the implementation. Cilk workers have no concept of terminating their Pthreads early. Early versions of Cilk had an `abort()` function [17] that could be used in specific settings, but newer versions of Cilk do not support it as it requires the programmer to write their code defensively which can be cumbersome.

To make it easier to reason about thread exits, I limited the flexibility of when `thrd_exit()` can be used. To have safe and reasonable behavior, I decided that cilks can have calls to `thrd_exit()` if a Cilk worker is not the calling thread. In other words, `thrd_exit()` should not be called within any Cilk computation. This means that we are using the application-level definition for `thrd_exit()`. If the calling thread is a Cilk worker, then the function errors. It remains to be explored how constraining this limited flexibility is in practice.

```
thrd_t thrd_current(): // Multicilk definition
    if (calling_thread is not Cilk_worker)
        return thrd_current(); // glibc definition
    return identifier of boss thread
```

Figure 4-3: C-like pseudocode for implementing the Multicilk definition of `thrd_current()`.

Pausing a cilk

The C11 thread interface provides a way to pause a thread and let it sleep with the function

```
int thrd_sleep(struct timespec* duration, struct timespec* remaining)
```

The thread is guaranteed to sleep for at least the time specified by `duration` [6]. As with `thrd_exit()`, the semantics of `thrd_sleep()` within a cilk are vague. If a Cilk worker Pthread encounters a call to `thrd_sleep()`, should just the Pthread sleep or should the entire cilk sleep (thus every Cilk worker)? I decided on the former behavior because I did not see a potential use-case of calling `thrd_sleep()` within Cilk computation. This is the service-level definition. Under this definition, `thrd_sleep()` requires no modifications for Multicilk support. For the same reason `thrd_yield` has no modification and uses the default service-level definition.

It is possible to implement an application-level definition, where every Cilk worker sleeps. When a worker calls `thrd_sleep()`, the worker first sets an exception pointer before sleeping. The set exception pointer causes all other workers to jump to an exception handler in the OpenCilk runtime logic when they are done with the computation in their current frame. In the exception handler, each worker can also sleep before returning to work.

An open question is which interpretation of `thrd_sleep()` (and by extension `thrd_yield` is useful in practice.

4.5 Function name interposition

The goal for having application-level functions is so the programmer can use the function when they want to operate on the scope of the entire thread, oblivious to

whether the thread is a cilk or not. This means the application-level functions should share the same symbols as the functions declared in the C11 threads interface. This idea is shown in Figure 4-3. But as shown in the same figure, we rely on a call to the standard glibc implementation within our new implementation. Thus there is a naming conflict since we want to use different definitions for a single symbol, `thrd_current()`.

Conceptually if the C11 library function definitions are dynamically loaded via glibc, we could rename the function names inside the shared object containing the glibc definitions and use the modified name in the Multicilk implementation of the function. However, the main issue with this approach is that renaming dynamic symbols is nontrivial, and to our knowledge, there is currently no tool that can do so for the ELF standard. Tools like `objcopy` can rename symbols in static libraries with `-redefine-syms`, but not dynamic symbols [32, 43].

Therefore we instead look at changing the way dynamic symbols are resolved at link-time or during run-time. There are three ideas that we considered. For each idea, I use the definition of `thrd_current()` from Figure 4-3 as a working example. Assume the user has some program source code defined in some source file `program.c` which uses Multicilk and makes a call to `thrd_current()`.

Using the Pthread interface

The first idea is to use the fact that with NPTL, the C11 threads interface is implemented with the Pthread interface. Therefore, instead of using the corresponding C11 function as `thrd_current()` in line 3 of Figure 4-3, we can use the corresponding Pthread function (albeit with appropriate casting). This is shown by example below which is the described modification of Figure 4-3.

```
1 // libidea1.so
2 thrd_t thrd_current() {
3     if (calling_thread is not Cilk_worker)
4         return (thrd_t)pthread_self();
```

```
    return identifier of boss thread
}
```

To successfully interpolate glibc's definition of `thrd_current()`, the user could compile their program with:

```
$ clang program.c -lidea1 -pthread -o program
```

As an alternative to recompiling their program, the user could also use the `LD_PRELOAD` environment variable to indicate to the linker to load the `libidea1.so` shared object before glibc, thereby using the definitions provided in `libidea1.so` and discarding any definitions that share the same symbol name that are found afterwards.

```
$ LD_PRELOAD=libidea1.so ./program
```

This solution works as long as the glibc library (which uses NPTL) implements the C11 interface using functions from the Pthread interface.

Using the GNU linker `-wrap` flag

Another idea I considered was using the `-wrap` flag provided by the GNU linker. When a program is linked with the `-wrap=symbol` flag, all previously undefined references to `symbol` resolve to `__wrap_symbol` and all previously undefined references to `__real_symbol` resolve to `symbol` [14]. We can use this to our advantage with the `thrd_current` definition shown below.

```
// idea3.c
thrd_t __real_thrd_current(void);
thrd_t __wrap_thrd_current() {
    if (calling_thread is not Cilk_worker)
        return __real_thrd_current();
    return identifier of boss thread
}
```

Thus any call to `thrd_current()` in the user program resolves to a call to `__wrap_thrd_current()`, which is the definition we intend for Multicilk users to

use. Similarly, the call to `__real_thrd_current()` on line 5, resolves to a call to `thrd_current()` which refers to the glibc definition.

To compile and link this definition with the `-wrap` flag, a programmer would use:

```
$ clang idea3.c program.c  -Wl,--wrap=thrd_current -o
   program
```

The `-wrap` flag is dependent on the linker that the OS uses, and requires code to be recompiled to be used. It does not resolve symbols in shared objects and only works with statically linked objects. Therefore, a user would have to statically link the Multicilk definitions with their program.

Using `dlsym()`

The last idea considered was using `dlsym(void* flag, char* symbol)`, which can obtain the symbol in a shared object at runtime [23]. One of the flags that can be used in the call to `dlsym()` is `RTLD_NEXT`, which forces `dlsym()` to find the next occurrence of the desired symbol outside of the current shared object. This can be used to find the glibc definition of a given function, as shown below.

```
// libidea2.so
static thrd_t (*libc_thrd_current)(void) = NULL;
thrd_t thrd_current() {
    if (libc_thrd_current is NULL)
       libc_thrd_current = dlsym(RTLD_NEXT, "thrd_current");
    if (calling_thread is not Cilk_worker)
       return libc_thrd_current();
    return identifier of boss thread
}
```

To successfully interpolate glibc's definition of `thrd_current()`, the user can compile their program with:

```
$ clang program.c -lidea2 -ldl -o program
```

or once again could use `LD_PRELOAD` with an existing binary.

`dlsym()` is a commonly supported library function and its glibc implementation does not rely on any functions from the C11 threads interface, so we don't have to worry about some cycle of dependence between the new definition and `dlsym()`. Due to this strategy's relative portability and the fact that it doesn't rely on any explicit dependence on a Pthread interface implementation, I consider this as the most promising of the three ideas and use this for my implementation.

External use of functions

Since I interpose the standard NPTL definition of common C11 threading-library functions with my own definitions, it is important to consider which other libraries use those functions, as they now use the modified definition unintentionally which may have terrible consequences. The two codebases I surveyed are glibc and the OpenCilk runtime logic. Glibc was considered since it contained commonly used functions such as `malloc` and `printf`, so it was necessary to make sure interposing thread-library functions did not break their correctness. I found that glibc does not use any functions from the C11 threads interface in its implementations. This means we do not have to worry about a glibc library using the Multicilk definition incorrectly. Likewise, since the OpenCilk runtime implementation does not use functions from the C11 threads interface, I do not have to worry about Multicilk's definitions of the C11 thread interface having unintended consequences when a Cilk runtime is running.

Chapter 5

Extending the Multicilk Interface

This chapter discusses how the Multicilk interface can be extended to support synchronization primitives found in the C11 threads library: mutexes, condition variables, and thread-local storage. Section 5.1 discusses why using default library implementations to lock and unlock mutexes within a cilk is problematic when trying to obtain mutual exclusion between cilks and threads. The section then explains how the existing NPTL implementation for locking can be changed easily for safe use within a cilk. Section 5.2 discusses the modifications needed to support for condition variables. Section 5.3 gives an overview of thread-local storage (TLS) and discusses the modifications necessary for Multicilk to support TLS.

5.1 Mutual Exclusion

A mutex is a commonly used synchronization mechanism. Using mutexes allows the programmer to protect data from being modified simultaneously by many threads, eliminating data races within the program.

For example in Figure 5-1, assume `f()` and `g()` are serial functions that each take an integer as an argument and return an integer. Suppose we have several threads running concurrently that each execute `thread_work()`. Mutexes allow threads to read and update `sum` atomically, which is necessary to achieve mutual exclusion since `sum` is a global variable. `mtx_lock()` and `mtx_unlock()` are part of the C11 threading

```c
int sum = 0;
mtx_t m; // initialized in main

int f(int);
int g(int);

void thread_work() {
    mtx_lock(&m);
    int x = f(sum);
    int y = g(sum);
    int sum += x + y;
    mtx_unlock(&m);
}
```

Figure 5-1: Example program using a mutex.

interface [7].

Types of mutexes

Mutexes can have a variety of different nuances based on the attributes of that mutex. There are three types of mutexes provided by C11 <threads.h>: normal, timed, and recursive mutexes [7].

Normal mutexes are the most basic form of mutexes with effectively no added safety. Deadlock detection is not enabled. Therefore, a deadlock will occur if a thread tries to lock a mutex it already had previously acquired. Timed mutexes block waiting threads until a certain amount of time has passed, after which the blocked threads will stop waiting, and the lock function will return with a timeout error. Finally, recursive mutexes allow for threads to acquire the same mutex multiple times, but for every successive lock of the mutex, there must be an eventual unlock by the same thread so that another thread may eventually acquire the mutex.

The Pthreads library provides attributes for different mutexes such as robustness or whether or not they support ellision. These additions are beyond the scope of the C11 threads interface. This thesis will focus strictly on the the three lock types provided by <threads.h>.

```
int sum = 0;
mtx_t m; // initialized in main

int f(int);
int g(int);

void thread_work() {
    mtx_lock(&m);
    int x = cilk_spawn f(sum);
    int y = g(sum);
    cilk_sync;
    int sum += x + y;
    m.unlock();
}
```

Figure 5-2: Example program of a mutex being used around Cilk computation.

Locking and unlocking within a cilk

Mutexes are a desirable mechanism for mutual exclusion between different cilks and threads. For example, consider we try to exploit the task parallelism of the code segment in Figure 5-1 with Cilk. In the code segment in Figure 5-2, we show the modified code segment where `thread_work()` is executed by concurrent cilks.

The use of mutexes is unchanged even with the use of Cilk as seen in Figure 5-2. This is intentional as I wanted to maintain the familiar C11 mutex interface for the programmer. However, the intersection of mutexes and Cilk contends with the guarantees of the NPTL implementation of `mtx_lock()` and `mtx_unlock()`.

The concept of thread ownership is a prevalent feature of many mutexes. In many cases, the same thread that locks the mutex is required to unlock that mutex. For example, in the NPTL implementation of `mtx_lock()` and `mtx_unlock()`, mutex ownership is maintained by saving the `tid` of the calling thread in the mutex metadata and comparing it with the `tid` of the calling thread afterward.

In Figure 5-2, it is entirely possible for a Cilk worker Pthread A to begin executing `thread_work()` and thus execute the call to `mtx_lock(&m)`. While worker A is executing the call to `f()`, it is possible for another Cilk worker B to execute the continuation and eventually execute the call to `mtx_unlock(&m)`. However, that is

erroneous behavior since Pthread A was the Pthread that acquired the mutex.

By the Separation-of-Layers Principle, the programmer should not have to be concerned with the behavior of Cilk workers within a cilk when using mutexes to protect shared resources between threads and cilks. Therefore, any necessary changes to mutex behavior are done by changing the mutex's lock and unlock logic.

With cilk impersonation, the desired semantics of locking and unlocking a mutex within a cilk in a Multicilk program is that all Cilk workers of the same cilk appear to share the same tid, specifically the `tid` of their boss thread.

The pseudocode for the NPTL implementation of mutex locking and unlocking is shown in Figure 5-3. The actual implementation has more logic to support other types of mutexes provided by the Pthread library, but the pseudocode shows the code paths for locking and unlocking normal mutexes and recursive mutexes. The `tid` of the calling thread is used to mark mutex ownership.

The modifications necessary to provide an application-level `mtx_lock()` and `mtx_unlock()` are surprisingly simple and the modified pseudocode can be seen in Figure 5-4. As this is an application-level function, if a non-Cilk worker Pthread is the calling thread, then they are redirected to using the standard locking implementation, which to avoid naming conflicts as described in Section 4.5, is the Pthread interface's corresponding function. If the calling thread is a Cilk worker, then instead of using its own `tid` for the remainder of the logic, it will use the `tid` of its boss thread. By using this modified locking/unlocking implementation for the program in Figure 5-2, we see that the mutex is safely locked and unlocked even if different Cilk worker Pthreads did them. The C11 interface also provides a `mtx_timedlock()` function for locking timed mutexes, and `mtx_trylock()` for non blocking calls. Both of those functions require the same modifications made to `mtx_lock()` to be supported by Multicilk.

One might notice the modified `mtx_lock()` and `mtx_unlock()` have determinacy races if multiple Cilk worker Pthreads from the same cilk make parallel calls to them. However, this is allowed behavior by the requirements of an application-level function. The application-level `mtx_lock()` and `mtx_unlock()` only provide mutual exclusion on the scope of normal threads and cilks, but do not do so on the granularity of a Cilk

```
// NPTL locking/unlocking implementation pseudocode
int mtx_lock(mtx_t* m) {
    tid = get_tid();
    if (m->type == NORMAL) {
        if (m->owner == tid)
            return ERROR;
        low_level_lock();
    } else if (m->type == RECURSIVE) {
        if (m->owner == tid) {
            ++(m->count);
            return SUCCESS;
        }
        low_level_lock();
        m->count = 1;
    }
    m->owner = tid;
    return SUCCESS;
}

int mtx_unlock(mtx_t* m) {
    tid = get_tid();
    if (m->type == RECURSIVE) {
        if (m->owner != tid)
            return ERROR;
        if (--(m->count) != 0)
            return SUCCESS
    }
    m->owner = NULL;
    low_level_unlock();
    return SUCCESS;
}
```

Figure 5-3: Pseudocode of NPTL implementation of mutex locking and unlocking [44,45].

worker Pthread. To provide mutual exclusion between workers of the same cilk, we need service-level locking and unlocking. This is effectively done using the standard locking function and unlocking function from the Pthread interface.

5.2 Condition Variables

Condition variables are synchronization primitives that allow threads to wait until they are signaled to wake up. The condition variable is usually protected by a mutex

```
// Multicilk locking/unlocking implementation pseudocode
int mtx_lock(mtx_t* m) {
    if (!is_Cilk_worker())
        return normal_mtx_lock(m);
    tid = get_boss_thread_tid();
    if (m->type == NORMAL) {
        if (m->owner == tid)
            return ERROR;
        low_level_lock();
    } else if (m->type == RECURSIVE) {
        if (m->owner == tid) {
            ++(m->count);
            return SUCCESS;
        }
        low_level_lock();
        m->count = 1;
    }
    m->owner = tid;
    return SUCCESS;
}

int mtx_unlock(mtx_t* m) {
    if (!is_Cilk_worker())
        return normal_mtx_unlock(m);
    tid = get_boss_thread_tid();
    if (m->type == RECURSIVE) {
        if (m->owner != tid)
            return ERROR;
        if (--(m->count) != 0)
            return SUCCESS
    }
    m->owner = NULL;
    low_level_unlock();
    return SUCCESS;
}
```

Figure 5-4: Pseudocode of Multicilk implementation of application-level mutex locking and unlocking.

so that threads can be selectively woken up.

Upon reaching a condition variable wait, a thread will unlock a previously acquired mutex before sleeping thus allowing other threads to access the condition variable and begin waiting if necessary. Sleeping threads can be signaled to wakeup.

Because condition variables involve locking and unlocking mutexes, potentially with Cilk computation between other calls to lock and unlock the same mutex, we must consider the situation in which different worker Pthreads will be attempting to access the same mutex. Fortunately, this problem is solved using the modified mutex lock and unlock methods described in Section 5.1 to implement the condition variable wait method, `cnd_wait()`. The same modifications are necessary to support `cnd_timedwait()`.

As with the Multicilk definition of `mtx_lock()`, these modifications to `cnd_wait()` make it strictly an application-level definition. It is not safe for multiple parallel Cilk workers to use this definition simultaneously.

This new definition of `cnd_wait()` does not support handling signals from functions like `pthread_kill()` or `pthread_cancel()` if a Cilk worker is waiting on a condition variable. It remains an area of future research to support this functionality.

5.3 Thread local storage (TLS)

The use of global and static variables is problematic in libraries used in threaded programs since the multiple threads may race on the state of those variables. Therefore, a commonly used technique in threaded programming is to define such static and global variables as ***thread local*** data, where each thread maintains and can access its copy of the allocated memory.

Thread-specific storage (TSS) overview

The Pthreads API supports thread local storage with the term Thread-specific storage. Thread-specific storage works akin to a hashtable, where threads can create keys with the function `pthread_key_create()` [26] and set a value for that key via `pthread_setspecific()`. Each thread must use `pthread_setspecific()` to associate their local value to the given key [31]. A destructor function can also be specified when creating the key, such that that destructor function will be called on a thread's corresponding value to that key upon the thread's exit [26]. A thread's data associ-

ated with a given key can be accessed via the `pthread_getspecific()` function [25]. The C11 `<threads.h>` header provides similar functions to access Thread-specific storage: `tss_create()`, `tss_get()`, and `tss_set()` [7].

`thread_local` *variables*

C and C++ also have extended their respective language specification to support TLS. C11 uses the keyword `_Thread_local` to define thread local variables. C++ uses the `thread_local` keyword. The C11 header, `<threads.h>` defines `thread_local` as a macro equivalent to using the `_Thread_local` keyword [7]. Certain compiler implementations also provide their equivalent keyword. For example, GNU C and Clang provide the `__thread` keyword [15]. Unlike the key-value structure on which the Pthread API is based, users can define various variables with the `thread_local` modifier and use those variables just as if they were static or global variables within function logic. Any writes or reads to a variable marked with this modifier will only reflect the state of that variable in the calling thread's TLS. This is usually done with the help of compiler support. The exact method the TLS variable is stored in memory depends heavily on the file format specification of the executable and object code and the specific machine architecture that the code is being compiled on. For example, many Unix and Unix-like systems use the ELF file-format standard. ELF provides an exact specification for how TLS works under the hood depending on the underlying machine architecture.

Issues with accessing TLS within a cilk

The general issue with supporting TLS is dealing with cilk impersonation. As with accessing mutexes, we have no guarantee of the underlying thread's identity before and after using some Cilk constructs such as a `cilk_spawn`. For example, consider the code segment in Figure 5-5.

The desired semantics of the program in Figure 5-5 is that the thread local variable `val` of the executing cilk is initially set to 0, and after parallel calls to `f()` and `g()`,

```
1  thread_local int val = 2;
2
3  int f(void);
4  int g(void);
5
6  void thread_work() {
7      val = 0;
8      int x = cilk_spawn f();
9      int y = g();
10     cilk_sync;
11     val += x + y;
12 }
```

Figure 5-5: Writes to a TLS variable before and after a `cilk_spawn`

whose results are summed, that sum is added to `val`, and thus `val`'s value for the cilk executing `thread_work()` after line 11 is expected to be `x+y`. However, this is not necessarily the case. It is entirely possible for one worker Pthread, Pthread A, to execute lines 6-8 and begin executing the call to `f()` while another worker Pthread, Pthread B, begins executing the continutation starting from line 9, and eventually line 11 after the first worker returns from `f()`. That means that on line 7, the `val` in Pthread A's local storage is set to 0, but on line 11 the uninitialized `val` in Pthread B is being incremented by `x+y`. Furthermore, the thread local `val` of the actual cilk's underlying C11 thread, is unchanged from the call to `thread_work()` as only Pthread B and A's TLS was accessed.

Ideally, we want the writes to the thread local variable `val` on lines 7 and 11 to occur to the exact location in memory, ideally a single thread's TLS, even though different worker Pthreads might do the writes. Furthermore, we want this change to be reflected on the TLS of the Cilk worker's boss thread.

Accessing TLS safely within a cilk

The functionality used to access TLS will need to be modified to be used safely within both threads and cilks with the intended semantics. In my thesis, I explored the necessary changes to the glibc implementations of TLS-related library functions.

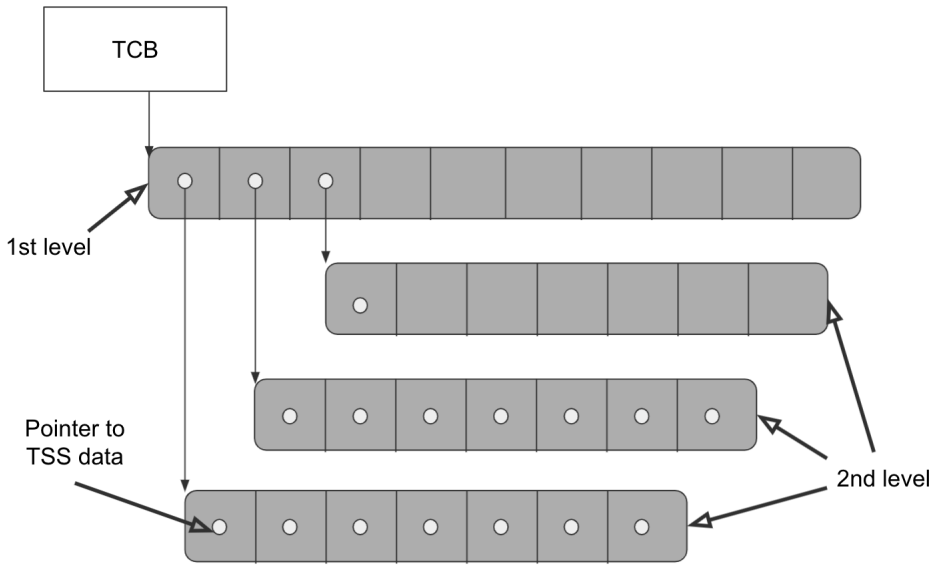

Figure 5-6: The two-level array data structure NPTL's implementation uses to store TSS data.

Supporting the TSS API

We only need to fix how TSS data is accessed in the underlying key-value system since the creation and destruction of keys do not depend on the thread that created/deleted a given key.

The functions that try to access TSS `tss_get()` and `tss_set()` will need to be linked with an alternate definition since the existing glibc definitions try to access the calling thread's thread control block (TCB) directly. The TCB is a data structure that holds information about its thread. The details of the information are OS-dependent. For example, on x86 machines, the current thread's TCB can be accessed directly with the `%fs` register.

NPTL implements TSS storage as a two-level array that is lazily allocated. Each element in the first-level array is a pointer to a different array in the second level. An illustration of this data structure can be seen in Figure 5-6. Both levels of this data structure are indexed with the passed-in key parameter. The elements of each of these second-level arrays are the pointers to the stored TSS data. This two-level array can be accessed as an element in the thread's TCB. The pseudocode for NPTL's implementation of `tss_set()` can be seen in Figure 5-7.

Cilk workers on a single cilk should access a common TSS instead of their own.

```
// NPTL tss_set() implementation pseudocode
int tss_set(tss_t key, void* value) {
    tcb = get_tcb();
    idx1 = key / SECOND_LEVEL_SIZE;
    idx2 = key % SECOND_LEVEL_SIZE;
    tss_1st_level = tcb->specific;
    if (tss_1st_level[idx1] == NULL)
        tss_1st_level[idx1] = calloc(SECOND_LEVEL_SIZE);
    tss_1st_level[idx1][idx2] = value;
    return SUCCESS;
}
```

Figure 5-7: Pseudocode for NPTL implementation of `tss_set()`.

```
// Multicilk tss_set() implementation pseudocode
int tss_set(tss_t key, void* value) {
    if (!is_Cilk_worker())
        return normal_tss_set(key, value);
    tcb = get_boss_tcb();
    normal_mtx_lock(tss_lock);
    idx1 = key / SECOND_LEVEL_SIZE;
    idx2 = key % SECOND_LEVEL_SIZE;
    tss_1st_level = tcb->specific;
    if (tss_1st_level[idx1] == NULL)
        tss_1st_level[idx1] = calloc(SECOND_LEVEL_SIZE);
    tss_1st_level[idx1][idx2] = value;
    normal_mtx_lock(tss_lock);
    return SUCCESS;
}
```

Figure 5-8: Pseudocode for Multicilk implementation of `tss_set()`.

Specifically, they should access the TSS of their boss thread/cilk. Therefore instead of using the TCB of the calling thread, we need to use the TCB of the boss thread, which can be obtained by using the modified Multicilk `thrd_current()` function described in Chapter 4. Because potentially multiple Pthreads will be trying to access and modify the same TCB, it is necessary to provide mutual exclusion, which can be done via service-level locking. This single normal mutex can be stored within the runtime of the cilk, and we only need one mutex per cilk since each cilk only has one common TSS to share between its workers. The pseudocode of the modifications can be seen in Figure 5-8.

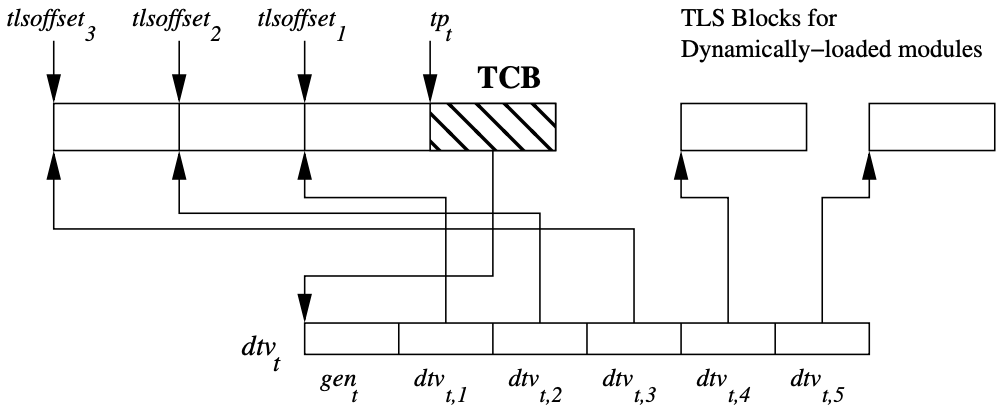

Figure 5-9: TLS data structure on x86-64. Image taken directly from [12].

Supporting the `thread_local` keyword

The ELF standard for x86-64 machines supports TLS with `thread_local` by using a data structure called the dynamic thread vector (DTV). The DTV can be indexed to determine the location in memory of the thread local variable of interest. Thread local variables themselves are stored in sections of memory called TLS blocks. TLS blocks for program modules that are available at compile-time and program startup time are located directly adjacent to the TCB and are referred to as ***static TLS blocks***. Modules that are dynamically loaded later during the program's execution, typically using `dlopen()` have TLS blocks created at runtime and are disjoint from the static TLS blocks. Generally, any TLS variable can be accessed by first indexing into the thread's DTV to find the appropriate TLS block and variable offset and then accessing that TLS block by the obtained offset. However, compilers optimize for the cases where the offset can be known ahead of time [12].

For `thread_local` to be supported in Multicilk programs, the workers of cilk must all have access to the common TLS of their boss thread. It was determined by an earlier report [20] that if a TLS variable was located in one of the static TLS blocks, then a Cilk worker can obtain the location of that variable in the boss thread's TLS by finding the offset between the worker Pthread's TCB and the boss thread's TCB, and adding that offset to the address of the desired variable in the Cilk worker's TLS. This strategy works because the static TLS is adjacent in memory to a thread's TCB,

and the layout of every thread's static TLS is identical since it is set up at program startup. Therefore if the compiler can tell if a TLS variable is in a static TLS block, then a worker Pthread can use this TCB offset method to access that corresponding variable in the boss thread's TLB.

However, the difficulty arises if the compiler cannot tell if the data exists in a static TLS block or if the data does not exist in a static TLS block. Using TCB offsets does not work for these cases in glibc since dynamically loaded modules will lazily allocate TLS blocks that are not guaranteed to have any expected relationship with the location of the thread's TCB. To deal with this, the ELF standard specifies using one of two dynamic TLS access models depending on how much information is known at program start time.

The first access TLS model is referred to as the ***General Dynamic TLS Model***. As its name implies, it is the most generic of the TLS models, and its resulting code can be used anywhere. This model assumes that a TLS variable's module and offset are not known at compile-time or link-time. At runtime, the dynamic linker will provide the module and offset, which will then be passed as an argument to `__tls_get_addr()` which will effectively retrieve the TLS data by indexing into the calling thread's DTV by the module id, and offsetting the output by the given offset. In a Multicilk program, Cilk workers of the same cilk will need to access the DTV of their boss thread instead of their own. Also, access to this DTV will need to be protected by some form of mutual exclusion, such as a mutex, to prevent multiple Cilk workers from racing on it in the case where multiple workers use `dlopen` and allocate Dynamic TLS Blocks simultaneously. For this case, the compiler must use the described modified definition when necessary.

The second access TLS model is referred to as the ***Local Dynamic TLS Model***. This is an optimization done by the compiler when it determines that the TLS variable is defined as the same object it is referenced in since the offsets are known at link-time, given that the object is not the main executable. However, since the object is not the main executable, the module id is still unknown, so at least one call to `__tls_get_addr()` is necessary with an offset of 0 to get the module's TLS block's

location. The optimization is that since the location of this TLS block does not change anymore, other TLS variables in the same module can be accessed via an offset to the saved TLS block location. For this model, the modifications necessary to support Multicilk are very similar to the needed changes to support the General Dynamic model – the modified `__tls_get_addr()` function will need to be used.

Keep in mind that any shared objects loaded before the program start will have their TLS variables in a static TLS block. If a TLS variable is defined in a shared library loaded before program start but not used in the main executable, the compiler will not know where the TLS variable is located, but the linker will. Therefore, if link-time optimizations are enabled, converting the either Dynamic TLS Model into the more straightforward TCB offset idea is possible.

I did not implement support for the modified `thread_local` keyword in Multicilk, but I list some of the required steps. The Tapir/LLVM compiler will need to be modified to support this application-level TLS (`thread_local`) and service-level TLS (`pthread_local`). The current `_Thread_local` keyword has the desired semantics of `pthread_local`, so it is enough to add a macro definition in `<thread.h>` that maps `pthread_local` to `_Thread_local`. To support the modified `thread_local` keyword, a new keyword such as `_Cilk_local` will have to be used for compiler use. The compiler logic will have to be changed as to use the modified `__tls_get_addr()` function when appropriate. The compiler will also have create a function that returns the offset between a Cilk worker's TCB and its boss thread's TCB, a value which is known only at runtime. Optimizing this logic and understanding all the required compiler changes to support this form of TLS remains to be explored.

Dealing with external libraries that use TLS

Because cilks will likely make calls to standard library functions such as `printf()` and `malloc()`, it is necessary to ensure that the function behaves as expected even when a Cilk worker Pthread is the calling thread. Many library function implementations rely on TLS, so it is essential to consider if each function uses the correct type of TLS appropriately (application-level TLS or service-level TLS). I focused on surveying

glibc as that library contained functions most commonly used by C programs.

After surveying various glibc libraries, there are generally three categories of implementations that use thread-local variables.

1. Category A: Implementations maintain thread-local variable state for the sake of performance. For example, glibc's implementation of `malloc()` falls under this category. It maintains a per-thread cache to hold free-lists [11].

2. Category B: Implementations use thread-local variables as per-thread flags. `errno` is an example of a thread-local flag [24].

3. Category C: Implementations depend on thread-local variable state between library function calls for correctness.

Since glibc only uses `__thread` and not `_Thread_local`, by default, all TLS variables use service-level TLS. This is fine for Category A as the TLS is used for performance, so there is no need to treat a Cilk worker thread differently. This is also fine for Category B since it doesn't make sense for Cilk workers to race on shared flags. It instead makes sense for each Cilk worker to use their copy of a flag. However, service-level TLS is inappropriate for Category C, since it relies on reading and modifying the state of TLS variables across function calls to be correct. But because these function calls may be executed by different Cilk workers with each their own TLS, it is possible to read incorrect TLS data. Fortunately, I did not find any functions that fit Category C in glibc or the OpenCilk runtime logic.

Other tooling libraries, however, likely fall into category C. For example, the Cilkscale scalability analyzer measures the parallelism of a Cilk program which indicates how well the program will scale with the number of processing cores [18, 33]. It is part of the Cilk tool suite. In current versions of OpenCilk, which has a single Cilk runtime, Cilkscale's implementation relies on having a global object to maintain metadata while the instrumented program runs. Given that Multicilk now deals with multiple cilks and thus multiple Cilk runtimes, it is natural to make this global object a thread local one. But because the now-instrumented program modifies the state

and data of this object, this means that the same object must be available for access at any point in the cilk – this directly suggests the need for application-level TLS.

In fact, every library that has functions that fall under Category C will need to switch over using application-level TLS, and recompiled with the modified Tapir/LLVM compiler. This task is admittedly cumbersome, and an open question exists on how this modification process can be further improved.

Chapter 6

Evaluation of Multicilk

In this chapter I evaluate Multicilk's ease of use and readability by first enumerating the steps necessary to install Multicilk and then describe possible changes to existing code in order to use Multicilk. I also evaluate Multicilk's simplicity of implementation by quantifying the number of changed lines of code in OpenCilk and NPTL, as well as the number of additional lines of source code from Multicilk's new definitions. Finally I address the limitations of Multicilk.

6.1 Ease of use and readability

Installing Multicilk

Installing Multicilk (without support for synchronization mechanisms) takes 3 steps.

1. Reinstall OpenCilk with the modified Cilk runtime implementation.
2. Add the Multicilk header to source code to access the Multicilk library.
3. Compile the program with `-fopencilk` and `-ldl` (to allow function interposition).

The following additional steps are necessary to support the use of synchronization mechanisms.

1. Compile the modified glibc.

2. When compiling the Multicilk program, link with the new glibc, and use the additional flag `-lgcc_s`.

As a future improvement, the step to compile the modified glibc can be completely avoided by providing a shared object that contains just the modified definitions for synchronization mechanisms, and simply linking against just that rather than an entirely new glibc shared object.

Using Multicilk

Because Multicilk's interface is based heavily on existing and commonly used threading interfaces like Pthreads and C11 threads, most provided functions are drop-in replacements with calls to existing threading functions. The other notable changes to the code are:

1. Using `cilk_thrd_create()` instead of `thrd_create()` to create cilks instead of threads.

2. Using `cilk_cfg_from_env()` to create a Cilk runtime configuration to use when creating cilks.

I do not expect Multcilk to negatively impact the readability of existing threaded logic. Since Multicilk's interface has a one-to-one correspondence with most of the available functionality in the C11 threads library.

Using environment variables to configure Cilk runtime parameters is bare-bones, and as the number of configurable parameters increases, using environment variables may be tedious for application designers. Looking for better ways to format configurations persistently is another area for future work.

Function	Type of modification
thrd_create()	Wrapper
thrd_exit()	Wrapper
thrd_join()	Wrapper
thrd_detach()	None. Not dependent on calling thread.
thrd_sleep()	None. Using service-level semantics.
thrd_yield()	None. Using service-level semantics.
thrd_equal()	None. Not dependent on calling thread.
thrd_current()	Wrapper
mtx_init()	None. Not dependent on calling thread.
mtx_lock()	NPTL modification
mtx_unlock()	NPTL modification
mtx_timedlock()	NPTL modification
mtx_trylock()	NPTL modification
mtx_destroy()	None. Not dependent on calling thread.
call_once()	None. Not dependent on calling thread.
cnd_init()	None. Not dependent on calling thread.
cnd_signal()	None. Not dependent on calling thread.
cnd_broadcast()	None. Not dependent on calling thread.
cnd_wait()	NPTL modification
cnd_timedwait()	NPTL modification
cnd_destroy()	None. Not dependent on calling thread.
tss_create()	None. Not dependent on calling thread.
tss_delete()	None. Not dependent on calling thread.
tss_get()	NPTL modification
tss_set()	NPTL modification

Figure 6-1: Modifications to support Multicilk definitions for the C11 threads interface.

Category	Definitions	Cilk runtime changes
Utility Functions	140	38
Multicilk Library Functions	115	2
Total	**255**	**40**

Figure 6-2: Breakdown of number of LOC changed in the Cilk runtime and the size of new function definitions to support threading functions.

6.2 Simplicity of implementation

Of the 25 functions provided by the C11 threads interface, 13 require no changes to their implementations, 8 functions require direct modification, and 4 functions are implemented as wrappers around their default definitions. The complete breakdown is laid out in Figure 6-1.

Figure 6-2 describes the breakdown of the number of lines of code changed in

Function	Diffs from NPTL	Cilk runtime changes
Utility functions	40	11
Mutex functions	28	3
Condition variable functions	9	0
TSS functions	26	8
Total	**103**	**22**

Figure 6-3: Breakdown of number of LOC changed in the Cilk runtime and NPTL to support synchronization mechanisms.

the Cilk runtime implementation to support Multicilk. The size of Multicilk's new function definitions are also shown. One change in the Cilk runtime logic involved modifying the global runtime object to be thread local. The LOC change counts in the runtime are inflated due to renaming that variable throughout the codebase. The utility functions include `cilk_thrd_init()` and `cilk_cfg_from_env()`. The latter's definition takes up over 120 LOC because it parses environment variables. Multicilk library functions consist of 5 wrapper definitions around C11 thread library functions provided in a single source file. The single change to the structure of the Cilk runtime state was the inclusion of the a `thrd_t` field to store the identifier of the cilk for workers to retrieve.

Minimal changes were also required to extend the Multicilk interface to support mutexes, condition variables, and TSS, and the breakdown is shown in Figure 6-3. Most of these changes were made directly in the glibc implementation of the given function. Of the 27 synchronization functions in the C11 interface, only 8 required any changes. Any line using the calling thread's TCB was switched to use the boss thread's TCB instead if the calling thread is a Cilk worker. For supporting mutexes and condition variables, a `pid_t` field was added to the structure of the Cilk runtime state to store the cilk's TID. For supporting TSS, a `pthread_mutex_t` object was added to the structure of the Cilk runtime state to provide mutual exclusion between workers when they access the cilk's TSS. Support for a modified `thread_local` keyword was not implemented.

6.3 Limitations

Since the Multicilk API provides alternate definitions to the functions provided in the C11 threads interface, we must be wary of how third-party libraries use threading-library functions. Specifically there are two cases of interest.

- Case 1: Libraries use C11 threading-library function symbols with Multicilk definitions, but semantically should be using the service-level function instead.
- Case 2: Libraries use functions from another interface (for example Pthreads), but semantically should be using application-level functions provided by Multicilk.

Neither case was found in a survey of NPTL or the OpenCilk runtime logic. If at least one of these cases is found in another third-party library, that library is incompatible with Multicilk unless the library source code is modified appropriately and recompiled.

Chapter 7

Conclusion

This thesis presents Multicilk as a framework for programmers to introduce task parallelism into the concurrent logic of threaded programs. Specifically, Multicilk allows users to use the Cilk programming language to parallelize the serial logic of individual threads.

I have demonstrated how Multicilk composes the task-parallel layer of my two-layer cake model with the threaded concurrency layer without confounding the use of threads between the layers by using the abstraction of a cilk. I introduce the Separation-of-Layers Principle as an ideal for how the two layers should coexist. I also introduce cilk impersonation as a mechanism to uphold the Separation-of-Layers Principle and maintain thread semantics when parallelizing thread logic with Cilk.

The Multicilk programming interface was directly adapted from the existing C11 thread interface. Multicilk uses function interposition to use the Multicilk definition of appropriate functions instead of the standard NPTL definition while keeping the C11 library symbols. Interposition allows Multicilk to be integrated into most threaded programs as a drop-in-replacement for most thread-library functions. The thesis demonstrates that Multicilk can be extended to support standard synchronization primitives such as mutexes, condition variables, and TLS.

The implementation of Multicilk is concise, spanning just over 400 lines of added or modified logic. Most of this logic contains the definitions of functions in the Multicilk API. I was pleasantly surprised to discover that we can implement most

thread-library functions as wrappers around their NPTL counterparts. The wrapper functions define the behavior of the library function if the calling thread is a Cilk worker. I was also surprised to see many functions do not need an alternate definition as a service-level definition is sufficient to maintain their expected semantics within a cilk.

This chapter concludes by presenting open questions and directions for future work.

Semantics of signalling a cilk

Thread signaling is a feature provided by the Pthread interface (`pthread_kill()`) [27]. Supporting it for signaling to cilks seems challenging. If a cilk is signaled while its boss thread is active, then the cilk should behave as expected. If the cilk is in the middle of some Cilk computation where the boss thread is waiting on a condition variable, however, then the semantics of what should occur become less clear.

When a cilk is signaled, should the boss thread be the only thread to handle the signal? Should it instead propagate the signal to each active Cilk worker? Perhaps it should only propagate the signal to a subset of workers? How should that subset be decided? To answer these questions it will be important to first consider the typical use-cases in which thread signaling is used.

Improving how Cilk runtimes are configured

Using environment variables to configure runtimes is not very expressive and can potentially be cumbersome if the number of configurable parameters in a runtime increases. Therefore, it would be interesting to identify other methods for programmers to specify parameters for runtime configurations. One such idea is to use file formats such as JSON.

Utilizing the time of waiting Cilk workers

Given that cilk logic might include using synchronization primitives such as mutexes or condition variables, there will be instances where a Cilk worker is blocked and waiting. Rather than staying blocked, is it possible to have the Cilk worker suspend its computation on that frame, try to steal other work, and return to the initial frame once the mutex can be acquired or the condition variable has been signaled?

Reducers in Multicilk

Cilk introduced objects called hyperobjects [16] that allow Cilk workers to share state safely without relying on the programmer using explicit synchronization primitives such as mutexes. A common type of hyperobject is called a reducer which allows associative operations to occur on a state in parallel. For example, multiple Cilk workers can increment a reducer's value in parallel. In Multicilk, it remains an open question of how reducers can be extended, given that various sets of workers exist. A conservative approach is to limit the use of reducers to a single cilk; therefore, a reducer cannot be used by multiple cilks concurrently. Can the capabilities of a reducer hyperobject be extended as a synchronization mechanism between cilks themselves? What are the semantics of using such a reducer?

Using Cilksan in Multicilk

Cilksan is a determinacy-race detector that, as the name implies, can be used to detect determinacy races within a Cilk program [13, 33]. An immediate question is what would should it report in a Multicilk program? As a start, it makes sense for it to report any determinacy races within the Cilk logic of each cilk in the program, but this becomes complicated as the cilks begin to interact with one another and possibly with other normal threads in the program. Multiple cilks can race, but because their Cilk runtimes are unrelated, the Cilksan race detector will not detect such races. What races should Cilksan report in the presence of multiple Cilk runtimes?

Using Cilkscale in Multicilk

The Cilkscale scalability analyzer measures the parallelism of a Cilk program which indicates how well the program will scale with the number of processing cores [18,33]. So the immediate question is, what should Cilkscale report in a Multicilk program? Again, the conservative approach is for it to report the parallelism of each cilk individually, but as cilks may interact with other threads in the program, only measuring the individual parallelism of each cilk may not give an accurate report of the scalability of the program as a whole.

More complex threading structures in Multicilk

In this thesis, I describe Multicilk as a way for programmers to spawn individual threads that can do Cilk computation concurrently. Given that more complex threading structures like thread pools exist and are commonly used, how should Multicilk adapt to support those? With the given Multicilk interface, it is possible to implement a thread pool or pool of cilks but the semantics of how the pool of cilks should behave remains unanswered.

Dynamically changing Cilk runtime configurations

One notable limitation of cilks is that once they are created with the user-specified configuration, the parameters of their Cilk runtime are fixed for the lifetime of the cilk. In other words, once the cilk is created, the Cilk runtime cannot alter parameters such as the number of workers. There might be a need, however, for some degree of dynamic modifications to the Cilk runtime. For example, consider two concurrent cilks running each using a non-overlapping set of 4 cores in an eight-core machine. If one cilk exits but the other cilk is still running, we are now wasting CPU resources since the four cores that the exiting cilk was using previously are now unused by the running cilk despite being available.

Generalized mutual exclusion

It is cumbersome for the programmer to decide when to use application-level or service-level locking to achieve the appropriate mutual exclusion. That decision violates the Separation-of-Layers Principle. Ideally, we should have a single locking routine that can behave correctly based on the context. To do this, the semantics of concurrency within a cilk need to be clearly established.

Bibliography

I archived every online source I used with Internet Archive and used the link to the archived sites below. Once can extract the original link, highlighted in blue within the web archive link, by dropping the "`https://web.archive.org/web/<number>/`" prefix from the web archive link.

[1] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[2] C++ reference. `thrd_exit`. `https://web.archive.org/web/20220122120822/https://en.cppreference.com/w/c/thread/thrd_exit`, 2015.

[3] C++ reference. `thrd_current`. `https://web.archive.org/web/20220122120528/https://en.cppreference.com/w/c/thread/thrd_current`, 2017.

[4] C++ reference. `thrd_detach`. `https://web.archive.org/web/20220122120736/https://en.cppreference.com/w/c/thread/thrd_detach`, 2017.

[5] C++ reference. `thrd_join`. `https://web.archive.org/web/20220122120634/https://en.cppreference.com/w/c/thread/thrd_join`, 2017.

[6] C++ reference. `thrd_sleep`. `https://web.archive.org/web/20220122121132/https://en.cppreference.com/w/c/thread/thrd_sleep`, 2018.

[7] C++ reference. C11 Thread Interface. `https://web.archive.org/web/20220122121318/https://en.cppreference.com/w/c/thread`, 2021.

[8] Stephen Cleary. *Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming*. O'Reilly Media, 2019.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, fourth edition, 2021.

[10] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan 1998.

[11] DJ Delorie. glibc Malloc Internals. https://web.archive.org/web/20220122052112/https://sourceware.org/glibc/wiki/MallocInternals, 2019.

[12] Ulrich Drepper. ELF handling for thread-local storage. pages 1–81, 2001. https://web.archive.org/web/20220125060736/https://www.akkadia.org/drepper/tls.pdf.

[13] Mingdong Feng and Charles E Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[14] Free Software Foundation. LD Command Line Options. https://web.archive.org/web/20220122053948/https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html, 1998.

[15] Free Software Foundation. Thread-Local Storage. https://web.archive.org/web/20220122052307/https://gcc.gnu.org/onlinedocs/gcc/Thread-Local.html, 2022.

[16] Matteo Frigo, Pablo Halpern, Charles E Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 79–90, 2009.

[17] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, 1998.

[18] Yuxiong He, Charles E Leiserson, and William M Leiserson. The Cilkview scalability analyzer. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 145–156, 2010.

[19] Intertwine. Best Practice Guide to Hybrid MPI + OpenMP Programming. *Intertwine Consortium*, pages 1–24, 2017. https://web.archive.org/web/20210129204331/http://www.intertwine-project.eu/sites/default/files/images/INTERTWinE_Best_Practice_Guide_MPI\%2BOpenMP_1.1.pdf.

[20] Tim Kralj. CilkLS: Integration and interoperability with pthreads with TLS. Unpublished internal working document, 2020.

[21] P-Fr Lavallée and Philippe Wautelet. Hybrid MPI-OpenMP programming. *CNRS IDRIS Lessons*, 2013.

[22] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The Design of a Task Parallel Library. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA'09)*. ACM SIGPLAN, September 2009.

[23] Linux Man Page. `dlsym`. https://web.archive.org/web/20220122053629/https://man7.org/linux/man-pages/man3/dlsym.3.html, 2021.

[24] Linux Man Page. `errno`. https://web.archive.org/web/20211223205449/https://man7.org/linux/man-pages/man3/errno.3.html, 2021.

[25] Linux Man Page. `pthread_getspecific`. https://web.archive.org/web/20220122052526/https://man7.org/linux/man-pages/man3/pthread_getspecific.3p.html, 2021.

[26] Linux Man Page. `pthread_key_create`. https://web.archive.org/web/20220122053119/https://man7.org/linux/man-pages/man3/pthread_key_create.3p.html, 2021.

[27] Linux Man Page. `pthread_kill`. https://web.archive.org/web/20220122120445/https://man7.org/linux/man-pages/man3/pthread_kill.3.html, 2021.

[28] Linux Man Page. `pthread_mutex_lock`. https://web.archive.org/web/20220122120303/https://man7.org/linux/man-pages/man3/pthread_mutex_lock.3p.html, 2021.

[29] Linux Man Page. `Pthreads`. https://web.archive.org/web/20220122121223/https://man7.org/linux/man-pages/man7/pthreads.7.html, 2021.

[30] Linux Man Page. `pthread_setaffinity_np`. https://web.archive.org/web/20220125061441/https://man7.org/linux/man-pages/man3/pthread_setaffinity_np.3.html, 2021.

[31] Linux Man Page. `pthread_setspecific`. https://web.archive.org/web/20220122052656/https://man7.org/linux/man-pages/man3/pthread_setspecific.3p.html, 2021.

[32] Ted Lyngmo. Binding failure with objcopy –redefine-syms. https://web.archive.org/web/20220125054317/https://stackoverflow.com/questions/54332797/binding-failure-with-objcopy-redefine-syms/54423018, 2019.

[33] OpenCilk. The Cilk tool suite. `https://web.archive.org/web/20220122121506/https://cilk.mit.edu/tools/`, 2018.

[34] OpenCilk. OpenCilk. https://web.archive.org/web/20220124124558/https://cilk.mit.edu/, 2020.

[35] OpenCilk. OpenCilk/Cheetah. https://web.archive.org/web/20200915142315/https://github.com/OpenCilk/cheetah, 2020.

[36] Prakash Panangaden. *Does Concurrency Theory Have Anything to Say about Parallel Programming?*, page 439–446. World Scientific Publishing Co., Inc., USA, 2001.

[37] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[38] Rolf Rabenseifner, Georg Hager, Gabriele Jost, and Rainer Keller. Hybrid MPI and OpenMP parallel programming. In *PVM/MPI*, page 11, 2006.

[39] Arch D Robison and Ralph E Johnson. Three layer cake for shared-memory programming. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, pages 1–8, 2010.

[40] Tao B Schardl, William S Moses, and Charles E Leiserson. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 249–265, 2017.

[41] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. *SIGPLAN Not.*, 28(7):13–22, Jul 1993.

[42] neboat uncttao. OpenCilk/opencilk-project/issues: Limit the number of parallelism. https://github.com/OpenCilk/opencilk-project/issues/71, 2021.

[43] Jorge Luis Williams. redefining dynamic symbols??? – why I want to. https://web.archive.org/web/20220123163942/https://sourceware.org/legacy-ml/binutils/2002-07/msg00417.html, 2002.

[44] Woboq. `pthread_mutex_lock.c`. https://web.archive.org/web/20220122053409/https://code.woboq.org/userspace/glibc/nptl/pthread_mutex_lock.c.html, 2019.

[45] Woboq. `pthread_mutex_unlock.c`. https://web.archive.org/web/20220122053321/https://code.woboq.org/userspace/glibc/nptl/pthread_mutex_unlock.c.html, 2019.