An Efficient Task-Parallel Platform for Interactive Applications

Kyle Singer
Washington University in St. Louis

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds

Recommended Citation
https://openscholarship.wustl.edu/eng_etds/910

This Dissertation is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in McKelvey School of Engineering Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.
# Table of Contents

List of Figures .................................................. v
List of Tables .................................................... viii
List of Algorithms ............................................... xi
Acknowledgments ................................................ xii
Abstract .......................................................... xvi

Chapter 1: Introduction .......................................... 1
  1.1 Motivation: Memcached Case Study ........................ 3
    1.1.1 Obstacle 1: The Rigid Structure of Fork-Join .......... 5
    1.1.2 Obstacle 2: Frequent Use of I/O ........................ 7
    1.1.3 Obstacle 3: Performance Criteria ....................... 10
  1.2 Contributions .............................................. 11
    1.2.1 A More Flexible Task-Parallel Abstraction ............ 11
    1.2.2 Asynchronous I/O with a Synchronous Interface ........ 13
    1.2.3 Prioritization of Tasks ................................ 14
  1.3 Outline .................................................... 14

Chapter 2: Preliminaries ......................................... 16
  2.1 Task Parallelism ............................................ 16
  2.2 Modeling Task Parallelism ................................ 18
    2.2.1 Traditional Task Parallelism ....................... 18
  2.3 Interactive Task Parallelism .............................. 19
  2.4 Scheduling Task Parallelism .............................. 20
    2.4.1 Traditional Task Parallelism ....................... 21
    2.4.2 Interactive Task Parallelism ....................... 22

Chapter 3: Futures and the Proactive Work-Stealing Scheduler .. 24
  3.1 Preliminaries .............................................. 27
  3.2 Proactive Work-Stealing ................................ 28
  3.3 Performance Bounds for ProWS ............................ 33
    3.3.1 Bound on Execution Time ............................ 34
3.3.2 Bounds on Deviations ........................................ 36
3.4 Cilk-F: A Prototype System ................................. 42
3.5 Empirical Evaluation of Cilk-F ............................... 45
3.6 Related Work ................................................. 49

Chapter 4: Using Futures to Efficiently Support I/O .......... 53
4.1 The System Implementation .................................... 57
4.2 Algorithm and Analysis ....................................... 61
4.3 Empirical Evaluation .......................................... 69
4.4 Related Work .................................................. 74

Chapter 5: Representing Priority and Ruling Out Priority Inversions with Futures ........................................ 75
5.1 A Graph Model for Responsiveness ............................ 78
  5.1.1 Preliminaries ............................................... 78
  5.1.2 Weak Edges ............................................... 80
  5.1.3 Well-Formedness and Response Time .................... 82
5.2 Type System for Responsiveness ............................... 87
  5.2.1 The $\lambda^f_i$ Core Calculus ............................. 87
  5.2.2 Cost Semantics and Time Bounds ......................... 90
5.3 Implementation of the C++ Type System in Adaptive I-Cilk .. 93
  5.3.1 Programming Interface ................................... 93
  5.3.2 Type System .............................................. 94
5.4 Evaluation of the C++ Type System ........................... 96
  5.4.1 Application Case Studies ................................ 97
5.5 Related Work .................................................. 99

Chapter 6: Greedy Scheduling for Responsive Applications ... 102
6.1 Background .................................................... 106
6.2 Adaptive Priority Scheduling ................................ 107
6.3 Response Guarantees of APS .................................. 109
  6.3.1 Response time of tasks with priority 1 .................. 112
  6.3.2 Response time of tasks with priority 2 .................. 114
  6.3.3 Generalizing to priority level $\ell$ ......................... 119
6.4 Empirical Evaluation ........................................... 121
  6.4.1 Evaluation of Microbenchmarks ........................... 123
  6.4.2 Evaluation of Application Benchmarks ..................... 126
6.5 Related Work .................................................. 130

Chapter 7: Practically Efficient Scheduler for Task-Parallel Interactive Applications ........................................ 133
7.1 Preliminaries .................................................... 136
7.2 Motivating Example: the Memcached Server ................ 139
Chapter 8: Conclusion ..... 158
8.1 Future Directions ..... 159

References ..... 161
List of Figures

Figure 1.1: An example of the parallel naive recursive fibonacci function written using spawn and sync. .......................................................... 2

Figure 1.2: Pseudocode showing an idealized control flow for handling client connections in Memcached. The spawn, at a high level, indicates that connections are handled by the drive-machine function in parallel. ...................... 4

Figure 1.3: A first attempt at parallelizing Memcached request processing. In this version, there is a race on c such that responses sent to the client can end up interleaved. ................................................................. 5

Figure 1.4: A second attempt at parallelizing Memcached request processing. This attempt can result in responses arriving out-of-order, which would break compatibility with many Memcached client applications. .............. 6

Figure 1.5: The state machine for a network connection in memcached, generated using PlantUML [1]. The structure of this diagram is a testament to how complex the state machine can get when attempting to write efficient modern interactive applications in an event-driven style. ...................... 8

Figure 1.6: Psuedocode showing the state machine code of the event handler used to handle I/O in memcached. ................................................. 9

Figure 1.7: Parallelization of Memcached request processing using futures. In this implementation, responses will not be interleaved and responses will be sent back to the client in the order they were received by the server. .... 12

Figure 2.1: The parallel naive recursive fibonacci function written using (a) spawn/sync and (b) fcreate/ftouch. ................................. 17

Figure 2.2: A directed acyclic graph (DAG) representing a possible task parallel program. Nodes are numbered in their serial execution order, a left-to-right depth first traversal by convention. .......................... 19
Figure 2.3: A directed acyclic graph (DAG) representing a possible task parallel program with priorities. The black nodes are High priority, and the white nodes are Low priority. .................................................. 20

Figure 3.1: A cactus stack. (a) The invocation tree, where function A invokes B and E, and B invokes C and D. (b) The view of the stack by each of the five functions. In a serial execution, only one view is active at any given time. In a parallel execution, however, if some of the invocations are spawns, then multiple views may be active simultaneously. ........................................... 43

Figure 4.1: Distributed map and reduce example. ........................................... 58

Figure 4.2: Speedups, compared to the one-core running time ($T_1$) of the respective baselines, of map-reduce running in ParWS, Cilk-F, Cilk-F (O), and Cilk-L with latencies of 1, 50, and 100 milliseconds. Ideal is the Cilk-F implementation that makes calls to read that returns immediately. For Ideal, Cilk-F, Cilk-F (O), and Cilk-L, we computed the speedup against the $T_1$ in Cilk-F. We computed the speedup for the ParWS against its corresponding baseline implemented in Parallel ML that is not shown. The x-axis shows the core counts ($P$) and the y-axis shows the speedup. Cilk-F (O) oversubscribes the system by using $2P$ workers instead of $P$, for $P$ number of cores (i.e. every core has two workers pinned, one per hyperthread context). Cilk-L pins one worker thread and one I/O thread per core, each on its own hyperthread context. ........................................... 71

Figure 5.1: graphs in which the main thread reads a valid thread handle (a) and NULL (b), and a graph with a weak edge representing a read of a valid thread handle (c). Vertices are labeled with the line of code they represent and threads are arranged in columns. ........................................... 80

Figure 5.2: (a) a graph that is not well-formed because of the strong path from $u_0$ to $t$ (b) a well-formed version of the graph with a weak path from $u_0$ to $t$. 83

Figure 5.3: (a) a graph; (b) its strengthening ................................. 84

Figure 5.4: Syntax of $\lambda_4^4$ ................................. 87

Figure 5.5: Selected command typing rules. ................................. 88

Figure 5.6: Stack, frame and state syntax. ................................. 90

Figure 5.7: Cost semantics for threads ................................. 91
Figure 6.1: The desire calculation algorithm for quantum \( q \) and priority level \( \ell \) . . . . 109

Figure 7.1: The 99\textsuperscript{th} percentile latency of handling memcached requests using Pthread, Adaptive I-Cilk, and Prompt I-Cilk versions of the memcached server. For Adaptive I-Cilk, a parameter sweep was performed and the data shown is drawn from the best configuration for each RPS. The dashed lines show the results for each set of parameters swept. Pthread and Prompt I-Cilk do not require parameter sweeps. . . . . . . . . . . . . . . . . . . . . . . . . . 134

Figure 7.2: The average number of high priority deques at quantum boundaries in memcached using Adaptive I-Cilk. Counts are drawn from the best performing configurations, as in Figure 7.1. . . . . . . . . . . . . . . . . . . . . . . . . . 144

Figure 7.3: The latencies of Memcached implemented with different schedulers. For Adaptive I-Cilk, Adaptive I-Cilk plus aging, and Adaptive Greedy the data points are drawn from the runtime parameter configuration with the best 99th percentile latency for the given RPS. Adaptive I-Cilk uses 5 different sets of parameters, while Adaptive I-Cilk plus aging and Adaptive Greedy both use 4 different sets of parameters. . . . . . . . . . . . . . . . . . . . . . . . . . 149

Figure 7.4: The 95th and 99th percentile latencies of the job server benchmark normalized to the latencies of Prompt I-Cilk, with Prompt I-Cilk also shown for reference. Tasks are shown in order of highest to lowest priority. Adaptive I-Cilk uses 3 different sets of parameters, and its variants use 2 different sets of parameters. . . . . . . . . . . . . . . . . . . . . . . . . . 149

Figure 7.5: The top row shows the the 95th and 99th percentile latencies of the email benchmark normalized to the latencies of Prompt I-Cilk, with Prompt I-Cilk also shown for reference. The bottom row shows the average and median latencies similarly. Tasks are shown in order of highest to lowest priority. Print and comp are at the same priority level. The Adaptive I-Cilk variants use 3 different sets of parameters. . . . . . . . . . . . . . . . . . . . . . . . . . 150
# List of Tables

Table 3.1: The execution times of benchmarks in seconds, running with Cilk Plus (fj and lp) and Cilk-F (sf and gf). $T_s$ shows the running time of the serial elision. $T_p$ shows the running time on $p$ processing cores, and the numbers in parenthesis are the speedups over $T_s$.  

Table 3.2: The execution times of different versions of fib, in seconds, running with Cilk Plus (fj) and Cilk-F (sf and -stack). The -stack row shows the running times of fib-st on Cilk-F but removes the stack-switch upon a fcreate. $T_p$ shows the running time on $p$ processing cores and the numbers in parenthesis are the scalability over $T_1$. The $T_s$ (serial elision running time) for fib is 2.46.  

Table 3.3: The number of deviations incurred on 16 processors. The data is the maximum out of 3 runs.  

Table 4.1: The execution times, in seconds, of map-reduce with different Fibonacci parameters running on Ideal and Cilk-L with different latencies. The $L$ column shows the simulated I/O latency; all latency values are in milliseconds. Two sets of Fibonacci parameters are tested; $n$ shows the input size and $b$ shows the serial base case size. The values in parentheses are overheads relative to the corresponding $T_p$ time of Ideal, which runs map-reduce on Cilk-F with zero latency and incurs no system overhead for latency hiding (same as the Ideal label shown in Figure 4.2).  

Table 4.2: The execution times, in seconds, of map-reduce with various configurations of Cilk-L with no I/O latency (i.e. reads do not block). The overheads are relative to the corresponding $T_P$ time of Ideal, which uses Cilk-F without latency-hiding.  

Table 5.1: The compilation times and resulting binary sizes of application code without and with priority. The compilation time is in seconds and the maximum out of the three compile runs. The binary size is in KB. The numbers in parentheses show overhead compared to the no priority version.
Table 6.1: Execution times of \texttt{fib-ep} (seconds) run using vanilla Cilk-L and Adaptive I-Cilk with $\delta = 0.9$ and $L = 1ms$. \texttt{fib-ep} was run with $\rho = 2$. Overhead, relative to \texttt{fib-ideal}, and standard deviation are in parentheses. ................................................. 123

Table 6.2: Execution times of \texttt{fib-rp} (seconds) run using vanilla Cilk-L and Adaptive I-Cilk with $\delta = 0.9$ and $L = 1ms$. \texttt{fib-rp} was run with $\rho = 1.2$. Overhead, relative to \texttt{fib-ideal}, and standard deviation are in parentheses. ................................................. 124

Table 6.3: Execution time of \texttt{fib-ep} and \texttt{fib-rp}, in seconds, run in Adaptive I-Cilk with various $\rho$ values on 20 processors with $L = 1ms$. Overhead (in parentheses) is relative to \texttt{fib-ideal}. ................................................. 124

Table 6.4: Execution times, in seconds, running with Adaptive I-Cilk using various quantum lengths ($L$) for \texttt{fib-ep} ($\rho = 2$) on 1 and 20 processors, and \texttt{fib-rp} ($\rho = 1.2$) when run on 20 processors. Overhead (in parentheses) is relative to \texttt{fib-ideal}. ................................................. 124

Table 6.5: The average (Avg.) and 95 percentile ($95\%$) flow time of different types of computations running on \texttt{job}, listed from highest to lowest priorities. All times are reported in milliseconds. The Jobs/s reports the throughput (how many instances per seconds executed). Times for Adaptive I-Cilk were collected with $\rho = 2$, $L = 500\mu s$, and $\delta = 0.9$. ................................. 127

Table 6.6: Response times of \texttt{email} tasks, listed from highest to lowest priorities. The resp reports the times elapsed between when a client sends a request to \texttt{email} reacts to the request by generating a computation. The send, sort, and print report the respective times elapsed between when a client sends the given request to when the corresponding task completes. The compress reports the time elapsed to perform a particular compression task. We report the average times (Avg), the $95^{th}$ percentile (95%), and the $99^{th}$ percentile (99%) for all categories. The times for resp and send are in microseconds. The times for sort are in microseconds per message. The rest are in milliseconds. Times for Adaptive I-Cilk were collected with $\rho = 2$, $L = 500\mu s$, and $\delta = 0.9$. ................................. 128
Table 6.7: Response times of proxy tasks listed from highest to lowest priorities. The `resp` reports the time elapsed between when a client sends a request to when proxy reacts to the request. The `hit` reports $\mu s/\text{byte}$ ($\mu s$ is microsecond) for responding with a website already in cache; the `miss` reports $\mu s/\text{byte}$ for those not in cache. The `stat` reports $\mu s/cach\text{eSize}$ for collecting cache statistics. Times for Adaptive I-Cilk were collected with $\rho = 1.2$, $L = 500 \mu s$, and $\delta = 0.9$.

Table 7.1: Wasted and running time, in seconds, accumulated across all processors. Running time is the processing time incurred by scheduling overhead and user code. The values in parentheses for Adaptive I-Cilk are the ratio of Adaptive I-Cilk to Prompt I-Cilk.
List of Algorithms

1. The main ProWS scheduling loop and helper functions .................................................. 31
2. The ProWS steal protocol ................................................................................................. 32
Acknowledgments

I never would have made it where I am today, and this dissertation would never have seen the light, without the support of many people. So many people, in fact, I am certain to accidentally forget to mention many in this section; to those of you unintentionally omit, I apologize.

First, I would like to thank my advisor, Dr. I-Ting Angelina Lee. It was thanks to Angelina that I had the opportunity to attend Washington University in St. Louis, and I couldn’t have asked for a more supportive advisor for this roller coaster known as "PhD life." Prior to working with Angelina, I had no research experience; thanks to her guidance I was able to grow as a researcher and make it to this point. I’ll miss the times we spent working on conference submissions, though I think I may have developed a vendetta against Anywhere-on-Earth (AoE) deadlines as a result. I am immensely grateful for all of Angelina’s guidance.

I would also like to thank my collaborators on the works contained in this dissertation: Drs. Umut A. Acar, Kunal Agrawal, Stefan K. Muller, and Yifan Xu. We never would have been able to advance the art on interactive task parallelism so far so quickly without these people. The people I have had the fortune to work with as a part of this dissertation remind me of an Isaac Newton quote: "if I have seen further, it is by standing on the shoulders of giants."

I thank the other members of my dissertation committee: Drs. Sanjoy Baruah and Christopher Gill. It is never easy to get five people together in the same meeting, especially when they are all professors.\(^1\) Thankfully, however, the members of my committee were very helpful in finding a time that worked after the scheduling poll didn’t turn up any time slots that worked. This may seem like a strange thing to recognize here, but I’ve been going through a bit of a rough patch and I am incredibly grateful that we managed to find a time so quickly.

I want to thank the other members of the Washington University in St. Louis (WUSTL) parallel computing group I have worked alongside: Aaron Handleman and Zhe Wang. I also thank former members Justin Deters, Dane Johnson, and Chen Xu. One thing I’ve missed

---

\(^1\) Many scheduling problems are in NP; I suspect, however, this scheduling problem lies beyond NP.
since the pandemic began 3 years ago in March, 2020, was the ease in which group members
could bounce ideas off of each other, both the good and the ridiculous.\(^2\)

I would like to thank the WUSTL engineering IT department, especially Mark Bober. From
recovering our servers from the (admittedly not so) few times I managed to lock them up, to
physically moving around our servers to directly connect secondary ethernet ports, Mark was
always quick to help. This definitely saved us on more than a few conference submissions.

I would also like to thank all of those I have worked with in industry as well. I would like to
thank those at Ciholas for the opportunity and support that helped me grow as a developer
during my time as an industry software engineer. My programming chops grew significantly
while working there, and that has helped me immensely with getting everything done as
a part of my PhD. I am also grateful to the people at Neural Magic and Facebook who
mentored me and helped me grow even further during my time as an intern.

I am very grateful to my undergraduate professors from the University of Evansville as
well, especially those from the CSE department: Drs. Dick Blandford, Deborah Hwang,
Robert Morse, Don Roberts, and Mr. Mark Randall. They provided me with a foundation
in computer science that I was able to use to succeed in graduate school. I am especially
thankful to Dr. Hwang for encouraging me to apply to PhD programs when I was considering
applying for Masters programs.

I have had the fortune of making many friends during my time in the PhD program at
Washington University in St. Louis: Ruixuan Dai, Justin Deters, Clayton Faber, Aaron
Handleman, Shenghua He, Dane Johnson, Jeff Jung, Qianyu Li, Aaron Park, Tom Plano,
Kathryn Sarullo, Wei Tang, Stephen Timcheck,\(^3\) Andrew Estornell, Zhe Wang, Bei Wu,
Zhihao Xia, Yifan Xu, Hao Yan, Rusi Yan, Dan Zeng, and Chenfeng Zhao. I am thankful to
each and every one of them; they have always been supportive and spending time with them
kept me sane. Whether we were playing video games, board games, eating together, dragon
dancing,\(^4\) or even just sitting around chatting,\(^5\) I always had a great time.

\(^2\)It is fun sometimes to entertain ridiculous ideas and think through them. Sometimes they don’t turn out
to be so ridiculous, and sometimes they lead to something sensible.

\(^3\)Alias: Stephen Ti-McHeck. The other half of the Smash Bros "dream team."

\(^4\)Not to be confused with lion dancing.

\(^5\)As an introvert, I was perhaps listening more than talking.
I would also like to thank my friends from before I joined WUSTL. They are regrettably far too many to list here, and I have fallen out of touch with most of them, especially these past 6 years. I would perhaps never have made it through to graduate school without having met them. I miss them all, and hope to meet them again; I have been too introverted and socially awkward to keep in touch, and that is something I need to work on. Despite this, I would like to especially thank Mason Blankenship, Jaehyuk Choi, David Clark, and John Kuc for keeping in touch throughout my time at WUSTL.

I would especially like to thank my girlfriend, Xiaojian Xu, for putting up with me. We entered the computer science PhD program at WUSTL at the same time, and she successfully defended her dissertation last year, so she definitely understands the time and stress related to this endeavor. Beyond that, she puts up with me daily, which certainly isn’t a walk in the park, especially of late. She is always incredibly supportive; I simply can’t find the words to express my gratitude, but I know without a doubt I am incredibly lucky to have had Xiaojian in my life.

Last, but not least, I would like to thank my family, both immediate and extended. I would especially like to thank my sister Shannon Oikawa, her husband Kenta Oikawa, my dad David Singer, and my mom Gail Singer. I appreciate their support throughout this process more than they could ever know. It is thanks to my family that I was able to develop from the problematic little boy who, although he liked conducting chemistry experiments using ingredients from the pantry, was a total mess at school who always lost his homework. It is especially thanks to them that I am who I am today, and that I have been able to come so far.

The work in this dissertation was funded in part by the following National Science Foundation (NSF) grants: CCF-1150036, CCF-1439062, CCF-1527692, CCF-1733873, CCF-1910568, CCF-1943456, and CCF-2107280.

Kyle Singer

Washington University in St. Louis
May 2023
Dedicated to my family, both immediate and extended.

In loving memory of Wallace Redman and Suzsanne Singer.
Because the performance of individual processor cores has been slowed by physical limitations that are difficult to overcome, the trend has been for processor manufacturers to instead increase the number of processing cores in a single chip. As a result, programmers must write parallel code to gain performance. While prior work on task-parallel platforms has simplified the process of writing throughput-oriented scientific applications, they have paid little attention to the types of workloads seen in modern interactive applications such as web services.

Interactive applications are programs which interact with users or other programs using input/output (I/O) operations and generally consist of tasks whose dependencies are more complex than scientific applications. Moreover, interactive applications have not only throughput requirements, but also responsiveness requirements for those tasks that interact with the user over I/O. Traditional task-parallel platforms, however, do not provide an efficient I/O interface, only focus on the execution time of the application as a whole, and either do not provide flexible enough abstractions for representing the dependencies of interactive applications or provide them in a manner that is not grounded in theory.

I contend that the broad notion of task parallelism, where the programmer expresses the logical parallelism of a program and the scheduling is handled by an underlying runtime, can
simplify how modern interactive applications are written. In this dissertation, I describe how a task-parallel platform can better support modern interactive applications and empirically demonstrate that the proposed solutions indeed lead to a task-parallel platform that is competitive with highly hand-tuned interactive applications using traditional means of parallelization. For encoding the complex parallel dependencies of such applications, this dissertation describes provably and practically efficient support for futures in task-parallel platforms. We then add provably and practically efficient I/O to meet the I/O-intensive demands of interactive applications. Next, we investigate prioritization of tasks, which allows us to bound task latencies and achieve responsiveness. Finally, we explore two efficient schedulers that can be used by task-parallel platforms to support interactive applications.
Chapter 1

Introduction

The performance of individual processor cores is no longer doubling every 2 years due to issues such as excess heat and power [118]. In fact, between 2012 and 2022, the Intel Core line of desktop processors saw an increase in clock speeds of less than 60% [53], which is a far cry from the nearly 3100% increase one would expect from Moore’s Law [120]. Instead, the trend has become to increase the number of processing cores per processor chip. Those looking to increase application performance must therefore look to parallelize their code.

Traditionally, parallel programs have been written using operating system (OS) provided abstractions for processor cores (e.g. threads) and synchronization (e.g. locks). At first glance, the use of these OS abstractions seems straightforward enough to achieve performance gains on multicore machines with little effort. Achieving good performance using these abstractions, however, is in reality an arduous task.

For those looking to perform throughput-oriented scientific computations, there is already several decades-worth of work on providing tools to simplify the task of writing performant parallel code without directly using OS abstractions. Task parallelism, for example, allows for straightforward programming models that allow the programmer to detach themselves from the logic for scheduling their code on multicore processors, allowing programmers to focus on core application logic.

Task-parallel platforms seek to provide lighter alternatives to thread creation that also allow programmers to write parallel code without worrying about processor oversubscription or scheduling concerns. Instead, task-parallel platforms adopt a straightforward syntax to express the logical parallelism of a computation, and provide a scheduler that decides how many threads to create and how to map the computation onto those threads. Some examples

---

While Moore’s Law actually states that the number of transistors per chip will double every 2 years, for decades this translated to a similar increase in clock speeds.
int fib(int n) {
    if (n <= 2) return n;
    int x, y;
    x = spawn fib(n-1);
    y = fib(n-2);
    sync;
    return x + y;
}

Figure 1.1: An example of the parallel naive recursive fibonacci function written using `spawn` and `sync`.

of task-parallel platforms include: Cilk dialects [38, 54, 85, 92, 106, 111], Parallel ML [22, 66, 77, 133, 142, 155], Threading Building Blocks (TBB) [90], Habanero dialects [28, 45, 88], OpenMP [129], the Java Fork/Join framework [105], MultiListp [78], NESL [36], TPL [110], Parallel Haskell [98, 103], and X10 [48, 148].

Much work has been done in providing provably and practically efficient schedulers for task-parallel code. A widely used scheduling algorithm is that of parsimonious work stealing [38, 44, 79, 102, 153], a scheduling technique that is popular because it provides a provably good execution time bound [23, 24, 39, 40] and is efficient at scheduling task-parallel programs in practice [70]. Even though work stealing and its execution time bound apply to programs that use flexible constructs such as futures [23, 24], a parsimonious work-stealing scheduler that provides provably good execution time bounds for these flexible constructs is both difficult to implement and incurs high scheduling overhead in practice.

Due to this high scheduling overhead, task-parallel platforms that support constructs such as futures have traditionally foregone provably good execution times in favor of practical performance. Prior works that want both provably good execution times and practical performance instead tend to implement rigidly structured forms of task-parallelism, such as fork-join parallelism, because they can be implemented in a manner that has both provably good execution times and practically good performance. Fork-join parallelism can be expressed with the `spawn` (fork) keyword to denote that the following function may execute in parallel with subsequent code (the `continuation`), and with the `sync` (join) keyword to
denote that all prior spawns in the function must complete before proceeding. Figure 1.1 shows an example of fork-join parallelism applied to parallelize the naive recursive fibonacci algorithm.

The traditional task-parallel platforms studied in prior works may work well for throughput-oriented scientific workloads, but they are ill-suited for modern interactive applications. Interactive applications are programs which interact with users or other programs using input/output (I/O) operations. They generally consist of tasks whose dependencies are too complex to be easily captured with fork-join semantics, and have different performance criteria than throughput-oriented scientific workloads. Although throughput is also important for modern interactive applications, the responsiveness of tasks that interact with users over I/O is also important.

I contend that the broad notion of task parallelism, where the programmer expresses the logical parallelism of a program and the scheduling is handled by and underlying runtime, can simplify how modern interactive applications are written. In this thesis, I describe how a task-parallel platform can better support modern interactive applications and empirically demonstrate that the proposed solutions indeed lead to a task-parallel platform that is competitive with highly hand-tuned interactive applications using traditional means of parallelization.

1.1 Motivation: Memcached Case Study

The Memcached server is an example of a modern interactive parallel application; it provides a networked in-memory key-value store for small objects. It is widely used as a caching backend for many performance-critical systems and is designed to be scalable. At a high level, it maintains a hash table; within each bucket, the objects are organized in (approximately) least-recently-used (LRU) order such that recently accessed objects are likely to be accessed more quickly. One can configure Memcached to support hash table expansion and writing of the cached content to an external persistent storage.

The original Memcached implementation is written in C and organized as follows. Upon startup, the main thread creates several background threads and a fixed number of worker

\footnote{In the model we are describing, an implicit \texttt{sync} would be added before returning from any function that uses \texttt{spawn}.}
void accept_new_connections(connection *c) {
    ...
    while (server_is_running()) {
        int conn_fd = accept(...);
        connection *new_conn = new_connection(conn_fd, ...);
        spawn drive_machine(new_conn);
    }
}

void drive_machine(connection *c) {
    while (server_is_running() && !c->closing) {
        command *cmd = read_command(c);
        response *resp = process_command(c, cmd);
        if (resp != NULL) {
            write_response(c, resp);
        }
    }
    close_connection(c);
}

Figure 1.2: Pseudocode showing an idealized control flow for handling client connections in Memcached. The spawn, at a high level, indicates that connections are handled by the drive_machine function in parallel.

threads to handle client connections. The background threads are designed to run periodically, and they perform tasks such as reorganizing the items in a bucket to maintain the LRU policy in the cache, assisting hash table expansion, crawling through the cache to collect statistics, and writing in-memory cached content to the external storage (if configured to do so). The main thread listens for connections requests from new clients; once a client is connected, a specific worker thread is assigned to handle its requests.

On the surface, it seems as though it would be straightforward to implement Memcached using spawn and sync. One would use spawn for each of the background threads and, instead of creating worker threads, one would use spawn to handle client connections in parallel, as shown in Figure 1.2. In reality, however, this would not work well. It turns out there are 3 major obstacles to using a traditional task-parallel platform with a modern interactive task-parallel platform such as Memcached: the rigid structure of fork-join parallelism, the
void command_handling_wrapper(connection *c, command *cmd) {
    response *resp = process_command(c, cmd);
    if (resp != NULL) {
        write_response(c, resp);
    }
}

void drive_machine(connection *c) {
    while (server_is_running() && !c->closing) {
        command *cmd = read_command(c);
        spawn command_handling_wrapper(c, cmd);
    }
    sync;
    close_connection(c);
}

Figure 1.3: A first attempt at parallelizing Memcached request processing. In this version, there is a race on c such that responses sent to the client can end up interleaved.

frequent use of I/O in interactive applications, and the performance criteria of interactive applications.

1.1.1 Obstacle 1: The Rigid Structure of Fork-Join

While Figure 1.2 shows code that allows for parallel processing of client connections, it does not allow for parallel processing of requests from the same client. This streamlined implementation makes it abundantly clear, however, that there is further room for parallelization by allowing multiple requests from the same client to be processed simultaneously.

A first obvious attempt to handle two (or more) requests $r_1$ and $r_2$ in parallel is shown in Figure 1.3. This implementation, however, has a bug. There is a race on the connection, c, which could lead to interleaving the responses sent back to the client. An interleaved message is worse than no response at all, as interleaving either results in messages that cannot be decoded by the client, or provides incorrect data to the client. This solution therefore needs to be rethought.

An obvious solution to the race on c would be to place a lock around sending the response, as shown in Figure 1.4. We now, however, have an ordering problem. Extending the Memcached server in this way allows sending the response to $r_2$ before $r_1$, which immediately breaks
void command_handling_wrapper(connection *c, command *cmd,
    lock *l) {

    response *resp = process_command(c, cmd);
    if (resp != NULL) {
        l->lock();
        write_response(c, resp);
        l->unlock();
    }
}

void drive_machine(connection *c) {
    lock my_lock;
    while (server_is_running() && !c->closing) {
        command *cmd = read_command(c);
        spawn command_handling_wrapper(c, cmd, &my_lock);
    }
    sync;
    close_connection(c);
}

Figure 1.4: A second attempt at parallelizing Memcached request processing. This attempt can result in responses arriving out-of-order, which would break compatibility with many Memcached client applications.
compatibility with clients that expect responses to arrive in the order requests were sent. Even if compatibility were not an issue, ordering responses according to the request order is often a desirable property as it greatly simplifies client application logic. Furthermore, this solution would be inefficient without support for locks built into the task-parallel platform.

This response ordering dependency shows that modern interactive applications have dependencies that cannot be expressed solely with spawn and sync. A task-parallel platform needs a more flexible abstraction to represent this and other dependencies that appear in modern interactive applications. Traditional task-parallel platform either do not support such a flexible abstraction, or they do so in a way that is not grounded in theoretical efficiency.

1.1.2 Obstacle 2: Frequent Use of I/O

Memcached, like any interactive application, makes frequent use of I/O. Each request that comes in from a client passes through multiple I/O handling states to read the entirety of a client request and to write back the response. These I/O operations, however, are not supported by traditional task-parallel platforms. Rather, whenever blocking I/O is encountered in a traditional task-parallel platform, the worker thread that encounters the I/O is forced to wait until the I/O operation completes.

Waiting for a blocking I/O operation to complete, however, is a waste of processing resources; for the majority of the waiting time, the blocked worker thread is not actively doing anything. During the time the thread is blocked on the connection for one client, many more requests may be coming in from other clients. Because the worker thread is blocked waiting for I/O, it cannot switch to process other requests, and thus overall performance would be poor.

Because traditional task-parallel platforms do not have efficient support for I/O operations, developers of interactive applications are unable to make use of these platforms. Instead, they are forced to turn to low-level threading and event-driven programming styles. In an event-driven interactive program, files are registered with an operating system-provided polling mechanism such as epoll [5] or a higher level wrapper such as libevent [2] and an event loop repeatedly checks for and handles I/O on the registered files.

In the case of Memcached, the developers used POSIX threads [89] and event-driven programming to implement asynchronous I/O via the libevent [2] library. In Memcached, a worker
Figure 1.5: The state machine for a network connection in memcached, generated using PlantUML [1]. The structure of this diagram is a testament to how complex the state machine can get when attempting to write efficient modern interactive applications in an event-driven style.

A worker thread time-multiplexes among multiple client connections at any given time via an event loop to handle events from multiple concurrent client connections. Each client, upon connecting to the server, is assigned to a particular worker, and a callback function is registered with the libevent library for events associated with that particular client connection. A worker thread never waits for a blocking I/O operation when handling a request. Instead, upon encountering a blocking I/O operation for a request \( r \), the worker thread returns from other connections, and only comes back to resume handling request \( r \) when the I/O can complete, which generates an event and causes the same callback function to be invoked again.

Due to this event-driven style, the control flow of the application is extremely complex. A given request \( r \) might require multiple I/O operations (with some computation interleaved between them). Thus, when the worker thread handling \( r \) encounters blocking I/O and leaves to handle other requests, it must leave enough bookkeeping information such that the proper
void drive_machine(connection *c) {
    bool stop = false;

    // Loop until we could move on to a different connection,
    // either because there is no available I/O, this connection
    // is closing, or too many commands have been consecutively
    // handled for this connection.
    while (!stop) {
        // c->state is one of the states from Figure 1.5;
        // all except ‘start’ and ‘stop’ are captured as cases
        switch (c->state) {
            case listening:
                // Accept new client connection
                ...
                case new_cmd:
                    // Transition to handling a new request
                    ...
        }
    }
}

Figure 1.6: Pseudocode showing the state machine code of the event handler used to handle I/O in memcached.
I/O operations can be identified when the I/O becomes unblocked. As a part of this process, for a single request \( r \), the client connection can pass through a number of I/O processing states, as shown in Figure 1.5. Therefore, the callback function effectively encodes a large state machine using a switch statement inside a loop as shown in Figure 1.6, where the states visited vary based on the request and at what point the request previously encountered blocking I/O. Such control flow obscures the application logic, as logic for handling a single request is scattered across different switch statement cases. This results in much more complicated code than the idealized code shown in Figure 1.2.

1.1.3 Obstacle 3: Performance Criteria

Many clients are expected to simultaneously connect to a Memcached server instance, each making frequent requests to, for example, get data from or insert data into the hash table. Like in throughput-oriented scientific applications, the number of requests per second (RPS), i.e. the throughput, a Memcached server can handle is important. The responsiveness in responding to individual requests, however, is also vitally important as the clients are often requesting data indirectly on behalf of a user interface that will update based on the result, so a user’s experience is impacted if the latency of a response is too high.

In other words, the foreground tasks in Memcached directly related to handling client requests need to finish quickly, whereas the background tasks do not. Crawling the cache for statistics is clearly something that could be delayed in favor of responding to a client request, for example.

Traditional task-parallel platforms, however, do not provide any means to prioritize one task over another. Rather, the scheduler of a traditional task-parallel platform would treat the task crawling the cache for statistics the same as the thread handling a client request. As such, traditional task-parallel platforms cannot provide a good guarantee on the responsiveness of the foreground tasks of the Memcached server.
1.2 Contributions

As Memcached exemplifies, traditional task-parallel platforms are ill-suited for interactive applications. Task parallelism can help, however, if we appropriately extend task-parallel platforms to deal with the above obstacles. To overcome the obstacles in Section 1.1, and to allow task-parallel platforms to provide support for modern interactive applications, I propose the following extensions in this dissertation: efficient support for a more flexible task-parallel abstraction, support for asynchronous I/O with a synchronous interface, and prioritization of tasks to provide good responsiveness.

1.2.1 A More Flexible Task-Parallel Abstraction

First, with a more flexible form of task-parallelism a task-parallel platform can express the dependencies of interactive applications. Rather than spawn and sync, an abstraction such as futures [27, 69, 79] can be used. The use of futures is similar to spawn and sync. The fcreate keyword, like spawn, can be used to indicate that a called function may execute in parallel with the continuation of the caller; fcreate, however, returns a future handle, which is an object in memory that represents the future task. Analogous to sync, ftouch can be used to wait for a future task; however, ftouch takes as an argument a single future handle on which to wait.

Using futures it becomes trivial to solve the race mentioned in Section 1.1.1, all while responding to requests in order. Figure 1.7 shows pseudocode for such an implementation. A task-parallel platform with support for a more flexible abstraction can thus be used to encode the more complex dependences of modern interactive applications, and in this case can do so without the need for locks.

The use of futures provides a flexible way to express parallelism and can generate arbitrary dependences among parallel subcomputations. The additional flexibility that futures provide comes with a cost, however. When scheduled using classic work stealing, a program with futures, compared to a program that uses only fork-join parallelism, can incur a much higher number of “deviations,” a metric for evaluating the performance of parallel executions that is closely related to practical scheduling overhead. All prior works assume a parsimonious
void command_handling_wrapper(connection *c, command *cmd,
    future *prev) {

    response *resp = process_command(c, cmd);
    if (resp != NULL) {
        // Wait for the prev task, if any, to write its response
        if (prev != NULL) ftouch *prev;
        write_response(c, resp);
    }
}

void drive_machine(connection *c) {
    future *prev = NULL;
    while (server_is_running() && !c->closing) {
        command *cmd = read_command(c);
        prev = fcreate command_handling_wrapper(c, cmd, prev);
    }
    close_connection(c);
}

Figure 1.7: Parallelization of Memcached request processing using futures. In this implementation, responses will not be interleaved and responses will be sent back to the client in the order they were received by the server.
work-stealing scheduler, however, where a worker thread (surrogate of a processor) steals work only when its local deque becomes empty.

In Chapter 3, we investigate an alternative scheduling approach, called proactive work stealing, where the workers perform proactive work stealing when handling future operations. We show that ProWS, for programs that use futures, can provide provably efficient execution time and equal or better bounds on the number of deviations compared to classic parsimonious work stealing. We implemented ProWS in a prototype system, and experimentally show that it achieves good practical performance.

1.2.2 Asynchronous I/O with a Synchronous Interface

As we saw with Memcached, interactive applications frequently rely on I/O operations that require few processing cycles but may incur significant latency to complete. In order to increase performance, when a particular thread of control is blocked on an I/O operation, ideally we would like to hide this latency by using the processing resources to do other ready work instead of blocking or spin waiting on this I/O. There has been limited prior work on hiding this latency and only one result that provides a theoretical bound for interactive applications that use I/O operations [121].

It turns out that futures serve as a nice abstraction for I/O operations. They allow programmers to write code with a synchronous I/O interface, while also allowing the runtime to efficiently switch execution contexts to overlap I/O communications with computation. This provides the performance of asynchronous I/O, but obviates the need for complex event-driven code to achieve high performance with frequent I/O use.

In Chapter 4, we propose a task parallel platform that supports I/O operations using the futures abstraction and a corresponding scheduler that schedules the I/O operations while hiding their latency. We provide a theoretical analysis of our scheduling algorithm that shows our algorithm provides better execution time guarantees than prior work. We also implemented the algorithm in a practically efficient prototype library and performed experiments that demonstrate the efficiency of our implementation.
1.2.3 Prioritization of Tasks

Like other modern interactive applications, Memcached has responsiveness requirements for a set of foreground tasks that interact with a user over I/O. As mentioned in Section 1.1.3, however, traditional task-parallel platforms treat all tasks in the system equally, and provide no guarantees on responsiveness. By incorporating the notion of priorities into tasks, however, the runtime scheduler of a task-parallel platform can be built to better support both responsiveness and throughput. Responsiveness guarantees when assigning priorities to tasks, however, require that there are no priority inversions where a high priority task waits on lower priority work.

In Chapter 5, we present a type system that allows for assignment of priorities to futures while ruling out priority inversions. While prior works[122, 123, 126] have done the same for functional languages, we extend this to imperative programs by adding state to the type system. We implement this type system and incorporate it into a task-parallel platform.

We then, in Chapters 6 and 7, present schedulers that can be used to provide good responsiveness for prioritized tasks. Chapter 6 presents a greedy algorithm, adaptive priority scheduling (APS), which provides bounds on the latency of individual tasks. We implement an approximation of APS and experimentally evaluate it to show that it appropriately prioritizes the responsiveness of higher priority tasks. Chapter 7 presents a scheduler that, although it may not be provably efficient, is easier to work with than APS and addresses several practical performance issues of APS.

1.3 Outline

The rest of this dissertation will proceed as follows. Chapter 2 discusses background material necessary for the following chapters. Chapter 3 covers proactive work stealing, adding support for futures to our task-parallel platform. Chapter 4 addresses I/O support, extending our task-parallel platform such that applications can efficiently interact with I/O sources. Chapter 5 describes a type system that allows state, allows assignment of priorities to futures, and statically rules out priority inversions. Chapter 6 discusses a greedy algorithm that provides execution time bounds for prioritized tasks instead of just the entire application.
Chapter 7 describes a practical scheduler for interactive applications. Finally, Chapter 8 concludes this dissertation with a summary.
Chapter 2

Preliminaries

2.1 Task Parallelism

Task-parallel platforms seek to provide lighter alternatives to thread creation that also allow programmers to write parallel code without worrying about processor oversubscription or scheduling concerns. Instead, task-parallel platforms adopt a straightforward syntax to express the logical parallelism of a computation, and provide a scheduler that decides how many threads to create\(^8\) and how to map the computation onto those threads.

When using `spawn` and `sync`, for example, whenever a spawning function \(F\) calls a function \(G\) preceded with the `spawn` keyword this denotes that the spawned child \(G\) can run in parallel with the code between that `spawn` and the next `sync` (i.e. the continuation of \(F\)), as shown in Figure 2.1 (a). Any functions spawned in \(F\) prior to a `sync` statement in \(F\) must complete before the program can progress past that `sync` statement. Implicitly, every function that contains a `spawn` executes a `sync` before returning from the function.

A second, more flexible abstraction that can be used in task-parallel programs is that of futures [27, 69, 79]. The use of futures is similar to that of `spawn/sync`. Like `spawn`, when a function \(F\) precedes a call to function \(G\) with the `fcreate` keyword, this denotes that \(G\) can run in parallel with the continuation of \(F\). In this example, we refer to \(G\) as the future task. The `fcreate` keyword, as shown in Figure 2.1 (b) however, returns a future handle: an object in memory that represents both the status of the future task and, upon completion, its result. Analogous to the `sync` keyword, `ftouch` can be used to wait for the completion of future tasks. However, an `ftouch` operates on a single future rather than all future tasks previously generated in the function. Further, `ftouch` is not implicitly inserted

\(^8\)Generally a constant number of threads per core.
Figure 2.1: The parallel naive recursive fibonacci function written using (a) spawn/sync and (b) fcreate/ftouch.

before returning from $F$, allowing future tasks to continue to execute even after the function $F$ returns.

Given these constructs, one can derive a **serial elision** of the parallel code by removing spawn, removing sync, replacing fcreate with a simple function call, and replacing ftouch with the return value of the corresponding fcreate function call. The serial elision of a task-parallel program $P$ is effectively a serial version of the parallel program $P$, and is semantically equivalent to running $P$ on a single core.

The use of spawn or fcreate in a task-parallel platform does not dictate parallelism, it merely indicates the potential for parallelism. It is up to the underlying task-parallel platform whether or not to realize that parallelism. Furthermore, these keywords are much lower overhead than thread_create, so the programmer can use them more liberally than thread_create. Because they do not create new threads, neither does the programmer need to worry about oversubscribing the system when using spawn or fcreate.

---

9An unrestricted use of futures may not have a valid serial elision. However, when restricted to a "structured" use of futures, as discussed in Section 3.1, there is always a valid serial elision.
2.2 Modeling Task Parallelism

2.2.1 Traditional Task Parallelism

The executions of task-parallel programs are often represented using the directed acyclic graph (DAG) model [40], as shown in Figure 2.2. In the DAG model, each node represents a series of instructions without parallel control (e.g. \texttt{spawn} or \texttt{sync}) that takes unit time to execute, and each edge represents a dependence between two nodes. As shown in Figure 2.2, for example, the instructions represented by node 11 cannot execute until after all instructions in nodes 5, 8, and 10 have executed. When using only \texttt{spawn}/\texttt{sync}, the computation forms a \textit{series-parallel (SP) DAG} with a single source and a single sink. An SP DAG can be constructed recursively using the following rules:

- \textbf{base case:} a single strand (a node) is an SP DAG;

- \textbf{series composition:} given two SP DAGs $G_1$ and $G_2$, compose them in series by adding an edge between the sink of $G_1$ and the source of $G_2$;

- \textbf{parallel composition:} given two SP DAGs $G_1$ and $G_2$, compose them in parallel by adding a new source $s$ and a new sink $t$, with edges from $s$ to the sources of $G_1$ and $G_2$, and edges from the sinks of $G_1$ and $G_2$ to $t$.

When futures are also used, the resulting DAG is a set of SP DARGS with non-series-parallel edges formed by the use of \texttt{fcreate} and \texttt{ftouch}. That is, if $F$ spawns $G$ via \texttt{spawn}, then the SP DAG of $G$ if part of the SP DAG of $F$. On the other hand, if $F$ spawns $G$ via \texttt{fcreate}, then $F$ and $G$ are independent SP DAGs, with the first strand of $G$ being the source of a separate SP DAG and the last strand of $G$ being the sink.

Given that each node of a computation DAG executes in a unit amount of time, we can determine the \textit{work} of the computation ($W$) by counting the number of nodes in the DAG. This corresponds to the amount of time it would take to execute the computation on a single processor, and is also commonly denoted as $T_1$ in the literature. We can also determine the \textit{span} of the computation ($S$) from the length of the longest path from the source of the DAG (e.g. node 1) to its sink (e.g. node 18). The span, also commonly denoted as $T_\infty$, corresponds
Figure 2.2: A directed acyclic graph (DAG) representing a possible task parallel program. Nodes are numbered in their serial execution order, a left-to-right depth first traversal by convention.

to the minimum amount of time the computation can be completed in if executed on an infinite number of processors.

Prior works on scheduler analysis typically assume that a DAG consists of nodes with at most two outgoing edges (e.g., [10, 23, 40, 84, 143]). One can transform the dag by replacing a node with multiple outgoing edges into a chain of nodes, each with two outgoing edges. This becomes relevant if a future handle can be touched multiple times. Provided that the number of touches per handle is constant, however, such a transformation does not adversely impact the span.

2.3 Interactive Task Parallelism

Similar to traditional task-parallel programs, interactive task-parallel programs can be represented with DAGs [122, 123, 125, 126, 140]. While in traditional task-parallel programs the goal is generally to consider the execution time of the program as a whole, in interactive applications we want to be able to bound the execution time of individual tasks in the
program, so nodes are additionally assigned priorities. We can define a task, $\tau$, in the DAG model as an SP DAG with a single source and sink where all nodes have the same priority, $l$. For example, the DAG in Figure 2.3 contains two tasks: 1 at High priority consisting of nodes 3 through 12, and 1 at Low priority consisting of nodes 1 through 18. In relation to our programming model of spawn/sync and fcreate/ftouch, we consider a fcreate as always creating a new task whereas spawn only creates a new task if the spawned child, $G$, is of a different priority than that of the spawning function, $F$.

2.4 Scheduling Task Parallelism

In the works described in this thesis, we will be considering dynamic scheduling algorithms that do not know the full computation DAG a priori. This means that the scheduling algorithm only has knowledge of which nodes are currently enabled and ready to execute, that is the set of nodes for which there are no ancestor nodes with unexecuted instructions. It is the job of a scheduler to determine how to best map the computation to processing cores in a way that respects the dependences specified by the parallel constructs. We say that $u$ is an immediate predecessor of $v$ if there is an edge from $u$ to $v$; in this scenario
we refer to $v$ as the **immediate successor** of $u$. A node $v$ is **ready** when all its immediate predecessors in the DAG have executed, and only ready nodes can be executed.

### 2.4.1 Traditional Task Parallelism

One class of online scheduler that has good theoretical execution time bounds for task-parallel applications is that of **greedy schedulers**. The defining characteristic of a greedy scheduler is that it greedily keeps processors busy such that no processor is ever idle if there is an available ready node that can be executed. When executing a computation with $W$ work and $S$ span on $P$ processors, a greedy scheduling algorithm executes the computation in time $T_P = O(W/P + S)$ [42, 75, 76]. Because $W/P$ and $S$ are both lower bounds on the execution time of the computation on $P$ processors, this is asymptotically optimal. However, this does not take into account the overhead inherent in greedy schedulers; to keep all processors busy, there must be a heavily contended shared data structure that processors can use to find nodes to execute.

A more widely used scheduling algorithm is that of parsimonious work stealing [38, 44, 79, 102, 153], a scheduling technique that is popular because it provides a provably good execution time bound [23, 24, 39, 40] and is efficient at scheduling task-parallel programs in practice [70]. In parsimonious work-stealing algorithms the nodes of the computation DAG are dynamically load balanced across **worker** threads, which are surrogates of the processors in the system. Each of the workers maintains a doubly-ended queue (deque) of ready nodes. Whenever a worker enables nodes by executing their last remaining ancestor, it pushes those nodes onto the bottom of its own deque. The worker continues executing by popping the bottom node off of its deque. If, however, the deque becomes empty, the worker turns into a **thief** and selects another **victim** worker from which it will attempt to steal an ready node. When stealing nodes from another worker, the thief operates on the top of the victim’s deque to reduce contention. When executing a computation with $W$ work and $S$ span on $P$ processors, parsimonious work stealing achieves an expected execution time $T_P = O(W/P + S)$ [23, 24, 39, 40], achieving the same asymptotic bound as that of greedy schedulers.

As is customary, we shall assume that the spawned function or future task is always the left child of the spawn node and the continuation strand is the right child. Thus, a **serial**
(one-worker) execution of a computation DAG follows the left-to-right depth-first traversal. This also means that we assume eager evaluation of futures, where the future task is always evaluated before the continuation of fcreate under serial execution.

2.4.2 Interactive Task Parallelism

In order to guarantee good execution times for latency-sensitive tasks it is important for the scheduler to respect priorities, preferentially scheduling the ready nodes at the highest available priority level. Only in recent years have researchers begun to investigate how to do this in the setting of task parallelism [122, 123]. These prior works propose lambda calculi representing functional languages as well changes to the traditional graph-based cost models to allow for reasoning about performance in the presence of priorities, and they analyze the performance guarantees of schedulers that follow the prompt scheduling principle. In a prompt scheduler, processors are kept busy so long as there are nodes available to execute (greedy), while also respecting the priority of nodes (prompt). In other words, a prompt scheduler will not schedule lower priority nodes so long as there are sufficient higher priority ready nodes to keep all processors busy, and no processor remains idle so long as there are ready nodes available. Much like the traditional greedy scheduler, however, prompt schedulers suffer from the need for frequent accesses to a shared data structure to find work. Moreover, prompt schedulers may additionally require frequent preemption in order to ensure that the highest priority work is being executed. As such, there is no known practical scheduler that faithfully follows or closely approximates the prompt scheduling principle.

To bound the execution time of an individual task $\tau$ executing at priority level $l$ we then need separate terms from that of the traditional work and span. The first term we need is the competitor work of the task $(W_{\tau})$, which can be defined as the count of the nodes in $\tau$ and all nodes that are logically parallel with $\tau$ that are of an equal or greater priority than that of $\tau$. As an example, in Figure 2.3 the High priority task has competitor work 10 (the count of High priority nodes), whereas the Low priority task has competitor work 18 (the count of nodes in the High priority task and the Low priority task).

Because we are trying to bound individual task execution times, we need to define an alternative to the traditional span, the $a$-span, $S_{\tau}$. The $a$-span of a task $\tau$ can be defined as a longest path to the sink of the sub-DAG of $\tau$ that does not include an ancestor of the
source of said sub-DAG. The $a$-span of the High priority task in Figure 2.3, then, passes through only high priority nodes and is 6, whereas the Low priority $a$-span is 9, passing through the High priority task’s sub-DAG.

With a prompt scheduler, given a task $\tau$ with competitor work $W_\tau$ and $a$-span $S_\tau$ executing on $P$ processors, the execution time of $\tau$ is $T_P(\tau) = O(W_\tau/P + S_\tau)$ [122, 123]. This bound only holds so long as no high priority task waits on a lower priority task.
Chapter 3

Futures and the Proactive Work-Stealing Scheduler

The use of future constructs provides a flexible way to express parallelism. Similar to fork-join parallelism, one can spawn off a future task that executes logically in parallel with the continuation of the spawning statement. Unlike fork-join parallelism, however, the termination of a future task is not restricted to a lexical scope. Rather, the spawn statement returns a future handle that can be used to retrieve the value produced by the future task. When the handle is touched, the control is blocked until the corresponding future task terminates and returns a value.

The additional flexibility of futures allows one to write a wider range of parallel programs and/or provide a higher level of parallelism beyond what can be specified using only fork-join parallelism. For instance, Blelloch and Reid-Miller [37] show that one can asymptotically reduce the span of various tree operations using parallel futures. Since its proposal in the 70s [27, 69], future constructs have been incorporated into various task parallel languages and platforms [45, 47, 48, 66, 79, 102, 114, 144, 149], including the C++11 standard [94].

Even though work stealing and its execution time bound apply to programs that use futures [23, 24], the flexibility of futures can incur additional costs. Specifically, prior works [10, 84, 143] show that using work stealing with futures, compared to work stealing with only fork-join parallelism, can incur a much higher number of “deviations,” a better metric for evaluating the performance of parallel executions.

As articulated by Spoonhower et al. [143], the number of deviations provide a better metric for evaluating performance bounds because it is highly correlated to the additional cache misses and scheduling overheads of parallel executions. Informally, a deviation occurs during a parallel execution when a processor executes an instruction whose ordering in the instruction
stream deviates from that of the serial execution. A deviation forces the scheduler to perform additional bookkeeping to keep track of the events that cause the deviations. Moreover, the number of deviations can be used to bound the extra cache misses incurred on the private caches during parallel executions (as first shown by Acar et al. [10]) — intuitively, the bound holds by considering each deviation to execute with an empty private cache.

Given a computation that employs only fork-join parallelism, Acar et al. [10] show that the expected number of deviations\(^{10}\) incurred by a work-stealing scheduler is \(O(PS)\). In contrast, given a computation that employs \(k\) future operations, Spoonhower et al. [143] show that the expected number of deviations incurred is \(O(kS + PS)\), an additional \(k\) multiplicative factor. More recently, Herlihy and Liu [84] show that if futures are used in a restricted fashion one can bound the number of deviations to be \(O(PS^2)\).

Although these bounds suggest that futures can indeed incur much higher overhead than strict fork-join parallelism, all these prior works assume a parsimonious work-stealing scheduler, where each worker maintains a single deque and only steals to load balance when its deque becomes empty. Due to the parsimonious nature of work stealing, each future touch can lead to \(O(S)\) number of deviations, contributing to the \(O(S)\) multiplicative factor in the deviation bound.

In this chapter we propose proactive work stealing: whenever a worker thread encounters a future touch that is not ready, it suspends the execution of its current task and tries to find something else to do. By proactively suspending the computation instead of expanding what’s already on the deque, one can minimize the deviations and their corresponding scheduling overhead and cache misses.

We show that the proposed proactive work-stealing algorithm, called ProWS, can provide a comparable execution time bound to the parsimonious variant, as well as equal or better bounds on the number of deviations for programs that use futures. Given a computation that employs futures with \(W\) work and \(S\) span, the proposed algorithm executes the computation on \(P\) processors in \(O(W/P + S \lg P)\) time, which is asymptotically comparable to the parsimonious version (except for the \(\lg P\) overhead on the span term). For structured use of futures, where the future is single-touch with no races on the future handle, the algorithm incurs \(O(PS^2)\) number of deviations, the same as the bound for the parsimonious variant. For

\(^{10}\)Acar et al. refer to deviations as drifted nodes in [10].
general use of futures, where the only restrictions are a constant number of touches per future and deadlock-freedom during one-worker execution,\textsuperscript{11} the algorithm incurs $O(m_k S + PS \log P)$ deviations, where $m_k$ is the maximum number of future touches that are logically parallel. This bound is better than the bound for the parsimonious variant if $m_k = \Omega(P \log P)$ and is smaller than $k$, the total number of touches in the entire computation; these assumptions hold true for all the benchmarks examined. Because proactive and parsimonious work stealing behave the same for programs that utilize only fork-join parallelism, they have the same bounds for such programs.

We have implemented a work-stealing runtime system called Cilk-F, extending the task parallel runtime system of Cilk Plus [92] to incorporate support for parallel futures scheduled using ProWS. Even though futures have been incorporated into various task parallel platforms [45, 47, 48, 66, 79, 102, 114, 144, 149], Cilk-F is the first provably efficient proactive work-stealing runtime that supports futures. An interesting design question arises when considering the implementation of the “cactus stack” [106] in supporting futures, which we discuss in more detail in Section 3.4. No prior stack space bound exists for parallel futures when scheduled using parsimonious work stealing. Interestingly, when scheduled using proactive work stealing, it’s possible to bound the stack space, albeit loosely.

We empirically evaluate Cilk-F with nine benchmarks and compare it to Cilk Plus. For benchmarks where the use of futures does not provide additional parallelism, Cilk-F performs comparably to Cilk Plus, indicating that the additional $\log P$ term in front of the span does not adversely impact performance in practice. For benchmarks where the use of futures provides additional benefit, Cilk-F can obtain better speedup. Nevertheless, future operations do incur higher scheduling overhead compared to pure fork-join parallelism, and we empirically analyze this overhead.

Contributions

In summary, this chapter describes the following contributions:

\textsuperscript{11}Prior work by Spoonhower et al. [143] assumes single-touch per future, but constant number of touches does not change their bound.
• We propose ProWS, a proactive work-stealing algorithm for scheduling computations with futures (Section 3.2). Even though the algorithm is stated in terms of futures, the algorithm works for computations with general synchronization patterns.

• We show that ProWS provides equal or better bounds on the number of deviations than the parsimonious variant (Section 3.3).

• We describe the implementation of Cilk-F, the first provably efficient proactive work-stealing runtime system that supports the use of futures (Section 3.4). We discuss in detail how Cilk-F supports futures in the cactus stack and the resulting stack space usage bound.

• We empirically evaluate Cilk-F and show that ProWS can be implemented efficiently (Section 3.5).

This work was completed with the help of Yifan Xu, who helped with development of the algorithm and with bounding the number of deviations incurred by the proactive work-stealing scheduler described in this chapter. This chapter originally appeared in: K. Singer, Y. Xu, and I.-T. A. Lee. Proactive work stealing for futures. In Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, PPoPP ’19, pages 257–271, New York, NY, USA, 2019. ACM.

3.1 Preliminaries

In this chapter, we consider two distinct types of futures. A structured use of futures imposes the following restrictions: 1) single touch, meaning that only a single ftouch is invoked on each future handle, and 2) no race on a future handle, meaning that there is a directed path between a future spawn node to the local parent of its corresponding touch. Note that this restriction is the same as prior work [84] and does not preclude a future task to execute in parallel with the function that performs its touch before the ftouch keyword. It simply means that the spawning of the future (which writes to the future handle) must be in series with the invocation of the corresponding ftouch (which reads the future handle). A general use of futures imposes the following restrictions: each future is touched a constant number of times and all the join edges are forward pointing, namely, a fcreate is always before its corresponding ftouch in a serial execution.
3.2 Proactive Work-Stealing

This section describes the proactive work stealing algorithm, which we shall refer to as ProWS in the rest of the section. We will refer to the original parsimonious algorithm analyzed by Arora et al. [23] (described in Section 2.4.1) as ABP.

The main distinction between ProWS and ABP is as follows. When a worker executes a \texttt{ftouch}, the associated future task may not be ready, so executing the \texttt{ftouch} does not enable the subsequent future join node. With ABP, this simply falls under the case of enabling zero nodes, and the worker continues execution by popping off the bottommost node to execute next. ProWS handles the execution of \texttt{ftouch} differently. If its future task is not ready, the worker suspends the entire deque and tries to find work elsewhere. An important consequence of such behavior is that there can be more than $P$ deques in the system, where $P$ is the number of workers.

In ProWS, suspended deques are still stored in a distributed fashion, thus each worker now manages a single active deque that it actively works on and a set of stealable deques that are not being actively worked on but contains ready nodes. When stealing, once a victim is chosen, a thief can steal from any deque that belongs to the victim with equal probability (including its active deque).

Data Structures Used

We shall first discuss the data structures used by the algorithm. Each deque supports the following operations:

- \texttt{popTop}: remove and return the node from the top;
- \texttt{popBottom}: remove and return a node from the bottom;
- \texttt{pushBottom}: insert a node onto the bottom;
- \texttt{pushBottomImplicit}: insert a node onto the bottom of the deque and mark the node as suspended; and
- \texttt{isEmpty}: return true if there are no ready nodes in the deque (but may contain one suspended future join node).
Just as in ABP, we assume that multiple workers can make calls to a deque concurrently; if more than one worker tries to pop the same element off the deque, one of them succeeds and the other one fails in a constant number of time steps.

Throughout the lifetime of a deque, it can be in one of the following four states:

- **active**: it is actively been worked on by a worker;

- **suspended**: it is suspended due to a `ftouch` call; every node in the deque is ready, except for the bottommost node, which is the corresponding suspended future join node;

- **resumable**: it contains only ready nodes, but it is not actively being worked on by a worker; and

- **muggable**: similar to a resumable deque, except that the entire deque can be stolen and resumed.

These states are exhaustive, and a deque can only transition: 1) from active to suspended due to execution of a `ftouch` call, 2) from suspended to resumable due to termination of the future task enabling the join node at the bottom, 3) from resumable to active if the worker who finishes the future task has an empty deque and resumes one of the now-resumable deques suspended with the future handle; 4) from resumable to muggable after a thief steals from it once, and 5) from muggable to active when a thief mugs it and resumes its execution. Since a resumable deque transitions to muggable once it is stolen from, only its top item may be stolen before transitioning. If a thief steals into a muggable deque, it takes the entire deque and resumes its execution from the bottommost node.

The stealable deques belonging to a worker are maintained as a set. Each future handle also maintains a deque set with references to suspended deques, allowing any deques suspended with the handle to be resumed when the future task completes. A deque set supports the following operations:

- **add(deq)**: add deque `deq` into the set;

- **remove(deq)**: remove deque `deq` from the set;

- **removeRandom()**: remove and return a deque from the set, chosen uniformly at random; and
- `pickRandom()`: return a reference to a deque in the set, chosen uniformly at random (but does not remove it).

We assume that one can make concurrent calls to a given set, and an operation will finish in constant amortized time. When operating on the stealable set of a worker, the worker is always chosen uniformly at random among the $P$ workers. Thus, the contention can be resolved in a constant number of time steps in expectation (e.g., see lemma 6 in [40]). In practice, a set can be implemented as a growable array (performing array doubling when necessary), which maintains a constant amortized insertion cost.

The Algorithm

Algorithm 1 shows the main scheduling loop for ProWS and its helper functions. Ignoring the special handling of future operations in lines 25–32, ProWS behaves the same as ABP. Each worker starts out with one active deque; it operates off the bottom of the deque (line 18 and lines 22–24) and steals when it runs out of work to do (lines 19–20). A worker, when enabling two nodes, pushes the right node (i.e., continuation) first (line 23) and then the left node (i.e., the spawned task) (line 24), which means that the left node gets executed next.

Future operations are handled differently. If the execution of this strand terminates with `ftouch` (lines 25–30) and the corresponding future task $f$ has not terminated, `ftouch` enabled zero nodes. The worker then pushes the corresponding future join node $j$ (the immediate successor of $n$) onto the bottom of the deque via `pushBottomImplicit` (line 27) and suspends the deque (line 28). The reference to the suspended deque is stored with the future handle of $f$ (line 29) and the worker’s active deque is set to `null` (line 30). It will be set to something else after the steal. On the other hand, if the executed strand terminates with `fput`, then its corresponding future task $f$ has terminated and all suspended deques stored with $f$ can now be resumed (lines 31–35). At this point, if the worker executing `fput` has an empty active deque, it will set its active deque to one of the suspended deques, $d_s$, stored with the future handle and resume execution at the bottom of $d_s$ (lines 33–35).

The implementation of `suspend` is shown in lines 1–6. Since ProWS may potentially suspend many deques, it takes extra steps to ensure that the number of stealable deques are roughly balanced among workers. Instead of suspending with the current worker $w$, it chooses a
Algorithm 1 The main ProWS scheduling loop and helper functions

Function suspend\( (\text{deq}) \) \( // w \) is the executing worker
1 \( \text{deq.status} \leftarrow \text{SUSPENDED} \)
2 \textbf{if} \( \text{dep.isEmpty()} \) \textbf{then} \( \text{deq.worker} \leftarrow \text{null}; \)
3 \textbf{else}
4 \( v \leftarrow \text{ChooseRandomVictim()} \); //can include \( w \) itself
5 \( v.stealable.add(\text{deq}) \)
6 \( \text{deq.worker} \leftarrow v \)

Function setToActive\( (\text{deq}) \) \( // w \) is the executing worker
8 \textbf{if} \( \text{deq.worker} \) then
9 \( \text{rebalanceStealables(\text{deq.worker})} \)
10 \( \text{deq.worker.stealable.remove(\text{deq})} \)
11 \( \text{deq.worker} \leftarrow \text{null}; // \text{deq} \) is not in any stealable set
12 \( \text{deq.status} \leftarrow \text{ACTIVE} \)
13 \textbf{if} \( \text{w.active is not null} \) \textbf{then} \( \text{freeDeque(w.active)}; \)
14 \( \text{w.active} \leftarrow \text{deq} \)

Function SchedulingLoop\( w \) \( // w \) is the executing worker
16 \textbf{while} computation is not done \textbf{do}
17 \( n \leftarrow \text{null}; // n \) points to next strand to execute
18 \( // \text{w.active} \) points to either \text{null} or its active deque
19 \textbf{if} \( \text{w.active is not null} \) \textbf{then} \( n \leftarrow \text{w.active.popBottom()} \);
20 \textbf{if} \( n \) is \text{null} \textbf{then}
21 \( \text{steal()} ; // \text{steal} \) returns when work is found
22 \textbf{else} // execute \( n \)
23 \textbf{if} \( \text{right is not null} \) \textbf{then} \( \text{w.active.pushBottom(right)} ; \)
24 \textbf{if} \( \text{left is not null} \) \textbf{then} \( \text{w.active.pushBottom(left)} ; \)
25 \textbf{//special case: } \( f \) is a future handle
26 \textbf{if} \( n \) terminated with \( f.\text{fcreate()} \) \textbf{then}
27 \textbf{if} \( f \) is \text{not ready} \textbf{then}
28 \( \text{w.active.pushBottomImplicit(j)} \)
29 \( \text{suspend(w.active)} \)
30 \( f.\text{suspended.add(w.active)} \)
31 \( \text{w.active} \leftarrow \text{null} \)
32 \textbf{else if} \( n \) terminated with \( f.\text{fput()} \) \textbf{then}
33 \textbf{Mark every deque in } \text{f.suspended} \text{ RESUMABLE}
34 \( \text{markSuspendedResumable(f.suspended)} \)
35 \textbf{if} \( \text{w.active is empty} \) \textbf{then}
36 \( \text{deq} \leftarrow \text{f.suspended.removeRandom()} \)
37 \( \text{setToActive(deq)} \)
target worker \( v \) uniformly at random (which can include \( w \) itself) and suspends the deque with \( v \). The reference to \( v \) is stored with the suspended deque so that when the deque gets resumed it can be removed from worker \( v \)'s stealable set.

If the suspended deque contains no ready nodes (line 2) we don’t store the deque in any worker’s stealable set, as it has nothing to be stolen. Such a deque, once it gets resumed, is inserted into the stealable set of a worker chosen uniformly at random (by `markSuspendedResumable` in line 32).

Finally, a key thing to note in `setToActive` is that it invokes `rebalanceStealables` (line 9), which is invoked whenever \( w \) is about to remove a deque from \( v \)'s stealable set — it randomly chooses another victim \( v' \); if \( v = v' \), \( w \) is done; otherwise \( w \) moves a stealable deque from \( v' \) to \( v \) if \( v' \) has one. Section 3.3 explains why we do such a rebalance.

Algorithm 2 shows the implementation of the steal protocol that a worker \( w \) invokes when its deque becomes empty or after it loses its deque due to suspension. The steal function performs steal attempts until \( w \) finds work successfully.

When stealing, \( w \) chooses a victim \( v \) uniformly at random (line 37, which again includes \( w \)) and chooses a deque uniformly at random among \( v \)'s deques (line 38). If the chosen deque, \( deq \), is muggable, \( w \) takes the whole deque and sets \( w \)'s active deque to \( deq \). Otherwise, \( w \) steals from the top (line 43). After `popTop`, if the deque runs out of ready nodes, it is
removed from v’s stealable set and possibly destroyed if there isn’t even a suspended future
join node at the bottom, which may happen when stealing from a resumable deque (line 45).
Moreover, rebalanceStealables is invoked again. If the deque is resumable and not
empty, it is marked as muggable (line 48). After a successful steal, w may need to allocate a
new deque (lines 50 and 51).

3.3 Performance Bounds for ProWS

This section analyzes ProWS to show that, 1) for a computation with W work and S span, it
executes the computation in expected time $O(W/P + S \log P)$, and 2) the number of deviations
is bounded by $O(PS^2)$ for a program that uses structured futures and $O(mkS + PS \log P)$ for
a program that uses general futures.

Before we analyze the bounds, we first show that, at any point during the execution, the set
of stealable deques are roughly evenly distributed across workers, which we utilize when we
discuss the bounds. We use the following lemma on the classic balls-into-bins problem, which
is not hard to show (see e.g., [119, Chp. 5]):

Lemma 1. When m balls are thrown independently and uniformly at random into n bins,
the probability that the maximum load is more than $\frac{m}{n} + O(\log n)$ is at most 1/n. Similarly,
the probability that the minimum load is less than $\frac{m}{n} - O(\log n)$ is at most 1/n.

Lemma 2. Given P workers and DS number of stealable deques in the system, with probability
1 – o(1) each worker has at most $DS/P + O(\log P)$ deques.

Proof Sketch. One can model the number of stealable deques per worker as the classic
balls-into-bins problem, where the workers are modeled as bins and the stealable deques are
modeled as ball tosses. Our process also includes muggings, however, which changes the size
of the stealable sets, and thus the analysis requires additional care.

We model the entire process as two separate ball-toss processes: a deque-suspension process,
where a suspended deque is modeled as a ball toss into a randomly-chosen bin (worker to
leave the deque with), and the deque-removal process, where removing a deque is also
modeled as a ball toss into a randomly-chosen bin (worker to remove the deque from). Then
the size of a given stealable set is the number of balls resulting from the deque-suspension

process minus the number of balls resulting from the deque-removal process. The upper and lower bounds on the maximum and minimum loads in Lemma 1 thus give us the desired bound.

It is not hard to see that the worker from the deque-suspension process is chosen uniformly at random. What remains to be shown is that the same holds true for the deque-removal process. There are two ways a deque can disappear from a stealable set: 1) a worker takes the whole deque to resume it (lines 35 and 41); or 2) a deque becomes empty after it is stolen from (lines 44–46). In both cases, we always invoke `rebalanceStealables`: if we are removing a deque from $v$, we randomly choose a victim $v'$ from which to move a stealable deque to $v$. If $v'$ has a stealable deque to move to $v$, it’s as if we removed the deque from $v'$. If $v'$ does not have a stealable deque, it’s as if we first moved the deque to $v'$ and then removed it. Pretending to move a deque from $v$ to $v'$ is ok, because $v$ has a larger stealable set at the moment, and doing so simply balances the load from a more-loaded worker to a less-loaded one. Even though such load balancing is conditioned on $v'$ not having any deque, doing so does not hurt the bound.

\[\square\]

### 3.3.1 Bound on Execution Time

Our time bound analysis follows a similar structure to the analysis done in [23] and [152]. We separately bound the number of time steps devoted to various activities: work, steal attempts, and muggings. By bounding how many time steps each activity takes, the final bound arises by summing all the time steps divided by $P$, the number of workers used. Obviously, the total work is bounded by $W$ time steps.

It remains to bound the number of steal attempt and mugging operations, each of which takes a constant number of time steps. In the original work stealing analysis by Arora et al. [23, 24], henceforth referred to as ABP, steal attempts are bounded by a potential function argument that states the following. Assuming there are $P$ deques in the system, after $O(P)$ steal attempts, the overall potential decreases by a constant fraction. This is because the topmost node in a deque contributes to a constant fraction of the overall potential among nodes within the deque.
More formally, the following lemma is a straightforward generalization of lemma 7 and 8 in ABP [23] which we utilize:

**Lemma 3.** Let $\Phi_i$ denote the potential at time $t$ and say that the probability of each deque being the victim of a steal attempt is at least $1/X$. Then after $X$ steal attempts, the potential is at most $\Phi(t)/4$ with probability at least $1/4$.

Effectively, this lemma says that the number of steal attempts is at most $O(XS)$, because the potential function is a function of $S$. For ABP, it is always the case that $X = P$, leading to a steal attempts bound of $O(PS)$.

The ABP analysis cannot be applied to ProWS directly because 1) ProWS can have more than $P$ deques in the system, and 2) a thief stealing into a muggable deque will resume the bottommost node in the deque instead of the topmost one, which may not contain sufficient amount of potential.

To resolve issue 1), we apply a similar technique to Utterback et al. [152] and divide the computation into two types of phases: a **steal-bounded phase** when there are at most $2P$ stealable deques, and a **work-bounded phase** when there are more than $2P$ stealable deques. During a steal-bounded phase, by Lemma 2, we know each worker has at most $O(\lg P)$ deques, leading to a steal attempts bound of $O(SP\lg P)$ by Lemma 18. During a work-bounded phase, the total number of deques in the system is more than $3P$. However, because there are many deques in the system distributed roughly equally among workers, steal attempts are likely to succeed, each followed by a unit of work. Thus, we can bound the steal attempts by $O(S)$ during a work-bounded phase. Overall, this leads to an execution time bound of $O(W/P + S\lg P)$.

We still need to resolve issue 2), and in addition bound the time spent on muggings. Recall that in ProWS, we enforce that every resumable deque has to be stolen from once before it becomes muggable. This may seem counter-intuitive — why not simply resume the deque from the bottom if it is already resumable? This steal-before-mug ensures that for each mugging there is a corresponding successful steal on the same deque to amortize against. Doing so prevents the worst case scenario where a deque with a high-potential node on top repeatedly becomes resumable and mugged but never stolen from. This scenario would prevent us from bounding steal attempts that lead to a successful mugging.
Thus, we can also bound the time steps spent on mugging against steals, resulting the following time bound:

**Theorem 4.** Consider a computation with $W$ work and $S$ span. The expected execution time is $O(W/P + S \log P)$.

### 3.3.2 Bounds on Deviations

We first define some notations. Given a computation DAG $G$, we say that $u$ is a *predecessor* of $v$ and $v$ is a *successor* of $u$ iff there is a directed path from $u$ to $v$.

We make the following assumption. Let $u$ be a node with two outgoing edges, meaning that $u$ can be a spawn node, a future spawn node, or a future put node. The only way for a future put node to have an out-degree of two is if the corresponding future is multi-touch, which creates a chain of put nodes, each with an out-degree of two.

Given a computation DAG $G$, the **sequential order** is a total ordering of nodes in $G$ that arises from the sequential (one-worker) execution. A **processor order** of a worker $w$ is the sequence of nodes processed by $w$ in a parallel execution of ProWS. We say $u <_1 v$ if $u$ is before $v$ and $u <_1 v$ if $u$ is immediately before $v$ in the sequential order. Similarly, we say $u <_w v$ if $u$ is before $v$ and $u <_w v$ if $u$ is immediately before $v$ in the processor order of $w$.

Given this notation, we now formally define deviation:

**Definition 5.** Let $u$ and $v$ be two nodes in a DAG and $u <_1 v$. We say that $v$ is a deviation in the parallel execution if for some worker $w$ that executed $v$, we have $u \not<_w v$.

Given the definition of SP DAGs (Section 2.2.1), it’s not hard to see that for every sync node $v$ in an SP DAG $G$, there is a corresponding spawn node $u$. Let $u.lchild$ denote the left child of $u$ and $u.rchild$ denote the right child of $u$. Similarly, let $v.lparent$ denote the left parent of $v$ and $v.rparent$ denote the right parent of $v$. Then, let $G_{\text{left}}$ be the SP sub-DAG that consists of the set of nodes $x$ in $G$ such that there is a path from $u.lchild$ to $x$ and from $x$ to $v.lparent$. We say $G_{\text{left}}$ is the SP DAG **enclosed** by $u.lchild$ and $v.lparent$.\(^1\) We define $G_{\text{right}}$ symmetrically. We first show properties of the sequential and parallel executions when scheduled with ProWS.

\(^1\)Recall that each future task is treated as its own SP DAG and thus if a node $x$ in $G$ spawns a future task via $fcreate$ none of the nodes belonging to the future task is in $G$.  

36
Lemma 6. Given an SP DAG $G$ enclosed by a spawn node $u$ and a sync $v$, let $x$ be a node in $G_{left}$ and let $y$ be a node in $G_{right}$. Then, $x <_1 u.rchild <_1 y <_1 v.rparent$. Moreover, $v.rparent <_1 v$.

Proof Sketch. Because each future task is modeled as a separate SP DAG, and a worker $w$ performs eager evaluation on a future task when it encounters a $fcreate$, we can show that this lemma holds by inducting on the number of independent SP DAGs.

Effectively, this lemma says that the sequential order of ProWS follows the depth-first left-to-right traversal of the DAG. Moreover, because for either structured or general use of futures the $fcreate$ of a future task $f$ must appear before the corresponding $ftouch$ in sequential order, the sequential execution of ProWS can never suspend due to an $ftouch$.

Now we prove lemmas about parallel executions.

Lemma 7. Let $v$ be a sync node and $u$ its corresponding spawn node. If $v$ is a deviation, then $u.rchild$ must be stolen.

Proof Sketch. A similar lemma has been shown by Acar et al. [10] for pure SP DAGs scheduled using classic work stealing (i.e., ABP). Here, we additionally need to consider how ProWS diverges from ABP when handling futures. Let $G$ be the SP DAG enclosed by $u$ and $v$ and a worker $w$ pushes $u.rchild$ onto its deque $deq$. By Lemma 6, sequentially $w$ will not pop $u.rchild$ off the deque before executing $v.lparent$. Then, the only way for $w$ to deviate from the sequential execution is to encounter a $ftouch$ and suspend $deq$. In this case, either $u.rchild$ is popped off the deque due to a successful steal, or the entire $deq$ is mugged, in which case the worker mugging the deque will resume execution from the bottom and follow sequential order for the rest of $G$.

Lemma 8. If a worker $w$ enables no children after executing the right parent of a sync node, then $w$'s deque is empty.

Proof. Let $v$ be the sync node and $u$ the corresponding spawn node. By 7, if $v$ is a deviation, $u.rchild$ is stolen. Consider the SP sub-DAG $G$ enclosed by $u.rchild$ (G’s source node) and $v.rparent$ (G’s sink node). Then any node in $G$ is executed before $v.rparent$ in any execution. We know $w$ must steal $u.rchild$ or any node in $G$, otherwise there is no possibility for $w$ to process $v.rparent$. Suppose the deque is not empty after executing $v.rparent$. Let $z$ be
bottommost node on the deque. We must have \( z \) outside \( G \) because any node in \( G \) has been executed. Furthermore, \( w's \) deque is empty when \( w \) performs the steal. Then everything in the deque afterwards is a descendent of \( u.rchild \). So \( z \) can only be a node in a future DAG spawned by \( G \).

Worker \( w \) will turn to the future sub-DAG immediately after executing the corresponding future spawn node. There are two ways that \( w \) can resume the execution of \( G \): (1) the future completes, or (2) \( w's \) deque is empty again and it performs a steal targeting a node in \( G \). In both cases, \( z \) cannot be on \( w's \) deque, which contradicts our supposition.

At a high-level, we bound the number of deviations as follows. We define the notion of “traces” that divide the sequence of nodes executed by a worker based on the types of nodes. We then show that only the first node in a trace can incur a deviation. Lastly, we show that, such a node is either the direct result of a successful steal or can be amortized against a successful steal.

**Definition 9.** Consider a sequence of nodes processed by \( w \), which we then separate into a set of traces, where each trace begins with one of the following nodes: (1) a sync node, (2) a node that gets executed immediately after \( w \) performs a successful steal, and (3) a node that gets executed immediately after \( w \) performs a successful mugging.

**Observation 1.** Given a node \( n \) in the DAG, \( n \) can be one of the following:

1. \( n \) is a regular node: \( n \) has one child in the DAG, and the child has only \( n \) as a parent;
2. \( n \) is a spawn node: \( n \) has two children in the DAG, where each child has only one parent;
3. \( n \) is a future put node: \( n \) can have either one child (single touch future) or two children (chain of put nodes for multi-touch futures);
4. \( n \) is the parent of a sync node: \( n \) has one child, where the child has two parents;
5. \( n \) is the local parent of a future join node: \( n \) has one child, where the child has two parents.

It can be seen that these types are exhaustive by enumerating all the possible combinations of the out-degree of \( n \) and in-degree of \( n's \) children. Note that we can never have a spawn
node $n$ leading to a child who is a sync node or join node — the act of invoking $ftouch$ or $sync$ terminates the current strand and creates a new node with an in-degree of two. Thus, a $ftouch$/$sync$ that immediately follows a $spawn$/$fcreate$ will have a node inserted between them.

**Lemma 10.** Consider a sequence of nodes executed by a worker $w$ during parallel execution scheduled using ProWS, which we separate into traces according to Definition 9. For a given trace $t = (n_1, n_2, \ldots, n_l)$, only $n_1$ can be a deviation.

**Proof.** Let $s = (n_1, \hat{n}_2, \ldots, \hat{n}_l)$ be the sequence of $l$ nodes that starts with $n_1$ in sequential execution. We show that $n_i = \hat{n}_i$ for $i = 2 \ldots l$ by inducting on the length of the trace and argue that either processor $w$ behaves exactly as sequential execution, or the trace ends.

Inductively, assume $n_i = \hat{n}_i$ for $i = 2 \ldots j - 1$ and $w$ behaves exactly the same as the sequential execution up to that point (i.e., each $n_i$ enabled exactly the same nodes as $\hat{n}_i$). Now we consider $n_j$. Based on Observation 1, $n_j$ can be one of the following.

- **Regular node:** $n_j$ must enable its only child and execute it next, just as in sequential execution.
- **Spawn node:** $n_j$ must enable both children, executing the left one and pushing the right one onto the deque, just as in sequential execution.
- **Future put node:** if $n_j$ is a put node for a single-touch future or one at the end of a put chain, then $n_j$ can either enable nothing or the corresponding future join. If $n_j$ enables nothing, this is the same as sequential execution, and $w$ either pops its bottom deque (which leads to the same $n_{j+1}$ by inductive hypothesis), or trace $t$ ends at $n_j$ if $w$'s deque is empty, because the next node has to follow from either a successful steal or mugging. If $n_j$ enables the corresponding future join node $j$, that means the local parent of $j$ executed and couldn’t enable $j$ and thus pushed $j$ onto the bottom of some (now suspended) deque. Even though $n_j$ enabled $j$, note that in ProWS, it does not push $j$ onto $w$’s deque. Instead, it simply marks the deque as resumable.

On the other hand, $n_j$ can be a put node for a multi-touch future that enables the next put node and may or may not enable the corresponding future join node. Enabling the next put node is exactly the same as sequential execution, and whether the corresponding
future join node is enabled or not does not matter, following a similar argument as above.

- **Parent of a sync node:** If executing \( n_j \) enables a sync node, then trace \( t \) ends at \( n_j \) (by the definition of a trace). If in both sequential and parallel executions \( n_j \) enables no child, then \( w \) tries to pop a node off the bottom of its deque, which leads to the same \( n_{j+1} \) by the inductive hypothesis. On the other hand, if \( n_j \) enables no child but in sequential execution, \( \hat{n}_j \) enables a sync node, then \( n_j \) must be the right parent of the sync node. Then by Lemma 8 \( w \)'s deque must be empty and thus trace \( t \) ends at \( n_j \).

- **Local parent of a join node:** In the sequential execution, because the local parent always enables the join node, \( \hat{n}_{j+1} \) will be the join node. In the parallel execution, either \( n_j \) also enables the join node, which means \( n_{j+1} = \hat{n}_{j+1} \), or it enables no child. In the latter case, \( w \) will push the join node onto the bottom of the deque and suspend the deque, which means trace \( t \) ends at \( n_j \) because the next node has to follow from either a successful steal or mugging.

Finally, a key theorem to bound the number of deviations:

**Theorem 11.** Given an execution of ProWS. Let \( n \) be the number of successful steals in the execution. Then, the number of deviations is \( O(n) \).

**Proof.** From the definition of traces and Lemma 10, we know a deviation may only occur at the beginning of a trace. Each trace begins with a sync node, a stolen node, or a node processed after a successful mugging. Thus, the number of deviations is bounded by the sum of the numbers for deviated sync nodes, successful steals, and muggings.

From Lemma 7, we know each deviation at a sync nodes has a unique corresponding stolen node, thus we can bound the number of deviated sync nodes by the number of successful steals. Also recall that in ProWS, a resumable deque has to be stolen from successfully before it becomes muggable. Thus, the number of successful muggings is also bounded by the number of successful steals. Thus, the total number of deviations is bounded by \( O(n) \), where \( n \) is the number of successful steals.
Given Theorem 11, we can now bound the deviations by bounding the number of successful steals, which is less than the number of steal attempts during the computation. Recall Lemma 18, which effectively states that the number of steal attempts can be bounded by $O(XS)$, when a deque can be stolen into with probability at least $1/X$. We provide a bound on $1/X$ by bounding the the maximum number of stealable deques possible during the execution.

**Lemma 12.** Given a computation that uses structured futures scheduled using ProWS, there can be at most $O(PS)$ stealable deques during execution.

**Proof.** In the case of the structured use of futures, the stealable deques can only include suspended deques. Recall that a structured use of futures is restricted to single touch and no race on the handle (i.e., the future spawn node has a directed path to the local parent of the join node). Due to the single touch restriction, there can exist only one suspended deque per future handle $f$. Moreover, due to the directed path, the continuation of the future spawn node that spawned $f$ must be stolen in order for a corresponding $ftouch$ on $f$ to suspend. Thus, whenever a worker $w$ eventually executes the $fput$ that completes $f$’s corresponding future task, $w$’s deque must be empty. By Algorithm 1, $w$ will then resume the single deque suspended with $f$, making it active, and thus there cannot be resumable or muggable deques in the stealable set.

Whenever a worker has to suspend a deque due to $ftouch$, the corresponding future task $f$ is either being actively worked on by another worker (due to eager evaluation), or $f$ is also suspended because $f$ itself invoked an $ftouch$ on a different future, creating a chain of suspended future tasks. Such a chain has length at most $S$ (as does any chain in the DAG). Moreover, at least one worker is working on the future task at the beginning of the chain. Therefore, there can be at most $O(PS)$ suspended (stealable) deques. \qed

**Lemma 13.** Given a computation that uses general futures scheduled using ProWS, there can be at most $m_k$ stealable deques during execution.

**Proof.** By Algorithm 1 lines 25–30, a deque can only suspend when encountering an $ftouch$ (future touch). When a future touch node $n$ causes a deque to suspend, no descendant of $n$ can execute until the deque becomes active again. By definition, because there can be at most $m_k$ number of future touches executing in parallel, this leads to a maximum number of $m_k$ stealable deques at any given time. \qed

41
Finally, we can prove the following deviation bounds:

**Theorem 14.** Given a computation that uses structured futures with span $S$ and scheduled using ProWS on $P$ workers, the number of deviations is $O(PS^2)$ in expectation.

*Proof.* By Lemma 12, we know the maximum number of deques possible during execution is $O(PS)$. By Lemma 2, each worker can have up to $O(S + \lg P)$ deques. Thus, a deque is stolen into with probability of at least $O\left(\frac{1}{PS + P\lg P}\right)$. Then by Lemma 18, the steal attempts across the computation is at most $O(PS^2 + PS\lg P)$, or $O(PS^2)$ assuming $S = \Omega(\lg P)$, which is likely the case.

**Theorem 15.** Given a computation that uses general futures with span $S$ and scheduled using ProWS on $P$ workers, the number of deviations is $O(m_ks + PS\lg P)$ in expectation, where $m_k$ is the maximum number of future touches that are logically parallel.

*Proof.* By Lemmas 13 and 2, we similarly derive the probability that a deque is stolen into to be at least $O\left(\frac{1}{m_k + P\lg P}\right)$. Then by applying Lemma 18 we obtain the bound.

### 3.4 Cilk-F: A Prototype System

This section describes Cilk-F, our prototype implementation of ProWS that extends Intel Cilk Plus [91, 92], a C/C++-based task parallel platform. An interesting design issue that arises is how to best support the “cactus stack” [82] abstraction needed by a task parallel platform. This section discusses the issue, our design choice, and a bound on the stack space based on our particular implementation.

#### The Cactus-Stack Abstraction

A serial language such as C [99] or C++ [146] utilizes an array-based linear stack because the activation frames of all child functions of a given parent can reuse the same stack space repeatedly. In a task parallel language, because a parent can have multiple spawned children executing simultaneously, their activation frames can no longer occupy the same space. Thus,
the underlying system must maintain a cactus stack [82] abstraction that supports the stack views of multiple children that are active simultaneously, such as in Figure 3.1.

A natural question that arises is how to best support a cactus stack while providing a good resource usage bound. While multiple mechanisms exist to support a cactus stack, the answer to this question is nuanced, and different implementations make different tradeoffs [106]. For programs that use strictly fork-join parallelism scheduled using work stealing, the best possible bound is \( P N \) [39, 40], where \( N \) is the stack space needed for serial execution.

It turns out that there are inherent tradeoffs between space usage, time bound, and interoperability with serial binaries [106]. Known mechanisms that allows for such space efficiency (e.g., “heap-based” cactus stack [70, 74]) do not allow for interoperability with serial binaries. Cilk Plus’s runtime system utilizes a strategy that allows for an efficient time bound, but exchanges a somewhat worse space bound for interoperability with serial binaries, resulting in a bound of \( PN F \) [106], where \( F \) is the maximum number of parallel functions nested on the stack during serial execution. Unfortunately, for a program with futures, neither space bound holds, because work stealing no longer maintains the necessary “busy-leaves” property [39].

Maintaining Cactus Stack for Futures

Cilk-F utilizes the same strategy that Cilk Plus uses, called stack stitching, where a worker grabs a new stack when necessary, and the cactus stack is maintained by stitching together linear stacks. With the work-first policy and fork-join parallelism, a linkage point, the stitching of two linear stacks, is created only upon a successful steal, because that’s when actual parallelism is realized. In Figure 3.1 for instance, if a worker \( w1 \) executes \( A \), which
spawns B, a different worker w2 that steals the continuation of A will grab a new stack to invoke E and remembers that its parent is A. With this strategy, the runtime maintains the invariant that only a parallel function with stealable continuations (i.e, spawn) can form linkage points and needs to know about the cactus stack; a serial function can operate with the old calling convention assuming a linear stack.

The use of futures raises unique issues with the maintenance of cactus stacks. In this example, say A is an ordinary C function which calls a parallel function B, and B subsequently spawns C via fcreate. Because the termination of a future task C is not confined within its enclosing lexical scope, without a corresponding ftouch, B can return (and so can any of its ancestors) without waiting for C to complete.

The first issue that arises is that C may access resources allocated on its ancestor’s stack. We won’t be too concerned with this issue, because in this case, it’s only appropriate that the ftouch on C is invoked within the subcomputation of the said ancestor; otherwise it’s a programming error.\(^{13}\)

Second, and more importantly, the continuation of B can be stolen and resumed by a different worker, and B can subsequently return while C executes on the original worker. In this case, however, all three functions are still on the same linear stack used by the original worker, and resumption of A can clobber the execution of C.

This is a function of both the work-first policy, which allows a future task to be allocated on the same stack, and the fact that that a future allows its ancestor, which may be an ordinary C function, to resume execution while the outstanding future task uses the same stack. To get around this, whenever a worker executes a future, we opt to spawn the future task on newly allocated linear stack and let the parent (continuation of fcreate) reuse the stack.

**Cilk-F’s Cactus Space Bound**

Let F be the maximum number of nested parallel functions (functions containing parallel keywords) on the stack during serial execution. Cilk-F provides the stack space bounds of \(O(PSNF)\) for structured futures and \(O(mNF)\) for general futures. Effectively, the maximum number of suspended deques at any given time (Lemmas 12 and 13 in Section 3.3)\(^{13}\)

\(^{13}\)The future and async constructs in C++ follow similar philosophy [156].
allows us to bound the stack space. Due to the proactive nature of ProWS, a deque contains only executions that correspond to a particular path in the invocation tree. For a given path, there can be at most $F$ linkage points, each keeping $N$ stack space alive. Thus, the bound follows from the bounds on the maximum number of deques.

If we had taken the heap-based cactus stack approach, we would obtain a space bound of $O(PSN)$ for structured futures and $O(m_kN)$ for general futures, following similar arguments: the stack space taken up by the activation frames fallen on a single path in the invocation tree is at most $N$, getting rid of the $F$ factor. However, the system would not allow for interoperability with serial binaries.

Lee et al. [106] and Yang and Mellor-Crummey [161] proposed virtual memory mechanisms to allow for a work-stealing scheduler to maintain the cactus stack abstraction in a way that bounds physical space and provides interoperability. Although their mechanisms are designed for fork-join parallelism, they can be adapted to programs with futures. Using Lee et al.’s mechanism, a space bound of $O(PS(N + F))$ for structured futures and $O(m_k(N + F))$ for general futures can be achieved, though the proposed VM mechanism requires special Operating System support. Mechanisms proposed by Yang and Mellor-Crummey can provide similar bounds on physical space consumption though not virtual address space. Both mechanisms require invoking system calls for each successful steal, suspension, and mugging.

3.5 Empirical Evaluation of Cilk-F

This section empirically evaluates Cilk-F, a prototype implementation of ProWS, across nine benchmarks. The use of futures provides additional flexibility in expressing parallelism, but the flexibility comes with some additional cost compared to fork-join parallelism, namely a higher number of deviations and higher cost in maintaining the cactus stack abstraction. To evaluate the overhead in using futures, we compare the execution times of benchmarks running on Cilk-F (future implementations) against Cilk Plus (fork-join implementations) and analyze its overhead. We show that ProWS can be implemented efficiently. Despite the slightly worse theoretical execution time bound compared to the classic work stealing that Cilk Plus implements, Cilk-F performs comparably to Cilk Plus.
**Experimental setup.** We ran our experiments on an Intel Xeon E5-2665 with 16 2.40-GHz cores on two sockets. Each core has a 32-KByte L1 data cache, 32-KByte L1 instruction cache, and a 256-KByte L2 cache. There is a total of 64 GB of memory, and each socket shares a 20-MByte L3-cache. All benchmarks are compiled with LLVM/Clang 3.4.1 with \(-O3 -flto\) running on Linux kernel version 4.4. Each data point is the average of 10 runs.

**Benchmarks.** We evaluate Cilk-F using nine benchmarks implemented with both fork-join parallelism and futures that fall into three different categories: A) benchmarks where futures do not provide additional benefits and both versions effectively do the same thing; B) benchmarks that can be implemented using both, but the versions with futures expose slightly higher parallelism; C) benchmarks that can only be implemented with futures.

Benchmarks under category A) include: \texttt{mm} (matrix multiplication, input 4k-by-4k), \texttt{smm} (the Strassen matrix multiplication algorithm, input 4k-by-4k), and \texttt{sort} (parallel mergesort, input 10^8). Benchmarks under category B) include: \texttt{heartwall} (adapted from the Rodinia benchmark suite \cite{49} that tracks the movement of a mouse heart over a sequence of ultrasound images, input 104 frames), \texttt{lcs} (dynamic programming solving longest common subsequence, input 32k with base case size 512), \texttt{sw} (dynamic programming that implements Smith-Waterman for sequence alignment, input 2k with base case 32), \texttt{bst} (the pipelined tree merge using parallel futures from \cite{37}, input 8e6 and 4e6). The benchmark under category C) is \texttt{ferret} (adapted from the PARSEC benchmark suite \cite{30} that implements a content-based similarity search on images with pipeline parallelism, with input size native). Even though \texttt{ferret} cannot be implemented using fork-join parallelism, it can be implemented as a pipeline program; thus, we compare the future implementation running on Cilk-F with the pipelined version running on Cilk Plus's implementation of Piper \cite{93}, a provably efficient work-stealing scheduler that supports (linear) pipeline parallelism \cite{107, 108}. The benchmarks are suffixed to indicate their implementation, with \texttt{fj} for fork-join parallelism, \texttt{lp} for (linear) pipeline parallelism, \texttt{sf} for structured futures, and \texttt{gf} for general futures. The serial elisions are obtained by removing parallel keywords from the \texttt{fj}/\texttt{lp} versions, but the serial elisions for the future implementations are comparable.

\footnote{With the exception of \texttt{bst-fj} and \texttt{bst-gf}, which were compiled with \texttt{-O0} and without \texttt{-flto} due to a compiler bug triggered by the code.}
As Table 3.1 shows, the performance of Cilk-F is comparable to the performance of Cilk Plus in all our benchmarks.

For the category A benchmarks this is to be expected; the future versions are direct conversions from fork-join parallelism by replacing spawn with fcreate and sync with ftouch. This structure ensures that when a ftouch causes a deque to suspend, that deque is always empty. Because all suspended deques are empty, they never get added to the stealable set, and thus the Cilk-F implementations have the same expected execution time as Cilk Plus, $W/P + O(S)$ for these benchmarks.

The category B benchmarks also have comparable results between Cilk Plus and Cilk-F, even though Cilk-F has a harsher expected time bound of $O(W/P + S \log P)$. We expect different versions of the heartwall benchmark to perform similarly because the use of futures only provide slightly higher parallelism by some constant amount (i.e., the same work and span asymptotically). On the other hand, the Cilk-F implementations of lcs and sw both demonstrate slightly better speedups than their Cilk Plus counterparts. The future

Table 3.1: The execution times of benchmarks in seconds, running with Cilk Plus (fj and lp) and Cilk-F (sf and gf). $T_s$ shows the running time of the serial elision. $T_p$ shows the running time on $p$ processing cores, and the numbers in parenthesis are the speedups over $T_s$.

<table>
<thead>
<tr>
<th>bench</th>
<th>$T_s$</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_4$</th>
<th>$T_8$</th>
<th>$T_{16}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm-fj</td>
<td>87.61</td>
<td>89.49</td>
<td>44.74</td>
<td>(1.96×)</td>
<td>22.35</td>
<td>(3.92×)</td>
</tr>
<tr>
<td>mm-sf</td>
<td>88.88</td>
<td>44.45</td>
<td>(1.97×)</td>
<td>22.22</td>
<td>(3.94×)</td>
<td>11.10</td>
</tr>
<tr>
<td>smm-fj</td>
<td>17.54</td>
<td>17.75</td>
<td>9.06</td>
<td>(1.94×)</td>
<td>4.68</td>
<td>(3.75×)</td>
</tr>
<tr>
<td>smm-sf</td>
<td>17.96</td>
<td>9.14</td>
<td>(1.92×)</td>
<td>4.73</td>
<td>(3.71×)</td>
<td>2.61</td>
</tr>
<tr>
<td>sort-fj</td>
<td>13.69</td>
<td>14.16</td>
<td>7.10</td>
<td>(1.93×)</td>
<td>3.58</td>
<td>(3.83×)</td>
</tr>
<tr>
<td>sort-sf</td>
<td>14.02</td>
<td>7.03</td>
<td>(1.95×)</td>
<td>3.55</td>
<td>(3.86×)</td>
<td>1.85</td>
</tr>
<tr>
<td>hw-fj</td>
<td>170.83</td>
<td>171.20</td>
<td>87.23</td>
<td>(1.96×)</td>
<td>45.04</td>
<td>(3.79×)</td>
</tr>
<tr>
<td>hw-sf</td>
<td>170.12</td>
<td>86.56</td>
<td>(1.97×)</td>
<td>44.77</td>
<td>(3.82×)</td>
<td>23.45</td>
</tr>
<tr>
<td>hw-gf</td>
<td>170.66</td>
<td>86.88</td>
<td>(1.97×)</td>
<td>44.58</td>
<td>(3.83×)</td>
<td>23.50</td>
</tr>
<tr>
<td>lcs-fj</td>
<td>8.80</td>
<td>9.06</td>
<td>4.63</td>
<td>(1.90×)</td>
<td>2.49</td>
<td>(3.53×)</td>
</tr>
<tr>
<td>lcs-sf</td>
<td>9.06</td>
<td>4.57</td>
<td>(1.92×)</td>
<td>2.42</td>
<td>(3.63×)</td>
<td>1.53</td>
</tr>
<tr>
<td>lcs-gf</td>
<td>8.84</td>
<td>4.54</td>
<td>(1.94×)</td>
<td>2.34</td>
<td>(3.75×)</td>
<td>1.31</td>
</tr>
<tr>
<td>sw-fj</td>
<td>14.51</td>
<td>12.58</td>
<td>6.43</td>
<td>(2.26×)</td>
<td>3.37</td>
<td>(4.31×)</td>
</tr>
<tr>
<td>sw-sf</td>
<td>14.47</td>
<td>7.27</td>
<td>(2.00×)</td>
<td>3.69</td>
<td>(3.93×)</td>
<td>1.89</td>
</tr>
<tr>
<td>sw-gf</td>
<td>13.64</td>
<td>6.93</td>
<td>(2.09×)</td>
<td>3.54</td>
<td>(4.10×)</td>
<td>1.82</td>
</tr>
<tr>
<td>bst-fj</td>
<td>3.31</td>
<td>3.41</td>
<td>2.42</td>
<td>(1.37×)</td>
<td>1.96</td>
<td>(1.69×)</td>
</tr>
<tr>
<td>bst-gf</td>
<td>3.42</td>
<td>2.43</td>
<td>(1.36×)</td>
<td>1.98</td>
<td>(1.67×)</td>
<td>1.79</td>
</tr>
<tr>
<td>fer-lp</td>
<td>191.54</td>
<td>192.72</td>
<td>97.62</td>
<td>(1.96×)</td>
<td>50.00</td>
<td>(3.83×)</td>
</tr>
<tr>
<td>fer-gf</td>
<td>190.66</td>
<td>96.44</td>
<td>(1.99×)</td>
<td>49.85</td>
<td>(3.84×)</td>
<td>25.43</td>
</tr>
</tbody>
</table>
versions of lcs and sw have smaller spans than the fork-join versions: for an input size $N$ and base case $B$, the fork-join versions have a span of $O(N \cdot B + \frac{N}{B} \cdot \lg \frac{N}{B})$ whereas the future versions have a span of $O(NB)$.\textsuperscript{15} The final benchmark in category B, bst, does not show noticeable improvement in the Cilk-F implementation compared to the Cilk Plus implementation. Even though the span is reduced from $O(\lg^2 N)$ to $O(\lg N)$ (with work $O(N)$), this is not a large enough improvement to be apparent in the data we collected, especially because to reduce the overhead of $T_1$ over $T_s$, we had to limit the parallel recursion depth (to depth 5).

In the case of the category C benchmark, ferret, one would expect the linear pipeline version to outperform the general future version because the expected execution time using Piper is $W/P + O(S)$ [107] which is asymptotically better than that of ProWS. In this case, unlike the fork-join case, Cilk-F would incur some number of suspensions with non-empty deques. However, ferret-gf turns out to perform slightly better than ferret-lp running on Piper, likely due to the fact that Piper automatically throttles (which we set to $8P$ for $P$ workers), whereas Cilk-F does not. Thus, ferret-gf can take better advantage of the parallelism in the ferret computation at the risk of increased memory usage.

**Overhead in Using Futures**

To compare the overhead of creating futures implemented in Cilk-F, we use the microbenchmark fib, which calculates the $n^{th}$ Fibonacci number by recursively spawning $\text{fib}(n-1)$ and $\text{fib}(n-2)$. By design, fib does not coarsen the base case so that the majority of the execution time is due to spawn or fcreate overhead. We implemented three separate versions of fib: fib-fj (with spawn/sync), fib-sf (with fcreate and ftouch), and -stack, which is identical to fib-sf but without the overhead of switching to a new stack on every fcreate. This “optimization” is not generally applicable due to the issue discussed in Section 3.4, but is still correct in this particular case (due to where ftouch is invoked); we simply use it to gauge how much overhead allocating a new stack for each fcreate incurs. The results are shown in Table 3.2.

Compared to fib-fj, the overhead primarily comes from two sources: the stack-switch that occurs on a fcreate, and synchronization between the ftouch and fput operations. By

\textsuperscript{15}This difference in span decreases as $B$ approaches $\sqrt{N}$. 

48
Table 3.2: The execution times of different versions of fib, in seconds, running with Cilk Plus (fj) and Cilk-F (sf and -stack). The -stack row shows the running times of fib-st on Cilk-F but removes the stack-switch upon a fcreate. $T_p$ shows the running time on $p$ processing cores and the numbers in parenthesis are the scalability over $T_1$. The $T_s$ (serial elision running time) for fib is 2.46.

<table>
<thead>
<tr>
<th>bench</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_4$</th>
<th>$T_8$</th>
<th>$T_{16}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib-fj</td>
<td>22.05</td>
<td>10.84</td>
<td>5.43</td>
<td>2.71</td>
<td>1.35</td>
</tr>
<tr>
<td>fib-sf</td>
<td>34.30</td>
<td>17.09</td>
<td>8.77</td>
<td>4.36</td>
<td>2.29</td>
</tr>
<tr>
<td>-stack</td>
<td>26.78</td>
<td>13.47</td>
<td>6.99</td>
<td>3.46</td>
<td>1.74</td>
</tr>
</tbody>
</table>

Table 3.3: The number of deviations incurred on 16 processors. The data is the maximum out of 3 runs.

<table>
<thead>
<tr>
<th>bench</th>
<th>fj</th>
<th>sf</th>
<th>gf</th>
</tr>
</thead>
<tbody>
<tr>
<td>hw</td>
<td>10286</td>
<td>13419</td>
<td>24904</td>
</tr>
<tr>
<td>lcs</td>
<td>3672</td>
<td>4231</td>
<td>8197</td>
</tr>
<tr>
<td>sw</td>
<td>4086</td>
<td>4086</td>
<td>12068</td>
</tr>
</tbody>
</table>

comparing the numbers between fib-sf and -stack, we see a big drop in $W$, which also helps with $T_p$. By removing the synchronization overhead between ftouch and fput, we see $W$ drops further from 26.78 seconds to 21.06 seconds, which is comparable to that of fib-fj.\footnote{Though naturally things cannot run correctly in parallel if there is not synchronization between ftouch and fput.} This shows that these are the two major sources of additional overhead when using Cilk-F futures compared to Cilk Plus spawn/sync.

To further investigate the overhead of Cilk-F futures compared to Cilk Plus spawn/sync, we instrumented Cilk-F to collect the number of deviations We chose benchmarks where all three versions are available to show the number of deviations; their code structure is similar between versions although the future versions do achieve slightly more parallelism. As Table 3.3 shows, matching the theory, structured futures generate many fewer deviations than general futures, but fork-join parallelism generates fewer still.

### 3.6 Related Work

**Work-stealing runtime for synchronization primitives:** Futures have been incorporated into many parallel platforms (e.g., [45, 47, 48, 66, 79, 102, 114, 144, 149]). Many use parsimonious work stealing [47, 66, 79, 102, 144].
Researchers have also proposed runtime schedulers for scheduling programs with blocking synchronizations. For instance, Agrawal et al. [19] proposed a work-stealing runtime system for helper locks — where when a worker tries to acquire a lock that is not available, it tries to help complete the critical section that is currently holding the lock. They proved that this scheduler was efficient if large critical sections had sufficient internal parallelism.

Other variants of work stealing have also been implemented to support synchronization primitives that can cause suspension, but none of them provide provably efficient scheduling bounds. In variants of X10 [48, 148] and Habanero dialects [45, 86], various synchronization primitives other than futures are provided that can cause the execution to block while the executing worker’s deque is not empty. In the initial release of X10 [48], little support was provided — a blocking synchronization primitive blocks the executing worker, and to compensate, the runtime spawns a new worker thread to replace the blocked worker, effectively suspending the deque. Over time, the system could be oversubscribed. Tardieu et al. [148] subsequently developed a version of X10 with better support for suspension. In their system, if a worker is blocked the worker suspends the blocked task, but uses a centralized queue to allow resumptions of suspended tasks. A similar approach is taken by the initial release of Habanero Java [45]. In a later version, Imam and Sarkar [86] describe support for suspension in Habanero Java. In their system, suspended tasks are stored with the blocking synchronization primitives (similar to how we handle futures), but once the tasks get resumed, they all get pushed onto the ready deque of the worker who unblocks them.

Zakian et al. [164] extend Intel Cilk Plus [91] to provide support for a low-level library which allows a worker to suspend the current execution context upon encountering a blocking I/O operation and find something else to do. Due to how the deques with the suspended execution context are handled, however, their system does not provide a provably efficient execution time bound. In their system, the multiple suspended contexts (deques) are stored with the worker which suspended them, causing the deques among workers to potentially become extremely imbalanced. Moreover, a thief stealing into a victim always checks the active deque first before stealing into deques with suspended execution contexts. In addition, once a suspended execution context becomes ready to be resumed, the deque holding the context may become unstealable, but the worker who owns the deque is busy working on something else and cannot resume it in a timely manner. Thus, a high potential work item may have little or no chance to be stolen into despite many steal attempts.
Work-stealing with multiple deques per worker: Various work-stealing runtime systems have used multiple deques per worker for different reasons. The runtime system for helper locks [19] (discussed above) used multiple deques per worker. When a worker is blocked on a lock, it is only allowed to work on the critical section that is holding the lock (assuming this critical section has internal parallelism) and does so by allocating another deque specifically for this critical section. Therefore, in a program with nested locks with nesting depth \( d \), workers could have as many as \( d \) deques each. However, the scheduler is designed so that each worker can steal from at most one deque of each of the other workers. In a similar vein, Agrawal et al. [14] proposed the Batcher runtime system to handle parallel programs that access shared data structures. In this case, workers can be working on either the program work or the data structure work, and these types of work are kept on different deques. But again, at any given time a worker steals randomly among \( P \) deques.

Researchers have proposed provably efficient work stealing schedulers where the execution allows for suspensions [121, 152]. Muller and Acar [121] studies a work stealing scheduler that hides latency of I/O operations. When a worker encounters I/O, it may suspend the currently executing task. Their scheduler is parsimonious, but due to the possibility of suspension, there can be more than \( P \) number of deques in the system. Their scheduler provides a bound of \( O(W/P + SU(1 + \log U)) \), where \( U \) is the maximum number of parallel I/O operations. Utterback et al. propose [152] Porridge, a processor-oblivious record-replayer for fork-join parallel computations that utilize locks. During replay, if a lock-acquire is not “ready” to be replayed, the executing worker suspends its current deque and steals. Our time-bound analysis takes inspiration from theirs, but we need to additionally handle muggable deques. In their system, the number of suspended deques can be unbounded, and the scheduler provides the time bound of \( O(W/P + S \log \log P) \). Instead of randomly choosing a victim to deposit the suspended deques, they utilize the power-of-two choices, choosing two victims and depositing the deque with the one with the lighter load, thereby obtaining a slightly better bound (\( \log \log P \) in front of the \( S \) term instead of \( \log P \)). We cannot apply the same strategy, because the power-of-two choices does not seem to help with bounding the minimum load [159].

Finally, with respect to space bounds, beyond what’s already stated in Section 3.4, Blelloch et al proposed a Parallel Depth-First (PDF for short) scheduler [33, 34] that is more space efficient than work stealing. Unfortunately, a PDF scheduler is challenging to implement.
efficiently in practice because it requires workers to synchronize with each other at a much higher frequency.
Chapter 4

Using Futures to Efficiently Support I/O

Even though task parallelism simplifies the job of programming multicore machines, modern parallel applications such as desktop software or web services are not well supported by existing task-parallel platforms. Research on task-parallel platforms has traditionally focused on supporting applications from the domain of high-performance and scientific computing, which has very different execution characteristics from that of modern parallel software. In particular, a modern parallel application tends to incur frequent interactions with the external world, done in the form of input/output (I/O), such as obtaining user input through key strokes or mouse clicks, waiting for a data packet to arrive on a network connection, or writing output to a display terminal or network.

Existing task-parallel platforms typically implement work stealing to schedule parallel computations. In the absence of I/O operations, work stealing provides provable bounds on execution time [23, 24, 39, 40], good space usage [40], good cache locality [9, 10], and allows for an efficient implementation [70]. I/O operations are typically done via low-level system libraries (e.g., the GNU C library) or through system calls provided by the operating systems (OS). While one can directly invoke functions provided by these libraries within a task-parallel program, doing so has performance implications. In particular, the classic work-stealing scheduler does not understand the use of I/O operations nor does it account for it in the scheduling algorithm. When a worker thread — the surrogate of a processing core managed by the scheduler — encounters an I/O operation, it can block for an extended period of time, leaving one of the physical cores underutilized while the worker waits for the I/O operation to complete. Thus, when a task-parallel computation includes I/O operations, a work-stealing scheduler can no longer provide the same provable guarantees.

Alternatively, one can utilize low-level system support to perform asynchronous (non-blocking) I/O operations. However, handling the asynchronous I/O completion can be complex — it involves utilizing mechanisms such as signal handling, event-driven style
programming, or explicit polling, all of which require restructuring the program and can severely complicate the program control flow.

In this chapter, we study the problem of how to best support interactions with the external world in the form of I/O operations in a task-parallel platform. Specifically, we design a work-stealing scheduler that hides the I/O latency — when a worker encounters blocking I/O, it simply suspends the current execution context and works elsewhere in the computation. When the I/O operation completes, some worker (not necessarily the worker that suspended it) picks up the suspended context and resumes it. To communicate to the underlying scheduler the use of latency-incurring I/O operations, the proposed task-parallel platform wraps the I/O operations with the future abstraction. As a result, both blocking and nonblocking I/O operations can be seamlessly integrated into the task-parallel programming model, and their uses are composable with other high-level parallel constructs supported by the platform. Finally, we show that the scheduler provides provably good performance bounds and empirically demonstrate the efficiency of our prototype implementation.

As far as we know, only a couple prior works study the problem of supporting I/O operations in task-parallel code. On the system side, Zakian et al. [164] extend Intel Cilk Plus [91] to provide support for a low-level library that allows a worker to suspend the current execution context upon encountering a blocking I/O operation and find something else to do. In this work, however, the proposed mechanism does not allow for provable bounds for both time and space usage due to the way workers handle active work items and blocked execution contexts.

On the scheduling algorithm front, only one prior result provides a provably efficient scheduling bound for task-parallel programs with I/O operations. Muller and Acar [121] present a cost model for reasoning about latency incurring operations (such as I/O) in task-parallel programs. In their work, given a computation with work \( W \) — the total computation time on one core — and \( \text{span}^{17} \ S \) — the execution time of the computation on infinitely many cores — the scheduler executes the computation in expected time \( O(W/P + SD_S(1 + \lg D_S)) \), where the \( D_S \) is the maximum number of latency incurring operations that are logically in parallel. Their bound is \textbf{latency-hiding} in that the latencies of I/O operations only appear in the span term and not the work term. If no latency-incurring operations are used, their bound is asymptotically equal to the standard work stealing bound of \( O(W/P + S) \).

\footnote{The term span is sometimes called “critical-path length” or “computation depth” in the literature.}
In this work, we improve the latency-hiding bound by extending a scheduling algorithm that supports futures based on ProWS [141], described in Chapter 3. We implement I/O operations seamlessly within task-parallel code using futures while getting nearly asymptotically optimal completion time. In particular, we were able to prove that our latency-hiding scheduler provides an execution time bound of $O(W/P + S \log P)$ in expectation; this bound is independent of the number of I/O operations in the system. Compared to the standard work-stealing bound, it just has an additional term of $\log P$ on the span term. This implies that while the standard work-stealing scheduler provides linear speedup when $W/S = \Omega(P)$, our scheduler provides linear speedup when $W/S = \Omega(P \log P)$. ProWS has the same bound, but its original analysis does not directly apply here due to the use of latency-incurring I/O operations. We extend ProWS’s analysis and performance bounds to allow for latency-incurring I/O operations using futures.

The high-level intuition on why we provide a better bound compared to prior work [121] is as follows. The work-stealing algorithm by Muller and Acar is parsimonious — a worker never steals unless it runs out of work to do. In contrast, our work-stealing algorithm based on ProWS, discussed in detail in Chapter 3, is proactive — whenever a worker encounters a blocking I/O operation, it suspends the execution context and finds something else to do by stealing. This behavior may seem counter-intuitive because it potentially increases the number of steal attempts. It turns out, however, by carefully managing deques one can sometimes amortize the steal cost against the work term, thereby obtaining a better bound.

More importantly, the use of proactive work stealing also provides a better bound on the number of deviations [143], defined as execution points at which the parallel execution of a program differs from its sequential execution. As articulated by prior literature [9, 143], the number of deviations provides a good metric for evaluating practical performance because it is highly correlated to scheduling overheads and cache misses during parallel executions. For fork-join parallelism, one can relate the number of deviations to the number of steal attempts [9]. This relationship does not hold with parsimonious work-stealing, however, if the program uses unstructured blocking operations like futures [143], making it difficult to bound the number of deviations. The use of proactive work stealing (as in our algorithm and as in ProWS) allows one to again bound deviations using the number of steals, thereby allowing for a better bound on deviations. With proactive work stealing, ProWS and our algorithm by extension, guarantee that the expected number of deviations is $O((P \log P + m_k)S)$ where $m_k$ is the total number of futures that are logically in parallel.
Our prototype system, Cilk-L, extends Cilk-F [141], an extension of Intel Cilk Plus [91] that supports futures and implements ProWS. Cilk-L defines a special type of future, called an IO future, which utilizes the parallelism abstraction provided by futures to schedule I/O operations in a latency-hiding manner that is composable with the rest of the parallel constructs supported (i.e., spawn, sync, fcreate, and ftouch). When a worker invokes an I/O operation using an IO future, a handle is returned, and the I/O operation can be done either synchronously by calling ftouch on the handle immediately, or asynchronously, calling ftouch at a later time when the result is needed for the control to proceed.

We empirically evaluated Cilk-L with microbenchmarks that interleave compute-intensive kernels with I/O operations. The empirical results indicate that Cilk-L is effective at latency hiding. When we compare the execution times of Cilk-L with the “idealized” execution times (where I/O operations do not incur latency), we find that Cilk-L incurs little overhead, indicating that the I/O latencies are mostly hidden and occur in the background. To support I/O futures, Cilk-L necessarily needs to incorporate additional system support for scheduling I/O operations asynchronously. We also provide a detailed breakdown of overhead.

Contributions

In summary, the contributions of this chapter include:

- We have developed Cilk-L, a task-parallel platform that incorporates support for scheduling I/O operations in a latency-hiding way. By utilizing the abstraction of futures, one can perform asynchronous I/O operations in task-parallel code in a way that is composable with other parallel constructs (Section 4.1).

- We extend the scheduling algorithm of Cilk-F to incorporate the latency-hiding cost model, and show that with I/O latency the algorithm can schedule the computation in time $O(W/P + S \lg P)$ on $P$ cores, independent of the number of I/O operations active in parallel. This bound is an improvement over the prior state-of-the-art by Muller and Acar. Since $\max\{W/P, S\}$ is a lower bound on the execution of this program on $P$ processors, this bound is nearly asymptotically optimal except for the $\lg P$ overhead on the span. Moreover, our algorithm provides bounds on stack space and deviations, whereas the algorithm by Muller and Acar does not (Section 4.2).
• We empirically evaluated Cilk-L using microbenchmarks. The empirical results indicate that Cilk-L hides I/O latency effectively and incurs little scheduling overhead in doing so (Section 4.3).


4.1 The System Implementation

This section describes Cilk-L, a prototype system that extends Cilk-F [141] to incorporate support for performing I/O operations with latency hiding. The I/O support in Cilk-L consists of two main components: the IO futures library and runtime support for doing asynchronous I/O operations. We first discuss the programmer API for using IO futures, its implementation, and then the runtime support. Since I/O operations are typically supported via low-level system libraries and by the underlying operating system, currently Cilk-L only targets Linux platforms and utilizes various I/O related facilities from Linux.

The IO Futures Library

We use an example to illustrate the programming API provided by the library. Figure 4.1 shows the distributed map-and-reduce example. In this code, the function distMapReduce takes in five parameters: $f$, $g$, $id$, $lo$, and $hi$. The computation works as follows. The code obtains a set of input values from $n = hi - lo$ different network connections. The call to openConnection in line 6 abstracts away the sequence of steps to open a network connection, which returns a file descriptor representing the network connection once it is open. For each value $x$ in the set, the code applies the map function $f(x)$ and then combines the resulting values from $f(x)$ using a binary reduction operation, $g$.

The IO futures library exposes one data type to the programmer, the handle for IO futures, io_future, and two I/O functions, cilk_read and cilk_write, for reading from and
writing to the file indicated by the file descriptor (i.e., the first argument, \texttt{fd}). In Linux, all I/O devices are presented as files, including network connections, which allows for a uniform interface for performing I/O operations [43, Chp. 10]. That means \texttt{cilk\_read} and \texttt{cilk\_write} work with any I/O device that can be represented as a file. The \texttt{cilk\_read} and \texttt{cilk\_write} functions are analogous to the Unix \texttt{read} and \texttt{write} system calls, except that they are \textit{asynchronous}, i.e., non-blocking. Both functions return an \texttt{io\_future} handle representing the ongoing I/O operation, but the function itself does not block — the I/O is initiated and linked to the \texttt{io\_future} handle, and the handle is immediately returned.

```plaintext
1 Function distMapReduce(f, g, id, lo, hi)
2 n ← hi − lo;
3 if n = 0 then return id; //return identity.
4 else if n = 1 then
5   char buf[NBYTES]; //buffer for input data.
6   fd ← openConnection(lo) //open network connection.
7   io\_future fut ← cilk\_read(fd, buf, NBYTES)
8   fcreate(fut);
9   return f(buf);
10 else
11   mid ← (lo + hi)/2;
12   r1 ← spawn distMapReduce(f, g, id, lo, mid);
13   r2 ← distMapReduce(f, g, id, mid, hi);
14   sync;
15   return g(r1, r2)
```

Figure 4.1: Distributed map and reduce example.

A call to \texttt{cilk\_read} or \texttt{cilk\_write} first creates an \texttt{io\_future} to represent the I/O request. The corresponding data required to carry out the I/O request (such as the file descriptor \texttt{fd} and the buffer \texttt{buf} to store input) is bundled up with the \texttt{io\_future}. This data bundle is inserted into a \textit{communication queue} to be processed by the runtime. The \texttt{io\_future} is then returned to the caller. If the user needs the result from the future or wants to ensure that it has completed, they can perform an \texttt{ftouch} on this \texttt{io\_future} handle. The continuation of \texttt{ftouch} (the future join node) cannot execute until the I/O operation has completed.
Runtime Support to Hide I/O Latency

At runtime startup, normally a work-stealing runtime creates \( P \) persistent threads as \( P \) workers, one per processing core. In Cilk-L, \( 2P \) persistent threads are created — for every worker a corresponding I/O thread is created, and this persistent thread is pinned to the same core as its worker.\(^{18}\) The I/O thread is only used to process I/O requests (via the IO futures library) generated by the worker’s execution of user code. Thus, in the library implementation described above, the communication queue is implemented as a lock-free single-producer/single-consumer queue and used as a means for the worker thread to communicate I/O requests to its I/O thread.

When an I/O thread runs, it dequeues items from the communication queue and attempts to perform an I/O operation as soon as it is received. If an I/O operation cannot be completed immediately (e.g., the next packet has not yet arrived on the network socket), then the I/O thread puts the request aside and processes it later when the I/O device becomes ready (e.g., it has more input to be consumed).

In order to describe how the actual mechanism works, we need to briefly discuss how I/O operations work on Linux. As mentioned earlier, any I/O device on Linux (e.g., network sockets, mice, and keyboards) can be represented as a file descriptor. Obtaining input (read) from a file descriptor is effectively copying data from the corresponding device into memory (e.g., the \( \text{buf} \) in the example). If the device is not ready to be read (e.g., the next packet has not arrived on the network channel yet), the system call \( \text{read} \) will block. One could mark the file descriptor with the correct flag such that the system call would simply return instead of blocking, with a return value indicating input not ready. However, in this case, we must periodically make the system call again to know when the device becomes ready.

One possibility is to periodically wake up the I/O thread and have it poll the device via non-blocking \( \text{read} \). This scheme is not ideal, as a system call can be expensive. Moreover, if the device is not ready, checking would simply cause the I/O thread to take up processor cycles that could be better used by its worker working on the actual computation. Thus, we would like to avoid the periodic wake up and the unnecessary system calls. Ideally, we would like the I/O thread to simply sleep and not use any processor cycles unless one of the

\(^{18}\)If the hardware has hyperthreading enabled, Cilk-L pins them to separate hardware threads (hyperthreads) associated with the same physical core.
following conditions happen: (a) one of the I/O devices with pending operations becomes ready; or (b) its worker inserts a new I/O request into the communication queue.

To achieve part (a), we use the Linux `epoll` facility which allows the I/O thread to monitor a set of file descriptors (an `epoll` set). Adding a file descriptor to be monitored takes $O(\log n)$ time, where $n$ is the number of file descriptors currently in the `epoll` set. The I/O thread can go to sleep by calling `epoll_wait`, and it will be woken up when one of the I/O devices corresponding to one of the monitored file descriptors becomes ready. Determining which file descriptors have become ready takes $O(1)$ time — adding a file descriptor to the `epoll` set registers a callback with the file's underlying system driver; this callback will move the file into a ready list and wake the monitoring thread when I/O on that device becomes possible. Once the I/O thread is woken up, it can query `epoll` to obtain the list of ready file descriptors, which allows the I/O thread to determine which pending I/O operations can continue. In summary, each I/O thread maintains its own `epoll` set. When an I/O thread receives an I/O request but the corresponding file descriptor is not ready, the I/O thread adds the file descriptor to its `epoll` set to be monitored. Once an I/O thread has processed all I/O requests in the communication queue, it goes to sleep via `epoll_wait`. Doing so achieves part (a).

One last piece of the puzzle is how to avoid having the I/O thread check the communication queue periodically and yet still allow submitted I/O requests to be processed quickly whenever they are received. We solve this by using an event wait/notify mechanism called `eventfd` provided by Linux [6]. The `eventfd` mechanism is used to create a file descriptor that can be read by an I/O thread and written to by its worker. This file descriptor can be opened with semaphore-like semantics, in which writes will increment a backing counter and reads will decrement the same counter. When used with `epoll`, a write to an `eventfd` file descriptor will cause the I/O thread to wake up whenever the backing counter is incremented from 0 to 1. By writing to an `eventfd` file descriptor associated with a communication queue whenever an I/O operation is enqueued, and by symmetrically reading from the same file descriptor whenever an operation is dequeued, `epoll` can also be used to monitor the state of the communication queue. Thus, we use this combination of `eventfd` and `epoll` to achieve part (b).

By combining these mechanisms, we achieve the effect that an I/O thread takes up processor cycles only when either there is a new I/O request from the worker or when one of the
previously dequeued (and unprocessed) I/O operations can be performed. When an I/O thread completes an I/O operation, it performs an \texttt{fput} on the corresponding \texttt{io\_future} handle. From its worker’s perspective, a call to \texttt{ftouch} can cause the current execution to suspend, but the worker will just go find something else to do. Cilk-L schedules the execution of the IO futures in the same manner Cilk-F schedules ordinary futures, which we briefly review in Section 4.2.

### 4.2 Algorithm and Analysis

This section describes how to represent a program with I/O operations abstractly, the high level scheduling algorithm, and the runtime analysis. For scheduling, we will use ProWS, the proactive work-stealing algorithm described in Chapter 3. The ProWS algorithm schedules programs with futures in a time and space efficient manner, and we will briefly recap the algorithm here for completeness. However, the analysis in Chapter 3 handles futures but not I/O operations. Here we will show how that analysis can be extended to appropriately account for I/O latencies.

**Execution DAG**

We extend the model from Section 2.2.1 and add weighted edges in a manner similar to [121]. In our model, I/O operations are performed within future tasks. The invocation of an I/O function (\texttt{cilk\_read} and \texttt{cilk\_write}) creates an \texttt{io\_future}, sets up the necessary data for the I/O request, inserts the request into the communication queue, and returns (discussed in Section 4.1). We will call the last node of this future task before it returns the **I/O setup node**. However, unlike in non-I/O future tasks, this future itself is not ready. The future is ready when the I/O thread executes \texttt{fput} upon the I/O completion — we will call the put node of an I/O future an **I/O put node**. We will have a \texttt{heavy} edge between the I/O setup node and the corresponding I/O put node — the weight on this edge represents the amount of time elapsed between when the I/O function returns and when the I/O completes (including the time that the I/O thread takes to handle the I/O request). All other edges are \texttt{light} with weight of one.
We can define work and span. The work is unchanged, i.e., the total number of nodes in the DAG. Therefore, it is unaffected by the latencies on the edges. The span of the weighted DAG is the longest weighted path in the DAG and is the only parameter affected by the latencies.

Again a node is ready if all its predecessors have executed, except for the I/O put node. An I/O put node is suspended once its predecessor (the corresponding I/O setup node) finishes executing. If $\ell$ is the weight of the incoming edge to the put node, it remains suspended for $\ell$ time steps. After these $\ell$ time steps, it is considered to have finished executing because the I/O thread will write the result into the future handle after these $\ell$ time steps. This definition of suspension of a put node is simply for the ease of analysis and has no impact on the scheduler because the put node is executed by an I/O thread and not by the worker thread.

**Proactive Work Stealing**

We use ProWS unchanged. The main difference between proactive and parsimonious work stealing is the handling of a blocked future touch. In parsimonious work stealing, when a worker’s current node executes an $\texttt{ftouch}$ and the future is not ready, the subsequent future join node is not enabled. Therefore, the current node enables 0 children and (as described in Section 2.4.1) the worker pops the next node from the bottom of the deque and continues working. The algorithm by Muller and Acar [121] is a variant of this — when a worker blocks on an I/O operation, it pops the next node off its deque and keeps working on it.

ProWS behaves differently on executing an $\texttt{ftouch}$ where the future handle $h$ is not ready.\(^{19}\) Instead of popping the next node from its active deque $d$, the worker work steals. In particular, the worker (1) marks the current deque suspended; (2) it randomly picks another worker to donate this suspended deque to; and (3) allocates a new active deque $d'$ for itself and randomly work steals. When the handle $h$ becomes ready (the future finishes), then the corresponding put node marks the deque $d$ resumable and pushes the future join node to the bottom of $d$.

\(^{19}\)There are other circumstances where the execution of the node enables no other nodes, such as when a worker returns from a spawned or future function — in all these circumstances proactive work stealing behaves as the parsimonious one and pops the bottom node from its deque.
Therefore, in ProWS each worker $p$ has potentially many deques. One of these is active — this is the deque the worker is currently working on. In addition, it may have many suspended and resumable deques — collectively, the suspended and resumable deques are called the worker $p$'s inactive deques. Any suspended deques that have no ready nodes are unstealable; all other deques are stealable. The reason for this distinction is that unstealable deques have no ready nodes, so stealing from them is a waste of time. Note that any resumable deques with no ready nodes are simply de-allocated. However, a suspended deque $d$ with no ready nodes cannot be deallocated for the following reason. Deque $d$ is suspended because some `ftouch` executed, but the corresponding future has not completed. When this future completes, the corresponding put node will enable the future join node and push it at the bottom of $d$ and mark it resumable. Therefore, if we deallocate it, we would not have a targeted place to push this future join node.

A steal attempt also works slightly differently compared to traditional work stealing. When work stealing, a thief first picks a random victim and then picks a random stealable deque to steal from among the deques that the victim has. If the target deque is suspended, then the worker simply steals the top node from the deque. If the deque has no more ready nodes, then this deque is marked unstealable. There are additional details on how to handle resumable deques in order to get the correct bounds on running time and deviations — however, these details do not change in our analysis and we refer the reader to Chapter 3 for those details.

The important bits from the perspective of our understanding are the following: (1) Every worker has potentially many deques: one deque is active, and there are potentially many inactive deques (either suspended or resumable and some of the suspended deques may be unstealable); and (2) due to random throws when the deques are suspended, all workers have an approximately equal number of deques. We will use these two facts in the analysis.

**Analysis**

The analysis of ProWS with I/O operations is, to a large extent, an extension of the analysis of Section 3.3 with proper accounting for latency edges. Muller and Acar do account for latency on edges, but do not use futures for I/O operations, use parsimonious work stealing, and do not rebalance deques between workers. Therefore, the running time on $P$ processors is $O(W/P + SU(1 + \lg U))$, where $U$ is the maximum suspension width — the number of
I/O operations that can be pending at the same time in the DAG. There is no bound on the number of deviations. The way they handle the potential function to hide the latency is a little different from our method.

Here, we are using ProWS and want to get a running time bound of \( O(W/P + S \lg P) \) and the deviation bound of \( O((P \lg P + m_k)S) \) where \( m_k \) is the total number of futures logically in parallel. For the special case where all futures are I/O futures, \( m_k = U \).\(^{20}\) The analysis of Section 3.3 doesn’t work out of the box, however, because it does not consider the latency on heavy edges. Therefore, here, we will rely on the lemmas proved in that section, but modify the potential function to handle the heavy edges appropriately.

In general, in work stealing, a worker is always either working or stealing. The main point of the analysis is to bound the total number of steal attempts, say by \( X \). Because the total work is \( W \), the total running time is \( (W + X)/P \). In addition, bounding the total number of successful steals also gives us a bound on the total number of deviations (for proactive work stealing, though not for parsimonious work stealing).

**ProWS potential function and analysis:** Section 3.3’s analysis uses a potential function similar to the one used by Arora et al. [23] (henceforth called ABP) to bound the number of steal attempts. The potential function there is based on the enabling tree — we say that \( u \) enables \( v \) if \( u \) is the last parent of \( v \) to execute and, in this case, we add an edge between \( u \) and \( v \) in the enabling tree. It turns out that, for technical reasons, we cannot use the enabling tree for proactive work stealing. Instead, we will use the DAG itself to decide the potential of the node.

The potential function is based on the depth of nodes in the DAG. The depth of the node \( u \) with one parent \( v \) is \( d(u) = d(v) + 1 \). The depth of a node with multiple parents is similar, except that we add 1 to the depth of the deepest parent. The weight of node \( u \) is \( w(u) = S - d(u) \).

We say that a node \( u \) is the **assigned** node for deque \( d \) if \( d \) is the active deque for some worker \( p \) and \( p \) is currently executing \( u \). The potential of a node \( u \) is defined as follows:

\[
\Phi(u) = 3^{w(u)} - 1 \quad \text{if } u \text{ is assigned and } \Phi(u) = 3^{w(u)} \quad \text{if } u \text{ is ready.}
\]

For technical reasons, we will\(^{20}\)Section 3.3 provides separate deviation bounds for structured and general futures and they both carry over. If all our I/O operations are done using structured futures, then the bound is better than this — here we only provide the general bound.
say that the assigned node for deque \( d \) is at the bottom of deque \( d \) even though it cannot be stolen. The total potential of a deque \( d \) is the sum of the potential of all nodes on \( d \) including the assigned node if \( d \) is active. The total potential of the computation is the sum of the potentials of all the ready and assigned nodes on all the deques.

Some of the key results from ABP carry over with these changes in definitions.

**Lemma 16.** The initial potential is \( 3^{2S-1} \) and the final potential is 0. In addition, the potential never increases.

**Lemma 17.** Top Heavy Deques The top most node in the deque has a constant fraction of the total potential of the deque.

The intuition is that the top of the deque contains the node that was pushed on the deque farthest in the past and, therefore, it is the shallowest node in the DAG. Because the potential decreases geometrically with the depth, this node contains most of the potential of the deque.

The following lemma is a straightforward generalization of Lemmas 7 and 8 in ABP [23]. The high-level intuition is that because the top node of each deque contains a constant fraction of its potential, if we steal and execute the top node from each deque with reasonable probability, the overall potential is likely to reduce by a constant fraction.

**Lemma 18.** Let \( \Phi_t \) denote the potential on deques at time \( t \) and say that the probability of each deque being a victim of a steal attempt is at least \( 1/X \). Then after \( X \) steal attempts, the potential of deques is at most \( \Phi(t)/4 \) with probability at least \( 1/4 \).

In ABP, because there are only \( P \) deques, one for each worker, this lemma shows that \( P \) random steal attempts reduce the potential by a constant factor with constant probability. However, in ProWS, there are potentially many deques. Therefore, we may need many more steal attempts to reduce the potential. In addition, it is difficult to design a way to steal from all deques with equal probability if the deques are distributed across many workers.

In ProWS, however, recall that when a deque is suspended, the worker picks a random worker and donates the deque to that worker. Therefore, even if one worker suspends many deques, it does not hold on to them — the deques are approximately evenly distributed among all workers. When a worker steals, it picks a random victim worker and then a random stealable deque from the victim. Therefore, each deque has an approximately equal chance of being a victim of a steal attempt. In particular, Section 3.3 shows the following:
Lemma 19. Given $P$ workers and $D$ stealable deques in the system, each worker has at most $D/P + O(\lg P)$ stealable deques with probability at least $1 - o(1)$.

Another insight Section 3.3 uses is that a steal attempt from a stealable deque is generally successful if it is not an active deque. Therefore, if there are many (more than $3P$) stealable deques in the system (and only $P$ of them are active), then most processors have at least one stealable deque and most steal attempts are successful. These periods are called **work-bounded phases** and Section 3.3 argues that the total number of steal attempts in work-bounded phases can be bounded by $O(W)$ in expectation.

Therefore, we only need worry about decreasing the potential when there are not too many stealable deques — these times are called **steal-bounded phases**. Section 3.3 uses Lemma 19 to argue that, during a steal-bounded phase, each stealable deque has at least $c/P \lg P$ chance of being the victim of a steal attempt (for some constant $c$) because no worker has more than $O(\lg P)$ stealable deques. Therefore, using Lemma 18, the potential of deques reduces by a constant factor after $P \lg P$ steal attempts (because unstealable deques are empty and have no potential). Given that the initial potential is $3^{2S-1}$, the expected number of steal attempts during steal bounded phases is $O(P \lg PS)$. Therefore, considering both work- and steal-bounded phases, the total number of steal attempts is $O(W + P \lg PS)$. In addition, they also separately bound the expected number of successful steals in work bounded phases by $O(mkS)$. This allows them to bound the deviations by $O((P \lg P + mk)S)$.

**Changes to potential and analysis to handle weighted edges:** We want to show the same bounds when we use futures for I/O operations. The bounds on steals in work-bounded phases carry over unchanged. In particular, the expected number of steal attempts in work-bounded phases is still $W$ and the expected number of successful steals is still $mkS$. However, for steal bounded phases, where we rely on potential to bound the number of steal attempts, the analysis that bounds the steals does not apply directly for somewhat technical reasons.

Consider the following scenario. Some worker $p$ with active deque $d$ executes an `ftouch` on an I/O future handle $f$ and blocks because the future is not ready. It suspends this deque and steals. At some point, the I/O thread completes the I/O operation corresponding to $f$, executes the put, enables the future join node (say $u$) for $f$, and puts it at the bottom of $d$. Note that this deque’s potential now suddenly increases, and our analysis strongly depends on the potential never increasing.
This is not a problem for Section 3.3 for the following reason: If a particular future join node $u$ is not ready, then some deque must have some ancestor of $u$ on its deque (either as a ready or an assigned node). Therefore, $u$ does not appear on $d$ from nowhere — some ancestor executes, this ancestor’s potential is larger than $u$’s potential and therefore, even though $u$ becomes ready, the overall potential of the computation does not increase. In our case, no ancestor of $u$ is ready or assigned anywhere in the system because the reason $u$ is not ready is due to the latency on an I/O edge. This is problematic because $u$ being enabled increases the potential of the system.

To fix this problem, we have to give potential to put nodes for I/O futures (even though they are executed by I/O threads) and handle them in a special way. In particular, recall that the only heavy edges in our DAG are between I/O setup nodes and the corresponding I/O put nodes. For I/O put nodes, we will define two notions of depth: the initial depth $id(u)$ of an I/O put node with enabling parent $v$ (which is always an I/O setup node) is $id(u) = d(v) + 1$. The depth $d(u)$ starts out as $id(u)$ and increases on every time step while the I/O operation is pending and this I/O put node is suspended. If the weight of the heavy edge (the latency of the corresponding I/O operation) between $v$ and $u$ is $\ell$, then $u$ is suspended for $\ell$ steps. Therefore, $u$’s final depth is $fd(u) = d(v) + \ell$. When the I/O operation completes, this put node completes.

Now consider the child node $j$ of the I/O put node $u$ — $j$ is always a future join node. When deciding the depth of $j$, we always use $u$’s final depth $fd(u)$. That is, if $x$ is $j$’s other parent (the node generated by the $ftouch$ operation), then $d(j) = \max\{fd(u), d(x)\} + 1$.

The potential of a pending put node is defined just like other ready nodes. At any time, if the depth of the put node is $d(u)$, its weight is $w(u) = S - d(u)$ and potential is $3^{2w(u)}$. However, because the depth of the node changes over time, so does its weight and potential. The total potential of the computation is the sum of the potentials of all the ready and assigned nodes on all the deques as well as the potentials of all the put nodes (which are not on any deque).

We now get back the following lemma:

**Lemma 20.** The potential never increases.
Proof. We only need consider the case when a future join node $v$ is enabled by an I/O put node $u$. By definition, $u$ has lower depth and thus higher potential than $v$. $v$ is only enabled once $u$ finishes. Therefore, the potential of the system does not increase.

However, adding these put nodes creates a problem. These put nodes are not on any deque; therefore, steal attempts do not reduce the potential associated with these nodes directly. We must also now argue that the potential of I/O put nodes decreases appropriately during steal-bounded phases. This is the reason why we designed the potential of these put nodes in the funny way where their potential starts out high and reduces on every time step.

**Lemma 21.** During steal-bounded phases, if the total potential at time $i$ is $\Phi_i$ (including potential of assigned, ready and suspended put nodes), then after $cP \lg P$ steal attempts, for some constant $c$, the potential is at most $\Phi(t)/4$ with probability at least $1/4$.

*Proof.* Section 3.3 already argued that the potential of deques reduces appropriately. Therefore, we only need to consider the suspended I/O put nodes. Note that if the latency of a weighted edge is $\ell$, then the corresponding put node $u$ remains suspended for $\ell$ time steps. Its potential starts at $3^{2(S-\text{id}(u))}$ and decreases by a factor of $1/9$ on every time step. When the I/O operation completes and the future is ready, the potential of $u$ is $3^{2(S-\text{fd}(u))}$. Because it takes at least $c \lg P \geq 1$ time steps to do $cP \lg P$ steal attempts, the potential of this put node $u$ reduces by a large fraction during this time. This is true for all put nodes, giving the desired result.

This lemma allows us to bound the expected number of steal attempts (and therefore, expected number of successful steals) during steal bounded phases by $P \lg PS$ — the same result as Section 3.3. Because the expected number of steal attempts and successful steals for work-bounded phases remains unchanged, we get the same time and deviation bounds as Section 3.3.

**Theorem 22.** The expected number of steal attempts is $O(W + P \lg PS)$. Therefore, the expected running time is $O(W + P \lg PS)$. In addition, the expected number of deviations is $O((P \lg P + m_k)S)$. 68
4.3 Empirical Evaluation

This section empirically evaluates our prototype implementation of Cilk-L using a microbenchmark map-reduce that closely resembles the example shown in Section 4.1, Figure 4.1. We would like to answer the following three questions in the evaluation: (1) how much benefit can one obtain from latency hiding; (2) how well can Cilk-L hide latency compared to an idealized system that hides the latency entirely and incurs zero overhead for hiding latency; and (3) how much does each mechanism used in Cilk-L to hide latency contribute to its overhead. How we measure each is explained in its respective subsections. Overall, the empirical results indicate that one can obtain substantial performance benefit from latency hiding. Cilk-L hides latency well when the application has ample parallelism. As the theory predicts, when the application has insufficient parallelism and the weighted span term (i.e., $S \lg P$) dominates, the performance of Cilk-L can lag behind the idealized version. Finally, the empirical results show that Cilk-L is lightweight, incurring minimal system overhead compared to the idealized version.

Experimental setup: We ran our experiments on a machine with two Intel Xeon Gold 6148 processors, each with 20 2.40-GHz cores, with a total of 40 cores. Each core has a 32KB L1 data cache, 32KB L1 instruction cache, and a 1MB L2 cache. Hyperthreading is enabled. Dynamic frequency scaling is disabled. Both sockets have a 27.5MB shared L3 cache, and 768GB of main memory. Cilk-L and map-reduce are compiled with LLVM/Clang 3.4.1 with -O3 -flto running on Linux kernel version 4.15. Each data point is the average of 10 runs. All data points have standard deviation less than 5% except for a handful of data points, which we note later as we explain the data.

Benchmark: We use a microbenchmark with very similar code structure to the map and reduce example (map-reduce) described in Section 4.1 (Figure 4.1), which is also used by Muller and Acar [121]. To allow for comparison with the system by Muller and Acar [121], we have used the same computation kernels and parameters used in their experiments, unless noted otherwise. Like Muller and Acar, we emulate 5000 remote server connections with simulated delays. At line 6 in Figure 4.1, rather than opening a true network connection, we used a timed file descriptor which becomes ready for I/O when the timer expires.\footnote{This functionality is provided by the Linux \texttt{timerfd} [7].} We replace the parameter $f$ with a parallel version of the naive recursive implementation of

\footnote{This functionality is provided by the Linux \texttt{timerfd} [7].}
Fibonacci with a serial base case of 25, and used it to compute the 30th Fibonacci number \( \text{fib} \). In place of calling function \( g \) (line 15), we return the sum of \( r1 \) and \( r2 \) modulo 1000000000.

**The Benefit of Latency Hiding**

To evaluate the benefit of latency-hiding, we compare Cilk-L with the baseline system Cilk-F, described in Chapter 3, which supports futures and provides the same provably efficient time and deviation bound, but does not hide I/O latency (i.e., a worker encountering an I/O function blocks until the I/O operation completes). Specifically, we compare to two different versions of Cilk-F: one uses the same number of workers (Cilk-F) and one uses twice as many workers so as to oversubscribe the system (Cilk-F (O)) and let the underlying OS perform scheduling to hide latency. The Cilk-L version of `map-reduce` uses IO futures to hide latencies whereas the two versions of Cilk-F execute the baseline code that simply uses a blocking `read`.\(^{22}\) We have also compared Cilk-L with the state-of-the-art latency-hiding work stealing scheduler proposed by Muller and Acar [121], denoted as ParWS (parsimonious work stealing). However, note that the comparison to ParWS is not strictly an apples-to-apples comparison, as their implementation is done in a variant of Parallel ML that implements parsimonious work stealing [145] whereas our implementation is C/C++-based. Nonetheless, because their implementation is the only other existing task-parallel scheduler that supports latency-hiding I/O operations with provably efficient execution time bounds, we thus include the comparison for completeness.

We ran `map-reduce` with simulated I/O latencies of 1, 50, and 100 milliseconds. Figure 4.2 shows the speedup of Cilk-L compared to the one-worker execution time of running the baseline version of `map-reduce` on Cilk-F. Unlike what was observed by Muller and Acar [121], we do see some advantage to using Cilk-L to hide I/O latency at 1 millisecond. When the latency is short, however, one could benefit from simply using the oversubscription strategy (i.e., Cilk-F(O)). The Cilk-F(O) version outperforms Cilk-L and even the Ideal at 1-millisecond latency because it utilizes all available hyperthreads to do work. Its performance slows once the computation incurs cross-socket communication (i.e., \( P > 20 \)). We did separate

\(^{22}\)The code executed by Cilk-F and Cilk-F (O) contains only `spawn` and `sync` because `read` is used in place of IO futures.
Figure 4.2: Speedups, compared to the one-core running time (T₁) of the respective baselines, of \textit{map-reduce} running in ParWS, Cilk-F, Cilk-F (O), and Cilk-L with latencies of 1, 50, and 100 milliseconds. Ideal is the Cilk-F implementation that makes calls to \texttt{read} that returns immediately. For Ideal, Cilk-F, Cilk-F (O), and Cilk-L, we computed the speedup against the T₁ in Cilk-F. We computed the speedup for the ParWS against its corresponding baseline implemented in Parallel ML that is not shown. The x-axis shows the core counts (P) and the y-axis shows the speedup. Cilk-F (O) oversubscribes the system by using 2P workers instead of P, for P number of cores (i.e. every core has two workers pinned, one per hyperthread context). Cilk-L pins one worker thread and one I/O thread per core, each on its own hyperthread context.

We also ran \textit{map-reduce} using the Parallel ML implementation of Muller and Acar [121] and plotted the speedup relative to their Parallel ML baseline application. Although this is not quite an apples-to-apples comparison, as there is inherent overhead when using functional languages, the significant increase in speedup for Cilk-L compared to ParWS for P ≤ 20 is still worth noting. For P > 20 it is an especially unfair comparison, as even the Parallel ML baseline application used by Muller and Acar [121] achieves poor speedup once processor socket communication overhead makes memory management very costly for Parallel ML. For ParWS, the standard deviations for P ≥ 35 (for all latency configurations) are quite high, ranging from 16 – 30%.
Table 4.1: The execution times, in seconds, of map-reduce version incurs neither latency nor the overhead for hiding latency. The values in parentheses are overheads relative to the corresponding Cilk-L’s Proximity to Ideal

| n=35 | 1  | 362.73(0.98x) | 72.59(0.98x) | 36.32(0.98x) | 24.22(0.98x) | 18.19(0.98x) | 14.56(0.98x) | 12.15(0.99x) | 10.42(0.99x) | 9.13(0.99x) |
| b=25 | 50 | 362.99(0.98x) | 72.76(0.99x) | 36.49(0.99x) | 24.42(0.99x) | 18.38(1.00x) | 14.74(1.00x) | 12.34(1.00x) | 10.61(1.00x) | 9.31(1.01x) |
| b=100| 100| 362.99(0.98x) | 72.76(0.99x) | 36.49(0.99x) | 24.42(0.99x) | 18.38(1.00x) | 14.74(1.00x) | 12.34(1.00x) | 10.61(1.00x) | 9.31(1.01x) |

Cilk-L’s Proximity to Ideal

Now we evaluate how close Cilk-L is to an “idealized” version at hiding I/O latencies. We obtain the Ideal measurement by running Cilk-F with a timed file descriptor with zero delay (i.e., in place of cilk_read the benchmark invokes a read that returns immediately). Moreover, because it is run with Cilk-F, it does not incur any overhead of setting up I/O futures, epoll, or waking up and context switching to I/O threads. That is, the idealized version incurs neither latency nor the overhead for hiding latency.

Figure 4.2 also includes the idealized version (labeled as “Ideal”) in the plots. As we can see, when the latency is large Cilk-L lags behind Ideal when running fib of 30 with a serial base case of 25. We suspect that this is because the benchmark running on Cilk-L does not have sufficient parallelism and becomes span-dominated (i.e., the span term \(S \lg P\) dominates the work term \(W/P\) in the time bound). By measuring fib of 30 running on one worker, the total amount of work is about 7 milliseconds, which is small compared to the 100 millisecond latency. Note that the computation running on Cilk-F variants do not have the same issue, as the I/O latency is incurred both on the work and span terms (or, in the Ideal case, there is zero latency). Thus, the parallelism profile is not the same when running on Cilk-L versus on Cilk-F variants. This shows up as Cilk-L lagging behind Ideal, because Ideal still achieves near-linear scalability whereas Cilk-L does not (scalability computed as \(T_P\) over \(T_I\) running on the same platform).

23Technically we used a 1 nanosecond latency, which is the smallest latency one could specify with the timed file descriptor on Linux, but it effectively causes the read to become ready immediately.
Table 4.2: The execution times, in seconds, of map-reduce with various configurations of Cilk-L with no I/O latency (i.e. reads do not block). The overheads are relative to the corresponding $T_P$ time of Ideal, which uses Cilk-F without latency-hiding.

To verify this, we also run both Ideal and Cilk-L with additional parameters, i.e., computing the Fibonacci number with different input size ($n$) and serial base case ($b$). Figure 4.1 shows the raw execution times of the Ideal and Cilk-L with different latencies. The first four rows are the times used to plot Figure 4.2. With input of 30 and base case of 25, the overhead ($T_P$ of Cilk-L divided by $T_P$ of Ideal) starts out small and increases as $P$ gets larger. With input of 35 and base case of 15, the computation has sufficient parallelism to hide the latency fully, and we see the discrepancy between Ideal and Cilk-L disappears. We also note that, for data points where Cilk-L runs fib of 30 with $P \geq 35$, we see higher standard deviations, between 6 – 10%.

Cilk-L’s Overhead in Latency-Hiding

The use of IO futures in Cilk-L has some inherent overhead: (1) setting up and tearing down IO futures, (2) invoking the epoll mechanism, which has its inherent system call overheads, and (3) waking up and context switching into I/O threads. These overheads likely contribute to both the less desirable performance compared to oversubscribing Cilk-F when the latency is small and the additional overhead comparing to the ideal version. To figure out how much overhead is contributed by each source, we measure different versions of Cilk-L and compare that to Ideal (Cilk-F running map-reduce where read returns immediately). The +future version is similar to Ideal except that we added back the overhead of using IO futures ($fcreate$, placing the result into future handles, and $ftouch$). Building on the +future version, the +epoll version then adds back the overhead of using epoll. Finally, the Cilk-L version adds back the overhead of using a separate thread to handle the I/O operations.\textsuperscript{24} Note that because the latency is effectively zero, the I/O thread will be woken up only once per request when the request is inserted into the communication

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline
overhead & $T_1$ & $T_5$ & $T_{10}$ & $T_{15}$ & $T_{20}$ & $T_{25}$ & $T_{30}$ & $T_{35}$ & $T_{40}$ \\
\hline
Ideal & 32.87 (1.00x) & 6.58 (1.00x) & 3.29 (1.00x) & 2.20 (1.00x) & 1.66 (1.00x) & 1.32 (1.00x) & 1.11 (1.00x) & 0.95 (1.00x) & 0.83 (1.00x) \\
+future & 31.58 (0.96x) & 6.32 (0.96x) & 3.16 (0.96x) & 2.11 (0.96x) & 1.58 (0.96x) & 1.27 (0.96x) & 1.06 (0.96x) & 0.91 (0.96x) & 0.80 (0.96x) \\
+epoll & 32.88 (1.00x) & 6.58 (1.00x) & 3.29 (1.00x) & 2.20 (1.00x) & 1.66 (1.00x) & 1.32 (1.00x) & 1.10 (1.00x) & 0.95 (1.00x) & 0.83 (1.00x) \\
Cilk-L & 32.38 (0.99x) & 6.50 (0.99x) & 3.27 (0.99x) & 2.20 (1.00x) & 1.66 (1.01x) & 1.35 (1.02x) & 1.13 (1.02x) & 0.97 (1.03x) & 0.86 (1.03x) \\
\hline
\end{tabular}
\caption{The execution times, in seconds, of map-reduce with various configurations of Cilk-L with no I/O latency (i.e. reads do not block). The overheads are relative to the corresponding $T_P$ time of Ideal, which uses Cilk-F without latency-hiding.}
\end{table}

\textsuperscript{24}Cilk-F is effectively a stripped-down version of Cilk-L that removes all mechanisms to support latency-hiding. We obtain each version by incrementally adding back each mechanism that incurs the overhead.
queue. Figure 4.2 shows the comparison. The empirical results show that the overhead from the use of futures is negligible. The overhead from epoll and the I/O thread are comparable, but both are small.

4.4 Related Work

**Interesting use of futures:** Since its proposal [27, 69], the use of futures has been incorporated into various task parallel platforms [45, 47, 48, 66, 79, 102, 114, 144, 149]. Futures are typically used as a high-level synchronization construct to allow parallel tasks to coordinate with one another in a way that is more flexible than pure fork-join parallelism.

Researchers have proposed interesting uses of futures. Blelloch and Reid-Miller [37] used futures to generate non-linear pipelines. Using futures to pipeline the split and merge of binary trees, they developed a parallel algorithm of tree merge with better span than a fork-join parallel merge algorithm. Surendran and Sarkar [147] proposed using futures to automatically parallelize pure function calls in programs and developed the corresponding compiler analyses. Kogan and Herlihy [101] proposed linearizable futures that allow a concurrent data structure to be shared among threads via the use of futures and formalized the correctness guarantees for such uses. Milman et al. [117] proposed an algorithm for a batched lock-free queue using futures and proved correctness guarantees for the batched queue. By exploiting the semantic requirements of a queue, they can optimize the batched operations.

Prior work has used futures as an abstraction for I/O operations [61] in server software. However, their work does not address how the use of futures are scheduled nor what kind of performance bounds the scheduler can provide.
Chapter 5

Representing Priority and Ruling Out Priority Inversions with Futures

Task parallelism is well suited for compute-intensive jobs and may be used to maximize throughput by finishing a job as quickly as possible, but modern applications also include interactive jobs where a thread may needed to be completed as quickly as possible. For such interactive applications, the main optimization criterion is responsiveness — how long each thread takes to respond to a user. To meet the demands of such applications, the systems community has developed techniques which focus on hiding the latency of blocking operations by multiplexing independent sequential threads of control [31, 65, 72, 83].

Historically, these latency-hiding techniques have been researched independently of task parallelism. With the mainstream availability of parallel computers, this separation is now obsolete: many jobs today include both compute-intensive tasks and interactive tasks. In fact, applications such as games, browsers, design tools, and all sorts of interesting interactive systems involve both compute-heavy tasks (e.g., graphics, AI, statistics calculations) and interaction. Researchers have therefore started bridging the two worlds. Muller et al. [122, 123, 126] have developed programming-language techniques that allow programmers to write task-parallel programs and also assign priorities to threads. By using a type system [123] and a cost model, the authors present techniques for reasoning about the responsiveness of parallel interactive programs.

All of this prior work has made some progress on supporting interactive applications in task-parallel platforms, but it makes an important assumption: pure functional programming. Specifically, the work does not allow for memory effects, which are crucial for allowing threads to communicate. This restriction can be significant, because nearly all realistic interactive applications rely on mutable state and effects. As an example, consider a basic server consisting of two entities: a high-priority event loop handling queries from a user and
a low-priority background thread for optimizing the server’s database. Under Muller et al.’s work, the event loop and background thread can only communicate by synchronizing, but such a synchronization would lead to a priority inversion. If effects were allowed, then the threads could communicate by using a piece of shared state.

In this chapter, we overcome this restriction by developing programming language support for priorities and task-parallel threading in the presence of state. To this end, we consider $\lambda^4$, a core calculus for an implicitly parallel language with mutable state in the form of references. The parallel portion of the calculus is based on futures, which represent asynchronous computations as first-class values. Futures can be created and synchronized in a very general fashion. The calculus also allows programmers to assign priorities to futures, which represent their computational urgency. Because it combines futures and state, $\lambda^4$ is very expressive and enables writing conventional nested-parallel programs as well as those with more complex and dynamic dependencies. For example, we can parallelize a dynamic-programming algorithm by creating an initially empty array of future references and then populating the array by creating futures, which may all be executed in parallel. Similarly, we can express rich interactive computations, e.g., a network event can be delegated to a future that sends asynchronous status updates via a piece of shared state.

The high degree of expressiveness in $\lambda^4$ makes it tricky to reason about the cost due to priority inversions and non-determinism due to scheduling: because of the presence of state, the computation may depend on scheduling decisions. For these reasons, traditional graph-based cost models of parallel computations [32, 35, 144] do not apply to programs that mix futures and state. Such models typically do not take priorities into account and assume that scheduling does not change the computation graph.

We tackle these challenges by using a combination of algorithmic and formal techniques. On the algorithmic side, we extend traditional graph-based cost models to include information about priorities as well as “happens-before” edges that capture certain dependencies by reifying execution-dependent information flow through mutable state. We then prove that if a computation graph has no priority inversions, then it can be scheduled by using an extension of greedy scheduling with priorities to obtain provable bounds on the response time of any thread (Theorem 25). Priority inversions are not simple to reason about, so we present a type system for $\lambda^4$ that guarantees that any well-typed program has no priority inversions. To establish the soundness of the type system (Theorem 27), we model the structure of the
computation by giving a dynamic semantics that, in addition to evaluating the program, creates a computation graph that captures both traditional dependencies between threads and also non-traditionally captures certain happens-before dependencies to model the impact of mutable state.

Because $\lambda^4_i$ is a formal system, it can in principle be implemented in many different languages. For this chapter, we chose to implement such a system in the context of C/C++ because many real-world interactive applications with stringent performance requirements are written in C/C++. Specifically, we have developed Adaptive I-Cilk, a task parallel platform that supports interactive parallel applications. Adaptive I-Cilk is based on Cilk, a parallel dialect of C/C++. As with traditional task-parallel platforms, Adaptive I-Cilk consists of a runtime scheduler that dynamically creates threads and maps them onto available processing cores. Unlike traditional task-parallel platforms, however, Adaptive I-Cilk allows the programmer to specify priorities of tasks. Perhaps somewhat unexpectedly, Adaptive I-Cilk also includes an implementation of the $\lambda^4_i$ type system to rule out priority inversions. The type system is implemented by using inheritance, template programming, and other features of C++ to encode the restrictions necessary to prevent priority inversions. Because C++ is not a safe language, this implementation of the type system expects the programmer to obey certain conventions.

The thread scheduler of Adaptive I-Cilk aims to implement the scheduling principle that Theorem 25 relies on. This is challenging to do efficiently because it requires maintaining global information within the scheduler that can only be achieved via frequent synchronizations. Instead, Adaptive I-Cilk approximates optimal scheduling by utilizing a two-level adaptive scheduling strategy that re-evaluates the scheduling decision at a fixed scheduling quantum.

We evaluate the type system of Adaptive I-Cilk by implementing three moderately-sized application benchmarks (about 1K lines each). These applications fully utilize the features of Adaptive I-Cilk (including I/O and prioritization of tasks). We will dive into one application in detail to illustrate the use of future references and mutable states. We defer the full description of the Adaptive I-Cilk implementation and its eval to Chapter 6.
Contributions

In summary, the contributions of this chapter include:

- a cost model for imperative parallel programs that incorporates scheduler-dependence through mutable state (Section 5.1);

- a calculus $\lambda^4$ for imperative parallel programs, equipped with a type system that guarantees absence of priority inversions (Section 5.2);

- Adaptive I-Cilk, a C/C++-based task parallel platform that supports interactive parallel applications with a type system and scheduler that embody the ideas of the threading model, type system, and cost model of $\lambda^4$ (Section 5.3); and

- an empirical evaluation of the C++ type system implemented in Adaptive I-Cilk using three large case studies written with Adaptive I-Cilk (Section 5.4).


5.1 A Graph Model for Responsiveness

5.1.1 Preliminaries

For the purpose of this chapter, we will consider programs with first-class threads that implement futures. Because our models and scheduling algorithms are largely independent of the language mechanisms by which threads are created, we will simply refer to “threads” here. We assign threads a priority, written $\rho$, drawn from a partially ordered set $R$, where $\rho_1 \leq \rho_2$ means that priority $\rho_1$ is lower than priority $\rho_2$ or $\rho_1 = \rho_2$. We write $\rho_1 < \rho_2$ for the strict partial-order relation that does not allow for reflexivity. Note that a total order is a
partial order by definition and threads can be given priorities from a totally ordered set, e.g., integers.

Threads interact with each other in two ways. First, a thread \(a\) may create a thread \(b\), after which the two threads run in parallel. We call this operation, which returns a handle to \(b\), “future-create” or simply \(\text{fcreate}\). Second, a thread \(a\) may wait for a thread \(b\) to complete before proceeding. We call this operation “future-touch” or \(\text{ftouch}\). This model subsumes the classic fork-join (spawn-sync) parallelism.

As is traditionally done, we can represent the execution of a parallel program with a directed graph.\(^{25}\) A vertex of the graph represents an operation (without loss of generality, we will assume that a single vertex represents a uniform unit of computation time, such as a processing core cycle). A directed edge from \(u\) to \(u'\), written \((u, u')\), indicates that the operation represented by \(u'\) depends on the operation represented by \(u\). We write \(u \supseteq u'\) to mean that \(u\) is an ancestor of \(u'\), i.e., there is a (directed) path from \(u\) to \(u'\) (it may be that \(u = u'\)). If it is the case that \(u \not\supseteq u'\) and \(u' \not\supseteq u\), then \(u\) and \(u'\) may run in parallel.

A schedule of a graph is an assignment of vertices to processing cores at each time step during the execution of a parallel program. Schedules must obey the dependences in the graph: a vertex may only be assigned to a core if it is ready, that is, if all of its (proper) ancestors have been assigned on prior time steps. The goal of an efficient scheduler for parallel programs is to construct as short a schedule as possible. Constructing an optimal schedule is impossible when, as in many real programs, the graph unfolds dynamically during execution and is not known ahead of time (even a relaxed offline version of the problem in which the graph is known ahead of time is NP-hard [150]). However, prior results have shown that schedules obeying certain scheduling principles are within a constant factor of optimal length while making decisions based only on information available online (i.e., they need only know the set of ready vertices at any point in time). One such scheduling principle for graphs with priorities is prompt scheduling. At each time step, a prompt schedule assigns to a core a ready vertex \(u\) such that no currently unassigned vertex is higher-priority than \(u\) repeatedly until no cores remain or no ready vertices remain.

It is worth noting that, in the remainder of this section, as we consider schedules of graphs with mutable state, we assume the graphs are acyclic (though we still use the more general

\(^{25}\)Typically, the representation is in the form of a Directed Acyclic Graph, or DAG. The presentation in this chapter allows for cycles in the graphs, so our graphs are not DAGs, but many of the ideas remain the same.
term graph, rather than DAG). A graph with a cycle cannot have a schedule because it is impossible to meet the dependency requirements of a vertex in a cycle; this corresponds to a deadlock in a parallel program. Avoidance and detection of deadlocks are outside the scope of this chapter.

5.1.2 Weak Edges

Traditionally, cost models for parallel programs assume that scheduling does not change the graph of a parallel computation. This assumption is reasonable for deterministic programs and provides a nice layer of abstraction over scheduling — we can assume that any schedule of a graph corresponds to a valid execution. This fundamental assumption breaks in our setting where threads are first class values and state can be used to communicate in an unstructured fashion, leading to determinacy races.

Consider as an example the following program.

```c
1 thread t = NULL;
2 void g() {}
3 void f() {
4   t = fcreate (g);
5 }
6 }
7 void main() {
8   fcreate (f);
9   if (t != NULL) {
10      ftouch (t);
11   }
12 }
```

The graph for this program, in particular whether there is an edge from `g` to `main` representing the `ftouch` on line 10, depends crucially on whether `f` performs the `fcreate` and assignment to `t` before `main` reads `t`, that is, on whether the conditional on line 9 returns true or false.
In fact, depending on the outcome of the condition, this program gives rise to one of two graphs, both shown in Figure 5.1: one in which the conditional is true and one in which it is false. Applying the traditional separation between graphs and schedules, given graph (a), the scheduler could execute the vertices in the following order: 8, 9, 5, 3, 10. But under this schedule, the read on line 9 should read \texttt{NULL}, and thus line 10 should not be executed at all! Similarly, the scheduler could execute graph (b) in the order 8, 5, 3, 9, in which case the read would read a valid thread handle.

The issue is that each graph is valid for only certain schedules but not all. To encode this information, we extend the traditional notion of graphs with a new type of edge we call a \textit{weak edge}. A weak edge from $u$ to $u'$ records the fact that the given graph makes sense only for schedules where $u$ is executed before $u'$. We call such a schedule \textit{admissible}. As an example, graph (c) of Figure 5.1 includes a weak edge (shown as a dotted line) from 5 to 9. The schedule 8, 5, 9, 3, 10 is an admissible schedule of graph (c), but 8, 9, 5, 3, 10 is not.

At first sight, the reader may feel that we can replace a weak edge with an ordinary (\textit{strong}) edge. This is not quite correct, as strong and weak edges are treated differently in determining whether a schedule is prompt. Recall that a schedule is prompt if it assigns ready vertices in priority order. In the presence of weak edges, we define a vertex $u$ to be ready when all of its \textit{strong parents}, that is, vertices $u'$ such that there exists a strong edge $(u', u)$, have executed.

Consider again graph (c) from Figure 5.1, but now suppose we wish to construct a prompt schedule on two cores. By the above definition, a prompt schedule must execute vertex 8, followed by 5 and 9 in parallel, followed by 3, followed by 10. This is, in fact, the only prompt schedule of graph (c), but it is not admissible because it does not execute 5 before 9. We thus conclude that there are no prompt admissible schedules of graph (c) on two cores and graph (b) is the only valid graph for a two-core execution of this program (as graph (b) has no weak edges, any prompt schedule of it is admissible). If we were to replace the weak edge (5, 9) with a strong edge, there would be a prompt schedule of graph (c) that executes 8, followed by 5, followed by 9 and 3, followed by 10. As always, a strong edge forces vertex 9 to wait for vertex 5, but this violates the intended semantics of the program as a simple read operation should not have to block waiting for a write.

In summary, strong edges determine what schedules are valid for a given graph, while weak edges determine whether a graph is valid for a given schedule. That is, weak edges internalize
information about schedules into the graph, breaking what would otherwise be a circular dependency between constructing a graph and constructing a schedule of it.

We extend the notions of ancestors and paths to distinguish between weak and strong edges. We say that a path is strong if it contains no weak edges. If \( u \sqsupseteq u' \) and all paths from \( u \) to \( u' \) are strong, then we say that \( u \) is a strong ancestor of \( u' \) and write \( u \sqsupseteq^s u' \). On the other hand, if there exists a weak path (i.e., a path with a weak edge) from \( u \) to \( u' \), we say \( u \) is a weak ancestor of \( u' \) and write \( u \sqsupseteq^w u' \). We will continue to drop the superscript if it is not important whether \( u \) is a weak or strong ancestor.

In formal notation, we represent a graph \( g \) as a quadruple \( (T, E^c, E^t, E^w) \). The first component of the quadruple is a mapping from thread symbols, for which we will use the metavariables \( a, b \) and variants, to a pair of that thread’s priority and the vertices it comprises. We use the notation \( \vec{u} \) for a sequence of vertices \( u_1 \cdot \ldots \cdot u_n \) making up a thread, and write \( \emptyset \) when \( n = 0 \). Such a sequence implies that \( g \) contains the edges \( (u_1, u_2), \ldots, (u_{n-1}, u_n) \). We will refer to such edges as continuation edges. For a thread with priority \( \rho \) and vertices \( \vec{u} \), we write \( a, \rho \rightarrow \vec{u} \in T \). We write \( \text{Prio}_g(u) \) to refer to the priority of the thread containing vertex \( u \) in \( g \).

The remaining three components are sets of edges. The set \( E^c \) contains \( \text{fcreate} \) edges \( (u, a) \) indicating that vertex \( u \) creates thread \( a \). It is shorthand for \( (u, s) \) where \( s \) is the first vertex of \( a \). The set \( E^t \) contains \( \text{ftouch} \) edges \( (a, u) \) indicating that vertex \( u \) touches thread \( a \). It is shorthand for \( (t, u) \) where \( t \) is the last vertex of \( a \). Finally, the set \( E^w \) contains weak edges.

5.1.3 Well-Formedness and Response Time

Our goal is to bound the response time \( T(a) \) of a thread \( a \) in a graph. If \( a, \rho \rightarrow s \cdot \ldots \cdot t \in g \), for a particular schedule of \( g \), we define \( T(a) \) to be the number of time steps between when \( s \) becomes ready and when \( t \) is executed, inclusive.

Intuitively, in a well-designed program and an appropriate schedule, if thread \( a \) has priority \( \rho \), its response time should depend only on parts of the graph that may happen in parallel.
Figure 5.2: (a) a graph that is not well-formed because of the strong path from $u_0$ to $t$ (b) a well-formed version of the graph with a weak path from $u_0$ to $t$.

with a (i.e. are not ancestors or descendants of a) and have priority not less than $\rho$. This is known as the competitor work $W_{\neq}(a)$ of a thread $a$ and is defined formally:

$$W_{\neq}(a) \triangleq \{|u \in g | u \not\sqsubseteq s \land t \not\sqsubseteq u \land \text{Prio}_g(u) \neq \rho\}$$

We must also define a metric corresponding to the critical path of $a$. We will call this metric the $a$-span, because it corresponds to the traditional notion of span in a parallel cost graph, but we will defer its formal definition for now, because we will need other definitions first.

Bounding the response time of $a$ in terms of only the competitor work and $a$-span is not possible for all graphs: if $a$ depends on lower-priority code along its critical path, this code must be included in the response time of $a$. This situation essentially corresponds to the well-known idea of a priority inversion. Our response time bound guarantees efficient scheduling of any graph that is well-formed, that is, free of this type of priority inversion. Well-formedness must, at a minimum, require that no ftouch edges go from lower- to higher-priority threads. This requirement is formalized in the first bullet point of Definition 1. There is another, more subtle, way in which priority inversions could arise. Consider the graph in Figure 5.2(a), in which shaded vertices represent high-priority work. Although no ftouch edges violate the first requirement of the definition, it would be possible, in a prompt schedule of the graph, for high-priority vertex $t$ to be delayed indefinitely waiting for low-priority vertex $u_0$ to execute due to the chain of strong dependences through $u$. Note that the problem is not that $u$ depends on a lower-priority vertex—as this is a fcreate edge, such a dependence is allowed. The issue is that $u$’s thread is then ftouched by $t$ with no other dependence relation between $u_0$ and $t$. The second bullet point of Definition 1 requires that, in such a situation, this dependence be mitigated by, e.g., the weak edge added in Figure 5.2(b).
We note that this second requirement actually places no additional restrictions on programs. Graphs such as the one in Figure 5.2(a) could not arise from real programs because in order for \( t \) to touch \( u \)'s thread, it must have access to its thread handle, which will have been returned by the \( fcreate \) call represented by \( u_0 \). This thread handle must be propagated to \( t \) through a chain of dependences including at least one dependence through memory effects. There must therefore be a weak path from \( u_0 \) to \( t \), as in graph 5.2(b), which reflects a write (\( w \)) of the thread handle followed by a read (\( u' \)).

Definition 1 formalizes the above intuitions.

**Definition 1.** A graph \( g = (\mathcal{T}, E^c, E^t, E^w) \) is well-formed if for all threads \( a \xrightarrow{\rho} s \cdot \ldots \cdot t \in \mathcal{T} \),

- For all \( u \in g \), if \( u \geq^s t \) and \( u \not\geq s \), then \( \rho \preceq \text{Prio}_{g}(u) \).
- For all strong edges \((u_0, u)\) such that \( u \geq^s t \) and \( u_0 \not\geq s \) and \( \text{Prio}_{g}(u) \not\preceq \text{Prio}_{g}(u_0) \), there exists \( u' \) such that \( u_0 \geq^w u' \geq^s t \) and \( u \not\geq u' \).

To a first approximation, we may define the \( a \)-span of a thread \( s \cdot \ldots \cdot t \) as the longest path ending at \( t \) consisting of non-ancestors of \( s \) (i.e., the longest chain of vertices that might delay the completion of \( a \)). In the presence of weak edges, however, the definition is not so simple. Consider the graph on the left of Figure 5.3, in which shaded nodes are high-priority. Under the above definition, the \( a \)-span includes low-priority node \( u_0 \), but in any admissible schedule, \( u' \) runs after \( u_0 \), so \( u_0 \) is not actually on the critical path. We thus transform the graph into one, the *strengthening*, that reflects this implicit dependence.

**Definition 2.** Let \( g \) be a well-formed graph with a thread \( a \xleftrightarrow{\rho_a} s \cdot \ldots \cdot t \). We derive the \( a \)-strengthening, written \( \hat{g}_a \), from \( g \) as follows. For every strong edge \((u_0, u)\) such that \( u \geq^s t \) and \( \text{Prio}_{g}(u) \not\preceq \text{Prio}_{g}(u_0) \) and \( u \not\geq s \),
• Remove the edge \((u_0, u)\).

• Let \(u' \in g\) such that \(u' \sqsupseteq t\) and \(u_0 \sqsupseteq u'\). If \(u' \not\sqsupseteq s\), then add the edge \((u', u)\) in place of the weak edge between \(u_0\) and \(u\).

The strengthening of the example graph is shown in the right side of the figure. For a thread \(a \mapsto s \cdot \ldots \cdot t \in g\), we define the \(a\)-span, written \(S_a(\rho a)\), to be the length of the longest path in \(\hat{g}_a\) ending at \(t\) consisting only of vertices that are not ancestors of \(s\). More generally, we write \(S_a(V)\) to be the length of the longest path in \(\hat{g}_a\) ending at \(t\) consisting only of vertices in \(V\). Intuitively, the \(a\)-span corresponds to the critical path of \(a\) because, in a valid and admissible schedule, it is possible that all of the vertices along this path may need to be executed sequentially while \(a\) is being executed. Lemma 23 formalizes this intuition.

**Lemma 23.** If, at any step of an admissible schedule, a vertex \(u\) is ready in \(\hat{g}_a\), then it is ready in \(g\).

**Proof.** For this to not hold, it would need to be the case that there exists a strong edge \((u_0, u)\) \(\in g\) such that \(u_0\) has not been executed and \((u_0, u) \not\in \hat{g}_a\). Then \((u_0, u)\) has been replaced in \(\hat{g}_a\) with an edge \((u', u)\) where, by assumption, \(u'\) has been executed (because \(u\) is ready in \(\hat{g}_a\)). By construction, \(u_0 \sqsupseteq u'\). Because the schedule is admissible, \(u_0\) has executed and \(u\) is ready in \(g\).

Lemma 24 is another auxiliary result about \(a\)-strengthening that will be necessary to prove the bound. It shows that, for a thread \(a \mapsto s \cdot \ldots \cdot t \in g\), any strong path ending at \(t\) in the \(a\)-strengthening starts at a high-priority vertex (in a well-formed but not strengthened graph, this is true for strong ancestors of \(t\)—the strengthening essentially removes weak ancestors of \(t\)).

**Lemma 24.** Let \(a \mapsto s \cdot \ldots \cdot t \in g\). If there exists a strong path from \(u\) to \(t\) in \(\hat{g}_a\) and \(u \not\sqsupseteq s\) in \(\hat{g}_a\), then \(\rho \preceq \text{Prio}_g(u)\).

**Proof.** If \(\rho \not\preceq \text{Prio}_g(u)\), there must be some strong edge \((u_1, u_2)\) on the path from \(u\) to \(t\) in \(\hat{g}_a\) such that \(\text{Prio}_g(u_2) \not\preceq \text{Prio}_g(u_1)\). But all such edges are removed in the construction of \(\hat{g}_a\).
Theorem 25 gives a bound on the response times of threads in admissible, prompt schedules of well-formed graphs. The intuitive explanation of the bound also gives a sketch of the proof: at every time step, such a schedule is doing one of two types of work: (1) executing $P$ vertices of competitor work or (2) executing all available vertices on the $a$-span. The amount of work of type (1) to be done is bounded by the competitor work divided by $P$. Work of type (2) can only be done during $S_a(\notin a)$ time steps, during which $P - 1$ of the $P$ cores might be idle. Adding these amounts of work together gives the bound on response time.

**Theorem 25.** Let $g$ be a well-formed graph and let $a$ be a thread of priority $\rho$ in $g$. For any admissible prompt schedule on $P$ processing cores,

$$T(a) \leq \frac{1}{P} [W_{\not\prec \rho}(\notin a) + (P - 1)S_a(\notin a)]$$

**Proof.** Let $s$ and $t$ be the first and last vertices of $a$, respectively. Consider the portion of the schedule from the step in which $s$ becomes ready (exclusive) to the step in which $t$ is executed (inclusive). For each core at each step, place a token in one of two buckets. If the core is working on a vertex of a priority not less than $\rho$, place a token in the “high” bucket; otherwise, place a token in the “low” bucket. Because $P$ tokens are placed per step, we have $T(a) = \frac{1}{P}(B_l + B_h)$, where $B_l$ and $B_h$ are the number of tokens in the respective buckets after $t$ is executed.

Each token in $B_h$ corresponds to work done at priority $\not\prec \rho$, and thus $B_h \leq W_{\not\prec \rho}(\notin a)$, so $T(a) \leq \frac{W_{\not\prec \rho}(g)}{P} + \frac{B_l}{P}$. We now need only bound $B_l$ by $(P - 1) \cdot S_a(\notin a)$.

Let step 0 be the first step after $s$ is ready, and let $\text{Exec}(j)$ be the set of vertices that have been executed at the start of step $j$. Consider a step $j$ in which a token is added to $B_l$. Consider a strong path in $\hat{g}_a$ ending at $t$ consisting of vertices in $g \setminus \text{Exec}(j)$. Any such path, by definition, begins at a vertex that is ready in $\hat{g}_a$ at the start of step $j$. By Lemma 23, this vertex is also ready in $g$, and by Lemma 24, it has priority greater than or equal to $\rho$. By promptness, this vertex must therefore be executed during step $j$. Thus, the length of the path decreases by 1 and so $S_a(g \setminus \text{Exec}(j + 1)) = S_a(g \setminus \text{Exec}(j)) - 1$. Therefore, the maximum number of steps in which $B_l$ increases is $S_a(g \setminus \text{Exec}(0))$, and at most $P - 1$ cores are idle while $a$ is active, so $B_l \leq (P - 1) \cdot S_a(g \setminus \text{Exec}(0))$. Because $\text{Exec}(0)$ contains all ancestors of $s$, any path excluding vertices in $\text{Exec}(0)$ also excludes all ancestors of $s$, and therefore $S_a(g \setminus \text{Exec}(0)) \leq S_t(\notin a)$, so $B_l \leq (P - 1) \cdot S_a(\notin a)$.

86
Constraints  \( C ::= \rho \preceq \rho \mid C \land C \)

Types  \( \tau ::= \text{unit} \mid \text{nat} \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid \tau \text{ ref} \mid \tau \text{ thread}[\rho] \mid \tau \text{ cmd}[\rho] \)

Values  \( v ::= x \mid \langle \rangle \mid \pi \mid \lambda x. e \mid (v, v) \mid \text{inl} \ v \mid \text{inr} \ v \mid \text{ref}[s] \mid \text{tid}[a] \mid \text{cmd}[\rho] \{m\} \)

Expressions  \( e ::= v \mid \text{let} \ x = e \ in \ e \mid \text{ifz} \ v \{e; x.e\} \mid v \ v \mid \text{fst} \ v \mid \text{snd} \ v \mid \text{case} \ v \{x.e; y.e\} \mid \text{fix} \ x: \tau \text{ is} \ e \)

Commands  \( m ::= \text{fcreate}[\rho; \tau\{m\}] \mid \text{ftouch} \ e \mid \text{dcl} [\tau] \ s := e \ in \ m \mid !e \mid e := e \mid x \leftarrow e; m \mid \text{ret} \ e \)

Figure 5.4: Syntax of \( \lambda^4_i \)

5.2 Type System for Responsiveness

We describe a type system that can be used to ensure that a program results in a well-formed cost graph, by way of a core calculus \( \lambda^4_i \), which extends \( \lambda^4 \) [123], with the key addition of mutable references (memory locations). Section 5.2.1 presents the calculus and type system. Section 5.2.2 equips \( \lambda^4_i \) with a cost semantics that evaluates a \( \lambda^4_i \) program to produce a cost graph of the form described in Section 5.1. We prove that, for a well-typed program, the resulting graph is well-formed, and thus the program is free of priority inversions.

5.2.1 The \( \lambda^4_i \) Core Calculus

The syntax of \( \lambda^4_i \) is shown in Figure 5.4, in A-normal form (for most expressions, any subexpressions that are not under binders are values; computations can be sequenced using let-bindings). We differentiate between expressions, language constructs that do not depend on the state of memory or threads, and commands, which do.

The non-standard types of \( \lambda^4_i \) are a type \( \tau \text{ ref} \) indicating references to memory locations holding values of type \( \tau \); a type \( \tau \text{ thread}[\rho] \) representing handles to running threads of type \( \tau \) at priority \( \rho \) and a type \( \tau \text{ cmd}[\rho] \) representing encapsulated commands which run at priority \( \rho \) and have return type \( \tau \). Priorities are drawn from a given fixed (partially ordered) set \( R \).
The novel values of the calculus are references \texttt{ref[s]}, which allow access to a memory location \texttt{s}; thread handles \texttt{tid[a]}, which reference a running thread referred to by \texttt{a}; and \texttt{cmd[\rho] \{m\}}, which encapsulates the command \texttt{m} at priority \texttt{\rho}. The expression layer is otherwise standard.

Commands include operations to manipulate threads and state, including commands to create and touch threads\footnote{While the syntax for \texttt{fcreate} and \texttt{ftouch} is drawn from the fact that our threading model is based on \texttt{futures}, we simply use the term “threads” to refer to running asynchronous threads of control and “thread handles” to refer to the first-class values that refer to threads. This avoids some terminological confusion frequently associated with \texttt{futures}.}. The command \texttt{dcl [\tau] \{s := e in m\}} declares a new mutable memory location \texttt{s}, initialized with the expression \texttt{e}, in the scope of \texttt{m}. The read command \texttt{! e} evaluates \texttt{e} to a reference \texttt{\texttt{ref[s]}} and returns the current contents of \texttt{s}. The assignment command \texttt{e_1 := e_2} evaluates \texttt{e_1} to a reference \texttt{\texttt{ref[s]}} and writes the value of \texttt{e_2} to \texttt{s}; the command also returns the new value.

Commands are sequenced with an operator \texttt{x \leftarrow e; m}, which evaluates \texttt{e} to an encapsulated command, executes the command, binds its return value to \texttt{x} and continues as \texttt{m}. Expressions may be embedded into the command layer using the command \texttt{\texttt{ret e}} which evaluates \texttt{e} and returns its value. These commands may be thought of as the monadic bind and return operators, respectively.

Figure 5.5 shows the key rules of the type system for \texttt{$\lambda_i^4$}, namely the rules for threads and references. We refer the reader to [124] for discussion of standard features, which are omitted here.
The command typing rules in the figure define the judgment $\Gamma \vdash_R^\Sigma m \sim \tau@\rho$. The signature $\Sigma$ tracks type information for threads and memory locations, as well as the priorities of threads. The typing judgment is also parameterized by a partially-ordered set $R$ of priorities and a typing context $\Gamma$. The context $\Gamma$, as usual, contains premises of the form $x:\tau$, indicating that the variable $x$ has type $\tau$. In addition to the return type $\tau$ of the command, the typing judgment indicates that the command may run at priority $\rho$. The rules CREATE and TOUCH contain notable features relating to priorities. In particular, TOUCH requires that $e$ be a handle to a thread running at priority $\rho'$ and that this priority be higher than or equal to the priority $\rho$ of the current thread. It is this requirement that prevents priority inversion. The CREATE rule requires that a command run in a new thread at priority $\rho'$ indeed be able to run at priority $\rho'$. Note, however, that the $\text{fcreate}$ command itself may run at any priority; the language does not enforce any priority relationship between a thread and its parent. We refer the reader to the presentation of $\lambda^4$ [123] for a more thorough description of these rules.

We describe the rules for allocating and accessing references in more detail. Rule DCL types the initialization expression $e$ at type $\tau$ and introduces a new location $s$ in typing $m$. Rule GET requires that its subexpression have reference type. Rule SET requires that $e_1$ have type $\tau\text{ ref}$ and that $e_2$ have type $\tau$. Note that this requires memory locations to have a consistent type throughout execution. The return type of an assignment to a $\tau$ reference is $\tau$. All of these commands may type at any priority as state operations and priorities are orthogonal.

The judgment $\Gamma \vdash_R C$ indicates that the premises contained in $\Gamma$ entail the priority constraints $C$. The rules (which follow standard rules of logic) are found in the full version [124]

The typing rules admit several forms of substitution. Lemma 26 shows that all of these substitutions preserve typing. The proof largely follows from that of the equivalent substitution lemma for $\lambda^4$ [123].

**Lemma 26** (Substitution).

1. If $\Gamma, x: \tau \vdash_R^\Sigma e_1: \tau'$ and $\Gamma \vdash_R^\Sigma e_2: \tau$, then $\Gamma \vdash_R^\Sigma [e_2/x]e_1 : \tau'$.
2. If $\Gamma, x: \tau \vdash_R^\Sigma m \sim \tau'@\rho$ and $\Gamma \vdash_R^\Sigma e: \tau$, then $\Gamma \vdash_R^\Sigma [e/x]m \sim \tau'@\rho$.
3. If $\Gamma, \pi \text{ prio} \vdash_R^\Sigma e: \tau$, then $[\rho/\pi]\Gamma \vdash_R^\Sigma [\rho/\pi]e : [\rho/\pi]\tau$. 

89
Frames $f ::= \text{let } x = - \text{ in } e \mid x \leftarrow m \mid \text{ftouch } - \\
| \text{dcl } [\tau] s := - \text{ in } m \mid i - \mid :- := e \\
| v := - \mid \text{ret } -$

Stacks $k ::= e \mid k; f$

States $K ::= k \triangleright e \mid k \triangleleft v \mid k \triangleright m \mid k \uparrow v$

Figure 5.6: Stack, frame and state syntax.

4. If $\Gamma, \pi \vdash^R m \sim \tau @ \rho$, then $[\rho'/\pi] \Gamma \vdash^R [\rho'/\pi] m \sim [\rho'/\pi] \tau @ [\rho'/\pi] \rho$.

5. If $\Gamma, \pi \vdash^R C$, then $[\rho'/\pi] \Gamma \vdash^R [\rho'/\pi] C$.

5.2.2 Cost Semantics and Time Bounds

In this section, we equip $\lambda_4$ with a small-step dynamic semantics that tracks two notions of cost. First, in a straightforward sense, the number of steps taken by the semantics to execute a program gives an abstract measure of execution time. Second, we equip the dynamic semantics to construct a cost graph for the program that captures the parallelism opportunities in the execution, and also uses weak edges to record happens-before relations as described in Section 5.1.

We present the dynamic semantics of $\lambda_4$ as a stack-based parallel abstract machine that serves as a rough model of the program’s execution time on realistic parallel hardware. A stack $k$ consists of a sequence of stack frames $f$ (or is the empty stack, $\epsilon$). Each frame is a command or expression with a hole, written $-$, to be filled with the result of the next frame. The stack thus represents the continuation of the current computation. At each step, each thread active in the machine is either executing a command or expression from the top of the stack (“popping”) or returning a resulting value to the stack (“pushing”). These states are represented by $k \triangleright e$ and $k \triangleleft v$, respectively, for expressions (and similar syntax with filled triangles for commands). The syntax of stack frames, stacks, and stack states is given in Figure 5.6.

A full configuration of the stack machine includes the current heap $\sigma$ and set of threads $\mu$. A heap, essentially, is a mapping from memory locations to values. For technical reasons, we also record two pieces of metadata in the heap at each location: the graph vertex that performed the last write to that memory location (which will be used to add weak edges to the cost graph) and a signature containing threads that one might “learn about” by reading this memory.
Figure 5.7 presents a subset of the rules for the command transition judgment

$$\sigma \mid \mu \otimes a \xrightarrow{K_{i}} K_{i} \Rightarrow \sigma'_{i} \mid \mu'_{i} \otimes K_{i}' \mid \mu'_{i} \otimes K_{i}'_{i} \otimes \mu'_{i} \otimes \Sigma''_{i} \mid \sigma'_{i} \mid g_{i}$$

In this judgment, $K_{i}'_{i}$ is the new state of thread $a_{i}$, $\Sigma''_{i}$ contains the memory locations allocated by the step and $\sigma'_{i}$ contains any heap writes performed by the step. The graph $g_{i}$ contains a vertex corresponding to this step as well as any additional $\text{fcreate}$, $\text{ftouch}$ or weak edges added by this step. The full semantics for the abstract machine also includes a single rule that steps some number of threads in parallel and combines the resulting states and graphs. The full set of rules can be found in the full version [124]

An $\text{fcreate}$ command simply creates a new thread symbol $b$ and adds a thread $b$ to the thread pool to execute the command $m$. It returns the thread handle, and adds a $\text{fcreate}$
edge to the graph. An ftouch command first evaluates its subexpression (D-TOUCH1). When the thread handle \( \text{tid}[b] \) is returned, rule D-TOUCH2 inspects the thread pool for the entry \( b \xleftarrow{\rho', \Sigma'} \epsilon \triangleright \text{ret} \ v \) (if \( b \)'s stack is not of this form, \( b \) has not finished executing and the ftouch will block until it does). The command returns the value \( v \) and adds the appropriate ftouch edge. It also adds \( \Sigma' \) to the set of threads that \( a \) “knows about,” because \( v \) might contain handles to threads in \( \Sigma' \).

Rule D-SET3 adds a binding to the heap for the new value of the memory location, and includes as metadata the new graph vertex \( u \) and the signature \( \Sigma \). Rule D-GET2 inspects the heap for the binding of \( s \), returns its value, adds a weak edge \((u', u)\) (recall that \( u' \) is the vertex corresponding to most recent write to \( s \)) and adds \( \Sigma' \) to the signature of \( a \).

The soundness theorem for the type system states that well-typed programs have well-formed cost graphs.

**Theorem 27.** Let \( m \) be such that \( \cdot \vdash^R m \sim \tau @ \rho \). If

\[
\cdot \mid \emptyset \mid \emptyset \mid a \xleftarrow{\rho'} \epsilon \triangleright m \Rightarrow^* \Sigma \mid \sigma \mid g \mid \mu
\]

then \( g \) is well-formed and acyclic.

The proof of this theorem consists of showing that all steps maintain two invariants:

1. No strong edges go from lower to higher priority

2. \( \Sigma \) correctly reflects the “knows about” relation motivated above.

These invariants respectively imply the two well-formedness requirements of Section 5.1. Full proof details are available in the full version [124].
5.3 Implementation of the C++ Type System in Adaptive I-Cilk

This section presents the design and implementation of Adaptive I-Cilk, our prototype task-parallel platform that supports parallel interactive applications. Adaptive I-Cilk is based on Cilk-L (previously described in Chapter 4), and uses futures to represent the threads described in $\lambda^4_i$. The implementation of Adaptive I-Cilk consists of two main components, a type system to rule out priority inversions (closely following the typing rules discussed in Section 5.2) and a runtime scheduler that automates load balancing while prioritizing high-priority futures over lower-priority ones. This section discusses the implementation of the type system; we defer description of the scheduler to Chapter 6. Before we discuss the implementation, we first briefly discuss the programming interface supported by Adaptive I-Cilk for writing interactive parallel applications.

5.3.1 Programming Interface

Future creation. In Adaptive I-Cilk, like in $\lambda^4_i$, a function $f$ can invoke another function $g$ with $\texttt{fcreate}$, which indicates that the execution of $g$ is logically in parallel with the continuation of $f$ after $\texttt{fcreate}$. A function invocation prefixed with $\texttt{fcreate}$ returns a handle to the new future, on which one can later invoke $\texttt{ftouch}$ to ensure that the future terminates before the control passes beyond the $\texttt{ftouch}$ statement. Since a future handle can be stored in a data structure or global variable and retrieved later, the use of $\texttt{fcreate}$ and $\texttt{ftouch}$ can generate irregular parallelism with arbitrary dependences. In Adaptive I-Cilk, as is common in C-like languages, it is possible to allocate a variable of future handle type without associating it to a future, and later pass this variable by reference to $\texttt{fcreate}$, to associate it with the created future. This is in contrast to $\lambda^4_i$, where the allocation of the handle and the creation of the future happen simultaneously.\footnote{Adaptive I-Cilk additionally supports $\texttt{spawn}$ and $\texttt{sync}$ for nested parallelism. The use of $\texttt{spawn}$ and $\texttt{sync}$ can be subsumed by $\texttt{fcreate}$ and $\texttt{ftouch}$ from the type checking perspective and hence we omit the discussion here.}

I/O Operations. Adaptive I-Cilk supports the use of I/O operations via a special type of future, called an $\texttt{io\_future}$, that performs an I/O operation in a latency-hiding way.
Specifically, Adaptive I-Cilk provides special versions of the \texttt{cilk\_read} and \texttt{cilk\_write} functions that behave similarly to the Linux \texttt{read} and \texttt{write} except that they return a \texttt{io\_future} reference representing the I/O operation. Upon invocation, \texttt{cilk\_read} and \texttt{cilk\_write} create a future to perform the I/O without occupying the processor, and then the returned \texttt{io\_future} can be used to wait on the I/O by calling \texttt{ftouch} on it.

5.3.2 Type System

The type system in Adaptive I-Cilk does not provide full type safety guarantees, as C++ is not type safe. Nevertheless, provided that the programmer follows a set of simple rules, the C++-based type system can ensure that a program that type checks will result in strongly well-formed graphs when executed. The type system enables us to type check moderately large benchmarks that implement interesting functionalities involving the use of low-level system calls and concurrent data structures (discussed in Section 5.4.1).

Enforcing Typing Rules

We utilize templates and other C++11 language features to encode the type system. In the C++ encoding, each priority is represented as a \texttt{class}. The relationship between two priorities is captured through the class hierarchy via inheritance; if priority $\rho$ inherits from priority $\rho'$ or some descendant of $\rho'$, then $\rho \succ \rho'$ (i.e., $\rho$ has higher priority than $\rho'$). Such relationships can be tested at compile time using \texttt{is\_base\_of}, which tests whether one \texttt{class} is either the same as or the ancestor of another. Unlike in $\lambda^1_4$, priorities are thus user-defined types rather than a pre-defined set of constants.

In $\lambda^1_4$, there is a separation between the command layer and expression layer. In Adaptive I-Cilk, the separation is not as clear. However, we must enforce restrictions on which functions can be invoked with \texttt{fcreate} (generating a handle that can be \texttt{ftouched} later) and which function can execute \texttt{ftouch}, because the priority of such functions must be retrievable at compile time in order to enforce the typing rules. We require these functions to be wrapped in a \texttt{command class} whose type relies on a template that specifies its execution priority. For ease of discussion, we will refer to such a function as a \texttt{command function}. Unlike in $\lambda^1_4$, \texttt{fcreate} is not a command — code at any priority may safely invoke a function with
fcreate; this causes no difficulties in enforcing the typing guarantees. Also unlike in $\lambda^4_i$, code in Adaptive I-Cilk does not require special syntax for invoking an expression (e.g., function that is not a command) within a command.

The encoding of the type system is realized by C++ macros that transform fcreate, ftouch, and declarations/invocations of command functions into the necessary C++ encodings.\textsuperscript{28} The templated types of command functions allow their priority to be known at compile time, and the type system checks for priority inversion at the execution of ftouch. First, a function invoked with fcreate (which must be a command function) returns a future handle whose type is templated with its priority and return type (i.e., what its corresponding future returns when done executing, which may be void). Second, an ftouch can only be executed from within a command function, and an ftouch on a future handle fptr is translated to:

```cpp
fptr->touch();
static_assert(is_base_of<this->Priority,
    fptr->Priority>::value,
    "ERROR: priority\_inversion\_on\_future\_touch");
```

The static assert ensures that the future invoking the ftouch has priority lower than or equal to that of the future whose handle is ftouched, causing a compiler error otherwise.

Lastly, we enforce that a command function $g$, if invoked by another command function $f$, must be invoked with fcreate or inherits the priority of $f$.\textsuperscript{29} Doing so ensures that another command function $h$ joining with $f$ (with lower priority than $f$ but higher priority than $g$) does not suffer from priority inversion by waiting on $g$. In $\lambda^4_i$ such an issue does not arise because call is an expression whereas fcreate is a command, and therefore the two do not mix. This issue is an artifact of the fact that the distinction between the command and the expression is not clear in Adaptive I-Cilk.

\textsuperscript{28}We additionally provide macros for declaring and defining a command function to ease the use of command functions.

\textsuperscript{29}Currently this is enforced by name mangling command functions which can be circumvented, but in principle this can be enforced with better compiler support.
Discussion: Type Safety

Ideally we would like to guarantee that programs which type check using our API will always generate strongly well-formed graphs when executed. However, we cannot make this guarantee in full because C++ is not a type-safe language. Nevertheless, provided that the programmer follows a few simple rules, our type system can statically prevent cases of priority inversions, and a program that type checks will result in strongly well-formed graphs when executed.

The first rule is that the programmer should not use unsafe type casts. Type casts circumvent the type system; the programmer can use \texttt{fcreate} to invoke a command function \texttt{foo} with a low priority, but at the point of \texttt{ftouch} to join with \texttt{foo} can type cast the future handle to be of a higher priority. The code would type check, as at the point of \texttt{ftouch}, the future handle is interpreted at a higher priority. Similarly, one could allocate a future handle \texttt{future\_pointer} of a high priority, but then type cast it to a lower priority at the point of \texttt{fcreate} to create a future with low priority. Because the priority check is done against the priority type of the \texttt{future\_pointer}, one can trick the type system into thinking that a \texttt{ftouch} does not cause a priority inversion when the future associated with the handle is actually of a lower priority.

The second rule is that the programmer should always ensure that a future handle is already associated with a future (via \texttt{fcreate}) before invoking \texttt{ftouch} on it. This rule is important because a strongly well-formed graph must have a path between the vertex that invokes the \texttt{fcreate} and the vertex that invokes the \texttt{ftouch}. This is trivially satisfied in $\lambda^4_i$ because allocation and creation are inextricably linked, but in Adaptive I-Cilk a future handle allocation can be separate from its future creation. Thus, such a requirement is not trivially satisfied, and the programmer has to manually ensure the future has been created before an \texttt{ftouch}.

5.4 Evaluation of the C++ Type System

This section empirically evaluates the C++ implementation of $\lambda^4_i$. To evaluate the practicality and usability of the type system, we wrote three moderately sized application benchmarks: a proxy server (\texttt{proxy}, 1.5K LoC), a multi-user email client (\texttt{email}, 1.1K LoC), and a job
server (jserver, 1.1K LoC). The type system helps the programmer ensure that there is no priority inversion, which is not always easy to tell, as future handles are often used to coordinate interactions among different application components. The same applications to evaluate the efficiency of the scheduler by comparing Adaptive I-Cilk against Cilk-L, the baseline system that utilizes proactive work stealing but does not account for the priority of futures (and thus does not incur the two-level scheduling overhead). We use the I/O functionality of Cilk-L and Adaptive I-Cilk for the I/O operations in the benchmarks so that I/O-blocked threads do not hinder parallelism. The empirical results indicate that Adaptive I-Cilk was able to prioritize high-priority futures and thus provide shorter response times. However, thorough evaluation of Adaptive I-Cilk is deferred to Chapter 6, in Section 6.4.

**Experimental Setup.** Our experiments ran on a computer with 2 Intel Xeon Gold 6148 processors with 20 2.40-GHz cores. Each core has a 32-kB L1 data and 32-KB L1 instruction cache, and a private 1 MB L2 cache. Hyperthreading was enabled, and each core had 2 hardware threads. Both processors have a 27.5 MB shared L3 cache, and there are 768 GB of main memory. Adaptive I-Cilk and all benchmarks were compiled using the Tapir compiler [137] (based on clang 5.0.0), with -O3 and -flto. Experiments ran in Linux kernel 4.15.

**5.4.1 Application Case Studies**

We evaluate the type system with three applications representative of interactive applications in the real world in that they utilize interesting features commonly used to write such applications, such as low-level file system and network libraries, and concurrent data structures implemented using primitives such as fetch-and-add and compare-and-swap.

**Proxy server.** The first application, proxy, allows multiple clients to connect and request websites by their URL. The server fetches the website on the client’s behalf, masking the client’s IP address. As an optimization, the server maintains a cache of website contents using a concurrent hashtable. If a website is cached, the server can respond with it immediately. The application utilizes components with four priority levels, listed in order from highest to lowest: a) the loop that accepts client connections and the per-client event loop that handles the client requests, b) a component that fetches websites in the event of a cache miss, c) a

---

30 LoC exclude comments, system libraries, and runtime code.
component that logs statistics, and the lowest is d) the main function that performs server startup / shutdown. The priority specification favors response time for client requests.

**Email client.** The second application **email** is a multi-user, shared email client that allows users with individual mailboxes to sort messages, send messages, and print messages; a background task also runs periodically to reduce storage overhead by compressing each user’s messages using Huffman codes [52][Chp. 16.3]. The application contains components with six priority levels, listed in order from highest to lowest: a) an event loop to handle user requests, b) a send component that sends email, c) a sort component that sorts emails, d) a compress component to compress emails and a print component to uncompress and send the uncompressed emails to the printer, e) a check component that periodically checks for the need to compress and fires off compression, f) the main function that performs shutdown.

One interesting feature is that the application requires the print and compress to interact with one another — if the user asks to print a particular email but it is in the midst of being compressed, the print component needs to coordinate with the compress component and wait for it to finish. Similarly, the compress component may encounter an email that it is about to compress, but it is in the midst of being printed, and thus the compress needs to wait for the print to complete.

To enable this, within each user’s inbox data structure is an array indexed using the email ID where any future attempting to print or compress the email will store its own handle. For instance, say there is an ongoing print task for an email. The array slot corresponding to the email stores (a pointer to) the print task’s future handle. If a compress task for the same email is created, the first thing the compress task does is perform a compare-and-swap (CAS) on the same array slot, swapping out the pointer to the print task’s future handle and inserting a pointer to its own handle into the slot. Assuming that CAS returns a non-null reference, the compress task invokes `ftouch` on the reference to ensure that the printing is done before proceeding with the compress.

The simplified pseudocode for the compress task is shown below.

```c
1 int compress(int userID,
2     int emailID, future<PrintOfComp, int> *thisFut) {
3     future<PrintOrCompP, int> *prev =
4         CAS(&userInbox[userID][emailID], thisFut);
5     int emailState = DECOMPRESSED;
```
A print task performs similar operations on the array to coordinate with an ongoing compress task for the same email. Such an interaction is achieved by utilizing the future handles and mutable state in an interesting way.

**Job server.** The jserver application executes jobs that arrive in the system using a smallest-work-first policy, i.e., given different types of jobs, the server knows the amount of work entailed for each type, and it prioritizes jobs with the least amount of work. We simulate user inputs using a Poisson process to generate jobs at random intervals and execute them. The priority levels correspond to the types of jobs. We simulated four different types of jobs with fixed input size $n$, listed in order of priority (high to low): a) parallel divide-and-conquer matrix multiplication ($\text{matmul}, n = 1024$), b) fibonacci ($\text{fib}, n = 36$), c) parallel merge sort ($\text{sort}, n = 1.1 \times 10^7$), and d) Smith-Waterman for sequence alignment ($\text{sw}, n = 1024$). This application differs from the previous two in that tasks in different priority levels are independent of each other, and it is constructed so that we can easily modify the workload to simulate a server that is lightly loaded to heavily loaded.

**Compilation time.** Because the type system heavily utilizes templates, we measure its effect by comparing the compilation time and resulting binary sizes between code that uses priorities and code that does not.\(^{31}\) As shown in Figure 5.1, the use of templates for enforcing the typing rules incurs acceptable overhead.

### 5.5 Related Work

**Task Parallelism**

Many languages and systems have been developed for task parallelism over the years. A large number of these, such as Id [25], Multilisp [79], NESL [36] and parallel versions of

\(^{31}\)The use of template can increase code size as each type instantiation of a given template gets its own code clone.
Table 5.1: The compilation times and resulting binary sizes of application code without and with priority. The compilation time is in seconds and the maximum out of the three compile runs. The binary size is in KB. The numbers in parentheses show overhead compared to the no priority version.

<table>
<thead>
<tr>
<th>case study</th>
<th>compilation time</th>
<th>binary size</th>
</tr>
</thead>
<tbody>
<tr>
<td>proxy (w/out)</td>
<td>1.95 (1.00×)</td>
<td>824.0 (1.00×)</td>
</tr>
<tr>
<td>proxy (with)</td>
<td>2.48 (1.27×)</td>
<td>974.7 (1.18×)</td>
</tr>
<tr>
<td>email (w/out)</td>
<td>4.66 (1.00×)</td>
<td>1241.16 (1.00×)</td>
</tr>
<tr>
<td>email (with)</td>
<td>5.40 (1.16×)</td>
<td>1454.58 (1.17×)</td>
</tr>
<tr>
<td>jserver (w/out)</td>
<td>2.10 (1.00×)</td>
<td>851.2 (1.00×)</td>
</tr>
<tr>
<td>jserver (with)</td>
<td>2.67 (1.27×)</td>
<td>987.7 (1.16×)</td>
</tr>
</tbody>
</table>

Haskell [46, 98] and ML [66, 77, 95, 133, 155], have focused on functional programming languages, in which the issues of races and deadlock do not arise or are limited; progress, however, has also been made toward handling some effects efficiently [77, 155]. These languages typically assume the fork-join model of parallelism, but there have also been advances in generalizing them to include the broader set of parallelism primitives such as futures [12].

Some parallel language extensions have targeted popular imperative programming languages such as C [70] and Java [41, 48, 88, 105]. Many papers have been devoted over the years to taming races (e.g. [64, 109, 134, 151, 160]) and deadlock (e.g. [13, 50, 51, 154]). None of these languages allow the tasks to be prioritized; doing so, as we do in this work, requires reasoning about priority inversions in addition to the problems mentioned above.

**Scheduling for Responsiveness**

Responsiveness has long been a concern in the systems community, as operating systems must schedule processes and threads, many of which are interactive. A thorough overview of this topic can be found in a text by Silberschatz et al. [138]. In contrast to task-parallel systems, OS schedulers deal with relatively small numbers of threads.

Many threading systems for which responsiveness is a concern incorporate some notion of priority. The problem of priority inversion has been noted in systems as early as Mesa [104].
Babaoğlu et al. [26] formalized the idea of priority inversions and discussed some techniques by which they could be prevented.

Recent work [122, 123] has introduced thread priorities into a task-parallel system and developed type systems for ruling out priority inversions that arise through touching a future. That work, however, targets purely functional programming, and so future handles can essentially only be passed through calls and returns, leading to a well-behaved DAG (Directed Acyclic Graph) structure. In this chapter, future handles can additionally be passed through mutable state, leading to much more complicated reasoning about priority inversions.

Cost Semantics

Cost semantics (e.g., [112, 135, 136]) are used to reason statically about the resource usage, broadly construed, of programs. Cost semantics for parallel programs [11, 32, 35, 144] typically represent the parallel structure of the program as a DAG. Offline scheduling results bound the time required to execute such a DAG on \( P \) processors in terms of the work and span of the DAG. Classic offline scheduling results have shown that a “level-by-level” schedule [42] and any greedy schedule [60] are within a factor of two of optimal. Although the full details of the DAG model are usually reserved for proofs, the metrics of work and span and the scheduling results above are quite useful in practice for analyzing programs by thinking in terms of the parallel structure of the underlying algorithm (e.g., the branching factor and problem size in a divide-and-conquer algorithm). Even in cases where the input is unknown, one can reason asymptotically about work and span, much like asymptotic reasoning in sequential algorithms. Recent work has extended parallel cost semantics to reason about I/O latency [121] and responsiveness [122, 123]. This chapter further extends the state of the art by adding weak edges that allow graphs (which may now additionally contain cycles) to reflect information passed between threads through global state.

Much of the above prior work has drawn a distinction between the cost semantics, which uses a very abstract evaluation model to produce a cost DAG from a program, and a provably-efficient or bounded implementation [35, 122, 123], which counts the steps of an abstract machine. A proof that the abstract machine actually meets the bounds promised by the cost semantics can be quite technical and involved. In this work, we present one dynamic semantics that both counts steps and produces a cost graph. This semantics reflects execution
ordering, which is important in our calculus, and simplifies the proof that the steps of the abstract machine are bounded by the cost semantics.
Chapter 6

Greedy Scheduling for Responsive Applications

Many modern parallel applications have a mixture of different types of tasks occurring concurrently that may have different requirements for responsiveness: interactive tasks that interact with the external world (e.g., the user) and therefore must provide fast response; and other (possibly compute intensive) tasks that run in the background to support these interactive tasks, but may not need to be quite as responsive. We want such interactive applications to be able to effectively utilize commodity multicore hardware because multicore processors are widely deployed from personal computers to mobile devices to cloud platforms.

In order for the application to be responsive, the scheduler must be able to discern and prioritize tasks that require faster response time over tasks with looser or no responsiveness requirements. Consider for example a modern desktop application such as an email client application; it likely has a graphical user interface (GUI) component that interacts with the user by continuously listening to keyboard or mouse inputs and reacting to them. The user may type in a search string; the email client reacts by performing a compute-intensive search across all the emails in the inbox. From time to time, the email client may trigger background tasks such as compressing existing emails as an archive to save storage space. In this example, the GUI component generates high-priority tasks, because they need to be most responsive. The search constitutes a medium priority task, as it needs to be done to respond to the user but is not as latency-sensitive as the GUI component. Finally, the compression is a low-priority task, as it does not directly interact with the user.

Most scheduling algorithms used by existing task-parallel platforms for multicore hardware do not work well for such interactive applications, because they are designed for throughput-oriented applications, as opposed to applications with latency-sensitive components that must be responsive. Thus, the scheduler has no notion of priorities and treats all ready
tasks as equal — in the above example, if the archiving is ongoing and occupying the entire machine, a GUI task may not get to execute at all. If these schedulers are used for interactive applications, the interactive latency-sensitive components may not execute as promptly as we would wish.

As alluded to in Chapter 1, because the schedulers of traditional task-parallel platforms do not work well for interactive parallel applications, most interactive applications are currently programmed using persistent threads such as pthreads. Well-designed task-parallel platforms with proper scheduling support, however, have the potential to significantly simplify the programming of interactive applications, particularly due to the presence of frequent I/Os within these applications.

In recent years, researchers have begun to address how to best support such interactive parallel applications in task-parallel platforms. Muller et al. [122, 123, 126] describe language extensions to parallel ML [144, 145] to express priorities of tasks. The languages are accompanied by a cost semantics that allows one to bound the response time of high-priority tasks, provided that the computation is well-formed and does not contain any priority inversions, where a high-priority task may wait for a low-priority task to complete to make progress. For well-formed computations, Muller et al. show that a “prompt” scheduler can provide provably efficient response times for high-priority tasks and bounded execution times for low-priority tasks. The accompanying type system checks for the well-formed-ness of programs; if a program type checks, the resulting computation is guaranteed to be well-formed and the corresponding cost semantics hold.

A shortcoming of the prior work by Muller et al. [122, 123, 126] is that the performance guarantees provided by its cost semantics hold only if a scheduler adheres to the prompt scheduling principle which requires that the scheduler is both greedy or work-conserving — no core is idle as long as some work is available to be done — and prompt — strict prioritization such that no core does low-priority work if high-priority work is available.

No prior work has developed a scheduler that strictly maintains, or even provably approximates, prompt scheduling in all cases, however, and doing so is challenging in practice. For example, strict adherence to the protocol would require extremely frequent preemption (so that high-priority work can immediately preempt lower-priority work) and a global queue (so that priority determinations can be made over all the work in the system). The context switching and contention in such an approach would result in unacceptable overhead. Thus, a
practical system must approximate a prompt schedule using coarser-grained preemption and
decentralized queues, much like approaches such as randomized work stealing [23, 24, 39, 40]
approximate work-conserving schedules [42, 76]. The most recent work by Muller et al. [126]
describes an implementation based on distributed load balancing that aims to approximate
prompt scheduling in common cases, but it does not formally analyze the algorithm. Indeed,
in certain cases, their implementation can be shown not to match the theoretical bounds
provided by the cost semantics.

In this chapter, we present Adaptive Priority Scheduling (APS for short), a scheduling
algorithm that approximates the prompt scheduling principle for scheduling computations
with priorities. APS takes inspiration from A-GREEDY [16], an online two-level scheduling
algorithm originally designed for scheduling multiple independent parallel jobs on a single
multicore. Like in A-GREEDY, APS assumes that time is broken into a sequence of
scheduling quanta consisting of $L$ time steps and utilizes an adaptive scheduling strategy
with two-level scheduling. The top level is a processor allocator that determines how to best
assign cores to each priority level for the next quantum. For each priority-level, a work-
conserving scheduler [42, 76] maps available work within the priority level onto the assigned
cores during the quantum. APS can be implemented efficiently because it does not follow
the prompt scheduling principle strictly and the resource allocator changes the allocation of
cores between different priority levels at the granularity of the scheduling quantum of length
$L$ and not at every timestep. The quantum length $L$ can be large enough to amortize the
scheduling and preemption overheads while approximating prompt scheduling.

Even though APS does not follow the prompt scheduling principle strictly, we show that
APS provides provably efficient response time for tasks at all priority levels. To formally
state the bounds, we use a cost model similar to the one based on work and span from prior
work [23, 24, 39, 40]. We model the computation as a directed acyclic graph (or DAG)
where a node denotes a unit-time instruction\(^32\) and an edge denotes the dependence between
a pair of nodes. Like most prior work, we assume that each node has out-degree at most two.

Because interactive applications can be long running, we are concerned with bounding the
execution time of each task as opposed to the overall execution time, where a task $\tau$ is defined
as a sub-DAG with a single source ($\text{source}(\tau)$) and a single sink ($\text{sink}(\tau)$) and consists of

\(^{32}\)This assumption is without loss of generality because a larger node can be represented by a chain of
unit-work nodes.
nodes with the same priority level. Given a task $\tau$ at priority level $\ell$, we define the competitor work $W_{\tau}$ of $\tau$ as the number of nodes in $\tau$ and the nodes logically parallel with $\tau$ that have the same as or higher priority than $\tau$. We say that an induced sub-DAG for a given task $\tau$ is the sub-DAG consisting of $\tau$ and all nodes that have a directed path to sink($\tau$) and that are not proper ancestors of source($\tau$). Then, the $a$-span $S_\tau$ of $\tau$ is defined as the length of the longest path in the induced sub-DAG of $\tau$. Given the competitor work $W_{\tau}$ and $a$-span $S_\tau$ of task $\tau$ with priority level $\ell$, APS with scheduling quantum length $L$ executes $\tau$ in time $T_P(\tau) = O(W_{\tau}/P + S_\tau + (\ell + k)L \log P)$, where $k$ is the number of edges between two nodes with different priority levels in the induced sub-DAG of $\tau$ and $P$ is the total number of cores in the system. Here, the highest priority level has $\ell = 1$, and lower priorities take on a higher value for $\ell$. Thus, a highest-priority task $\tau$ has a response time of $T_P(\tau) = O(W_{\tau}/P + S_\tau + L \log P)$, because $\ell = 1$ and $k = 0$ (in a well-formed DAG with no priority inversion, the induced sub-DAG of a task with priority level $\ell$ has only nodes with priority level $\ell$ or smaller).

To put these bounds into perspective, work by Muller et al. bounds the execution time of a task $\tau$ at any priority level on a “prompt” system with $P$ cores to be $O(W_{\tau}/P + S_\tau)$. Therefore, theoretically, APS incurs an additive overhead of $O((k + \ell)L \log P)$, which is typically small.

We implemented APS in a Cilk-based task-parallel platform [85] called Adaptive Interactive Cilk (or Adaptive I-Cilk for short). Even though APS is analyzed assuming a work-conserving scheduler for each priority level, a strict work-conserving scheduler is difficult to implement efficiently. Thus, Adaptive I-Cilk instead implements a distributed, approximately work-conserving scheduler based on work-stealing [40] to schedule tasks within a priority level. To provide provable bounds, APS necessarily assumes that the DAG is well-formed, which is checked by Adaptive I-Cilk’s C++-template based type system [125], as described in Chapter 5. Finally, to support applications that perform I/O operations, Adaptive I-Cilk provides a library that allows the scheduler to overlap computations with I/O.

With Adaptive I-Cilk, we empirically demonstrate the following. First, Adaptive I-Cilk can provide practically efficient response times for highest-priority tasks and bounded execution times for lower-priority tasks. Second, APS can be implemented efficiently as Adaptive I-Cilk incurs negligible overhead for utilizing the two-level adaptive scheduling algorithm. Finally, when compared to a vanilla scheduler that does not account for priorities, Adaptive I-Cilk

\[33\] Competitor work $W_{\tau}$ and $a$-span $S_\tau$ are defined similarly.
provides better response times for highest-priority tasks and more stable running times for all tasks.

Contributions

In summary, this chapter describes the following contributions:

• **Algorithm**: We propose APS, an adaptive scheduling algorithm that provides provably good response times.

• **System**: We developed Adaptive I-Cilk, a Cilk-based task-parallel platform for interactive parallel applications. Adaptive I-Cilk implements APS and utilizes a practically efficient distributed protocol for load balancing tasks in a given priority level.

• **Evaluation**: Empirical evaluation of Adaptive I-Cilk demonstrates that APS can be implemented efficiently — it incurs little scheduling overhead for its adaptive scheduling strategy and can provide practically efficient response times for high-priority tasks.

This chapter originally appeared in: .

6.1 Background

In this section we explain how interactive applications which contain tasks with different priorities are modeled and also explain the relationship of APS to A-GREEDY.

**Modeling Adaptive I-Cilk Computations.** In a program written using Adaptive I-Cilk, which supports both fork-join parallelism (using `spawn`/`sync` keywords) and futures (using `fcreate`/`ftouch` keywords), tasks have priorities. When one invokes `fcreate`, the future function is always considered a separate task from its caller and has its own priority (potentially different from the caller). This may or may not be the case for `spawn`— the spawned function is considered as a separate task only if it has a different priority than its caller. In terms of the DAG model, one can think of a **task** being a SP DAG with a single source and a single sink whose nodes all have the same priority.
Adaptive I-Cilk incorporates the type checking module that statically checks for priority inversions [125] as described in Chapter 5. For scheduling analysis, we assume that there are no priority inversions — namely, a node (a sync or ftouch) with higher priority never waits for a node with lower-priority to finish. Recall that in the bound presented at the beginning of this chapter, the running time of a high-priority task does not depend on the work of any lower-priority task. However, if a high-priority task can block on a lower-priority task, then the latency of the high-priority task necessarily depends on the latency (and therefore, work) of the low-priority task — therefore, the no priority inversion assumption is essential to prove these bounds.

6.2 Adaptive Priority Scheduling

This section describes APS, the algorithm for running an interactive application with multiple priority levels on a multicore machine with $P$ processors. As mentioned in the introduction to this chapter, APS takes inspiration from A-GREEDY, an adaptive scheduling algorithm designed for scheduling multiple parallel jobs on a shared platform. Like A-GREEDY, APS uses a two-level scheduler. The top level processor allocator operates at the granularity of a scheduling quantum $L$, and decides how many processors to allocate to each priority level for the next quantum. This allocation does not change for the duration of the quantum.

At the second level, each priority level has its own scheduler for mapping its ready nodes onto its currently allocated cores. This scheduler is a work-conserving scheduler — given $p$ allocated processors for the current quantum, at each step, if at least $p$ nodes are ready, it arbitrarily picks any $p$ and schedules them. Such steps are called complete steps because all allocated processors are being used to do work. If fewer than $p$ nodes are ready, then it schedules all ready nodes. Such steps are called incomplete steps because not all processors may be used to do work.

To aid the processor allocation, APS also consists of a desire-calculation module for each priority level $\ell$. This module monitors the scheduler of each priority $\ell$ during each quantum and calculates the desire $d^\ell_q$ — the number of cores to request for priority $\ell$ tasks (all together) for the next quantum $q$. Given the desires from each priority, the processor allocator assigns cores to each priority level — it simulates promptness by always assigning available
cores to the highest priority-level (up to the limit of its desire) before assigning cores to the lower priority levels.

This desire-calculation algorithm is similar to A-GREEDY [16]. The algorithm uses two performance parameters which can be used to make trade-offs. The first is called the **efficiency parameter** $\delta$ and the second is a **responsiveness parameter** $\rho$. If the priority $\ell$ scheduler achieved at least $\delta L$ complete steps in quantum $q - 1$, it decides that the quantum was **efficient** for the priority $\ell$ scheduler; otherwise it declares that the quantum was **inefficient**. In addition, if the quantum $q - 1$ was allocated $p_{q-1}^{\ell} = d_{q-1}^{\ell}$ processors, then the quantum is said to be **satisfied**, otherwise, it is a **deprived** quantum. Thus, each scheduler classifies each quantum as 4 possible classifications: efficient-satisfied, efficient-deprived, inefficient-satisfied and inefficient-deprived. As done by A-GREEDY, we use three of these: efficient-satisfied, efficient-deprived and inefficient — we do not care if an inefficient quantum is satisfied or deprived.

Based on this classification and the desire of quantum $q - 1$, the desire of quantum $q$ is calculated using the algorithm described in Figure 6.1. The idea is simple. When the previous quantum is inefficient, then we assume that the desire was too high and reduce it. When the previous quantum is efficient and satisfied, we speculate and ask for more processors. When the previous quantum was efficient, but deprived, we used the allocated processors efficiently, but because the allocation was smaller than what we asked, we don’t know if our calculation of desire was too low or too high. Therefore, we do not change our demand.

There is one important difference from the original A-GREEDY algorithm. A-GREEDY doesn’t consider the possibility of a program having no ready nodes as an unfinished job has at least one ready node. However, in APS, there may be no ready nodes of a particular priority level at certain times. In this case, the scheduler makes its desire 0 and restarts the desire at 1 when new ready nodes appear.

Based on the desire of all the schedulers, the processor allocator allocates $p_q^{\ell}$ processors to the scheduler at priority level $\ell$ for quantum $q$. The processor allocator gives $p_q^1 = \min\{d_q^1, P\}$ processors to the highest-priority scheduler because this scheduler is scheduling the most latency-sensitive tasks. It then gives $p_q^2 = \min\{d_q^2, P - p^1\}$ processors to the priority 2 scheduler, $p_q^3 = \min\{d_q^3, P - (p_q^1 + p_q^2)\}$ processors to the priority 3 scheduler, and so on. In practice, if the total desire of all levels is smaller than $P$, the remaining cores can be allocated arbitrarily — it does not impact the analysis.

---

34 In practice, if the total desire of all levels is smaller than $P$, the remaining cores can be allocated arbitrarily — it does not impact the analysis.
if no ready nodes of priority $\ell$ then $d^\ell_q = 0$ return
2 if $d^\ell_{q-1} = 0$ then $d^\ell_q \leftarrow 1$ return
3 if more than $(1 - \delta)L$ incomplete steps in $q - 1$
4 then $d^\ell_q \leftarrow d^\ell_{q-1}/\rho$ return $\triangleright$ inefficient
5 if $p^\ell_{q-1} = d^\ell_{q-1}$ $\triangleright$ efficient and satisfied
6 then $d^\ell_q \leftarrow \rho d^\ell_{q-1}$ return
7 $d^\ell_q \leftarrow d^\ell_{q-1}$ return $\triangleright$ efficient and deprived

Figure 6.1: The desire calculation algorithm for quantum $q$ and priority level $\ell$.

other words, it satisfies the desire of the priority 1 scheduler first, then tries to satisfy the
desire of the priority 2 scheduler, and so on until it runs out of processors.

6.3 Response Guarantees of APS

This section describes the analysis of the response time provided by APS. We will consider
an individual task $\tau$ at priority level $\ell$, and we want to show a bound on the responsiveness
or latency of the task — the amount of time that the task can be active or, in other words,
the time that can pass between when the source node of the task becomes ready and its sink
node is executed by APS.

Although APS’s desire calculation algorithm is almost identical to A-GREEDY, there are
significant differences in the analysis and the results. First, A-GREEDY does not dictate
anything about how the processor allocator behaves and provides guarantees for individual
jobs based on the behavior of the processor allocator. On the other hand, in APS, the
processor allocator tries to simulate promptness by preferentially giving processors to higher
priority levels. More significantly, A-GREEDY is designed for independent jobs sharing a
multicore — these jobs do not interact and can be analyzed independently. APS is designed
for a single interactive application where each second level scheduler is scheduling nodes of
a particular priority. Tasks at different priority levels are not independent; for instance, a
low-priority task may wait for the completion of a high-priority task. Thus, the performance
of these tasks cannot be analyzed independently.

Definitions. There are a few important parameters that play a role in a task’s response
time: Competitor work $W_\tau$ includes the work of the task itself and any work of the same
or higher priority that may execute in parallel with the task. This is all the work that the
schedulers at the same or higher priority level can be executing while \( \tau \) is active. The second
is the \textit{a-span} \( S_\tau \) which is the longest chain of nodes that can hold up the completion of the
task. Note that the longest chain may not contain only the nodes of the task itself because
nodes of this task may wait on nodes of other tasks especially in the presence of futures.
Formally, to define the a-span, we will define an \textit{induced DAG} \( G_\tau \) for task \( \tau \), which consists
of all nodes that have a path to the sink node of \( \tau \), but do not have a (non-empty) path to
the source node of \( \tau \). The a-span \( S_\tau \) is the longest path in the induced DAG \( G_\tau \).

We will assume that our interactive application has \textit{no priority inversions} — formally,
this means that the induced DAG \( G_\tau \) of task \( \tau \) at priority level \( \ell \) has no nodes of lower
priorities (priorities with levels \( > \ell \)). Therefore, higher priority tasks do not wait for the
completion of lower priority nodes. In addition, we will define \( k \) as the number of edges that
go between nodes of different priority levels in \( G_\tau \). For tasks with \( \ell = 1 \) (highest priority),
\( k = 0 \) by definition because their induced DAGs can only contain nodes with \( \ell = 1 \). Using
these definitions, we will prove the following theorem:

\textbf{Theorem 28.} A task \( \tau \) with priority level 1 (the highest priority level) has a response time
of at most \( \frac{W_\tau}{\delta P} + O\left(\frac{S_\tau}{(1-\delta)} + L \lg P\right) \). A task \( \tau \) with priority level \( \ell > 1 \) (a lower priority level)
has a response time of at most \( (1 + \rho) \cdot \frac{W_\tau}{\delta P} + O\left(\frac{S_\tau}{(1-\delta)} + (\ell + k)L \lg P\right) \).

\textbf{Potential Function.} We will prove the theorem using an amortized analysis with a potential
function. For every node \( u \in G_\tau \), we define the weight \( w(u) \) of the node \( u \) as the length
of the longest path from \( u \) to the sink of \( \tau \). We say that a ready node \( u \) has a potential
\( \Phi(u) = 2^{2w(u)} \); nodes which are not ready or have already executed have no potential.\footnote{Because we consider a single task \( \tau \) at a time, we omit it from the definition, even though the same node
may be part of induced DAGs of multiple tasks and have potential with respect to all those tasks.}
For task \( \tau \), the potential of the task at time \( t \) is \( \Phi_\tau(t) = \sum_{u \in G_\tau} \Phi(u) \).

The following Lemma follows from the fact that the longest path in the induced DAG \( G_\tau \) is
of length \( S_\tau \).

\textbf{Lemma 29.} The total potential of the task when the task becomes active is at most \( 2^{2S_\tau} \); the
final potential when the task ends is 0 and the potential of a task never increases.

\textit{Proof.} A task becomes active when its source node becomes active. The potential of the
source node is at most \( 2^{2S_\tau} \) because the source is part of \( G_\tau \) and thus can have weight at
most \( S_\tau \). In addition, some other nodes \( u \) which are part of the induced subdag \( G_\tau \) may be ready when the source becomes active and all of these can have potential at most \( 2^{2S_\tau} \) for the same reason. However, all these nodes must be part of other futures which will eventually (recursively, possibly) join with \( \tau \) — otherwise, there cannot be a path from \( u \) to the sink of \( \tau \). Because only two futures can participate in a join at a time, and the joins must eventually occur on some chain of length at most \( S_\tau \), the total number of such nodes that can join before sink of \( \tau \) is \( 2^{S_\tau} \). Therefore, the total potential of all these nodes together is at most \( 2^{3S_\tau} \) when the task becomes active. When the task completes, all of the nodes in its induced subdag have completed — therefore, the potential is 0.

At any point, the potential only changes when a node, say \( u \), finishes executing, and enables either 0, 1 or 2 children. It is easy to check that even if it enables two children, say \( v \) and \( v' \), the reduction in potential is \( 2^{w(u)} - 2^{w(v)} - 2^{w(v')} \geq 2^{w(u)} - 2^{w(u)-1} - 2^{w(u-1)} \geq \Phi(u)/2 \). Therefore, the potential always reduces.

**Counting Efficient-Satisfied and Inefficient Quanta.** We want to amortize certain quanta against others during certain time intervals, but we first need the following Lemma which is indirectly proved for A-GREEDY; here we state and prove it directly.

**Lemma 30.** Each inefficient quantum \( q \) with desire \( d_q \) can be mapped to a prior efficient and satisfied quantum with desire \( d_q/\rho \) for the same scheduler such that no two inefficient quanta are mapped to the same efficient and satisfied quantum.

**Proof.** Consider any inefficient quantum \( q \) with desire \( d_q \). We go backward in time to find the nearest efficient and satisfied quantum \( q' \) with desire \( d_q/\rho \) and we say that inefficient quantum \( q \) is mapped to efficient and satisfied quantum \( q' \). We can always find such a quantum because the desire cannot increase to \( d_q \) without there being a prior efficient and satisfied quantum with desire \( d_q/\rho \). The only way another quantum \( q'' \) could be mapped to the same quantum \( q' \) is if \( q'' \) had desire \( d_q \) and was between \( q \) and \( q' \), so that \( q' \) is the closest efficient and satisfied quantum for both \( q \) and \( q'' \). However, if such a \( q'' \) existed, then the desire would have reduced to \( d_q/\rho \) after \( q'' \) and we would need another efficient and satisfied quantum \( q''' \) after \( q'' \) with desire \( d_q/\rho \) so that the desire can increase to \( d_q \) again, which is a contradiction because \( q''' \) is closer to \( d_q \) than \( q' \).
We now show how to amortize certain quanta against others during certain time intervals in the following lemma. Consider a period of time from $t_1$ to $t_2$ and say $I^\ell(t_1, t_2)$ is the number of inefficient quanta within this period for priority $\ell$ and say $ES^\ell(t_1, t_2)$ is the number of efficient-satisfied quanta.

Lemma 31. Consider a scheduler at priority $\ell$ and say that the desire of this scheduler is never 0 between time $t_1$ and $t_2$. Say $I^\ell(t_1, t_2)$ is the number of inefficient quanta between time $t_1$ and $t_2$ and $ES^\ell(t_1, t_2)$ is the number of efficient and satisfied quanta within the same time. (We ignore partial quanta at the beginning and end of the period). Then,

$$I^\ell(t_1, t_2) \leq ES^\ell(t_1, t_2) + \log_\rho P$$

Proof. The two statements are somewhat symmetric — we consider the first one. From Lemma 30, we can find a mapping from inefficient quanta to efficient and satisfied quanta. There are two cases: Either $q$’s mapped quantum $q'$ is after time $t_1$ and therefore within the interval $t_1$ to $t_2$ — there can be only $ES^\ell(t_1, t_2)$ such quanta. Or this mapped quantum is before $t_1$ and we will now argue that there can be at most $\log_\rho P$ such quanta. We know that there were no other inefficient quanta with desire $d_q$ between $q'$ and $q$ and therefore, no other inefficient quanta with desire $d_q$ between $t_1$ and quantum $q$. Therefore, there can be only one inefficient quantum with desire $d_q$ which cannot be mapped to an efficient and satisfied quantum after time $t_1$. There are only $\log_\rho P$ different possibilities of desire because desire increases and decreases multiplicatively by factor $\rho$. Hence there can only be $\log_\rho P$ inefficient quanta that cannot be mapped. Considering both cases, we have

$$I^\ell(t_1, t_2) \leq ES^\ell(t_1, t_2) + \log_\rho P.$$ 

We can do a similar sort of mapping from efficient and satisfied quanta to inefficient quanta. We can uniquely map an efficient and satisfied quantum $q$ with desire $d_q$ to the closest inefficient quantum $q'$ later in time with desire $\rho d_q$ — unless $q$ is the last quantum with desire $d_q$ in the time period (or ever). Again there are only $\log_\rho P$ quanta that cannot be mapped, giving us

$$ES^\ell(t_1, t_2) \leq I^\ell(t_1, t_2) + \log_\rho P.$$ 

6.3.1 Response time of tasks with priority 1

To build intuition, we will begin by analyzing the response time of highest priority tasks (level 1). We first bound the number of efficient-deprived quanta for the priority 1 scheduler
while priority 1 task $\tau$ is running. Note that the priority 1 scheduler is only deprived when it has a desire larger than $P$.

**Lemma 32.** The total number of efficient and deprived quanta for priority 1 scheduler during the execution of $\tau$ is at most $W_\tau/(\delta LP)$

**Proof.** During deprived quanta, the priority 1 scheduler has $P$ processors. Because they are also efficient, at least $\delta L$ steps are complete steps where all $P$ processors are doing ready work. Thus, the work done during these quanta is at least $\delta LP$. Because all work done by the priority 1 scheduler while $\tau$ is executing is part of competitor work $W_\tau$, we can have at most $W_\tau/(\delta LP)$ such quanta.

We next show that each inefficient quantum reduces the potential of task $\tau$.

**Lemma 33.** During any incomplete step of the priority 1 scheduler, the potential of all currently active priority 1 tasks decreases by a factor of 2. Thus, during an inefficient quantum for priority 1 scheduler, the potential of a currently active priority 1 task $\tau$ decreases by a factor of $2^{(1-\delta)L}$.

**Proof.** While a task $\tau$ is active, some node(s) in its $G_\tau$ must be ready and these nodes are priority 1 nodes (there are no priority inversions). During an incomplete step for the priority 1 scheduler, all priority 1 ready nodes and, therefore, all ready nodes $u \in G_\tau$ are executed. A particular ready node $u$ enables at most two children, say $v$ and $v'$. After $u$ executes, the potential due to $v$ and $v'$ is $\Phi(v) + \Phi(v') = 2^{2w(v)} + 2^{2w(v')} \leq 2 \times 2^{2w(u)-2} = \Phi(u)/2$. Thus, after an incomplete step, the potential due to all ready nodes $u \in G_\tau$ reduces by at least a factor of 2, reducing $\Phi_\tau$ by at least a factor of 2. The potential decrease for inefficient quanta follows because they contain at least $(1-\delta)L$ incomplete steps.

The following lemma follows from Lemmas 33 and 29.

**Lemma 34.** The total number of inefficient quanta during the execution of a priority 1 task $\tau$ is at most $3S_\tau/((1-\delta)L)$.

**Proof.** When a task $\tau$ starts executing, the total potential of the task is at most $2^{3S_\tau}$. (It can be lower because some nodes $u \in G_\tau$ may already have been executed before the task
starts if they are part of other tasks.) In every inefficient quantum, it reduces by a factor of $2^{(1-\delta)L}$. Therefore, if the total number of inefficient quanta before the task finishes is $I$, then $2^{3S_\tau} / 2^{I(1-\delta)L} < 1$. Solving for $I$ gives us the number of inefficient quanta.

Combining with Lemma 31 gives us the following corollary because the priority 1 scheduler’s desire is never 0 while $\tau$ is active.

**Corollary 35.** The total number of efficient and satisfied quanta during the execution of a high-priority task $\tau$ is at most $3S_\tau/((1-\delta) L) + \log P$.

Lemmas 32, 33 and Corollary 35 gives us the result stated in Theorem 28 for priority 1 tasks.

### 6.3.2 Response time of tasks with priority 2

Analyzing the response time of lower priority tasks is more challenging for the following reasons.

1. Lemma 32 depends on the fact that when the priority 1 scheduler has an efficient and deprived quantum, all $P$ are allocated to it and are doing competitor work. When a priority 2 scheduler has an efficient and deprived quantum, its allocation is not necessarily $P$ because some processors may have been allocated to the priority 1 scheduler. Therefore, an analog of Lemma 32 does not directly hold.

2. The induced DAG $G_\tau$ of a priority 2 task $\tau$ can have both priority 1 and 2 nodes and some of the potential of $\tau$ belongs to priority 1 nodes. Therefore, an analog of Lemma 33 does not hold directly — an inefficient quantum for the priority 2 scheduler does not reduce the potential of an active priority 2 task if most of its potential is in the priority 1 nodes (which may not execute during this quantum). Therefore, we must consider both priority 1 and 2 schedulers when analyzing a priority 2 task.

When analyzing a priority 2 scheduler, we must consider the quantum classifications of both priority 1 and 2 schedulers — giving us 9 types of quanta. However, we will divide them into two types of quanta. The first type — we call it type $A$ — is when either the priority 1 or the priority 2 scheduler is efficient and deprived. We will bound the number of type A quanta using the competitor work of $\tau$. The second type — called type $B$ — is when
neither is efficient and deprived and therefore priority 1 and 2 schedulers are either efficient and satisfied or inefficient during these quanta. We will bound the number of type B quanta as a function of the a-span $S_\tau$. Note that if either scheduler has desire 0, the other scheduler can still completely classify the quantum. This covers all cases because, while $\tau$ is active, one or the other scheduler must have ready nodes.

**Lemma 36.** The total number of type A quanta while a priority 2 task $\tau$ is executing is at most $(1 + \rho)W_\tau/(\delta LP) + \log_\rho P$.

**Proof.** We consider a few sub-cases of type A quanta.

**Priority 1 scheduler was efficient and deprived:** Say there were $X$ such quanta and the total priority 1 work done during these quanta was $W_1$ (no lower priority work executed because the priority 1 scheduler got all cores). During each quantum, at least $\delta LP$ priority 1 work was done; therefore, $W_1 \geq X\delta LP$.

**Priority 1 scheduler was efficient and satisfied and priority 2 scheduler was efficient and deprived:** Say there were $Y$ such quanta and say $W_2$ priority 1 and 2 work was done during these quanta. Again all $P$ processors were allocated jointly to the two schedulers and both schedulers were efficient. Therefore, again, we have $W_2 \geq Y\delta LP$.

Considering just these two cases, all the work done in these quanta is competitor work for $\tau$. Therefore, we have $W_\tau \geq W_1 + W_2 \geq (X + Y)\delta LP$. Therefore, we have $X + Y \leq W_\tau/(\delta LP)$. We have one more case.

**Priority 1 scheduler was inefficient and priority 2 scheduler was efficient and deprived:** This is the complicated case. Consider one such quantum $q$ and suppose the desire of the priority 1 scheduler during this quantum was $d_{1q}^1$. The allocation for the priority 1 scheduler during quantum $q$ was $p_{1q}^1 = d_{1q}^1$. Therefore, the allocation to the priority 2 scheduler was $p_{2q}^2 = P - d_{1q}^1$ (because it was deprived, it got all remaining cores). Because the priority 2 scheduler was efficient, the total work done during that quantum was at least $W_{2q}^2 \geq \delta L p_{2q}^2 = \delta L(P - d_{1q}^1)$.

Now we look for the previous quantum $q'$ when the priority 1 scheduler was efficient and satisfied with desire and allocation exactly $p_{1q'}^1 = d_{1q'}^1 = d_{1q}/\rho$. As argued in Lemma 31, we can always find such a quantum and there is a unique mapping from $q$ to $q'$. We have two cases: First, $q'$ occurred after $\tau$ became active. The total priority 1 work done during
\( q' \) is at least \( W_{q'}^1 \geq \delta L_{q'}^1 = \delta L_{q'}^2/\rho \) and all this work is part of competitor work \( W_{\tau} \) because it is executed by a higher priority scheduler while \( \tau \) is active. Therefore, we have \( W_{q'}^1 + W_{q'}^2 \geq \delta L_{q'}^1 + \delta L(P - d_{q'}) \geq \delta LP/\rho \). All this work was part of \( W_{\tau} \) because it occurred while the task was active and was at a higher or the same priority. Second, \( q' \) may have occurred before \( \tau \) started executing — by logic similar to Lemma 31, there can be at most \( \log_\rho P \) such quanta while \( \tau \) was executing. Say there were \( Z \) such quanta (where priority 2 scheduler is efficient and deprived and priority 1 scheduler is inefficient) — we would get \( Z \leq W_{\tau}/(\delta LP) \). We get \( X + Y + Z \leq (1 + \rho) W_{\tau}/(\delta LP) + \log_\rho P \) type A quanta.

Now we must consider type B quanta. We will use the potential function to bound these quanta using the \( a \)-span of \( \tau \). The fundamental reason this analysis is complicated is as follows: A priority 2 task can have both priority 1 and 2 nodes in its \( G_{\tau} \) — therefore, some of the nodes that contain the potential of this task are being executed by each scheduler, potentially. If both priority 1 and 2 schedulers were inefficient in some quantum \( q \), then we could use an argument similar to the one in Lemma 33 to show that the potential decreased for the priority 2 task \( \tau \) during that quantum. However, if only one of the two schedulers is inefficient (and the other is efficient and satisfied) or if both schedulers are efficient and satisfied, then we cannot say anything about the total decrease in potential.

Therefore, we divide the priority 2 task potential into two components — one consisting of priority 1 nodes and one consisting of priority 2 nodes. That is, \( \Phi_{\tau}^1(t) = \sum_{\text{priority 1 node } u \in G_{\tau}} \Phi(u) \) and \( \Phi_{\tau}^2(t) = \sum_{\text{priority 2 node } u \in G_{\tau}} \Phi(u) \). The total potential \( \Phi_{\tau}(t) = \Phi_{\tau}^1(t) + \Phi_{\tau}^2(t) \).

Now, we hope to make an argument similar to the one in Lemma 33 to say that if, for instance, the priority 1 scheduler had an inefficient quantum, then \( \Phi_{\tau}^1 \) will decrease by a large fraction in that quantum. Unfortunately, this is not always true. Even though \( \Phi_{\tau} \) always decreases (as shown in Lemma 29), it is not true that either \( \Phi_{\tau}^1(\tau) \) or \( \Phi_{\tau}^2(\tau) \) always decreases. In fact, sometimes \( \Phi_{\tau}^1(t) \) may become 0 if there are no priority 1 ready nodes from \( G_{\tau} \) at time \( t \) and then increase again later when some priority 2 node executes and enables a priority 1 child. The same thing can happen to \( \Phi_{\tau}^2(\tau) \). In other words, potential sometimes gets transferred from \( \Phi_{\tau}^2(\tau) \) to \( \Phi_{\tau}^1(\tau) \) and vice versa, even though their sum never increases.

So, we first state a Lemma about how many times this potential transfer can happen — in other words, how many times can \( \Phi_{\tau}^1 \) or \( \Phi_{\tau}^2 \) increase while \( \tau \) is active. The proof of the following lemma relies on the fact that there are at most \( k \) edges between nodes of different
priority levels in $G_\tau$ and the potential of priority 1 can increase only if a priority 2 node enables a child node of priority 1 (and vice versa).

**Lemma 37.** The total number of times $\Phi^1_\tau$ or $\Phi^2_\tau$ can increase is at most $k$.

*Proof.* The only time $\Phi^1_\tau$ can increase is when a priority 2 node $u \in G_\tau$ executes and enables a child node $v$ which is a priority 1 node. By an argument similar to Lemmas 29 and 33, if a priority 1 node enables a priority 1 node, the potential only decreases because child nodes have much lower potential than their parents. There is a symmetric argument for priority 2 potential increase. Because there are only $k$ edges between nodes of different priorities in $G_\tau$, this increase can occur at most $k$ times. \qed

The quantum during which a potential transfer occurs is called a *transfer quantum*. We divide $\tau$’s execution into *phases* between these transfer quanta — the first phase starts when $\tau$ is created and ends when the first potential transfer happens. All subsequent phases begin when the previous phase ends and ends at the next potential transfer or when $\tau$ finishes executing. Therefore, there are at most $k$ transfer quanta while $\tau$ is executing and at most $k + 1$ phases.

We now consider each phase individually. Consider a phase $Q$ that begins at time $t_1$ and ends at time $t_2$. We want to bound the number of type B quanta during this phase (not counting the transfer quanta that begin and end the phase). The potential of task $\tau$ at time $t_1$ is $\Phi_\tau(t_1) = \Phi^1_\tau(t_1) + \Phi^2_\tau(t_1)$ and at time $t_2$ is $\Phi_\tau(t_2) = \Phi^1_\tau(t_2) + \Phi^2_\tau(t_2)$. From Lemma 29, we have $\Phi_\tau(t_2) \leq \Phi_\tau(t_1)$.

**Lemma 38.** Say there are $I^2 \geq 0$ inefficient quanta for priority 2 scheduler during the phase $Q$ and $I^1 \geq 0$ inefficient quanta for the priority 1 scheduler during the phase $Q$. Then $\Phi^1_\tau(t_2) \leq \Phi^1_\tau(t_1) / \left(2^{(1-\delta)L \min\{I^1, I^2\}}\right)$. Therefore, $\min\{I^1, I^2\} \leq \frac{\lg \Phi^1_\tau(t_1) - \lg \Phi^1_\tau(t_2)}{(1-\delta)L}$.

*Proof.* Because there was no potential transfer during the phase, every inefficient quantum decreased the priority 2 potential $\Phi^2_\tau$ by a factor of $2^{(1-\delta)L}$ by the same logic as Lemma 33. Therefore, $I^2$ inefficient quanta reduced $\Phi^2_\tau$ by a factor of $2^{(1-\delta)L I^2}$ — implying $\Phi^1_\tau(t_2) \leq \Phi^1_\tau(t_1) / 2^{(1-\delta)L I^2}$. Note that in this case, the potential $\Phi^2_\tau$ may become 0 at some point during the quantum — however, the above inequality trivially holds in that case and the potential cannot increase again during the quantum because it is not a transfer quantum.
Using a similar argument, we have $\Phi_1^\tau(t_2) \leq \Phi_1^\tau(t_1)/2^{(1-\delta)LI^1}$. Therefore, we have $\Phi_\tau(t_2) = \Phi_1^\tau(t_2) + \Phi_2^\tau(t_2) \leq \Phi_1^\tau(t_1)/2^{(1-\delta)LI^1} + \Phi_2^\tau(t_1)/2^{(1-\delta)LI^2} \leq (\Phi_1^\tau(t_1) + \Phi_2^\tau(t_2))/2^{(1-\delta)L\min\{I^1, I^2\}}$

More algebra gives the bound on $\min\{I^1, I^2\}$.

The following lemma follows from Lemma 31.

**Lemma 39.** The total number of type $B$ quanta during a phase $Q$ that goes from time $t_1$ to time $t_2$ is at most $2\log_\rho P$.

**Proof.** Recall that, by Lemma 31, we know that during the phase, the total number of efficient and satisfied quanta is at most $ES^2 \leq I^2 + \log_\rho P$ for the priority 2 task and $ES^1 \leq I^1 + \log_\rho P$. In addition, the total number of type $B$ quanta is at most $\min\{I^1 + ES^1, I^2 + ES^2\}$ because both schedulers have to inefficient or efficient and satisfied during these quanta. Therefore, the total number of type $B$ quanta is at most $2\min\{I^1, I^2\} + \log_\rho P$ giving us the required bound.

We can now bound the type $B$ quanta for task $\tau$ — the proof uses Lemmas 29 and 39 to argue about initial potential and change in potential.

**Lemma 40.** The total number of type $B$ quanta across the entire execution of the priority 2 task is $O(S_\tau/((1 - \delta)L) + (k + 1)\log_\rho P)$.

**Proof.** We know that the beginning potential of the task when it is created is at most $2^{3S_\tau}$ and the final potential right before the task ends is less than 2. Say that there are $X \leq k$ phases for task $\tau$. Say there are $B_i$ type $B$ quanta during phase $i$ and phase $i$ begins at time
$t_{i-1}$ and ends at time $t_i$. From Lemma 39, we know that $B_i \leq 2 \frac{\lg \Phi_\tau(t_{i-1}) - \lg \Phi_\tau(t_i)}{(1-\delta)L} + \log_\rho P$.

Therefore, if we add $B_i$’s over all phases, we get

$$
\sum_{i=1}^{X} B_i \leq \sum_{i=1}^{X} 2 \frac{\lg \Phi_\tau(t_{i-1}) - \lg \Phi_\tau(t_i)}{(1-\delta)L} + \log_\rho P
\leq 2 \frac{\lg \Phi_\tau(t_0) - \lg \Phi_\tau(t_X)}{(1-\delta)L} + X \log_\rho P
\leq 2 \frac{\lg 2^{3S_\tau}}{(1-\delta)L} + X \log_\rho P
\leq 2 \frac{3S_\tau}{(1-\delta)L} + (k + 1) \log_\rho P
$$

In addition, we have $k + 2$ transfer quanta, but that term is subsumed by the last term.

Adding type A and type B quanta and multiplying by $L$ gives us the bound on response time for priority 2 tasks as stated in Theorem 28. Intuitively, it turns out that the analysis of type B quanta does not change at all; however, for type A quanta, we get an additional $\ell \log_\rho P$ quanta for the following reason. Consider the proof of Lemma 36 and look at the third case where priority 1 quanta are inefficient and priority 2 quanta are efficient and deprived. For a scheduler at priority $\ell$, in the worst case, all schedulers of level $< \ell$ may be inefficient — if all of these quanta can be mapped to prior quanta that started after $\tau$, then like in the proof, we can account for them using competitor work. However, if any of them cannot be mapped, then we get an additive factor of $\log_\rho P$, giving us a total additive factor of $(\ell - 1) \log_\rho P$.

### 6.3.3 Generalizing to priority level $\ell$

Now we consider some arbitrary priority level. Again, we can divide quanta into two categories — type A quanta are those where some scheduler with priority $x \leq \ell$ is efficient and deprived and type B quanta are quanta where all schedulers with priority $x \leq \ell$ are either inefficient or efficient and satisfied.

It turns out that the analysis of type B quanta does not change as we increase the priority level $\ell$. In particular, again, we can divide the potential into $\ell$ different components and the number of transfer quanta is still bounded by $k$. In each phase (time between transfer
quanta), we can easily generalize Lemma 39 to show the same bound on the decrease in potential and then use it to show Lemma 40. In other words, increasing priorities does not increase the bound on type B quanta.

Type A quanta are a different beast because the bound depends on \( \ell \) — but only in the last term.

**Lemma 41.** The total number of type A quanta while a priority \( \ell \) task \( \tau \) is executing is at most \((1 + \rho)W_\tau/((\delta L P) + \ell \log \rho P)\).

**Proof.** Recall that if a scheduler with priority level \( x \) was efficient and deprived, no scheduler with priority \( > x \) gets any processors. First, let us consider the simple case: All schedulers which got any processors were efficient (either satisfied or deprived). This includes all cases where the scheduler at priority level \( x \leq \ell \) was efficient and deprived and all schedulers with priority level \( < x \) were efficient and satisfied. (In Lemma 36, this includes the first two cases.) Say there were \( A \) such quanta and \( W_1 \) work was done over all these quanta on priority \( 1 - \ell \) tasks. Because all \( P \) processors were allocated to priorities \( \leq \ell \) and the quanta were efficient, we have \( W_1 \geq \delta L A \). Because all this work is competitor work of \( \tau \), we get \( A \leq W_\tau/((\delta L)) \).

Now we must consider the case where some quanta were inefficient and some quanta were efficient — consider one such quantum \( q \). By definition of type A quanta — some scheduler with priority level \( x \leq \ell \) is efficient and deprived during \( q \). All schedulers with priority level \( < x \) are either efficient and satisfied or inefficient — the worst case is that they are all inefficient and we will consider that case. Consider the inefficient quantum \( y \) for priority level \( y < x \) and say that the allotment of that quantum was \( p_y^q = d_y^q \) (because there were processors left over for lower priority tasks). From Lemma 30, we can find a previous efficient and deprived quantum \( q'(y) \) with allotment \( p_{q'(y)}^y = d_{q'(y)}^y = d_y^q/\rho \). We can find a similar mapping for all priority levels \( y < x \) with inefficient quanta (though the \( q'(y) \) may be different for each \( y \)).

There are two cases: For all \( y \), this \( q'(y) \) occurred after task \( \tau \) became active. In this case, we know that \( \sum_{y<x} p_{q'(y)}^y = \sum_{y<x} d_y^q/\rho \). Therefore, the total work done by the priority \( 1 - x - 1 \) schedulers during these mapped quanta (which are efficient) is at least \( W_q' \geq \delta L \sum_{y<x} d_y^q/\rho \). In addition, the work done by the scheduler at priority level \( x \), which is efficient and deprived, is at least \( W_q \geq \delta L (P - \sum_{y<x} d_y^q) \). Adding these together, we get \( W_q + W_q' \geq \delta LP/\rho \). If there
are \( B \) such quanta, and all this work is competitor work of \( \tau \) (because it was all executed by a higher or equal priority scheduler while \( \tau \) was active), we get \( B \leq \rho W_\tau / (\delta L) \).

The final case to consider is that for some \( y \), the mapped quantum \( q'(y) \) occurred before task \( \tau \) became active. In this case, we cannot argue that the work done during \( q'(y) \) was competitor work. However, from Lemma 31, we know that we can find at most \( \log_\rho P \) such quanta for each priority level \( < x \). Therefore, the total number of quanta of this type is \( (\ell - 1) \log_\rho P \).

Adding all cases gives us the bound on type A quanta.

Because the bound on type B quanta does not change, combining Lemmas 41 and 40, and multiplying with \( L \) gives us the bound stated in Theorem 28.

6.4 Empirical Evaluation

This section empirically evaluates Adaptive I-Cilk, which implements APS. We evaluated Adaptive I-Cilk using two microbenchmarks and three moderately sized application benchmarks (that range from 1.1k to 1.5k lines of code). The microbenchmarks are written such that they can be configured to generate different workloads to allow us to better evaluate different aspects of the scheduler. The application benchmarks consist of richer workloads that simulate real-world interactive applications.

Implementation of Adaptive I-Cilk. Adaptive I-Cilk extends Cilk-L, a Cilk dialect that uses proactive work stealing [141], as discussed in Chapter 3, to schedule futures. Adaptive I-Cilk incorporates latency-hiding I/O support as described in Chapter 4, a priority type system as described in Chapter 5, and an adaptive processor allocator.

Adaptive I-Cilk implements a two-level adaptive scheduling strategy as described in Section 6.2. The top-level processor allocator adaptively allocates workers (surrogates of processing cores) to priority levels using a thread known as the conductor. The second level uses an extension of proactive work stealing to schedule tasks within a priority level. Each worker periodically updates its utilization based on how long it spent working versus looking for work to do. The conductor is a thread that sleeps for the duration of a quantum \( (L) \), collects utilization
reported, calculates the core utilization at each priority level, computes the desire of each priority level, and allocates workers to each level. If there are left-over cores not allocated based on desires, the runtime assigns half of the left-over cores to each priority level, starting from the highest level to the lowest until it runs out of cores. The conductor alerts a worker to switch priority by setting a per-worker flag, checked at each strand boundary (i.e., at a \texttt{spawn, sync, fcreate, and ftouch}) or at a user-specified yield via a runtime library call\textsuperscript{36}, and switches to its new priority level if the flag is set.

In classic work stealing, each worker has one deque storing its work items. In Adaptive I-Cilk, because each worker may work on different priority levels throughout execution, each worker ends up having multiple pools of deques, one for each priority level, and like in Cilk-L, there can be multiple deques within each pool. In Adaptive I-Cilk, however, a worker can generate additional deques when it needs to switch levels or when it generates a new task with a different priority level from its own.

**Experimental Setup.** We empirically evaluate Adaptive I-Cilk by comparing its performance against that of Cilk-L. Our experiments ran on a computer with 2 Intel Xeon Gold 6148 processors, each with 20 2.40-GHz cores. Each core has a 32-kB L1 data and 32-KB L1 instruction cache, and a private 1 MB L2 cache. Hyperthreading was enabled, and each core had 2 hardware threads. Each processors has a 27.5 MB shared L3 cache, and there are 768 GB of main memory. Adaptive I-Cilk and all benchmarks were compiled using the Tapir compiler \textsuperscript{[137]} (based on clang 5.0.0), with -O3 and -flto. Experiments ran in Linux kernel 4.15.

For the choices of runtime parameters, we expect $\delta$ (efficiency parameter) to range between 0 and 1, and is likely close to 1, and $\rho$ (responsiveness parameter, or how fast to grow the desire) to range between 1 and 2. For all evaluations, we have tested several sensible configurations of runtime parameters. Unless stated explicitly (i.e., when testing the sensitivities to parameters), we show the configuration that best benefits the high-priority tasks. We discuss how runtime parameters may impact execution time at the end.

\textsuperscript{36}This yield functionality is an optimization for programs with long strands, allowing the programmer to indicate to the runtime when it is safe to interrupt.
Table 6.1: Execution times of fib-ep (seconds) run using vanilla Cilk-L and Adaptive I-Cilk with \( \delta = 0.9 \) and \( L = 1 \text{ms} \). fib-ep was run with \( \rho = 2 \). Overhead, relative to fib-ideal, and standard deviation are in parentheses.

<table>
<thead>
<tr>
<th></th>
<th>( T_1 )</th>
<th>( T_3 )</th>
<th>( T_8 )</th>
<th>( T_{16} )</th>
<th>( T_{20} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>11.61 (1.00%, 0.0%)</td>
<td>5.98 (2.06%, 27.7%)</td>
<td>2.24 (1.54%, 35.9%)</td>
<td>1.27 (1.74%, 21.4%)</td>
<td>0.95 (1.64%, 27.2%)</td>
</tr>
<tr>
<td>Cilk-L M</td>
<td>23.22 (2.00%, 0.0%)</td>
<td>8.05 (2.77%, 7.1%)</td>
<td>3.67 (2.52%, 12.1%)</td>
<td>2.07 (2.84%, 7.8%)</td>
<td>1.73 (2.96%, 1.3%)</td>
</tr>
<tr>
<td>L</td>
<td>34.83 (3.00%, 0.0%)</td>
<td>8.62 (2.97%, 2.5%)</td>
<td>4.36 (2.99%, 0.0%)</td>
<td>2.18 (2.99%, 0.06%)</td>
<td>1.74 (2.98%, 1.1%)</td>
</tr>
<tr>
<td>H</td>
<td>12.53 (1.08%, 0.0%)</td>
<td>3.13 (1.08%, 0.2%)</td>
<td>1.57 (1.08%, 0.1%)</td>
<td>0.79 (1.08%, 0.4%)</td>
<td>0.63 (1.08%, 0.3%)</td>
</tr>
<tr>
<td>Adaptive I-Cilk M</td>
<td>25.05 (2.16%, 0.0%)</td>
<td>6.24 (2.15%, 0.1%)</td>
<td>3.13 (2.15%, 0.1%)</td>
<td>1.57 (2.15%, 0.1%)</td>
<td>1.26 (2.15%, 0.1%)</td>
</tr>
<tr>
<td>L</td>
<td>37.52 (3.23%, 0.0%)</td>
<td>9.36 (3.22%, 0.1%)</td>
<td>4.69 (3.22%, 0.0%)</td>
<td>2.34 (3.22%, 0.0%)</td>
<td>1.88 (3.22%, 0.1%)</td>
</tr>
</tbody>
</table>

6.4.1 Evaluation of Microbenchmarks

We use two microbenchmarks to answer the following questions. First, does Adaptive I-Cilk appropriately prioritize tasks in the order of their priorities? Second, how much overhead does the two-level adaptive scheduling strategy incur in Adaptive I-Cilk? Finally, how do the changes in the scheduling parameters of Adaptive I-Cilk (i.e., scheduling quantum length \( L \), responsiveness parameter \( \rho \), and utilization parameter \( \delta \)) impact execution times of tasks at different priority levels? To answer these questions, we utilized the microbenchmarks to generate different workloads and compare the average execution times of tasks at each priority level when running on Adaptive I-Cilk versus Cilk-L.

**Microbenchmark.** The first microbenchmark, fib-ep, computes the 42nd fibonacci number with a serial base case of 2, which has ample parallelism. The second microbenchmark, fib-rp, similarly computes the 44th fibonacci number but with a serial base case of 40, thus restricting the number of parallel strands to 8 for each priority level. In both microbenchmarks, fibonacci computations are spawned for high (H), medium (M), and low (L) priority in succession. The serial fibonacci base case in Adaptive I-Cilk uses yield calls to check if the worker needs to switch priorities.

Because fib-rp does not have much parallelism, it is an adversarial workload for Adaptive I-Cilk. Each task cannot fully utilize all the cores, so the desire for each priority level will oscillate between two different values, as falling slightly below the parallelism or overshooting and causing low utilization.

**Prioritization of Tasks.** We compare the runtime of tasks at each priority level running on Adaptive I-Cilk and Cilk-L. Here, we choose the optimal runtime parameters for Adaptive I-Cilk (sensitivity to parameters evaluated later). Figure 6.1 and Figure 6.2 show the execution times for tasks at each level, with overhead and standard deviation shown in parentheses. The
Table 6.2: Execution times of \(\text{fib-rp}\) (seconds) run using vanilla Cilk-L and Adaptive I-Cilk with \(\delta = 0.9\) and \(L = 1 ms\). \(\text{fib-rp}\) was run with \(\rho = 1.2\). Overhead, relative to \(\text{fib-ideal}\), and standard deviation are in parentheses.

<table>
<thead>
<tr>
<th></th>
<th>(T_{16})</th>
<th>(T_{20})</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(T_{16})</td>
<td>(T_{20})</td>
</tr>
<tr>
<td>H</td>
<td>0.95 (1.22x, 20.8%)</td>
<td>0.78 (1.00x, 0.5%)</td>
</tr>
<tr>
<td>Cilk-L M</td>
<td>1.10 (1.42x, 19.9%)</td>
<td>1.04 (1.34x, 20.6%)</td>
</tr>
<tr>
<td>L</td>
<td>1.43 (1.85x, 11.1%)</td>
<td>1.14 (1.47x, 14.1%)</td>
</tr>
<tr>
<td>Adaptive I-Cilk</td>
<td>0.87 (1.13x, 0.2%)</td>
<td>0.87 (1.12x, 0.3%)</td>
</tr>
<tr>
<td>M</td>
<td>1.14 (1.47x, 0.6%)</td>
<td>1.04 (1.12x, 0.1%)</td>
</tr>
<tr>
<td>L</td>
<td>1.68 (2.17x, 2.1%)</td>
<td>1.27 (1.63x, 0.2%)</td>
</tr>
</tbody>
</table>

Table 6.3: Execution time of \(\text{fib-ep}\) and \(\text{fib-rp}\), in seconds, run in Adaptive I-Cilk with various \(\rho\) values on 20 processors with \(L = 1 ms\). Overhead (in parentheses) is relative to \(\text{fib-ideal}\).

<table>
<thead>
<tr>
<th></th>
<th>(\rho = 1.2)</th>
<th>(\rho = 1.5)</th>
<th>(\rho = 1.75)</th>
<th>(\rho = 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\rho = 1.2)</td>
<td>(\rho = 1.5)</td>
<td>(\rho = 1.75)</td>
<td>(\rho = 2)</td>
</tr>
<tr>
<td>H</td>
<td>0.64 (1.10x)</td>
<td>0.63 (1.08x)</td>
<td>0.63 (1.08x)</td>
<td>0.63 (1.08x)</td>
</tr>
<tr>
<td>(\text{fib-ep}) M</td>
<td>1.26 (2.15x)</td>
<td>1.25 (2.15x)</td>
<td>1.25 (2.15x)</td>
<td>1.25 (2.15x)</td>
</tr>
<tr>
<td>L</td>
<td>1.88 (3.22x)</td>
<td>1.88 (3.22x)</td>
<td>1.88 (3.22x)</td>
<td>1.88 (3.22x)</td>
</tr>
<tr>
<td>H</td>
<td>0.87 (1.12x)</td>
<td>0.98 (1.26x)</td>
<td>1.16 (1.49x)</td>
<td>1.27 (1.63x)</td>
</tr>
<tr>
<td>(\text{fib-ep}) M</td>
<td>1.27 (1.64x)</td>
<td>1.41 (1.82x)</td>
<td>1.38 (1.77x)</td>
<td>1.37 (1.77x)</td>
</tr>
</tbody>
</table>

Table 6.4: Execution times, in seconds, running with Adaptive I-Cilk using various quantum lengths \((L)\) for \(\text{fib-ep}\) \((\rho = 2)\) on 1 and 20 processors, and \(\text{fib-rp}\) \((\rho = 1.2)\) when run on 20 processors. Overhead (in parentheses) is relative to \(\text{fib-ideal}\).

<table>
<thead>
<tr>
<th></th>
<th>(L = 100)us</th>
<th>(L = 500)us</th>
<th>(L = 1)ms</th>
<th>(L = 10)ms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\rho = 1.2)</td>
<td>(\rho = 1.5)</td>
<td>(\rho = 1.75)</td>
<td>(\rho = 2)</td>
</tr>
<tr>
<td>H</td>
<td>13.18 (1.13x)</td>
<td>12.57 (1.08x)</td>
<td>12.53 (1.08x)</td>
<td>12.50 (1.08x)</td>
</tr>
<tr>
<td>(\text{fib-ep}) M</td>
<td>26.37 (2.27x)</td>
<td>25.13 (2.16x)</td>
<td>25.05 (2.16x)</td>
<td>24.97 (2.15x)</td>
</tr>
<tr>
<td>L</td>
<td>39.11 (3.37x)</td>
<td>37.64 (3.24x)</td>
<td>37.52 (3.23x)</td>
<td>37.40 (3.22x)</td>
</tr>
<tr>
<td>H</td>
<td>0.63 (1.08x)</td>
<td>0.63 (1.08x)</td>
<td>0.63 (1.08x)</td>
<td>0.66 (1.12x)</td>
</tr>
<tr>
<td>(\text{fib-ep}) M</td>
<td>1.26 (2.16x)</td>
<td>1.25 (2.15x)</td>
<td>1.25 (2.15x)</td>
<td>1.27 (2.18x)</td>
</tr>
<tr>
<td>L</td>
<td>1.89 (3.23x)</td>
<td>1.88 (3.22x)</td>
<td>1.88 (3.22x)</td>
<td>1.88 (3.24x)</td>
</tr>
<tr>
<td>H</td>
<td>0.87 (1.12x)</td>
<td>0.88 (1.13x)</td>
<td>0.87 (1.12x)</td>
<td>0.89 (1.15x)</td>
</tr>
<tr>
<td>(\text{fib-ep}) M</td>
<td>0.87 (1.13x)</td>
<td>0.87 (1.13x)</td>
<td>0.87 (1.12x)</td>
<td>0.92 (1.19x)</td>
</tr>
<tr>
<td>L</td>
<td>1.28 (1.63x)</td>
<td>1.27 (1.64x)</td>
<td>1.27 (1.64x)</td>
<td>1.28 (1.65x)</td>
</tr>
</tbody>
</table>
overhead is computed by comparing to \textit{fib-ideal}, a single instance of the \texttt{fib} computation running on Cilk-L (as opposed to three contending for cores). Adaptive I-Cilk appears to prioritize tasks appropriately. For \texttt{fib-ep}, Adaptive I-Cilk across the board executes H and M tasks faster with a slight degradation for the L task. For \texttt{fib-rp}, the H task oscillates between desires that either undershoot its parallelism level or overshoot. When it overshoots, it takes cores away from M and L tasks yet doesn't utilize them fully. Cilk-L does not have the same issue as it treats all tasks equally. Consequently, Adaptive I-Cilk experiences higher degradation compared to Cilk-L, but not by too much.

The lower overhead and standard deviation on the M task in Adaptive I-Cilk indicate that the priority relationship between M and L is also being respected. To verify this, we ran the same Adaptive I-Cilk microbenchmarks with just the H and M tasks (not shown) and saw that the execution times for the H and M tasks were similar to that shown in Figure 6.1 and Figure 6.2.

\textbf{Overhead of the Adaptive Scheduling Strategy.} We can look at the overhead of the H task of Adaptive I-Cilk to gauge the overhead for adaptive scheduling. From the results in Figure 6.1 and Figure 6.2, we found the overhead be low. For \texttt{fib-ep} the overhead is also minimal for M and L tasks; they complete in the time it would take to execute 2 and 3 \texttt{fib} computations respectively. This reflects the fact that the medium priority work has to wait for the high priority work to complete, and low similarly has to wait for the medium priority work.

\textbf{Sensitivity to Responsiveness Parameter }\(\rho\). When there is ample parallelism, as in \texttt{fib-ep}, larger \(\rho\) reduce the execution time at every level because it is most efficient to be aggressive about growing the desire; it also benefits lower priorities because the H tasks back off their desire more quickly. When there is little parallelism, as in \texttt{fib-rp}, the correct values are less obvious. For the H task, it is important that \(\rho\) is large enough that the desire can grow quickly; however, it is in some cases even more important that the floor of the high priority desire can reach exactly the number of parallel strands in the H tasks to avoid oscillating between too many and too few processors. In Figure 6.3, this is the case only for a \(\rho\) of 1.2 or 2, and the overhead on the H task of \texttt{fib-rp} reflects this. For M and L tasks, however, their execution times suffer with \(\rho\) of 2 because the H task wastes more processing cycles (low utilization when it overshoots the desire) and deprives the M and L tasks.
Sensitivity to Scheduling Quantum Length $L$. Figure 6.4 shows the microbenchmark execution times of Adaptive I-Cilk compared to fib-ideal with different $L$. As $L$ increases there is less overhead seen on the one core execution. For parallel executions, a longer $L$ means that it takes longer for a priority level to reach the desired number of cores (especially evident in the H task). The M task also suffers slightly, as it takes longer for the H task to give up its cores once it’s done executing; this is especially evident with fib-rp, which has a low parallelism H task can waste cores due to low utilization.

6.4.2 Evaluation of Application Benchmarks

Our application benchmarks are designed to simulate real-world interactive applications. These benchmarks have much richer characteristics compared to the microbenchmarks in that they generate tasks of variable sizes and have different mixtures of I/O operations and computations. The first benchmark, job, most closely resembles traditional task-parallel workloads, but incorporates priorities; it simulates a job scheduler that schedules compute-heavy workloads. The second benchmark, email, simulates an email application that has a non-negligible amount of I/O operations, but also a healthy amount of computation. The last benchmark, proxy, simulates a proxy server that has very little background computation and a high I/O-to-compute ratio. These benchmarks are nondeterministic even when we fix the inputs (e.g., the sequence of requests sent by a client), as inputs arrive with nondeterministic timing. Note that these are the same benchmarks as described in Chapter 5; for completeness, however, we describe them again here.

The Job Server. The job server generates parallel jobs using a Poisson process at random intervals and schedules them. Priorities are assigned based on the smallest-work-first principle, so jobs with the smallest work (i.e., one-core execution time) are assigned the highest priority. Such a scheduling policy is designed to minimize the average flow time of jobs [20], where flow time is the time elapsed between when a job is generated and when it finishes executing. Four types of parallel jobs are used (from highest to lowest priority): a) matrix multiplication (mm, $n = 1024$), b) fibonacci (fib, $n = 36$), c) merge sort (sort, $n = 1.1e7$), and d) Smith-Waterman (sw, $n = 1024$).

We ran job on 20 cores with different $L$ and $\rho$. At 2, 3, 4, and 5 jobs per second, the machine utilization is about 50%, 70%, 95%, and > 95%, respectively. Compared to Cilk-L, Adaptive
Table 6.5: The average (Avg.) and 95 percentile (95%) flow time of different types of computations running on job, listed from highest to lowest priorities. All times are reported in milliseconds. The Jobs/s reports the throughput (how many instances per seconds executed). Times for Adaptive I-Cilk were collected with $\rho = 2$, $L = 500\mu s$, and $\delta = 0.9$. 

I-Cilk prioritizes tasks with higher priority. When the machine is not heavily loaded, Adaptive I-Cilk provides higher throughput than Cilk-L because it implements smallest-work-first using priorities. Adaptive I-Cilk compares favorably over Cilk-L regardless of the choice of $L$ and $\rho$.

As the machine gets more loaded, Adaptive I-Cilk prioritizes tasks at higher priority at the expense of the lowest priority tasks (e.g., sw). This generally translates to higher throughput for higher priority tasks but lower throughput for lower priority ones, and the overall system throughput may be lower than Cilk-L as a result. Here, the choice of $L$ and $\rho$ matters, and we have shown data with parameters that prioritize higher-priority tasks. With a longer $L$ and/or a lower $\rho$, it would take longer for the high-priority tasks to gain processing cores and in turn benefits the lower priority tasks. Thus, a longer $L$ and/or a lower $\rho$ can lead to higher flow time for high-priority tasks but also higher overall system throughput.

The Email Client. The email benchmark simulates a multi-user shared email client. It contains five priority levels (highest to lowest): a) a loop that handles user requests, b) a component to send emails, c) a component to sort emails, d) two equal-priority components: one to compress emails and one to decompress and print emails, e) a loop to periodically check if there are uncompressed emails (due to print or send) that need to be compressed and trigger the compress component.

We ran email on 10 cores, using the other 30 cores to simulate clients connecting to email. Figure 6.6 shows the results with different client configurations with each client sending 1500 requests, except for the 30 client configuration where each client sends 2000 requests to allow for a longer execution time.
Table 6.6: Response times of email tasks, listed from highest to lowest priorities. The \texttt{resp} reports the times elapsed between when a client sends a request to when email reacts to the request by generating a computation. The \texttt{send}, \texttt{sort}, and \texttt{print} report the respective times elapsed between when a client sends the given request to when the corresponding task completes. The \texttt{comp} reports the time elapsed to perform a particular compression task. We report the average times (Avg), the \(95^{th}\) percentile (95\%), and the \(99^{th}\) percentile (99\%) for all categories. The times for \texttt{resp} and \texttt{send} are in microseconds. The times for \texttt{sort} are in microseconds per message. The rest are in milliseconds. Times for Adaptive I-Cilk were collected with \(\rho = 2\), \(L = 500\mu s\), and \(\delta = 0.9\).

<table>
<thead>
<tr>
<th></th>
<th>30 Users</th>
<th></th>
<th>60 Users</th>
<th></th>
<th>90 Users</th>
<th></th>
<th>120 Users</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg 95% 99%</td>
<td>Avg 95% 99%</td>
<td>Avg 95% 99%</td>
<td>Avg 95% 99%</td>
<td>Avg 95% 99%</td>
<td>Avg 95% 99%</td>
<td></td>
</tr>
<tr>
<td>\texttt{resp}</td>
<td>466.00</td>
<td>344.58</td>
<td>7433.00</td>
<td>2248.61</td>
<td>3050.62</td>
<td>18597.10</td>
<td>18080.90</td>
</tr>
<tr>
<td>\texttt{send}</td>
<td>148.18</td>
<td>129.31</td>
<td>19415.30</td>
<td>2287.18</td>
<td>3282.31</td>
<td>18997.10</td>
<td>17248.80</td>
</tr>
<tr>
<td>\texttt{sort}</td>
<td>2.40</td>
<td>3.29</td>
<td>74.33</td>
<td>19.87</td>
<td>18.82</td>
<td>73.07</td>
<td>43.68</td>
</tr>
<tr>
<td>\texttt{print}</td>
<td>3.85</td>
<td>4.38</td>
<td>40.82</td>
<td>6.98</td>
<td>11.61</td>
<td>50.01</td>
<td>16.69</td>
</tr>
</tbody>
</table>

Table 6.7: Response times of proxy tasks listed from highest to lowest priorities. The \texttt{resp} reports the time elapsed between when a client sends a request to when proxy reacts to the request. The \texttt{hit} reports \(\mu s/byte\) (\(\mu s\) is microsecond) for responding with a website already in cache; the \texttt{miss} reports \(\mu s/byte\) for those not in cache. The \texttt{stat} reports \(\mu s/cacheSize\) for collecting cache statistics. Times for Adaptive I-Cilk were collected with \(\rho = 2, L = 500\mu s\), and \(\delta = 0.9\).

<table>
<thead>
<tr>
<th></th>
<th>36 Clients</th>
<th></th>
<th>72 Clients</th>
<th></th>
<th>108 Clients</th>
<th></th>
<th>144 Clients</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg 95% 99%</td>
<td>Avg 95% 99%</td>
<td>Avg 95% 99%</td>
<td>Avg 95% 99%</td>
<td>Avg 95% 99%</td>
<td>Avg 95% 99%</td>
<td></td>
</tr>
<tr>
<td>\texttt{resp}</td>
<td>103.78</td>
<td>74.61</td>
<td>341.17</td>
<td>231.44</td>
<td>79.61</td>
<td>529.18</td>
<td>343.11</td>
</tr>
<tr>
<td>\texttt{hit}</td>
<td>4.94</td>
<td>3.74</td>
<td>14.85</td>
<td>10.26</td>
<td>3.95</td>
<td>22.71</td>
<td>14.92</td>
</tr>
<tr>
<td>\texttt{miss}</td>
<td>2.01</td>
<td>2.30</td>
<td>60.18</td>
<td>2.12</td>
<td>3.04</td>
<td>67.79</td>
<td>1.98</td>
</tr>
<tr>
<td>\texttt{stat}</td>
<td>0.17</td>
<td>0.38</td>
<td>0.61</td>
<td>0.17</td>
<td>0.38</td>
<td>0.46</td>
<td>0.19</td>
</tr>
</tbody>
</table>

129
Here we only show data for one configuration of runtime parameters. Based on our evaluations however, Adaptive I-Cilk uniformly provides shorter response time and send time (the two highest priority tasks) compared to Cilk-L regardless of the choice of $L$ and $\rho$. When the number of clients are moderate (90 or less), Adaptive I-Cilk also provides better sort time. As the number of clients become large (120), Adaptive I-Cilk sacrifices the lowest priority tasks (periodic check for uncompressed messages).

That means Cilk-L may be doing more work (the two lowest priority tasks) than Adaptive I-Cilk, because email running on Adaptive I-Cilk can skip checks from time to time if the loop does not get scheduled within the timer period. As a result, the duration of compress tasks is also higher on Adaptive I-Cilk.

Similar to the observation in job, a longer $L$ and a lower $\rho$ means that it takes longer for the high-priority tasks to gain processing cores which leads to overall longer latency for higher priority tasks (e.g., response and send) but still outperforms that of Cilk-L.

The Proxy Server. The proxy server requests websites on behalf of clients, hiding the requestor’s IP address, and sends the websites back to the client. A concurrent hashtable is used to cache the websites once they are fetched. There are four priority levels (highest to lowest): a) accepting new client connections and handling requests from clients already connected (and immediately responding if the requested site is in already the cache); b) on a cache miss, fetching content from the site and storing it in cache; c) logging statistics about sites requested.

Results from running proxy are shown in Figure 6.7. In general, Adaptive I-Cilk schedules the tasks such that it favors response time of high priority tasks (hit) over lower priority ones. A $\rho$ value of 1.2 seems to work the best, as the high priority tasks in proxy have little parallelism.

Discussion. While it is true that the optimal parameter values will depend on the application characteristics, and the programmer should empirically evaluate a few configurations within the range if she wants the best performance possible, based on our empirical evaluation a few default values can work well based on the observations that follow. First, $\delta$ (the efficiency parameter) does not seem to make much difference as long as it’s on the high-end of the range (e.g., $> 0.75$) because its value is used as a threshold to qualify whether a quantum is efficient. Second, for $L$ (quantum length), generally something like 500 microseconds or
1 millisecond works well regardless of application characteristics because it should be long enough to amortize the cost of adaptive scheduling, but short enough that it can react to changes in parallelism within each priority level. For an application that lacks parallelism, lower $\rho$ (responsiveness parameter) values (e.g., 1.2) seems to work well, whereas for an application that has ample parallelism, higher values (e.g., 2) seem to work well.

### 6.5 Related Work

**Priority Scheduling.** Prior work by Muller et al. [122, 123, 126] provided a type system, the corresponding cost semantics, and the principle that a well-formed computation (from a program that type checked) can be scheduled with their stated execution time bounds as long as the scheduler used is prompt. This prior work does not describe a provably efficient online scheduling algorithm, however. Even though the prior work has an implementation, their implementation is not prompt (a strictly prompt scheduler would not be practical due to synchronization overhead), nor do they analyze the implementation to show any formal approximation of promptness.

In this chapter, we remove the assumption of strict promptness from our theoretical scheduling algorithm. Instead, our theoretical algorithm approximates promptness while being practical to implement and we provide an analysis that explicitly bounds the waste due to this approximation. In other words, our theoretical result provides an online scheduling algorithm with the same asymptotic performance bounds as Muller et al. Our scheduler approximates promptness in the same spirit as how a work-stealing scheduler approximates work conservation.

Beyond work by Muller et al., there is little work in the context of task parallelism. When priorities have been studied in the context of task parallelism, they have been used as heuristics to improve the throughput of computations (e.g., searches where the ordering of branches can improve performance) [87, 157, 158], rather than for responsiveness.

Priorities are also used for scheduling in Operating Systems (see an overview in [138]) and real-time systems (see a survey in [55]). In an OS, the scheduling entities are usually heavy-weight persistent threads (e.g., POSIX threads [89]) and the scheduling quantum lengths are longer. In this chapter, the scheduling entity is a task, which is much lighter weight, and we
use a much shorter scheduling quantum length. These differences necessarily lead to different mechanisms for ensuring responsiveness. Moreover, there are seldom any theoretical bounds on response time within OS threads.

In real-time scheduling, priorities are used in a different manner. Typically in a real-time system, the set of jobs is fixed \textit{a priori}, with known period (how quickly a job generates a task), known deadline (when a task must complete), and a known upper bound on the work of a task. These tasks tend to be independent of each other and do not interact (although they could share resources through the use of locks). The scheduler must provide an \textit{a priori} guarantee of meeting each deadline and priorities are used as a scheduling mechanism to meet these deadlines.

In the realm of real-time and embedded systems, Gadepalli et al. [71] propose Temporal Capabilities, or TCaps, to allow for partitioning of the system and its resources into subsystems. It does so in a manner that real-time and best-effort computations can coexist and coordinate to provide each other services (such as accessing an I/O device) while limiting the extent to which subsystems can interfere with each other. Instead of scheduling tasks, however, TCaps only decides whether or not to schedule pending interrupts, and subsystems must implement their own schedulers for their tasks. Similar to Adaptive I-Cilk, such decisions are based on priorities; in TCaps, however, interrupts are assigned priorities for a subset of the subsystems rather than being assigned a single system-wide priority. Instead of considering a single priority for scheduling interrupts, TCaps first does a set intersection on the sets of subsystems for which priorities are assigned to the interrupt and to the currently running code. Running code is only preempted and the interrupt scheduled if the priority of the interrupt for each subsystem in the intersection is higher than the priority of what it is interrupting. Thus, like in Chapter 5, priority comparisons in effect form a partial order.

**Adaptive Scheduling.** Our scheduling framework is based on the adaptive scheduling strategy A-GREEDY [16], which is designed for a very different context (scheduling independent parallel jobs in a multiprogrammed environment). A-GREEDY focuses on providing an algorithm for the second-level scheduler to best determine the desired number of cores for a job to request assuming an adversarial top-level scheduler. Without considering the top-level scheduler and how it assigns cores, our second-level scheduler essentially utilizes the same algorithm as A-GREEDY to determine the desired number of cores to request. However, the analysis of our algorithm is very different from that of A-GREEDY due to two key differences.
First, we have a top-level scheduler that accounts for priority when making core assignments. Second, in our context, tasks from different priority levels are not independent, whereas A-GREEDY considers independent parallel jobs.

A-GREEDY [16] assumes a second-level greedy scheduler. A similar adaptive scheduling framework for scheduling independent parallel jobs has been developed called A-STEAL [17, 18], where the analysis considers work stealing [23, 24, 39, 40] as the second-level scheduler. A similar adaptive scheduling algorithm using work-stealing has also been used to optimize for power and energy for parallel tasks [15]. In our framework, incorporating work stealing into the analysis will be complex because we would need to bound the steal overheads for different priority levels (a possible direction for future work).
Chapter 7

Practically Efficient Scheduler for Task-Parallel Interactive Applications

In order to support interactive applications in task-parallel platforms, one must provide efficient scheduling support specifically designed for interactive applications. Most prior work [122, 123, 125, 126] on supporting interactive applications, and Chapter 5, focuses on type systems to check for priority inversions where a higher-priority task must wait for a lower-priority task to complete. Assuming the program type checks, the corresponding cost semantics can guarantee bounded response times of tasks assuming a “prompt” and greedy / work-conserving scheduler. Here, we say that the scheduler is prompt if it follows strict prioritization such that no core does low-priority work when high-priority work exists, and it is greedy or work-conserving if no core is left idle when some work is available. Even though some of this work provides a prototype prompt scheduler for small applications written in ML, the focus of this work is not on how to design a low-overhead and scalable scheduler for large-scale task-parallel interactive applications.

In Chapter 6 we proposed an online adaptive scheduler for priority-oriented task-parallel computations written in C++, and approximated this scheduler in Adaptive I-Cilk. We wanted to evaluate how well the Adaptive I-Cilk scheduler works when running real-world application workloads, so we ported the Memcached object caching server [3] to run on Adaptive I-Cilk. As far as we are aware, no one has conducted empirical evaluation on such a large-scale task-parallel code for interactive applications. The Memcached server consists of over 18,000 lines of C/C++ code and the porting effort is non-trivial — it involved converting all the I/Os within the application, removing the event loop in favor of something more streamlined, and removing the concurrency mechanisms of pthreads and rewriting them to utilize task-parallel constructs (i.e., spawn, sync, and futures [79]). Unfortunately, when compared to the original pthreaded implementation the Adaptive I-Cilk scheduler fell short.
Figure 7.1: The 99th percentile latency of handling memcached requests using Pthread, Adaptive I-Cilk, and Prompt I-Cilk versions of the memcached server. For Adaptive I-Cilk, a parameter sweep was performed and the data shown is drawn from the best configuration for each RPS. The dashed lines show the results for each set of parameters swept. Pthread and Prompt I-Cilk do not require parameter sweeps.

As such, the goal of this chapter is to present a practically efficient scheduler for interactive applications that can provide comparable (or better) performance to that of hand-tuned code using low-level threading.

Figure 7.1 shows the performance of pthreaded, Adaptive I-Cilk, and our proposed new scheduler Prompt I-Cilk (discussed later) running the Memcached server for the 99th percentile latencies across different requests per second (RPS) values (higher RPS means heavier server load). As this data shows, Adaptive I-Cilk performs significantly worse than the pthreaded implementation.

In order to understand the shortcomings of Adaptive I-Cilk, let us briefly review its design. At a high-level Adaptive I-Cilk tries to approximate prompt scheduling by preferentially executing high priority tasks, while simultaneously trying to maintain efficiency and low overhead. In order to do so, the scheduler in Adaptive I-Cilk utilizes a two-level scheduling strategy. The top-level processor allocator, given runtime executions, adaptively determines how to best assign cores to priority levels; it re-evaluates the core assignments at every scheduling quantum (i.e., at every $L$ time steps where $L$ is an input parameter to the runtime) based on how well each priority level utilized its assigned cores from the previous quantum.

---

37 The experimental setup is in Section 7.4.
At the bottom-level, a randomized work-stealing scheduler is used to schedule tasks for a given priority level.\footnote{Even though the analysis assumes a greedy bottom-level scheduler for each priority level, the implementation utilizes a randomized work-stealing scheduler.}

Intuitively, the scheduling design in Adaptive I-Cilk makes sense. The re-evaluation of core assignments and moving workers across different priority levels incurs overhead; therefore, the granularity at which Adaptive I-Cilk does this can be adjusted to be infrequent enough to minimize overhead while being frequent enough to approximate promptness. In addition, randomized work-stealing has been shown to be provably efficient \cite{24, 40} and incurs little scheduling overhead in practice \cite{70}. So why does this well-designed scheduler perform poorly?

Upon investigation, we realized that a key contributor to the superior performance of the pthreaded version is its “aging heuristic.” Throughout the executions, a connection can become blocked due to its I/O operation and unblocked when the I/O operation completes. To obtain performance, the pthreaded version uses asynchronous I/O and event-driven style programming to time-multiplex among multiple concurrent client connections. However, its implementation preferentially handles execution contexts that got unblocked earlier over the later ones. In contrast, while Adaptive I-Cilk time-multiplexes among concurrent connections, the randomized work stealing means that the unblocked execution contexts are resumed in a randomized order.

Moreover, the Adaptive I-Cilk scheduler has multiple runtime parameters and the performance is extremely sensitive to these parameters. A parameter sweep is necessary to choose the right values, which can vary widely between applications and even within the same application (Figure 7.1 shows the best possible performance after the parameter sweep, so it actually paints an optimistic picture of Adaptive I-Cilk performance).

Based on our experience evaluating the Memcached server, we decided to rethink the scheduler design from scratch. Our resulting scheduler, Prompt I-Cilk, is surprisingly simple and yet exhibits superior performance with respect to responsiveness for Memcached, as can be seen in Figure 7.1, and for the other application benchmarks tested. Prompt I-Cilk does not use randomized work stealing. Instead, it utilizes something between work stealing and work sharing, and keeps a centralized deque pool to keep track of all execution contexts for a given priority level; this allows it to easily implement the aging heuristic. In addition, in Prompt
I-Cilk, each worker thread checks, somewhat frequently, whether it should switch levels, thus eliminating the need for a processor allocator and parameters.

The design of Prompt I-Cilk moves away from conventional wisdom about task-parallel scheduler design. The first bit of this wisdom was that randomized work-stealing provides better performance than centralized deterministic policies. However, in large-scale interactive applications, there are a large number of parallel contexts; therefore, contention is less of an issue. It is more important to find the “oldest” work, which is harder to do with a randomized policy. The second bit of wisdom was the “work-first” principle — first articulated in [70] — which has long been considered important in scheduler design; it states that we should avoid adding overhead when a worker has local work on its deque. It turns out that in order to minimize latency for high-priority tasks, it is better to abandon the work-first principle in favor of increasing promptness; therefore Prompt I-Cilk frequently checks to ensure that workers are working on the “correct” tasks. In particular, we make the following contributions:

• We ported the Memcached server, a large-scale real-world interactive application, to run on task-parallel platforms and conducted an empirical evaluation with it. Using Memcached as an example application, we argue that task-parallelism can simplify implementations of interactive applications. However, using empirical evaluation, we show that classic randomized work stealing may not work well (Section 7.2).

• Based on our experience with the Memcached server, we propose Prompt I-Cilk, an approximation of the prompt-greedy scheduler proposed in prior work and often thought of as infeasible to implement efficiently; its design involves using a centralized pool with aging heuristics and frequent checking of priority-level assignment (Section 7.3); and

• We conducted a detailed empirical evaluation of Prompt I-Cilk using the Memcached server and two other application benchmarks. Empirical results demonstrate that Prompt I-Cilk, albeit simple, is a highly effective scheduler for these applications (Section 7.4).

7.1 Preliminaries

This section briefly overviews the linguistics of Prompt I-Cilk and Adaptive I-Cilk — both systems are identical in terms of linguistic support and differ only in terms of scheduler design.
The section also briefly describes the scheduler design for Adaptive I-Cilk for completeness. A more in-depth discussion of Adaptive I-Cilk can be found in Chapter 6.

**Language support for interactive applications**

Like prior work on task parallelism, Prompt I-Cilk provides constructs for both fork-join parallelism and futures. The keyword `spawn` can precede a function invocation, indicating that the invoked function may execute in parallel with the continuation of the caller. The keyword `sync` serves as the counterpart for `spawn` and indicates that the control cannot pass beyond `sync` until all previously spawned functions have returned. The keyword `fcreate` is similar to `spawn`, but creates a future and returns a `future handle`, which can be used to wait for the termination of the function invoked and query its result using `ftouch`. These constructs express the logical parallelism; the underlying scheduler is responsible for efficient scheduling of the generated work.

As in Chapters 4 and 6, I/Os in Prompt I-Cilk are expressed using I/O futures, a special type of future. Therefore, Prompt I-Cilk essentially provides a *synchronous interface* for I/O operations which greatly simplifies the application logic. Generally synchronous I/Os are inefficient because workers may spin waiting for an I/O; however, because the Prompt I-Cilk runtime system understands these I/Os, it transparently and automatically time-multiplexes among concurrent subcomputations whenever the I/O operations block providing the *performance advantages of asynchronous I/Os*. In addition, when a task is created (via a `spawn` or `fcreate`), the programmer can specify its priority. The type system then checks for priority inversions — whether any higher priority task might wait for a lower priority task to complete — and the runtime system uses these priorities to preferentially execute higher priority tasks.

**The Adaptive I-Cilk scheduler design**

Although Prompt I-Cilk and Adaptive I-Cilk differ in important aspects of how they schedule tasks within a priority level, they both utilize deques to store execution contexts and operate on deques similarly. At any time, a worker has an *active deque*, and when the worker encounters a `spawn` or `fcreate`, the worker simply pushes the continuation of the spawning
parent onto the bottom of its own deque, and the worker continues to execute the spawned subroutine or future (assuming the subroutine is invoked with the same level of priority). If a worker encounters a failed \textit{sync}, just as in classic work-stealing, the deque must be empty, and the worker tries to find work via work stealing. If a worker encounters a failed \textit{ftouch}, the deque may or may not still contain ready work; regardless, the worker \textit{suspends} the deque and tries to find work via work stealing. Once the \textit{ftouch} that causes the deque to be suspended becomes ready, the deque becomes \textit{resumable} and can be resumed by any worker looking for work. If a suspended deque has ready nodes which can be stolen, it is called a \textit{stealable deque}.

This design in both Prompt I-Cilk and Adaptive I-Cilk is inspired by proactive work-stealing, which as we discussed in Chapter 3 was designed to handle futures (and we later extended to handle I/O futures in Chapter 4) efficiently. Proactive work-stealing is designed to provide efficient execution via reducing the number of “deviations” in the presence of futures; intuitively, reducing the number of deviations acts as a proxy for both reducing the scheduling overhead and improving locality via reducing the number of cache misses. However, unlike classical work-stealing [40] (where the number of deques is the same as the number of workers), proactive work-stealing can generate many more deques than workers.

Adaptive I-Cilk and Prompt I-Cilk differ considerably, however, in most other aspects of their design — in particular, how the workers are moved between different priority levels and how the deques are organized. Adaptive I-Cilk [140] utilizes a two-level scheduling system. At the bottom level, each priority has its own a randomized work-stealing scheduler. Since the number of deques is larger than the number of workers, instead of each worker having a single deque, each worker has its own \textit{deque pool} — a set of deques. Each worker has one \textit{active} deque, the deque it is currently working on; but may also have several suspended and resumable deques in its deque pool. When a worker tries to steal, it first picks a random worker and then picks a random deque in that worker’s deque pool as a victim. The Adaptive I-Cilk scheduler tries to ensure that the probability of stealing from a each deque is about the same by performing rebalancing of deques among workers from time to time. In addition, recall that any steals from suspended deques which are not stealable are completely unproductive; therefore, Adaptive I-Cilk removes these non-stealable suspended deques from workers’ deque pools and reinserts them when they become resumable.

\footnote{If the spawned subroutine belongs to a different priority level, a deque is generated to store the subroutine and tossed to the appropriate priority level.}
At the top-level, Adaptive I-Cilk has a centralized scheduler that moves workers between different priorities. At any particular time, each bottom-level scheduler is assigned a certain number of workers by the top-level scheduler and this allocation can change at quantum boundaries. At the end of each quantum, the system measures the utilization — percentage of the processing used for application work — of each bottom level scheduler and the top-level scheduler decides the allocation for each bottom-level scheduler for the next quantum based on this quantity. The intuition behind this design in Adaptive I-Cilk is as follows: If a lot of work is available at a particular priority level (as measured by utilization), more workers will be allocated to that level (with preference given to higher priorities). It simultaneously tries to minimize the overhead of moving workers between priority levels by doing so only infrequently.

7.2 Motivating Example: the Memcached Server

This section reports on our experience porting and running the Memcached object server [3], a widely used interactive application, on Cilk-based task-parallel platforms. We use the Memcached server as an example application to explain how task parallelism can simplify the implementation of interactive applications. We also use it to explain how the typical workload characteristics of interactive applications may differ from traditional parallel applications, thereby making randomized work-stealing non-ideal for these applications.

The Memcached server overview

The Memcached server provides an in-memory key-value store for small objects; it is widely used as a caching backend for many performance-critical systems and is designed to be scalable. At a high-level, it maintains a hash table; within each bucket, the objects are organized in (approximately) least-recently-used (LRU) order such that recently accessed objects are likely to be faster to access. One can also configure Memcached to support hash table expansion and writing of the cached content to an external persistent storage.

The original implementation is written in C with POSIX threads [89] and organized as follows. Upon startup, the main thread creates several background threads and a fixed number of
worker threads to handle client connections. The background threads perform tasks such as reorganizing the items in a bucket to maintain the LRU policy in the cache, assisting hash table expansion, crawling through the cache to collect statistics, and writing in-memory cache content to the external storage (if configured to do so); these are designed to run periodically. The main thread listens for requests from new clients; once a client is connected, a specific worker thread is assigned to handle its requests.

The Memcached server uses event-driven style programming with asynchronous I/O operations via the libevent library [2]. A worker thread time-multiplexes among multiple client connections at any given time via an event loop to handle events from multiple concurrent client connections. Each client, upon connecting to the server, is assigned to a particular worker, and a callback function is registered with the libevent library for events associated with that particular client connection. A worker thread never waits for a blocking I/O operation when handling a request. Rather, upon encountering a blocking I/O operation for a request \( r \), the worker thread returns from the callback function to handle requests from other connections, and only comes back to resume handling request \( r \) when the I/O can complete, which generates an event and causes the same callback function to be invoked again.

Due to asynchronous I/Os and event-driven style programming, the control flow of the application is extremely complex. A given request \( r \) might require multiple I/O operations before completion (with some computation interleaved between them). Thus, when the worker thread handling \( r \) encounters a blocking I/O and leaves to handle other requests, it must leave enough bookkeeping information so that it knows which I/O operation it blocked on when it returns to this callback function. Therefore, the callback function effectively encodes a large state machine using a switch-statement in a loop, where a read can transition to multiple different states depending on what the input is and what state the request was in previously. Such control flow obscures the application logic, as the logic for handling a single request is scattered across different switch statement cases.

As mentioned in the beginning of this chapter, the pthreaded version implements an aging heuristic that handles execution contexts in the order in which they become unblocked. This aging heuristic is implemented implicitly due to how libevent and the underlying OS handles asynchronous I/O operations. As the I/O operations become ready, the OS detects the available I/O events and returns them to libevent in the same order. Thus, as worker threads process the I/O events from libevent, they effectively process them in
the same order as they became ready, at least among clients handled by the same worker thread. There are exceptions to this aging heuristic: if a particular client connection has multiple outstanding requests all of which can be processed without blocking, then the worker thread will process them together upon entering the callback function corresponding to that connection, up to some threshold before the worker thread voluntarily “yields” and returns from that callback so as to not starve other connections.

Simplifications using task parallelism

We rewrote the Memcached server using task-parallel control constructs and found that the resulting control flow is much simpler and exposes the underlying program logic. Specifically, we removed the event loop and we rewrote the giant callback function that encodes the state machine to handle client requests with I/O futures, which provide a synchronous I/O interface with the performance of asynchronous I/O. This rewrite turns the function that handles client requests into something much more streamlined; the I/O futures allow the programmer to express the client request handling logic as sequences of synchronous I/O and, based on the I/O result, some amount of computation. Furthermore, we no longer need to explicitly assign client connections to a given worker thread. Instead, each client connection translates into a future routine that handles the requests for that connection.

This simplification is possible because the runtime scheduler understands the semantics of I/O futures. It transparently time-multiplexes among multiple concurrent connections, by switching away from one execution context when a client request is blocked on an I/O operation, and resuming it when the I/O completes.

Shortcomings of Adaptive I-Cilk

Task parallelism makes it easier to write interactive applications, but puts the burden on the scheduler to provide good performance by time-multiplexing between requests and providing low latency to high priority requests. As mentioned in Section 7.1, the Adaptive I-Cilk scheduler tries to satisfy these goals by providing good scheduling mechanisms for I/Os, futures, and by adding support for priorities. However, certain characteristics of interactive applications make this design inefficient.
Most importantly, the parallelism (one might call it concurrency in this context) in the Memcached server is purely due to the handling of multiple concurrent client connections. There isn’t much parallelism otherwise — requests for a given connection are handled sequentially and there are not any parallel operations needed within a single request. Therefore, the scheduler must juggle many execution contexts (futures) concurrently in order to get efficiency. In addition, because all these connections are active at once, the aging heuristic is absolutely essential.

Figure 7.2 shows the average number of non-empty deques across quantum when running the Memcached server with different server loads on Adaptive I-Cilk. As the numbers show, even for a lower server load, the number of non-empty deques at a given time is on the order of hundreds.

Recall that Adaptive I-Cilk is a randomized scheduler which tries to ensure that all deques are stolen from with about equal probability. The randomization in a classic work stealing scheduler serves two purposes. First, it avoids contention. When every thief randomly chooses a victim to steal from, one can show that probability of stealing from the same victim is low. Second, it allows the scheduler to amortize failed steal attempts against the span of the computation, the length of a longest dependence chain in the computation. This scheduler design makes sense for classic task-parallel applications, where parallelism in the applications is mostly expressed in the form of fork-join parallelism and the number of deques is approximately the same as the number of workers.

In the context of interactive applications, where there a lot of futures and I/O operations, and therefore deques, this design may not be ideal and can incur higher scheduling overhead than necessary. First, because the number of non-empty deques is abundant, contention is less of an issue. Second, there is little need to amortize the steal cost assuming a thief looking for work can locate a non-empty deque quickly without spending much time on failed steals, which is much easier to accomplish when the number of deques is large. For these two reasons, the randomization, at least for the purpose of practical considerations, no longer seems as important.

---

40 In principle, when the hash table gets resized, one can parallelize the moving of buckets from the old hash table to the new resized hash table, but we didn’t do that (nor did the pthreaded implementation) as a) the resizing occurs concurrently while the requests are being handled and does not seem to be a bottleneck and b) during our performance evaluation resizing doesn’t get triggered as the initial capacity is configured to be large enough for the workload.
Yet, the design of Adaptive I-Cilk incurs scheduling overhead in favor of ensuring randomization due to (1) rebalancing deques across processors; and (2) high constant overhead when a deque becomes empty or resumable because the deque pool of each processor is protected by a lock. In particular, the second point is subtle — in order to provide randomization, the deque pool must support random access and removal from arbitrary locations, which is difficult to implement without a lock. However, due to locking, accessing the deque pool can become expensive because a deque in its life time can repeatedly transition between being suspended / empty and resumable / non-empty (due to I/O operations), and each deque can thus be repeatedly removed from and re-inserted into different deque pools. In addition to the scheduling overhead, the randomization makes it quite challenging to implement the aging heuristic, which is absolutely crucial to get low latency.

When the number of deques in the system is large, which is typical for interactive applications that utilize frequent I/O operations, we argue that it is more important to provide a lightweight data structure to manage the many deques with fast insertion and removal than to ensure that the work stealing is randomized. As we will see in Section 7.3, our Adaptive I-Cilk scheduler design moves away from randomized work stealing while keeping some other aspects of work stealing from Chapter 3, and our design favors access efficiency for managing the many deques. It also makes it easier to implement the aging heuristic. In Section 7.4, we show that the empirical data implies that the Adaptive I-Cilk design incurs lower scheduling overhead than Adaptive I-Cilk.

### 7.3 The Prompt I-Cilk Scheduler

This section describes the construction of the Prompt I-Cilk scheduler. At any time, each worker is working on a particular priority level. We use a bitfield to keep track of which priority levels have available work and each worker, asynchronously and somewhat frequently, checks whether it should switch to a different priority level. Within a priority level, we use a simple queue based scheduler which can be viewed as a hybrid of work sharing and work stealing, with the goal of optimizing for efficient handling of many deques.

---

41We are not aware of any lock-free concurrent data structures that satisfy these requirements. For instance, queue data structures (e.g., [116, 162]) do not allow for random accesses, nor does it allow one to remove items from arbitrary locations. On the other hand, a concurrent vector may support random access, but it cannot support removal from arbitrary locations in constant time (e.g., [57, 63]).
Figure 7.2: The average number of high priority deques at quantum boundaries in memcached using Adaptive I-Cilk. Counts are drawn from the best performing configurations, as in Figure 7.1.

**Work stealing without randomization**

Within a priority level, our strategy can be viewed as a hybrid of work stealing and work sharing. The Prompt I-Cilk scheduler, like Adaptive I-Cilk, retains certain properties of proactive work stealing [141] (summarized in Section 7.1). Specifically, each worker operates on its own active deque (at the priority level that the worker is in); when the worker runs out of work or encounters a `ftouch` that requires suspending its deque, it tries to steal work from other deques at the same priority level. Due to deque suspensions, there can be many deques at each priority level at any given time. Prompt I-Cilk mainly differs from Adaptive I-Cilk in how workers perform steals and how it manages the many deques in the system.

Prompt I-Cilk uses a relatively simple strategy for finding work. Instead of each processor having its own deque pool and performing randomized work-stealing, Prompt I-Cilk maintains a single deque pool for each priority level — this deque pool is implemented using an efficient concurrent non-blocking FIFO queue. The queue utilizes fetch-and-add to implement fast insert (at the tail) and removal (from the head). It is organized as an array of arrays to allow for concurrent accesses while resizing. It uses the standard epoch-based reclamation technique [67] to ensure that no workers are still referencing the old arrays before recycling them.

Whenever a worker (thief) wants to steal, it pops the deque off the head of the queue. If the deque is suspended and stealable, the thief steals from the top of that deque. If the deque is
resumable, the thief simply **mugs** the whole deque and resumes the bottom-most frame. If the deque, after being stolen from / resumed, still contains stealable work, the deque gets pushed back onto the tail of the queue. Similarly, if a deque becomes resumable and if it is not already in the queue, the system pushes it to the tail of the queue.

Another key distinction between Prompt I-Cilk and Adaptive I-Cilk is that, while Adaptive I-Cilk ensures that no non-stealable (e.g., suspended and empty) deques are in any deque pool, Prompt I-Cilk does not maintain this strict condition. If a thief discovers an empty deque at the head of the queue, it simply doesn’t push it back and removes the next deque from the queue.

Prompt I-Cilk maintains the invariant that all non-empty deques (active, suspended or resumable) are in the deque pool; however, some empty deques may also be in the deque pool. An empty deque is never added to the queue, but a deque may become empty while in the queue and remain there for some time. For instance, worker $p$’s active deque may be non-empty and therefore in the queue, and $p$ may pop off the bottom frame, making it empty. This deque remains in the queue until it reaches the head of the queue, some thief pops it, discovers that it is empty and refrains from adding it back. While this mechanism means that sometimes workers have to do multiple queue accesses in order to find work, it makes the queue design simple and fast since we needn’t support removal or insert except at the head or tail respectively.

It is crucial, however, that all non-empty deques are in the queue since any available work on a deque must be discoverable by other workers. One possible design choice might be to keep all the deques in the queue — however, this would make the queue cluttered with many empty deques making it difficult for thieves to find work. Prompt I-Cilk strikes a balance — when a thief discovers an empty deque, it does not add it back to the queue. This, however, means that when an empty deque becomes non-empty, it might need to be added back. Therefore, whenever the system resumes a deque, it checks to see if this deque is already on the queue and pushes it back if it is not. Similarly, when a worker pushes something onto its active deque (via spawn or fcreate), it checks and pushes its active deque back onto the queue if necessary.

This checking of whether a deque needs to be pushed back onto the queue goes against the work-first principle [70]; however, in our experience, we find that this choice strikes the right balance. The added overhead is small enough that this choice is superior to both
alternatives, namely (1) keeping all deques in the queue (making it difficult to find work); and (2) maintaining the strict invariant (as Adaptive I-Cilk does) that no empty deque is in the queue (requiring complex data structures that support removal of deques from arbitrary locations in the queue).

Support for aging

Recall that the aging heuristic — handling requests in roughly old to new order — is crucial for performance in interactive applications. Intuitively, a FIFO queue naturally emulates the aging heuristic — when deques become resumable, they are inserted into the queue in FIFO order. However, it turns out that there are some subtle exceptions. Therefore, in practice, we actually utilize two queues to maintain the deque pool in order to implement aging properly.

One exception where the FIFO queue does not implement aging strictly is when a deque $d$ is already in the queue, $d$ becomes both suspended and empty, but $d$ has yet to be removed from the queue (because it has not yet reached the head). Now suppose $d$ becomes resumable again. In this case, $d$ might be handled before deques that became resumable before it, but were added to the queue later. It turns out that this doesn’t happen enough to create issues in practice as deques in interactive applications tend to be shallow (i.e., upon suspension it is either empty or becomes empty after a few steals), and workers go through deques quickly due to frequent I/O operations. Thus, a few deques can get “lucky” and cut in line in this fashion.

The other exception is due to abandoned deques and is more problematic. This happens when a worker $p$ is working on a lower priority, say $k$; some higher priority work becomes available; $p$ abandons its active deque at priority $k$ and starts work at a higher priority.\footnote{This is an issue only at lower priorities since a worker in the highest priority level will never abandon its active deque to go work on a different level.} We call such a deque an immediately resumable deque — it was suspended not because it was blocked on a $\mathit{foo}$; it was suspended because the worker has to abandon it to go work on a higher priority. At the point of suspension, this immediately resumable deque must be inserted into the queue, if it is not in the queue already. By inserting it at the tail, however, we are effectively “de-aging” the deque, putting it behind any deques that became resumable before the insertion.
To mitigate this issue, we use two queues to implement the centralized pool — a mugging queue that contains only immediately resumable deques that were abandoned, and a regular queue that operates as described above. If a worker needs to abandon its active deque, it always inserts the deque into the mugging queue. A thief trying to find work will always check the mugging queue first and mug the deque found in there before trying to steal or mug deques from the regular queue. Therefore, an immediately resumable deque may be in both queues until it is mugged by a thief, but this is not a problem; the regular queue operates as usual.

Supporting promptness and elasticization

The Prompt I-Cilk scheduler tries to approximate prompt scheduling — workers want to work on high priority work if any is available. In order to do so, the Prompt I-Cilk scheduler uses a bitfield to indicate which priority levels currently have work, where index $i$ in the bitfield is set to 1 if priority level $i$ has work. In practice, we use a 64-bit integer to represent the bitfield since it allows us to use the efficient atomic logical operations on x64 to manage the bitfield. In particular, we use fetch-and-or and fetch-and-and to update the bitfields; C++ atomic load with sequential consistent memory ordering to read the bitfield; and the compiler builtin __builtin_clzll (e.g., see [8]) to quickly retrieve the bit index for the highest bit set. Although this limits the max number of priority levels to 64 in our implementation, this is more than enough in the applications we examined.

Because of the centralized deque pools, it is easy in Prompt I-Cilk to check whether a given priority level contains work. A worker, when enqueueing a deque into a pool, always sets the corresponding bit. A thief, when encountering an empty pool (which requires checking two queues, as described above), clears the corresponding bit. Since checking the size of the pool and clearing the bit are not atomic, the thief performs double checking — if the pool is empty, it clears the bit, checks the pool again, and resets the bit if the pool is no longer empty, ensuring that the bit should not be left unset for an extensive period if a thief clearing the bit interleaves with an active worker generating new work.

Every worker checks this bitfield frequently — a thief checks this bitfield before attempting a steal; an active worker checks this bitfield at every spawn, sync, fcreate, and ftouch. If a worker realizes that it is working at a lower priority level than the highest level with
available work, it abandons its active deque in the current level and moves itself to the higher level (the abandoned work remains in the lower priority deque pool and will eventually be resumed). The frequent checking goes against the work-first principle [70]. However, somewhat surprisingly, we found that implementing the promptness with the frequent checking works well — in Section 7.4 we show that the frequent checking allows the scheduler to react to work quickly and does not incur undue scheduling overhead.

At times, thieves may find that there are no bits set in the bitfield. In this case, the thief is (perhaps temporarily) not currently needed by the application. Workers in this situation will, instead of stealing, wait on a global condition variable while the bitfield contains only 0 bits. As soon as an active worker sets the bitfield from zero to non-zero, that worker will broadcast the condition variable to wake up all sleeping workers. These workers then re-enter the scheduler loop and resume working on the highest priority work.

7.4 Empirical Evaluation of Prompt I-Cilk

This section empirically evaluates Prompt I-Cilk using the Memcached server [3] discussed in Section 7.2, an email server application, and a job server. We empirically evaluate Prompt I-Cilk by comparing it to Adaptive I-Cilk, the-state-of-the-art priority-oriented task-parallel platform discussed in Chapter 6. We additionally compare Prompt I-Cilk to two other variants of Adaptive I-Cilk, to evaluate how much the performance of Prompt I-Cilk may be attributed to each of its key differences from Adaptive I-Cilk. Empirical results indicate that generally Prompt I-Cilk outperforms Adaptive I-Cilk and other variants, and incurs lower waste.

Experimental setup and evaluation criteria

Hardware configuration. We ran our experiments on a computer with 40 2.40-GHz cores split across 2 Intel Xeon Gold 6148 processors, and 768-GB of main memory. Each processor has a shared (among 20 cores) 27.5-GB L3 cache. Hyperthreading was enabled such that each core had 2 hardware threads. Each core has a 32-KB L1 data cache, a 32-KB L1 instruction
Figure 7.3: The latencies of Memcached implemented with different schedulers. For Adaptive I-Cilk, Adaptive I-Cilk plus aging, and Adaptive Greedy the data points are drawn from the runtime parameter configuration with the best 99th percentile latency for the given RPS. Adaptive I-Cilk uses 5 different sets of parameters, while Adaptive I-Cilk plus aging and Adaptive Greedy both use 4 different sets of parameters.

Figure 7.4: The 95th and 99th percentile latencies of the job server benchmark normalized to the latencies of Prompt I-Cilk, with Prompt I-Cilk also shown for reference. Tasks are shown in order of highest to lowest priority. Adaptive I-Cilk uses 3 different sets of parameters, and its variants use 2 different sets of parameters.
Figure 7.5: The top row shows the the 95th and 99th percentile latencies of the email benchmark normalized to the latencies of Prompt I-Cilk, with Prompt I-Cilk also shown for reference. The bottom row shows the average and median latencies similarly. Tasks are shown in order of highest to lowest priority. Print and comp are at the same priority level. The Adaptive I-Cilk variants use 3 different sets of parameters.
cache, and a 1-MB L2 cache. Experiments ran in Linux kernel 5.15, and all benchmarks were compiled with -flto and -O3 using the clang 5.0.0-based Tapir compiler [137].

The Memcached server setup. We have already covered the characteristics of the Memcached server [3] extensively in Section 7.2. One of the most important considerations for a Memcached server is how many requests per second it can process while meeting guarantees on the tail latency, which we will refer to as the Quality of Service (QoS). For Memcached, we adopt the QoS criterion used by Palit et al. [131]: 95% of all client requests should be handled within 10 ms. We use the Memcached driver from [131] to find the maximum number of requests per second (RPS) that the server can handle using the pthreaded version, and then run experiments on nearby RPS values. As in [131], we find the maximum RPS that meets the QoS using a binary search on the simulated RPS using a fixed number of clients (600).

We ran Memcached on 4 cores and used 20 cores for client connections. For the pthreaded version, this configuration means the server creates 4 worker threads, plus the main thread and a few background threads. Even though the pthreaded version oversubscribes, the background threads rarely ran. For all other platforms, this configuration means that the platform creates 4 worker threads plus 4 I/O handling threads (which is based on the design of the prior work on handling I/O futures [139] discussed in Chapter 4). We chose to run the Memcached server on this configuration because the Memcached documentation [4] recommends one run the server with only 4 worker threads, as more worker threads would likely create a synchronization bottleneck on the shared cache.

The email and job servers setup. The email server and job server benchmarks were used to evaluate Adaptive I-Cilk [140], and they are specifically designed to test out priority-oriented task-parallel platforms, where each application consists of multiple tasks with multiple priority levels. These are the same benchmarks as described in Chapters 5 and 6, but slightly modified in order to ensure that the amount of the work done in each run is the same. Typically, if in an interactive application the input arrives with nondeterministic timing, the workload generated on the server side may differ depending on the internal server state when the input arrives. To enable more consistent timing and fair comparison between different platforms, we made changes to the client request generations and how the servers worked to ensure consistent workload generation. For completeness, here we again describe these benchmarks.
The email server benchmark simulates a multi-user email server and supports multiple operations, from highest to lowest priorities — a) send emails (send), b) sort emails (sort), and c) two operations with equal priority: compress emails (compress); and print emails (print), which requires decompressing the emails. We ran the email server on 4 cores and used 20 cores to simulate client connections.

The job server simulates a server that performs shortest-job-first scheduling, so shorter jobs get higher priorities, from highest to lowest: a) matrix multiplication (mm), b) Fibonacci (fib), c) sort (sort), and d) Smith-Waterman (sw). Because all these jobs are parallel tasks, we ran the job server on 20 cores.

**Variants of Adaptive I-Cilk.** We additionally implemented two variants of Adaptive I-Cilk to enable us to gauge how much of the performance of Prompt I-Cilk may be contributed by its key differences from Adaptive I-Cilk. The first one is Adaptive I-Cilk plus aging, which adds a per-worker aging heuristic to Adaptive I-Cilk; that is, each worker additionally maintains a queue of resumable deques in resumption order, such that each worker (or a thief stealing from this worker) can work on these deques according to the aging order. This allows for a per-worker approximation of the aging heuristic. The second one is Adaptive Greedy, which still uses the two-level scheduling system as in Adaptive I-Cilk but replaces the bottom-level scheduler with the work-stealing scheduler in Prompt I-Cilk, i.e., it maintains a centralized deque pool and steals without randomization, and therefore approximates aging better than Adaptive I-Cilk plus aging.

Like Adaptive I-Cilk, both variants still require setting runtime parameters. For all of these Adaptive variants of schedulers, we always perform the runtime parameter sweep for each benchmark and present numbers with the runtime parameters that have the best tail latency. This means the timing data shown for each RPS for a given benchmark may not use the same runtime parameters.

**Latency comparison**

Figure 7.3 shows the 95th and 99th percentile latencies of handling requests in the Memcached server for the implementations using pthreads, Prompt I-Cilk, and variants of Adaptive I-Cilk.

43 For the email and job servers, we used an average of 95th and 99th percentile latencies; for Memcached, we used the 99th percentile latency.
We only show the 95th and 99th percentile latencies because the plots for the average and 50th percentile latencies look very similar to the 95th percentile plot. Prompt I-Cilk, Adaptive I-Cilk plus aging, and Adaptive Greedy have similar performance to the pthreaded version, and outperform the pthreaded version at higher RPS, especially for the 99th percentile latency. Adaptive I-Cilk, on the other hand, has much higher latencies, creating an obvious gap between Adaptive I-Cilk and Prompt I-Cilk. This result indicates that the aging heuristic is really the crucial difference here. Interestingly, Adaptive Greedy slightly outperforms Prompt I-Cilk at the highest RPS, indicating that the promptness incurs slightly more overhead for Memcached at high RPS.

Figure 7.4 shows the 95th and 99th percentile latencies of the job server implemented in Prompt I-Cilk and the Adaptive variants of I-Cilk, with the latencies normalized to the latencies of Prompt I-Cilk. The server workload is configured to be low at 3 RPS, decent at 4 RPS, and high at 5 RPS. We only show the 95th and 99th percentile latencies, but we have also examined the average and 50th percentile latencies, and Prompt I-Cilk across the board outperforms other variants. The job server, unlike the Memcached server, contains more parallelism. Each job instance created by the server is a traditional task-parallel job. Even in this setting the scheduling mechanisms in Prompt I-Cilk seem to work well, namely aging and promptness.

First, let’s examine the aging mechanism. The aging mechanism really kicks in when the number of deques within a priority level gets large. In the job server, unlike in Memcached, each request generates a parallel task and thus can fully utilize more cores. Consequently, the higher priority tasks (i.e., \texttt{mm} and \texttt{fib}) will take up more cores, leaving the lower-priority-level tasks unattended to for longer periods. We observed that the average number of deques (across quanta) is high for lower priority tasks (i.e., \texttt{sort} and \texttt{sw}), especially at 4 and 5 RPS (on the order of hundreds per level). The aging mechanism really matters for providing better latencies in this setting, as demonstrated by the superior performance of Adaptive Greedy over other Adaptive variants for \texttt{sort} and especially \texttt{sw} at 4 and 5 RPS. We don’t see this phenomenon at 3 RPS because the number of deques is low across priority levels.

Second, the promptness in Prompt I-Cilk really helps. Generally, we see that the latency improvement of Prompt I-Cilk over other variants is the most pronounced for high server loads and for the higher priority levels, which makes intuitive sense for this workload. Because the job server contains many parallel jobs, the promptness of Prompt I-Cilk will cause it to
work on the highest priority work available, whereas in other Adaptive variants the highest priority level will take time to both ramp up its worker counts when the tasks come online and then take time to ramp back down when the tasks complete. This slower ramp up allows the lower priority level jobs to get some work done in the mean time, but the slower ramp down also means more wasted cycles, which we will examine in more detail later in this section.

The top row of Figure 7.5 shows the 95th and the 99th percentile latencies of the email server implemented with Prompt I-Cilk and the Adaptive variants of I-Cilk, with the latencies normalized to the latencies of Prompt I-Cilk. For this benchmark, we additionally show the the average and median latencies in the bottom row because, unlike other benchmarks, the shape of the average and median latencies do not resemble that of the higher percentiles.

The email server is an interesting application to study, as the data does not tell a story as clean as the job server. Let’s examine the aging heuristics first. In the email server, we observed that the lower priority levels (i.e., print and compress) contain large numbers of deques at only the highest load, 18K RPS. Thus, we see that the aging heuristics really help in that setting. At the 6K RPS, because the number of deques is low, the aging heuristics do not do much, and the different variants perform comparably. However, at 12K RPS, we see Adaptive Greedy perform better for high priority tasks but worse at lower priority tasks. It turns out that, at 12K RPS in particular, the data points we chose to show for Adaptive Greedy used a runtime parameter that ramps up and down the core counts more aggressively compared to other variants, leading to better latencies at higher priority tasks but also higher waste, which resulted in lower priority tasks getting delayed. At 6K RPS we don’t see this pattern because the best data points across variants happen to use the same runtime parameters.

At the 95th and 99th percentile, the promptness clearly helps as Prompt I-Cilk outperforms the other variants. For the 50th percentile, the Adaptive variants outperform Prompt I-Cilk at 6K RPS as well as for the lowest priority task, comp, at 12K RPS. Prompt I-Cilk still provided better or comparable average latencies for these data points, however, which indicates that Prompt I-Cilk provides more stable running times (i.e., lower variance).
Waste and scheduling overhead

To evaluate how well Prompt I-Cilk manages waste and its scheduling overhead, we additionally collected the waste times and running times for both Adaptive I-Cilk and Prompt I-Cilk. Here, the waste corresponds to time workers spent looking for and failing to find work. For Prompt I-Cilk, it also includes time a worker spent putting itself to sleep and waking up when the bitfield goes from non-zero to zero and vice versa. The running time corresponds to time workers spent doing useful work or scheduling overhead, such as successful steals, mugging, and in the case of Prompt I-Cilk, time spent checking the bitfield or checking if it needs to push its deque onto the queue while active.

Across benchmarks, Prompt I-Cilk incurs slightly higher running times, but makes up for it by reducing waste.

For the Memcached server and the job server, where the concurrency and parallelism are abundant, Prompt I-Cilk does not particularly incur higher scheduling overhead compared to Adaptive I-Cilk, which suggests that the extra scheduling overhead incurred in Prompt I-Cilk while a worker is active are not overly cumbersome. Moreover, compared to Adaptive I-Cilk, for the job server it seems to be managing waste a lot better.

The email server benchmark, on the other hand, creates sequential tasks and tasks with low parallelism in bursts. Such a workload causes Adaptive I-Cilk to incur slightly higher waste compared to the job server, and slightly higher running time compared to both of the other benchmarks. The savings on waste still outweighs the higher running times compared to Adaptive I-Cilk however. Interestingly, Prompt I-Cilk incurs its highest waste at 6K RPS for

Table 7.1: Wasted and running time, in seconds, accumulated across all processors. Running time is the processing time incurred by scheduling overhead and user code. The values in parentheses for Adaptive I-Cilk are the ratio of Adaptive I-Cilk to Prompt I-Cilk.
email. As the server load gets higher (in this case, higher concurrent requests), it is able to manage waste better than Adaptive I-Cilk.

7.5 Related Work

In the beginning of this chapter and in Section 7.1, we have discussed the most closely related work in detail. In addition, there is extensive work, going back many decades, on work-stealing for classical throughput oriented parallel applications; here we will focus primarily on some related work for interactive applications.

As mentioned in the beginning of this chapter, task parallelism for interactive applications is a relatively recent research topic, but has been considered from the perspective of type systems and cost models [122, 123, 125, 126]. Most of this work focuses on designing type systems that can detect priority inversions in order to ensure that, if scheduled using a prompt scheduler, the application gets appropriate performance guarantees. This work is synergistic and orthogonal with our work because Prompt I-Cilk is meant to be a practically efficient approximation of a prompt scheduler. Some of this work [122, 123, 126] implements a work-stealing scheduler with private deques in an extension of parallel ML [144, 145]; however, this scheduler is mainly designed to be proof of concept for expressiveness of the type system and not really evaluated for performance on larger applications. In Chapter 6, we proved theoretical performance guarantees which do not apply directly to Adaptive I-Cilk; however, we argue that a design similar to Adaptive I-Cilk, but with a greedy scheduler at the bottom-level is almost prompt.

Interactive applications are conventionally written with asynchronous I/O with event-driven programming. Runtime mechanisms have been proposed to allow for a synchronous I/O interface while providing the performance of asynchronous I/O [21, 81]; this work is concerned only with I/O performance and not load balancing. Researchers [73, 165] have also tried to incorporate work stealing into event-driven applications to allow for parallel execution but they are not concerned with the priorities of tasks.

Much work on interactive applications is in the context of web services, focusing on resource management to ensure quality of service (QoS) guarantees on responsiveness [56, 80, 96]. In web searches, a single search can generate multiple queries, where the response time is
dictated by the tail latency contributed by a few long queries. In this context, each query is
treated as a parallelizable job, and multiple scheduling mechanisms have been proposed to
selectively or incrementally parallelize certain queries to reduce the tail latency [80, 97, 100].
Often, interactive web services run in a multiprogrammed environment, where batch jobs
are co-located with latency-sensitive web services to increase CPU utilization. Various
scheduling techniques have been proposed for such systems to meet QoS while keeping
the CPU utilization high [29, 58, 59, 62, 68, 113, 115, 127, 128, 130, 132, 163]). This line
of work is largely orthogonal and complementary to ours — one can imagine using these
systems to manage resources in a multiprogrammed environment with some latency-sensitive
task-parallel applications written to run on Prompt I-Cilk.
Chapter 8

Conclusion

Because of a significant decrease in the rate at which the performance of individual processor cores improve, and the switch to instead adding more cores to a single processor chip, it is becoming more and more important for programmers to write parallel code. Low-level threading, however, can be difficult to implement effectively. While this has led to the development of task-parallel platforms to ease the burden on the developer, work on task-parallel platforms has largely been focused on throughput-oriented scientific computations and the results are ill-suited for interactive parallel applications.

This dissertation has shown that task-parallel platforms can be extended to better support modern interactive applications, and has empirically demonstrated that the proposed solutions lead to a task-parallel platform that is competitive with hand-tuned interactive applications that use traditional means of parallelization. We explored more practical support for futures in a task-parallel platform, efficient I/O support, how to represent interactive imperative parallel applications through a formal type system, and how to schedule interactive parallel applications.

Chapter 3 explored ProWS, adding futures to task-parallel platforms with a near-asymptotically-optimal execution time bound, but with a better bound on scheduling overhead than the classic work-stealing algorithm. This allows us to efficiently support complex parallel dependences that cannot be expressed with spawn/sync. We implemented ProWS in the Cilk-F runtime.

We built upon ProWS and Cilk-F to provide efficient I/O support using the futures abstraction in Chapter 4, providing the support we need for I/O operations in interactive parallel applications. Our extension to Cilk-F is implemented in the Cilk-L runtime.
In Chapter 5 we described a type system that allows for prioritization of future tasks, while statically preventing priority inversions. We implemented this type system in C++, and interfaced it with I-Cilk, a runtime we concurrently developed to support interactive parallel applications. Prioritizing tasks while preventing priority inversions is key to providing execution time bounds on not just entire programs, but to the individual tasks in the program.

We dive into the details of APS and I-Cilk, and provide execution time bounds for tasks at different priority levels when executed using APS in Chapter 6. This allows us to efficiently schedule interactive parallel applications, though with room for improvement both in performance and reducing the number of parameters that need to be tuned.

Finally, in Chapter 7 we proposed and implemented the Prompt Greedy scheduler, a scheduler that simultaneously improves practical performance over APS and removes the pesky parameters of the APS algorithm.

### 8.1 Future Directions

While this dissertation has shown task-parallel platforms can be implemented to efficiently support interactive applications, there are many directions in which one could build upon this work. From tools to support development and debugging, to further work on scheduling interactive applications, there is much left to explore. Below are but a few possible directions to take this work.

**First**, while we have presented two schedulers for interactive applications in this dissertation, one (APS) is theoretically efficient but would be practically inefficient if faithfully implemented, and the other (Prompt Greedy) is practically efficient but is theoretically inefficient because of how readily workers switch to lower priorities when there is no immediately available high priority work. In practice, our approximation of APS works fairly well, and Prompt Greedy’s shortcomings can be overcome to some degree by the programmer inserting manual interruption points to prevent high priority tasks from being delayed for too long. However, it would be better to have a scheduler that provides the best of both worlds without extra work on the part of the user (e.g. ideally reducing or eliminating manual interruption points).
Second, the schedulers presented in this dissertation has no fairness between priority levels. In some interactive applications there are tasks that can be delayed to a point, but too much delay would cause performance issues. In Memcached, for example, cache maintenance tasks can be delayed for some time, but if cache maintenance is never performed then the foreground tasks responsible for responding to client requests would severely degrade in performance. Ideally, a task-parallel platform would provide a means to handle this scenario by making sure lower priority tasks are not starved of resources for too long.

To my knowledge, the only prior work to deal with fairness in scheduling interactive task-parallel applications is that of Muller et al. [126]. In their work, they propose the fairly prompt scheduling principle for fair scheduling of interactive applications and implement a proof-of-concept scheduler. Their prototype implementation does not, however, strictly adhere to the fairly prompt scheduling principle. Much as the prompt scheduling principle was a good guide for our schedulers, however, I believe an effective and fair scheduler could be achieved by modifying the interactive schedulers proposed in this dissertation with the fairly prompt scheduling principle as a guide.

Finally, in this dissertation we only considered synchronization using futures and fork-join parallelism. Interactive applications, however, often use other synchronization primitives, such as locks, condition variables, and barriers. In fact, Memcached has quite a few locks and condition variables used to, for example, coordinate between background threads that maintain the LRU properties of the cache. While we did not implement a locking primitive in I-Cilk, we did need to implement a condition variable that would suspend an execution context and return to the task-parallel scheduler. This dissertation does not, however, provide any means to check for priority inversions when using locks or condition variables, and when analyzing APS we do not consider these types of synchronization in our analysis.

It would be interesting to integrate locks, condition variables, and other synchronization primitives into a type system extending what was presented in Chapter 5, guaranteeing there are no priority inversions from their use. This would then enable us to investigate methods to schedule such computations in a manner that provides good responsiveness for interactive tasks.

\[\text{We did not need to implement locks for good performance because the critical sections were, as is typically the case, incredibly short.}\]
References


167


[154] C. Voss, T. Cogumbreiro, and V. Sarkar. Transitive joins: A sound and efficient online
deadlock-avoidance policy. In Proceedings of the 24th Symposium on Principles and
Practice of Parallel Programming, PPoPP ’19, pages 378–390, New York, NY, USA,
2019. ACM.

programs. In Proceedings of the 47th Annual ACM Symposium on Principles of
Programming Languages (POPL), 2020.


scheduling strategies. In Proceedings of the 18th ACM SIGPLAN Symposium on
Principles and Practice of Parallel Programming, PPoPP ’13, pages 315–316, Shenzhen,
China, 2013. ACM.


for two-dimensional dags. In Proceedings of the 23rd ACM SIGPLAN Symposium on
Principles and Practice of Parallel Programming, PPoPP ’18, pages 368–380, Vienna,
Austria, 2018. ACM.

Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures,
SPAA ’16, pages 61–70, Pacific Grove, California, USA, 2016. ACM.

Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of
Parallel Programming, Barcelona, Spain, 2016. ACM.

for increased utilization in warehouse scale computers. In Proceedings of the 40th Annual
International Symposium on Computer Architecture, ISCA ’13, pages 607–618, Tel-Aviv,
Israel, 2013. ACM.

Cilk: Lazy promotion from tasks to threads in c/c++. In Revised Selected Papers of
the 28th International Workshop on Languages and Compilers for Parallel Computing -