Thesis Proposal
Responsive Parallel Computation

Stefan K. Muller

May 2017

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Umut Acar (Chair)
Guy Blelloch
Mor Harchol-Balter
Robert Harper
John Reppy (University of Chicago)
Vijay Saraswat (IBM TJ Watson Research Center)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Abstract

Multicore processors are becoming increasingly prevalent, blurring the lines between traditional parallel programs, which use *cooperative threading* to reduce execution time, and interactive programs which use *competitive threading* to increase responsiveness. To assist programmers in developing this new class of *responsive parallel programs* which use threads for both of these purposes, I propose a model that combines the cooperative and competitive paradigms. The proposed model spans many levels of abstraction. The first component is a cost model that extends existing models of cooperative threading in order to allow programmers to reason about the parallel running time and the responsiveness of responsive parallel applications. The second is a language that neatly combines abstractions for both forms of threading, and enables reasoning about efficiency and responsiveness at the level of the source code. Finally, I propose to implement the language as part of a compiler for Standard ML, and evaluate it on a benchmark suite including a number of realistic responsive parallel applications.

Contents

Chapter 1

Introduction

Threads, as an abstraction for multiple sequences of instructions running simultaneously, have been a familiar feature of programming for quite some time. While threads take many forms and have many uses, applications of threads generally fall into two classes, which we will call *competitive* and *cooperative*. The aim of competitive threading is to maximize responsiveness, whether to user input, external devices, inter-process communication, or other external stimuli. Threads in a competitive system are generally scheduled preemptively. The system may interrupt threads periodically to enforce some notion of fairness, or it may do so based on external events, or to correspond to an assignment of priorities to the threads. The aim of cooperative threading, by contrast, is to separate pieces of work that may be performed in parallel into separate threads that can be scheduled onto multiple processing units to decrease the overall execution time of the program. Cooperative threads tend to be fairly fine-grained, each containing only small amounts of work to expose the maximum amount of parallelism, and so cooperative systems generally eschew high-overhead preemption mechanisms in favor of non-preemptive scheduling algorithms, such as work stealing, to distribute work appropriately across multiple processors. Since cooperative threads do not generally require any sort of fairness or responsiveness guarantees, it is usually not important to prioritize or interrupt threads in such a system.

For much of the history of computing, it has sufficed to keep these two modes of threading separate, largely because they have been used in distinct applications. Applications that use competitive threading are primarily interactive. While they may also perform substantial computations, these computations are likely confined to single threads. In many cases, these applications are running on a single-core machine and are simply using threads for responsiveness. On the other hand, cooperative threading has generally been used in applications that process large amounts of data in a structured, easily parallelized way, and fundamentally use parallelism to leverage a multiprocessor architecture. These applications have come largely from domains such as scientific computing and simulations.

With the rise of consumer multicore machines, the landscape of concurrent and parallel computing is changing. It is now increasingly desirable for everyday, consumer applications to use parallel threads to speed up computation. Cooperative threading is ideal for this purpose, but if (as is likely) such applications also need to interact with the user, competitive threading should be used as well. Consider, for example, a real-time, multi-player computer game. Such a program must take input from many sources, for example a local player's mouse and keyboard, and

a network connection to remote players. It must respond in real time to these inputs in order to maintain a quality user experience. At the same time, the software may be controlling an AI player, computing its strategy in real time by solving a compute-intensive (and often, easily parallelized) search problem in the background. This computation must be completed in a timely fashion, but must not detract from the responsiveness of the program to user input.

This new application domain of responsive parallel programs indicates the need for a threading model that neatly combines cooperative and competitive threading, allowing both to be expressed and efficiently scheduled within the same language and runtime. I propose to develop such a model, in several forms, as summed up in my thesis statement.

Thesis Statement: *It is possible to extend existing language and cost models for cooperatively threaded parallelism in a natural way to account for competitive threading constructs, and to design and implement scheduling algorithms that account for both throughput and responsiveness.*

First, I propose an abstract model for responsive parallel computations, which builds on a standard method of modelling parallel computations as directed acyclic graphs. Next, I propose a language which allows both forms of threading to be expressed naturally, and which comes with a cost model that admits reasoning about execution time and responsiveness at the language level. Finally, I discuss scheduling algorithms to efficiently schedule both competitive and cooperative threads, and propose to implement and evaluate such algorithms in the context of an extension to the Standard ML language. After providing background on existing uses and models of threads (Chapter 2), Chapters 3 through 6 will discuss existing research I have conducted with my advisors on the four areas listed above (cost models, language models, scheduling algorithms and implementation). Finally, in Chapter 7, I will outline the future work to be conducted as part of the proposed thesis.

Chapter 2

Background

2.1 Cooperative Threading

Cooperative threading originated in early languages such as Id [4] and Multilisp [20] in the late 1970s and early 1980s. After a brief decline, the paradigm regained popularity, and has gained newfound importance with the rise of multicore architectures in consumer hardware. Cooperative threading constructs have arisen in everything from industrial high-performance computing languages such as X10 [13] to parallel extensions of languages such as Java [23, 26], Haskell [12, 25] and ML [19, 24]. While these languages and libraries and the threading constructs they use take many forms, they have at least two common features. First, threads are used primarily for the purpose of exposing parallelism in the program: if two operations can be performed in parallel, the programmer can place them in separate threads to indicate to the language runtime an opportunity to run the program more quickly by leveraging this parallelism. Second, the threading constructs allow programmers to express parallelism using high-level abstractions without explicitly specifying a mapping from threads onto the underlying hardware. Many such abstractions have been proposed. In Section 2.1.1, I will discuss two popular ones which will be used in this work.

While some of the languages and threading libraries that we have placed in the category of cooperative threading allow synchronization and accesses to shared memory, cooperative threading mechanisms generally encourage or force the programmer to maintain a separation between parallel threads and minimize or eliminate the concurrent use of shared resources. As such, functional programming is often considered to be a good paradigm for use with cooperative threading, and many parallel languages such as NESL [7] and Manticore [19] have been based on parallel languages like ML. In this work, we will follow the trend and use a functional, ML-like language as a starting point, though we must introduce non-functional features in order to express interaction.

Since, in cooperative threading models, the mapping of threads onto processors is not made explicit by the programmer, the language runtime must include a mechanism for scheduling the execution of the high-level threads. Since many cooperative threading paradigms encourage expressing as much parallelism as possible, a good scheduling mechanism must minimize overhead by implementing threads using lightweight, user-level structures and balancing them across the

available processors. In Section 2.1.3, I will discuss *work stealing*, a popular class of algorithms for accomplishng this scheduling.

2.1.1 Abstractions for parallelism

Two of the most common abstractions for expressing parallelism are *fork-join* or *nested* parallelism and parallel *futures*. In fork-join parallelism, a program forks two parallel threads which may be run in parallel. Both threads run to completion and then join, returning both results to the main thread. A common language construct for expressing fork-join parallelism, especially in higher-order languages, is the parallel tuple, often indicated with a command such as `fork` or `par`:

```
1 let (a, b) = par (f (), g ())
2 in
3   h a b
```

In the above code, the parallel tuple calls functions `f` and `g` in parallel, and returns a tuple of their results when both complete. The function `h` is called with the two results as its arguments. As a larger example, take the parallel Fibonacci function, which is commonly used in the literature as a simple parallel function. We use a parallel tuple to perform the two recursive subcalls in parallel.

```
1 function fib n =
2   if n <= 0 then n
3   else let (a, b) = par (fib (n - 1), fib (n - 2))
4        in a + b
```

Parallel futures, or simply futures, allow a programmer to spawn a computation that runs asynchronously with the main thread. The command to create a future, often called `spawn` or simply `future`, returns a handle to the running computation. Using this handle, the programmer may demand (also known as *touch* or *force*) the result elsewhere in the program. If the future has not completed running when it is forced, the call to `force` blocks until the result is ready.

```
1 let f = future (f ())
2 in
3   ...
4   force f
```

Futures are a strictly more general abstraction than parallel tuples, as one can implement parallel tuples using futures (here, we assume that the two components of the tuple are thunks which can be called with a unit value to start the computation).

```
1 function fork (f, g) =
2   let a = future (f ())
3       b = future (g ())
4   in
5     (force a, force b)
```

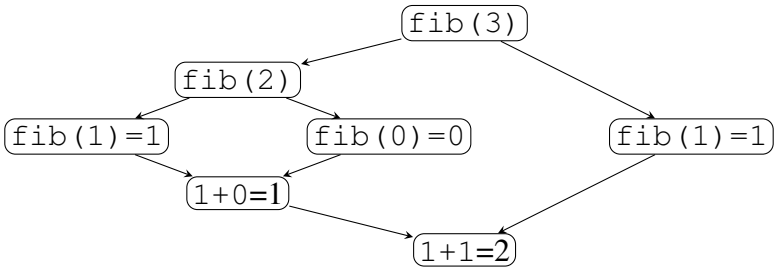


Figure 2.1: A dag representation of `fib(3)`.

Our language will use parallel tuples for most parallelism, since they are sufficient to encode many examples and this more restricted form of parallelism allows for optimizations and cleaner semantics. However, we will require a construct related to futures in order to implement prioritized computation.

2.1.2 Cost Models

In this section, I will take a brief detour from describing cooperatively threaded languages to discuss some of the abstract models that have been used for reasoning about the cost and parallel structure of cooperatively threaded programs. I will build on these models in the work described in this proposal[1].

It is common to represent parallel computations using directed acyclic graphs or *dags*. Vertices of the dag represent instructions of the computation, each of which executes in one unit of time, which we call a *step*. Edges represent dependencies between instructions: an edge from u to u' indicates that the instruction represented by u must execute before u'. We write $u \leq u'$ to indicate that u is an ancestor of u'.

For example, we can represent an execution of `fib(3)` as a dag, as shown in Figure 2.1. For brevity, each vertex represents a call to `fib` instead of an individual instruction, but can be expanded into a chain of instructions if desired. Vertices with out-degree two fork two parallel computations, which may be executed in two (cooperative) threads. Vertices with in-degree two join two parallel computations; a join vertex synchronizes its two in-neighbors by waiting for both of them to complete before executing.

In parallel computing with cooperative threads, two measures are typically used to discuss the running time and parallelism of a computation: the *work* of a dag g, which we write $W(g)$, is defined as the number of vertices in the dag and *span* $S(g)$ is defined as the length of the longest path in the dag. The work corresponds to the time required to perform the computation on one processor. The span corresponds to the amount of time required to perform the computation given an infinite number of processors. The quantity $\frac{W(g)}{S(g)}$ is often referred to as the *parallelism* of the computation: it expresses how much useful parallelism is exposed by the computation.

A *schedule* is an assignment of vertices to processors at each step such that each vertex is executed only when it is *ready* (meaning all of its ancestors have completed). A schedule is *greedy* if as many ready vertices as possible are executed at each step. While designing an

[1]This section is largely taken from our PLDI '17 paper

optimal schedule is NP-hard [37], a greedy schedule is guaranteed to have a run-time within a factor of 2 of optimal. Results of this form are often attributed to Brent [10], who proved a similar result for "level-by-level" schedules. The bound for greedy schedules was established in later work [8, 17].

Theorem 1. *The run-time of a greedy schedule of a dag g is bounded by $\frac{W(g)}{P} + S(g)\frac{P-1}{P}$.*

This run-time is within a factor of 2 of optimal because $\frac{W(g)}{P}$ and $S(g)$ are each, individually, lower bounds on the run-time.

While useful in getting a sense of the parallel structure of a program, dag-based models have two limitations. The first is that they must effectively be assembled "after the fact" using information from a particular execution of a program. They do not help us in directly analyzing the source code of a program. For that, we turn to language-based cost models. Such "cost semantics" have been in use for analyzing the complexity of programs in higher-order languages for nearly 30 years [31, 32]. Blelloch and Greiner [6] and Spoonhower et al. [35] use similar techniques to generate cost graphs, or dags, from a program written in a higher-order parallel language. We build on their tools in the work described in this proposal.

The second weakness of the dag model described above is that it considers only offline scheduling, in which the dag is known in advance. As difficult as this problem is, it does not take into account the complexity of assigning threads to processors as the dag unfolds dynamically at runtime, nor the additional time required to make these scheduling decisions at runtime. In the next section, we discuss practical online scheduling algorithms for parallel programs.

2.1.3 Scheduling

As the name suggests, threads in a cooperative threading system are scheduled cooperatively, i.e. non-preemptively. A common approach to load balancing in these systems is *work stealing* [11, 20]. In work stealing, each processor "owns" a set of parallel threads. When a processor is idle, it "steals" one or more threads owned by an overworked processor. In particular, a concrete strategy frequently used (e.g. [9]) is to store threads in a double-ended queue, or *deque*. A deque has two ends, the top and bottom. Threads can be pushed or popped from the bottom, and can be popped from the top. Since threads are only pushed to the bottom, popping from the top results in a first-in-first-out ordering, and popping from the bottom results in a last-in-first-out ordering. When a processor generates new work, it pushes it to the bottom of its own deque. When it needs to execute a thread, it pops the bottom thread from its own deque. When a processor's deque is empty, it pops the *top* thread from another processor's deque. Because other processors only access the top of a deque, synchronization is only required at the top or when the deque contains one element. Many efficient implementations of concurrent work stealing deques have been proposed [3, 14]. Recent work [2] proposes a variant of work stealing in which deques are private to a processor, and processors cooperate to send tasks using message passing idioms.

Work stealing algorithms have been shown to execute parallel programs in times that are within a constant factor of the bound given by Brent (Theorem 1), i.e. $O(W/P + S)$ [2, 3, 9]. An essential feature of these analyses is the so-called *deque invariant* or *deque discipline*, which maintains that, since threads are pushed onto deques in order with earlier threads farther toward the top, the first threads to be stolen are the ones closer to the root of the dag, which generally

represent larger pieces of work. This allows the cost of the steal to be amortized over the large pieces of work stolen.

Some prior work [22, 38, 39] has considered work stealing using priorities in order to improve performance in computational applications where the order in which subtasks are completed can reduce the overall work (e.g. search problems with heuristics that indicate which branches are more promising). As with traditional work stealing, these approaches are non-preemptive and do not consider responsiveness concerns.

2.2 Competitive Threading

Competitive threading is a more familiar paradigm to many programmers, and has been in common use since systems such as STAR [34] and Cedar [36]. While programs using threads in this paradigm may or may not run on multiprocessor hardware, the focus in competitive threading, as stated in a survey by Hauser et al. [21], is "the role of threads in program structuring rather than how they are used to exploit multiprocessors." This survey and Birrell [5] describe several uses of competitive threads, including reducing latency when a thread may have to wait on a long-running process or external device, and setting up pipelines to structure complex processes. These types of threads are largely useful in interactive systems, both for efficiency (since if one thread is blocked on an I/O operation, other threads can continue doing other work) and for structuring programs (a program can separate different interactive elements into different threads, rather than combining these various concerns into one large piece of code).

In modern systems, competitive threads are scheduled preemptively, usually by the operating system. The exact scheduling policy is generally unspecified, but such systems will try to maximize responsiveness, as this is a primary goal of competitive threading. The POSIX Threads (pthreads) API, a well-accepted standard for competitive threading systems, also allows programmers to set the priorities of threads and to control some aspects of the scheduling policy [1].

Chapter 3

A dag-based cost model for responsive parallelism

The first aspect of this thesis proposal is to build on the dag-based cost model of Section 2.1.2, extending it to represent important features of interactive computations which are generally not considered in models of cooperative threading. Our prior work in this area was done in two stages: in Section 3.1, we extend the model to handle computations with operations that may incur latency, such as I/O operations. This work was previously described in our paper "Latency Hiding Work Stealing" [27]. In Section 3.2, we further extend the model to assign subdags a higher, "foreground", priority, leading to a model for two-priority interactive computations. This work is described in more detail in our upcoming paper "Responsive Parallel Computation: Bridging Competitive and Cooperative Threading"[28].

3.1 Blocking I/O operations

Previous cost models for parallel computations assume that all operations require computational resources in addition to time. This assumption is violated by operations such as "input" and "sleep" which take a possibly substantial amount of time, which we call the *latency* of the operation, but which do not require computational resources for the duration of this latency. It is worthwhile to separate the latency from the computational work since, ideally, a schedule of the computation would not assign a processor to the operation during its latency, freeing up those computational resources for other tasks. This is sometimes known in the systems community as *latency hiding*. Furthermore, the cost metrics should reflect the fact that a latency-incurring operation need not take up a processor.

To model latency-incurring operations, we add edge weights to the edges of the dag. An edge is now represented as a triple (u, u', δ), indicating that the instruction corresponding to u must be completed δ steps before u' can be run. The edge weight δ corresponds to the latency incurred by the instruction u. In most cases, $\delta = 1$ and this reduces to the usual case in which u' can run immediately after u.

Consider the program in Figure 3.1. This program computes the 3^{rd} Fibonacci number and, in parallel, asks the user two questions and responds. The dag corresponding to this code is

```
function ask i =
  if i <= 0 then bg ()
  else
    let
       _ = output ``What is your name?''
       n = input ()
       _ = output ``What is your quest?''
       q = input ()
       _ = output (``Hello, '' ^ n)
    in ask (i-1)

function fib_ask () = par(fib 3, ask 1)
```

Figure 3.1: Fibonacci composed with an interactive process.

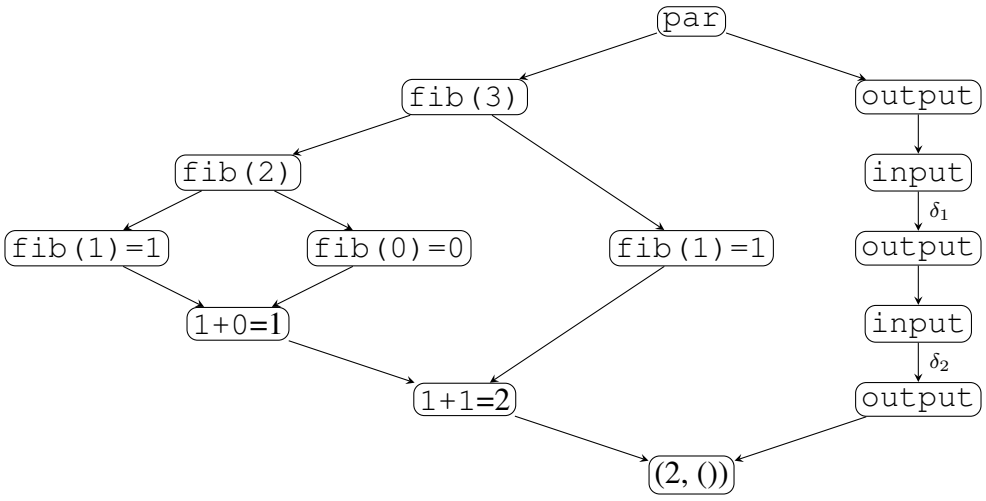


Figure 3.2: A dag representation of `fib_ask`.

shown in Figure 3.2. Most edges have a weight of 1 and these weights are omitted for clarity. The outgoing edges of the two input operations however, have latencies of δ_1 and δ_2, reflecting the latencies of the input operations.

It may seem strange to annotate the dag with information that can only be known "after the fact", such as the length of time it took the user to answer the questions. However, this is consistent with the existing usage of dag models, which reflect a particular execution of a program. Dags might, for example, reflect which branch of a conditional was taken at runtime, and other "post-hoc" information.

The definition of work of a *weighted dag* in this extended model remains the number of vertices in the dag. It does not take into account the edge weights, reflecting the desired property that latencies should not be counted as computational work. However, this means that, unlike in the traditional model, work no longer corresponds to the one-processor execution time of the program (since that execution time might include some latency). The span of a computation represented by a weighted dag is the longest *weighted* path in the dag. This reflects the fact that latencies on the critical path of a computation will delay the computation even though no computational resources are required because the results of the latency-incurring operations are required for the computation to proceed.

The definition of a greedy schedule of a weighted dag is largely unchanged from the standard dag model, with the exception that a vertex is now only considered ready when its parents have completed execution and their latency requirements (if any) have expired.

Theorem 2. *If a computation with work W and span S is scheduled using a greedy schedule on P processors, then the execution time is at most $\frac{W}{P} + S$.*

See the paper for the proof. Note that the bound given here is slightly worse than that of Theorem 1, which is $\frac{W}{P} + \frac{P-1}{P}S$. The reason for this is that, when operations incur latencies, it is possible for all processors to be idle at once, which is not possible in standard parallel computations.

3.2 Prioritized computations

In interactive programs, it is often the case that some threads (usually those that interact with the user) have a higher priority than others. If these threads are delayed, it will impact the responsiveness of the program, harming the user experience. Consdier the program `fib_ask` of Figure 3.1. If a large number is used as input for the Fibonacci computation, the many threads created by that parallel computation may overwhelm the single thread used for the interactive process, because the system has no way to distinguish these threads. Assume that we add a construct `fg` indicating that a block of code should be assigned higher priority. We refer to such a piece of code as a *foreground block*. For example, we might rewrite the last line of Figure 3.1 as follows:

```
function fib_ask () = par(fib 3, fg (ask 1))
```

Graphically, we represent the fact that the portion of the dag corresponding to `ask 1` is in the foreground by drawing a box around this piece of the dag, as shown in Figure 3.3. In our bounds, we also require that dags be free of priority inversions in which a foreground block may

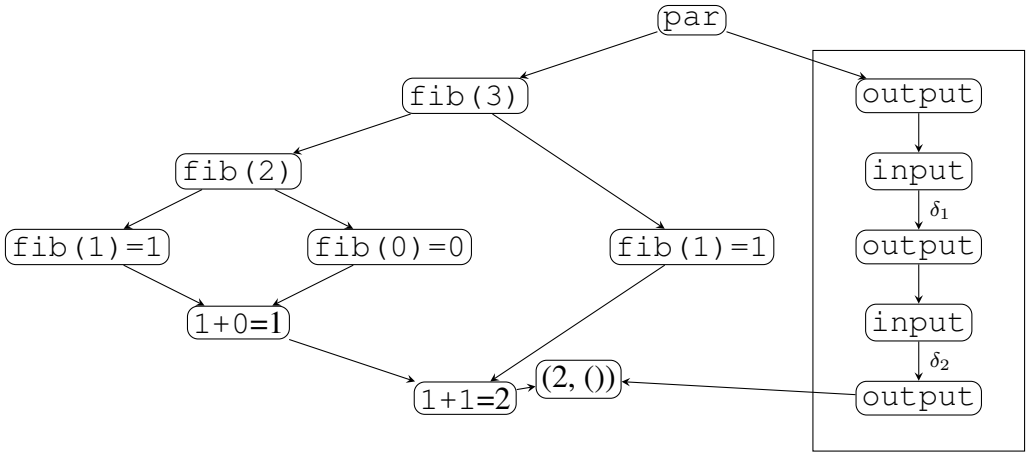


Figure 3.3: A dag representation of `fib_ask`.

have to wait on background code, a property that we call *well-formed*. Formally, this requires that foreground blocks are not nested and have no incoming edges from outside the foreground block except to the source of the foreground block.

In order to capture the responsiveness requirements of prioritized dags, we must extend the notion of the cost beyond simply execution time. To do this, we define the *response time* $R(f)$ of a foreground block f to be the number of steps between when the source of f becomes ready and when the sink is executed (inclusive). The total response time of a P-processor execution of a program is the sum of the response times of all of the foreground blocks in the program.

Ideally, the response time of an execution should depend only on the amount of foreground work, and should be (largely) unaffected by the amount of background work. To formalize this, we define $W^\circ(f)$ and $S^\circ(f)$ as the work and span, respectively, of a foreground block, and the total *foreground work (resp. span)* $W^\circ(g)$ (resp. $S^\circ(g)$) to be the sum of the work (resp. span) of all foreground blocks in the program. The definitions of work and span are otherwise unchanged from the weighted dag model. We also need one more metric of a dag, called the *foreground width*, $D(g)$, which corresponds to the maximum number of foreground blocks that can be active at a time.

We can now bound the total response time of an execution. To do this, we assume a *prompt* schedule, which generalizes a greedy schedule to dags with priorities: a prompt schedule will always assign as many ready foreground vertices as possible, followed by ready background vertices. By definition, prompt schedules are greedy, so Theorem 1 still holds for the total execution time.

Theorem 3. *If a computation represented by a well-formed dag g is scheduled using a prompt schedule on P processors, then the total response time is at most $D(g)\frac{W^\circ(g)}{P} + S^\circ(g)$.*

Chapter 4

A language and cost model for responsive parallelism

The second aspect of the proposed thesis is the design of a language for writing responsive parallel programs. In this chapter, I discuss the design of the "static" components of such a language—the syntax, type system and cost model. The next chapter discusses the runtime of the language, in particular a scheduling algorithm for responsive parallel threads. We have several design goals for a responsive parallel language:

- The language should have support for, at least, fork-join parallelism, and should have constructs for prioritizing threads.
- Any assumptions required for efficiency (e.g. the lack of priority inversions discussed in Chapter 3) should be enforceable at compile time by the type system.
- The language should be accompanied by a cost model allowing programmers to reason at the language level about the execution time and responsiveness of their programs.

The last item above is particularly important, even given the dag-based cost models of Chapter 3, because a dag represents only one possible execution of a program and so dag-based models allow reasoning only at the level of a particular execution. A language-based cost model of the form I will describe in this chapter allows reasoning about all possible executions.

Our prior work in this area is described in our PLDI '17 paper, which presents a functional core calculus, complete with a type system and cost semantics meeting the above requirements.

4.1 Language and type system

The PLDI '17 paper presents a language λ^{ip} which extends a simply-typed functional calculus with:

- Parallel tuples `par(e1, e2)` which spawns two fine-grained threads to execute `e1` and `e2` in parallel
- Generic input and output operations which abstract over various forms of I/O (e.g. console I/O, GUI interaction, etc.)

- Priority constructs `fg(e)` for denoting foreground blocks and `bg(e)` for spawning an asynchronous background thread from within a foreground block.

Operationally, `bg(e)` spawns a background thread to run `e` and returns a handle to this thread. Meanwhile, `fg(e)` evaluates `e` down to a background thread handle. It then blocks until the background thread completes, and returns the return value of the thread. This mechanism is similar to parallel futures in that an asynchronous computation is spawned (by `bg`) and "forced" or "touched" by `fg`.

As an example, consider the code below for a "Fibonacci server."

```
function fib_server () =
  let n = input () in
    if n < 0 then bg ()
    else
      bg (output (fib n));
      fib_server ()

fg (fib_server ())
```

The server consists of an interactive loop which runs in the foreground taking an integer as input. When the user enters an integer n, the server spawns a new thread to compute, and then asynchronously print, the n^{th} Fibonacci number. Meanwhile, the interactive loop continues in the foreground. If the user enters an invalid number, the loop terminates by returning a unit value (which, as it is being returned to the background, must be wrapped in a `bg`).

In the context of λ^{ip}, a priority inversion as mentioned in Chapter 3 can only occur if a background computation is demanded in the foreground. For example, consider the following "bad" variant of the Fibonacci server:

```
1 function fib_server_bad () =
2   let n = input () in
3     if n < 0 then bg ()
4     else let fibn = bg (fib n) in
5            output (fg (fibn));
6            fib_server_bad ()
7
8 fg (fib_server_bad ())
```

The function `fib_server_bad` receives the input from the user and then spawns the background Fibonacci computation `fibn` (Line 4). It then immediately demands the result for output (Line 5). This program might not be responsive because a foreground computation (function `fib_server_bad`) is waiting on a potentially long-running background computation.

The type system of λ^{ip} is designed to separate foreground and background computations in order to prevent such a priority inversion. It does this using a modal type system similar to prior type systems for staged computation (e.g. [15, 16, 18]). These type systems generally allow computation at earlier stages to encapsulate and pass around, but not demand the value of, computation that is to run at a later stage. Here, our foreground computations correspond

nicely to "earlier stages" and background computations correspond to "later stages". The primary difference is that, operationally, background computations are run asynchronously at the same time as the foreground computation while in multistage languages, encapsulated later-stage computations are delayed to be run only at the proper stage.

More concretely, our typing judgment is $\Gamma \vdash e : \tau @ w$, indicating that under context Γ mapping variables to types, expression e has type τ at world w, where w is either $\mathbb{F}$ for foreground computation or $\mathbb{B}$ for background computation. The rules for typing `fg` and `bg` transition between the two worlds.

$$\frac{\Gamma \vdash e : \tau @ \mathbb{B}}{\Gamma \vdash \texttt{bg}(e) : \bigcirc\, \tau @ \mathbb{F}} \qquad \frac{\Gamma \vdash e : \bigcirc\, \tau @ \mathbb{F}}{\Gamma \vdash \texttt{fg}(e) : \tau @ \mathbb{B}}$$

In particular, these rules only only allow background computations to be demanded (using `fg`) in the background, preventing priority inversions.

4.2 Cost semantics

While the dag-based cost model of Chapter 3 provides a convenient way to describe the parallel structure of a computation, it abstracts over many details of the program, and only applies to one execution of a program. We therefore wish to build a cost model that is based on the language. Such a model will allow programmers to reason about execution time and responsiveness at the level of the code of a program.

We have developed a language-based cost semantics for our language λ^{ip}. The cost semantics takes the form of an evaluation judgment $e \Downarrow v; g$ that, given a program expression e, produces both the value to which the program evaluates and a weighted, prioritized cost graph g of the form described in Section 3.2. The work, span, foreground work, foreground span and foreground width of the computation can be read off from this graph. More precisely, since the semantics can be nondeterministic (ours is, because of the way we handle inputs; more detail on how this nondeterminism is resolved can be found in the paper), the semantics gives a set of graphs

$$\{g \mid e \Downarrow v; g\}$$

each corresponding to a valid execution of the expression. We will take the maximum work (span, foreground work, etc.) of the graphs in this set to be the work (span, etc.) of the computation.

Our cost semantics is mostly based on the work of Spoonhower et al. [35]. Operations that are considered atomic produce a graph consisting of a single fresh vertex, written in the form $[u]$. Expressions that evaluate multiple subexpressions combine the graphs generated by the subexpressions, either in sequence using the sequential composition operator $\oplus$ or in parallel using the parallel composition operator $\otimes$. As an example, the cost rules for function application and parallel tuples are shown below:

$$\frac{e_1 \Downarrow \lambda x{:}\tau.e; g_1 \qquad e_2 \Downarrow v; g_2 \qquad [v/x]e \Downarrow v'; g_3 \qquad u \text{ fresh}}{e_1\ e_2 \Downarrow v'; g_1 \oplus g_2 \oplus [u] \oplus g_3} \qquad \frac{e_1 \Downarrow v_1; g_1 \qquad e_2 \Downarrow v_2; g_2}{e_1 \parallel e_2 \Downarrow \langle v_1, v_2 \rangle; g_1 \otimes g_2}$$

Given the cost semantics and the type system, we can show that all dags generated by the cost semantics are well-formed in the sense of Section 3.2 and we can therefore apply Theorem 3 to evaluate the cost of λ^{ip} computations. See the paper for the proof.

Theorem 4. *If* $\cdot \vdash e : \tau @ w$ *and* $e \Downarrow v; g$*, then* g *is well-formed.*

4.3 Operational semantics and realization

We have thus far established bounds on the responsiveness and run-time of prompt schedules of well-formed execution dags and defined a language for prioritized interactive parallelism whose cost semantics generates only such dags. The assignment of dags to computations provides a theory of the responsiveness and efficiency of λ^{ip} programs with which we can derive analytic results about particular programs. But such results are abstract until they can be validated in terms of a lower-level model that is closer to an implementation on a real machine. To this end, we present a transition semantics that specifies an abstract implementation of λ^{ip}, and show that the cost semantics may be validated with respect to it.

The transition semantics of λ^{ip} records and schedules the active threads. Threads are created by the $\texttt{bg}(e)$ construct and by the parallel tuple construct. A set of local transition rules define how expressions evaluate within a single thread, possibly spawning new threads in the process. A global transition rule selects a set of ready threads and schedules them for a "quantum" of local computation, modeled by a single transition. The number of global steps taken to run a program to completion is the run-time. The transition relation also tracks the total response time.

To prove the correspondence between the cost model of Section 4.2 and the operational semantics, we generate cost graphs for intermediate states of execution and show a correspondence, at each step of execution, between the state of the program as viewed by the operational semantics and the current computation dag, as determined by the cost model. We then use a similar argument to the proof of Theorem 3 to show a bound on the number of global steps and the total response time for a prompt scheduling strategy.

Chapter 5

Scheduling responsive parallel tasks

The next component of the thesis will be designing an efficient parallel scheduler for programs in the proposed language. Clearly, such a scheduler must schedule computations based on their priorities (approximating a prompt scheduler) in order to maintain the responsiveness of the program. We require another necessary property of the scheduler which, though no less important, is less clear from the cost models presented: the scheduler must be able to hide the latency of interactive operations by switching to another thread when one performs a blocking operation. This property is nontrivial: a standard work stealing scheduler will not remove a thread from a processor once it has been scheduled; a thread that blocks will continue consuming its assigned processor while it blocks. We have investigated the latter property in our SPAA 2016 paper. This work is described in Section 5.1. While we have not investigated prioritized scheduling in as much detail, our PLDI 2017 paper includes a prototype implementation of a prioritized scheduler. I will discuss the scheduling algorithm we used for this prototype in Section 5.2.

In designing these algorithms, we are concerned with the practicality of the scheduler and not just the cost bounds of the resulting schedule. A prompt latency-hiding scheduler could be implemented trivially by simply maintaining a global priority queue of currently ready threads, and reassigning ready threads to processors in priority order at periodic (ideally small) intervals. Such an implementation would display unacceptable overhead, however, because of the cost of synchronizing on the global queue and of frequent preemption. We therefore consider algorithms based on work stealing (Section 2.1.3, which uses decentralized data structures to regain efficiency.

5.1 Latency hiding for blocking operations

The first step in developing a latency-hiding runtime is replacing blocking I/O operations with nonblocking versions. For example, our version of an "input" operation would first check if bytes are available on the appropriate device. If not, it returns to the scheduler, allowing the runtime to schedule another thread on the processor. It also installs a callback which will reinsert the calling thread into the scheduler when the input is available.

When a thread suspends and returns to the scheduler, the scheduler places the suspended thread aside and returns to the deque to schedule the next available vertex. When the thread

resumes, assuming the processor has not run out of work in the meantime, the resumed thread is placed back in the deque. If, however, the processor runs out of work in its deque and needs to steal work while a thread is still suspended, the processor creates a new deque and puts the stolen work in the new deque. The old deque is suspended, waiting for the suspended thread to resume. When the suspended thread resumes, it is returned to the old, suspended, deque. Before stealing new work, a processor will check if it has other deques with work. If so, it will switch to working on that deque rather than stealing new work. Processors only steal work if all of their deques are supended. Thus, the number of deques a processor can have is limited to the maximum number of simultaneously suspended threads, a quantity we call the *suspension width* and notate U. Suspending deques in this way maintains the deque invariant that vertices in each deque are ordered by their depth in the overall computation dag.

In the SPAA paper, we use a version of the Arora, Blumofe and Plaxton analysis [3] to bound the running time of our latency hiding scheduler. While we did not achieve the theoretical greedy scheduling bound given by Theorem 2, our bound comes fairly close, as we expect U to be quite small for most practical purposes.

Theorem 5. *The running time of the latency-hiding work stealing scheduler on a computation with work W, span S and suspension width $U > 1$ is at most*

$$O\left(\frac{W}{P} + SU(1 + \lg U)\right)$$

5.2 Preliminary work on responsive scheduling

Our prioritized scheduler builds on a recent variant of work stealing [2] which replaces the shared work stealing deques with private deques and accomplishes load balancing through message passing between processors. In our scheduler, each processor maintains a private priority queue of threads. Each processor also has a mailbox for other processors to send it threads and a flag indicating the priority of the thread on which it is currently working. Each processor works on its highest-priority thread, interrupting its work periodically to check if a higher priority thread has resumed or been sent to its mailbox. Such preemption is necessary so that foreground threads cannot get delayed by long-running background threads.

Load balancing happens as follows. At set times, each processor picks a random target processor and attempts to send (*deal*) it work. Say that processor i is attempting to deal work to processor j. Processor i checks j's flag to see what j's highest-priority thread is. If i has a higher-priority thread to deal j (or any thread if j is idle), it places it in j's mailbox.

While we have not theoretically analyzed this algorithm, we have implemented a prototype language runtime using this scheduler (see Section 6) and it displays good scaling in practice.

Chapter 6

Preliminary Implementation and Evaluation

We have implemented prototype versions of both schedulers described in Chapter 5. Both implementations were built on a parallel extension of the MLton compiler for Standard ML, developed by Spoonhower et al. [35]. The implementations included ML libraries for non-blocking I/O and for the foreground and background constructs of λ^{ip}. The schedulers were implemented in ML. Since the goal of these implementations was primarily to establish plausibility and do initial performance tests, we did not extend the compiler to implement the type system of λ^{ip}.

6.1 Measuring responsiveness

To quantitatively evaluate our proposed techniques, we have developed an experimental framework for measuring the average responsiveness of an interactive program, as determined by its latency in responding to an interaction. Our framework simulates the user's interactions with the program (or the interactions with other aspects of the external environment) in order to provide a reproducible, consistent evaluation. The framework is also able to measure the latency of the program in responding to events triggered by the framework. The key component of this framework is a driver program, written in C, which takes as input a binary to evaluate and a trace file containing a sequence of events and runs the binary, performing the sequence of actions detailed in the trace file as if it were the user. The driver simulates standard input using a pipe ans mouse and keyboard actions through the X Window System using `libxdo`[1].

6.2 Benchmarks

While many benchmark programs exist for testing parallel schedulers (e.g. raytracers, parallel geometry and graph algorithms [33]), standard parallel benchmarks are heavily computational and not interactive. As such, an important part of evaluating the contribution of this thesis will be establishing a benchmark suite of responsive parallel programs. Such a program can be created

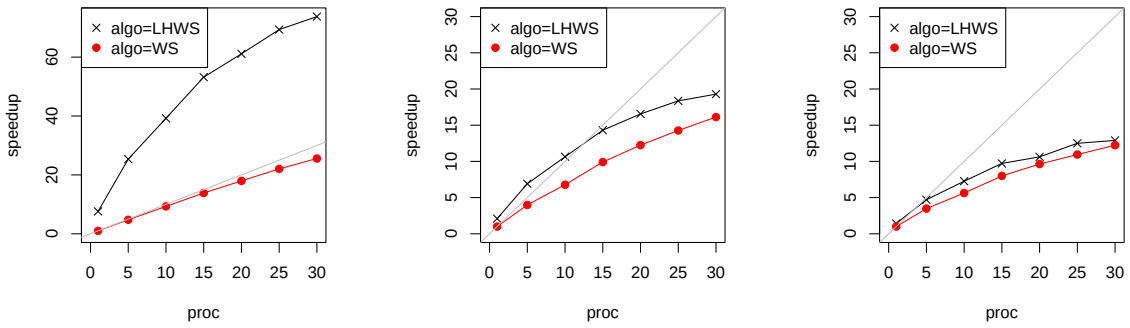
[1] http://www.semicomplete.com/projects/xdotool/

simply by composing a standard parallel program with an interactive process (for example, the `fib_ask` program of Section 3.2, which computes Fibonacci numbers while also responding to user input). Such a simple program is not without merit; it allows us to verify that the scheduler can allow background computation to scale without harming the responsiveness of foreground threads. However, the results of the thesis will not be convincing unless the language and runtime can scale to complex, realistic examples. We have already developed a number of benchmarks, ranging from simple to fairly complex.

Synthetic Benchmarks. The simplest benchmarks are the synthetic benchmarks, made by composing a standard parallel benchmark with an interactive process. For this purpose, we have selected two parallel benchmarks and two interactive processes: in each case, we have selected one "regular" benchmark and one "irregular" benchmark. The regular parallel benchmark is Fibonacci (as shown in Figure 3.1), which shows a regular, uniform parallel structure. The irregular parallel benchmark is unbalanced tree search (UTS) [29], a benchmark designed to be adversarial to load balancing. Meanwhile, the regular interactive process is terminal echo, which repeatedly asks for the user's name on standard input and greets the user, similar to the `ask` function of Figure 3.1. This benchmark always contains exactly one foreground thread. The irregular interactive process, network echo, listens for network connections and spawns a new foreground thread for each connection. Each thread interacts with the client as in terminal echo, above. We have generated four synthetic benchmarks using each combination of parallel and interactive benchmarks.

More realistic benchmarks. In the synthetic benchmarks, the parallel and interactive components are entirely separate. This makes it easy to run experiments by tuning the components individually, but it is not realistic. We have therefore developed a series of benchmarks which display more interplay between the parallel computation and the interaction. The benchmarks we have developed so far are summarized in the following table.

Fibonacci Server	Takes numbers from the user, computes their Fibonacci values asynchronously in the background
Convex hull server	Computes and displays the convex hull of a cloud of points as the user inserts more points by clicking
Interactive raytracer	Renders a simple, moving scene, as the user navigates around it
Web server	Responds to HTTP requests while performing parallel analytics computations in the background
Photo viewer	Scrolls through a folder of images while prefetching and decoding the next few images in the background
Streaming music server	Streams a music file to many simultaneous network clients of varying priorities



δ = 500ms δ = 50ms δ = 1ms

Figure 6.1: Experimental results of the prototype latency-hiding scheduler.

6.3 Initial results

Latency Hiding. To test the latency hiding scheduler, we used a benchmark which simulates performing a large computation over many inputs which must be fetched from remote servers. The benchmark forks many parallel threads, each of which simulates fetching the data by sleeping for a defined time δ, and then computes `fib(30)`. The results are then summed in parallel. We have run this benchmark using both our latency-hiding scheduler and Spoonhower's traditional work-stealing scheduler.

Figure 6.1 shows the self-speedup curves for our scheduler, labeled LHWS, and the standard work-stealing scheduler, labeled WS. For all curves, the speedup shown is relative to the one-processor run of WS. In all of these experiments, the number of elements (remote server connections) is 5,000. Note that, in this example, this is equal to the suspension width. The first (leftmost) plot shows the case where the latency δ is 500ms (a very high latency, which might represent waiting for user input or for data which requires some computation on a remote server). This plot shows that latency-hiding work stealing delivers superlinear speedups, as much as 3 times larger speedup than standard work stealing. The superlinear speedups are expected with latency hiding because the standard work stealer does not hide latency. In the second plot, δ is 50ms. Latency-hiding still provides substantial speedup benefit. The third plot shows that, when δ is smaller (1ms), there is less benefit to hiding latency. In the limit, if latency is expected to be small, programmers might wish to wait for operations to complete.

Prioritized Work Stealing. We tested the prioritized work stealing scheduler on several of the benchmarks described in Section 6.2. In Figure 6.2, we show the results for the synthetic benchmark that combines Fibonacci and terminal interaction. The left plot shows the speedup with respect to serial Fibonacci of the benchmark, where the number of interactions varies from 1 to 50 interactions per second. It also shows, for comparison, the speedup of Spoonhower's scheduler running a parallel Fibonacci computation with no interaction. Interaction causes the benchmark to scale slightly less than the non-interactive benchmark, but the interaction rate does not affect the scaling substantially, as would be expected. The right plot shows the response time

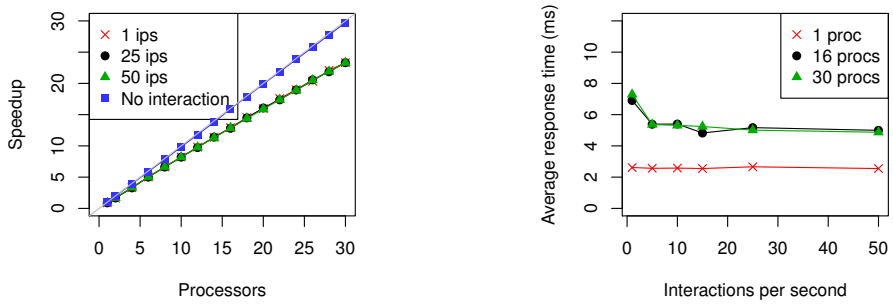


Figure 6.2: Speedup (l) and response time (r) for the Fibonacci-terminal benchmark.

of the terminal echo as the interaction rate increases. Again, interaction rate has little effect on the response time.

Chapter 7

Proposed Work

In the proposed thesis, I intend to develop:

1. A dag-based cost model for reasoning about interactive parallel programs, which generalizes the model of Section 3.2 to a more general notion of priority (more than simply foreground and background).
2. A language for writing responsive parallel programs which allows the programmer to express fork-join parallelism, input/output and a general notion of priority.
3. A scheduling algorithm for scheduling responsive parallel threads in order to optimize both throughput and responsiveness.
4. An implementation of the proposed language features and scheduler, and a thorough empirical evaluation.

Most of these components have several subcomponents, and design decisions made in one component or subcomponent can constrain or inform others. Figure 7.1 shows a rough visualization of the components of the project and some of the dependencies among them. The major components listed above are grouped by gray boxes. For example, it is possible to design the DAG model and language constructs independently, but these designs might place differing, possibly incompatible, constraints on the cost semantics, and so these components must, to some degree, be developed together. These cyclic dependencies make it difficult to present intermediate results, but in this section, I will sketch plausible directions for proceeding on each of the major components.

One design decision that impacts all components of the thesis is what notion of priority to use. Many options are plausible, from using a constant number of pre-defined priorities, to alowing the programmer to define a partial order between threads. I intend to mostly take an intermediate approach, associating priorities with natural numbers, with 0 being background and higher numbers corresponding to higher priorities. For the set of examples we have considered, this will be general enough. In practice, many systems use a constant number of priorities, and so in certain places where it is necessary for simplicity or efficienty, I may assume that the number of priorities is constant.

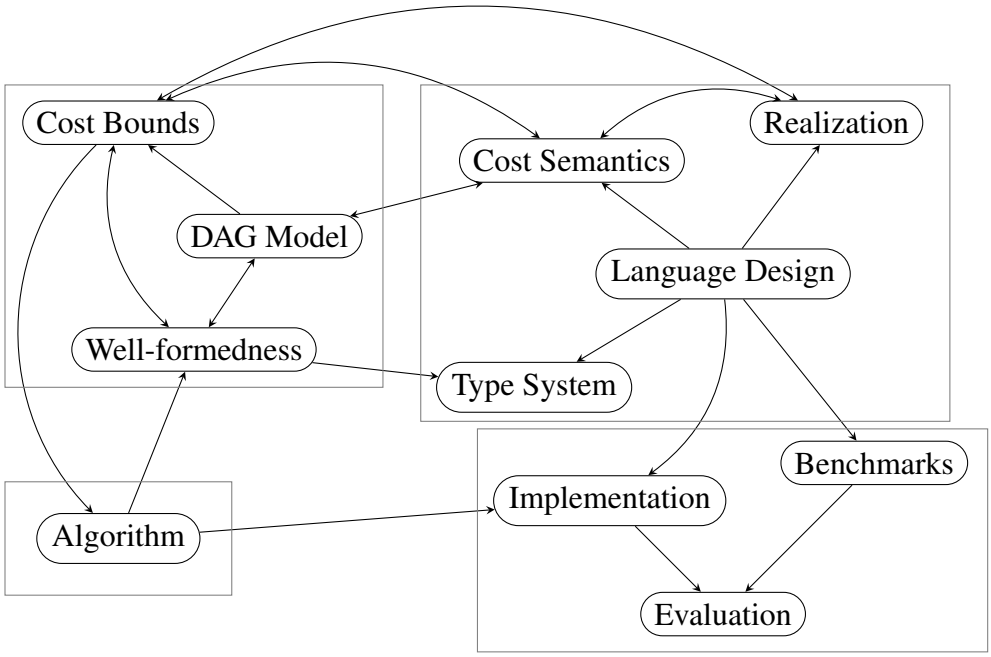


Figure 7.1: The major and minor components of the proposed thesis, with dependencies

7.1 Dag-based Cost model

The main open question in defining the dag model for natural number priorities is how to represent foreground blocks. One simple option would be to leave the representation of foreground blocks unchanged from the PLDI '17 model, but allow foreground blocks to be nested and define the priority of a vertex to be the number of foreground blocks within which it is nested. Other models are possible, but we will explore this particular idea further in this section.

A natural way to extend the cost metrics to this more general model is to consider the response time separately for each priority level. For a given priority level $r > 0$, consider any foreground block at priority r to be in the foreground and all other computation to be in the background, thus "truncating" the priority hierarchy at priority r. If we ignore all other foreground block structure, including nested foreground blocks of priority greater than r, we have reduced the dag to the model of Section 3.2. Thus, the response time R^r at a priority r is the sum of the response times of all of the foreground blocks at priority r. This leads naturally to generalizations of foreground work, span and width: the quantities $W^r(g)$ and $S^r(g)$ simply sum over the work and span of foreground blocks at priority r and $D^r(g)$ is the maximum number of simultaneous foreground blocks of priority r. Note that, since higher-priority foreground blocks must be nested inside lower-priority blocks, $W^r(g)$ includes vertices of priority greater than r. This corresponds to the fact that it must always be acceptable to delay lower-priority work in favor of higher-priority work, and so the response time of foreground blocks at priority r must include work at higher priorities.

The generalization of a prompt schedule is the obvious one: it should first schedule the highest-priority vertices, then the next-highest, and so on. For any fixed priority r, such a sched-

ule will always schedule vertices of priority at least r before it schedules lower-priority vertices, and so when we truncate the priority hierarchy at r, the schedule still meets the definition of a prompt schedule from Section 3.2. With these definitions in place, it should be straightforward to generalize Theorem 3 to higher priority levels.

Conjecture 1. *If a computation represented by a well-formed dag g is scheduled using a prompt schedule on P processors, then for all r, we have*

$$R^r \leq D^r(g)\frac{W^r(g)}{P} + S^r(g)$$

7.2 Language

Most of the proposed work in designing a language for responsive parallelism consists of extending the λ^{ip} language to the more general notion of priorities described above. If time permits, I will also consider adding other features to the language, including polling and cancellation of background threads, as discussed below.

General Priorities The idea explored above of defining priority by nesting foreground blocks works quite well with the language constructs for priority described in Chapter 4: the `fg` keyword increments the priority of an expression and the `bg` keyword spawns an asynchronous computation with its priority decremented. It seems that the type system should generalize naturally, since most prior type systems for staged computation are not limited to two stages (e.g. [15]), and it is common to denote stages by natural numbers. Using this work as an inspiration, hypothetical typing rules for the foreground and background constructs might look like the following.

$$\frac{\Gamma \vdash e : \tau @ n+1}{\Gamma \vdash \texttt{bg}(e) : \bigcirc\, \tau @ n} \qquad \frac{\Gamma \vdash e : \bigcirc\, \tau @ n}{\Gamma \vdash \texttt{fg}(e) : \tau @ n+1}$$

Note that the numbering of the stages in these rules is reversed from the numbering of the priorities we used in the previous section: in the above rules, stage 0 corresponds to the most-foreground priority and computations can be pushed arbitrarily far into the background (while our dag model follows the intuition that stage 0 is the background and computations can be pushed arbitrarily far into the foreground). It is possible this is a superficial issue and the numbering can simply be reversed with no deeper consequences; however, since this would deviate from prior type systems, this should be investigated more carefully. If only a constant number of priorities are allowed, we need not worry about this issue.

More Expressive Background Threads Another language feature that would be useful in a realistic language for responsive parallel computation would be the ability to, without blocking, poll whether a background thread has completed. Since this operation is nonblocking, it would be permissible to perform it in the foreground. For example, this would allow us to rewrite the Fibonacci server of Section 4.1 as follows in order to perform the outputs in the foreground:

```
function print_done threads =
  case threads of
```

```
  | [] -> []
  | th::rest ->
    (case poll th of
     | Done v -> output v; print_done rest
     | NotDone -> th::(print_done rest))

function fib_server threads =
  let n = input () in
    if n < 0 then
      case threads of
      | [] ->  bg ()
      | _ -> fib_server (print_done threads)
    else
      fib_server ((bg (fib n))::(print_done threads))
  end

fg (fib_server ())
```

The server now maintains a list of the currently active background threads. The function `print_done`, which is called in the server loop, prints out the results of any background threads that have completed and returns the ones that are still active. When the user is done entering inputs, the server loops until all of the background threads have completed and their results have been printed.

Implementing polling is fairly straightforward, but polling complicates the cost semantics in that it makes the cost, and result, of program evaluation, dependent on the scheduling decisions made at runtime. Consider the following program fragment:

```
function loop () = loop ()

function race () =
  case poll (bg e) of
  | Done v -> bg v
  | NotDone -> loop ()

fg (race ())
```

Function `race` spawns a background thread. If the runtime executes the background thread before the remainder of `race`, then the pattern match does a small amount of work and then terminates. If, however, the pattern match is executed before the background thread has completed, `race` does not terminate and the work of the program is infinite. While we are not aware of work that has considered the cost semantics of such programs, it is possible that we could handle this dependence on the schedule in the same way that we handle other nondeterminism, such as the latency of blocking operations, by taking the maximum work and span over a family of possible dags.

It might even be useful to allow background threads to return or "post" multiple values at different times before they complete, with the `poll` operation returning the most recently posted

value. For example, a background thread may be running an "anytime algorithm" which can, at any time during computation, produce a partial or inexact answer. If the user, interacting with a foreground thread, demands an answer at a certain time, the foreground thread can poll the most recent answer produced by the computation and return it. The pollable background threads described above can be seen as a special case of these "multi-post" background threads; they return `NotDone` as a default value and post `Done v` when they terminate with value `v`.

Some of our example programs could also be improved by the ability to cancel in-flight background threads. For example, the Convex Hull server benchmark discussed in Section 6.2 spawns a new convex hull computation every time the user clicks to add a new point to the cloud. If the user clicks twice in rapid succession, the first hull might not have been computed before its result was invalidated by the addition of the second point. In the current implementation, both hull computations are allowed to complete and the first is simply overwritten by the second. We could save work if we were able to cancel the first computation when the second point is added.

7.3 Scheduling algorithm

The scheduling algorithm sketched in Section 5.2 admits a natural generalization to more than two priorities. In this algorithm, each processor would have a priority queue of threads and a mailbox to which other processors can send threads. Each processor p would be interrupted periodically and go through the following steps:

1. Make a deal attempt:
   (a) Select a random target processor p'.
   (b) If p has a higher-priority thread than p', deal this thread.
2. Begin working on the highest-priority thread from p's queue, including its mailbox.

Working on threads in priority order poses a technical problem: because the scheduler may "jump around", scheduling threads from different parts of the dag, the standard work stealing deque invariant may not hold. Consider the following sequence of events:

1. Processor p executes a background vertex u_{B1}.
2. Processor p is dealt a foreground vertex u_{F1}, which it then executes.
3. Vertex u_{F1} spawns a foreground vertex u_{F2}, which is scheduled next, and a background vertex u_{B2}, which is placed in the deque.

At the background priority, p's deque now contains u_{B1} and u_{B2}, but since u_{B2}'s parent was stolen from another processor, there is no way to tell which of these is "heavier" and should be dealt next. The analysis of private deques work stealing [2] suggests a possible solution to this issue: instead of using a vertex's depth in the dag in its analysis of steals, it uses the *fork depth*, defined as the minimal number of fork vertices (vertices with out-degree two) in a path from the root to a vertex. It is possible to track the fork depth of vertices at runtime by incrementing a counter whenever a thread forks and storing the appropriate counter values in the deque with the threads. In this way, we can organize the deque by both priority and fork depth, always dealing the shallowest (and therefore heaviest) vertex of the highest priority. Because we are using private

deques and load balancing is handled by message passing, we need not worry about designing and implementing concurrent data structures for this purpose.

While proving execution time and response time bounds on the scheduling algorithm is not intended to be a required component of the thesis, I anticipate that we can extend the private deques analysis to prove that the execution time of our scheduler is within a constant factor of the prompt scheduling bound of $W/P + S$ since load balancing works much the same way in our algorithm as in the non-prioritized one. In a nutshell, the analysis assigns tokens to processors when they are idle and uses a potential argument to show that the idle tokens can be amortized over the span of the computation since the more processors that are idle in a given step, the greater the probability that a deal will occur if one is possible. As in most analyses of work stealing, this argument relies on the fact that, when a deal happens, enough work is transferred (because the "heaviest" vertex is dealt) to amortize the cost of the deal. In order to maintain this property when we are only dealing high priority vertices (which may be small pieces of work), we may have to alter the algorithm to, for example, deal half of the highest-priority threads.

A similar argument can likely be used to bound response time, by assigning a potential to each foreground block and collecting "response tokens" for each ready foreground block each step it is active. When all processors are busy with high-priority work, the response tokens can be charged to the high-priority work. When processors are not working on high-priority work, we can charge the response tokens to the potential of the foreground block, which is likely to decrease when its vertices are dealt to other processors.

7.4 Implementation, Benchmarks and Evaluation

For my final implementation, I intend to build on top of the mlton-parmem compiler, an updated and extended version of Spoonhower's MLton currently being developed in Umut Acar's research group. Based on recent results [30], it appears that this will be a robust compiler and runtime to use as a substrate on which to build. A robust, efficient private deques scheduler for fork-join parallelism and futures is currently being developed for mlton-parmem, and this scheduler would be a good starting point for the implementation of the scheduling algorithm described in the previous section (using futures to implement the foreground and background constructs).

Before the final evaluation, I hope to expand the benchmark suite of parallel, interactive programs to include more examples, including at least one large example. The large example might take the form of a real-time computer game, such as the one described in the introduction, which performs responsive interaction with one or more players while computing the strategy for an AI player in the background. Such a program would be a good test case for the "multi-post" background threads proposed in Section 7.2, since this would allow the AI process to continually expand its search depth, returning the best move found when the AI move is required. Since part of my thesis is that parallel interactive programs will become an important new domain, having a benchmark suite that others can use to evaluate future languages and schedulers could be of independent interest.

Finally, I intend to perform a thorough performance evaluation on the implementation, using the full suite of benchmarks. This will likely require extending our evaluation framework to be able to evaluate the responsiveness of complex programs like a game, the photo viewer or the

music server. For this, we could draw inspiration and measurement techniques from existing tools and techniques for quantifying user experience metrics in domains such as web servers.

7.5 Timeline

As discussed at the beginning of this chapter, the components of the thesis are interdependent and it is difficult to fix an order in which to complete them. The following timeline is thus subject to change.

June-August 2017	Finish designing a scheduling algorithm (Section 7.3), but do not yet prove bounds on it. Implement the proposed algorithm, and develop a complete benchmark suite and robust evaluation framework (Section 7.4) in order to do a thorough evaluation. Aim to submit to PPoPP or PLDI.
September-November 2017	Finish the above if it is not yet complete, in conjunction with fleshing out the design of the language features and type system (Section 7.2) and the cost model (Section 7.1).
December 2017-February 2018	If the above is complete, begin proving run-time and responsiveness bounds on the scheduling algorithm, and aim to submit to SPAA.
March-May 2018	Write thesis.

Chapter 8

Conclusion

In the proposed thesis, I aim to take a step toward unifying the paradigms of cooperative and competitive threading. The goal is for the resulting programming model to allow programmers to easily use threads to leverage multiprocessor hardware to complete computational tasks quickly, and also express interactive elements such as I/O. As part of this model, I will develop cost models that allow programmers to reason abstractly about the execution time and responsiveness properties of their code, and will show, using type systems as necessary to enforce any required well-formedness properties, that the abstract cost bounds derived from the models can be realized in practice. I also hope to have a robust implementation which has been thoroughly evaluated using a complete, realistic benchmark suite.

Bibliography

[1] The open group base specifications issue 7. IEEE Std. 1003.1-2008, 2016. 2.2

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *PPoPP '13*, 2013. 2.1.3, 5.2, 7.3

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001. 2.1.3, 5.1

[4] Arvind and K. P. Gostelow. The Id report: An asychronous language and computing machine. Technical Report TR-114, Department of Information and Computer Science, University of California, Irvine, September 1978. 2.1

[5] Andrew D. Birrell. An introduction to programming with threads. In Greg Nelson, editor, *Systems Programming with Modula-3*, pages 88–118. Prentice Hall, Upper Saddle River, NJ, 1991. 2.2

[6] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*, pages 213–225. ACM, 1996. 2.1.2

[7] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21(1):4–14, 1994. 2.1

[8] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998. 2.1.2

[9] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999. 2.1.3

[10] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2): 201–206, 1974. 2.1.2

[11] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Functional Programming Languages and Computer Architecture (FPCA '81)*, pages 187–194. ACM Press, October 1981. 2.1.3

[12] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: a status report. In *Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP 2007, Nice, France, January 16, 2007*, pages 10–18, 2007. 2.1

[13] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra,

Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538. ACM, 2005. 2.1

[14] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28, 2005. ISBN 1-58113-986-1. 2.1.3

[15] Rowan Davies. A temporal-logic approach to binding-time analysis. In *LICS*, pages 184–195, 1996. 4.1, 7.2

[16] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*, 48 (3):555–604, 2001. 4.1

[17] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computing*, 38(3):408–423, 1989. 2.1.2

[18] Nicolas Feltman, Carlo Angiuli, Umut A. Acar, and Kayvon Fatahalian. Automatically splitting a two-stage lambda calculus. In *Proceedings of the 25th European Symposium on Programming*, ESOP '16, Eindhoven, The Netherlands, 2016. Springer-Verlag. 4.1

[19] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5-6):1–40, 2011. 2.1

[20] Robert H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, 1985. 2.1, 2.1.3

[21] Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, and Mark Weiser. Using threads in interactive systems: A case study. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 94–105, New York, NY, USA, 1993. ACM. 2.2

[22] Shams Imam and Vivek Sarkar. Load balancing prioritized tasks via work-stealing. In *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing*, pages 222–234, 2015. 2.1.3

[23] Shams Mahmood Imam and Vivek Sarkar. Habanero-Java library: a Java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14*, pages 75–86, 2014. 2.1

[24] Suresh Jagannathan, Armand Navabi, KC Sivaramakrishnan, and Lukasz Ziarek. The design rationale for Multi-MLton. In *ML '10: Proceedings of the ACM SIGPLAN Workshop on ML*. ACM, 2010. 2.1

[25] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, 2010. 2.1

[26] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, 2000. 2.1

[27] Stefan K. Muller and Umut A. Acar. Latency-hiding work stealing: Scheduling interacting parallel computations with work stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016*, pages 71–82, 2016. 3

[28] Stefan K. Muller, Umut A. Acar, and Robert Harper. Responsive parallel computation: Bridging competitive and cooperative threading. In *ACM Sigplan Conference on Programming Language Design and Implementation*, PLDI '17 (To appear), 2017. 3

[29] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: an unbalanced tree search benchmark. In *Languages and Compilers for Parallel Computing, 19th International Workshop*, LCPC '06, 2006. 6.2

[30] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. Hierarchical memory management for parallel programs. In *ACM International Conference on Functional Programming (ICFP)*, 2016. 7.4

[31] Mads Rosendahl. Automatic complexity analysis. In *FPCA '89: Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, 1989. 2.1.2

[32] David Sands. Complexity analysis for a lazy higher-order language. In *ESOP '90: Proceedings of the 3rd European Symposium on Programming*, pages 361–376, London, UK, 1990. Springer-Verlag. 2.1.2

[33] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architecture*, SPAA '12, 2012. 6.2

[34] David Canfield Smith, Charles Irby, Ralph Kimball, Bill Verplank, and Eric Harslem. Designing the STAR user interface. *BYTE Magazine*, 7(4):242–282, 1982. 2.2

[35] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *International Conference on Functional Programming*, 2008. 2.1.2, 4.2, 6

[36] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A structural view of the Cedar programming environment. *ACM Trans. Program. Lang. Syst.*, 8(4):419–490, August 1986. 2.2

[37] J.D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975. 2.1.2

[38] Martin Wimmer, Daniel Cederman, Jesper Larsson Träff, and Philippas Tsigas. Work-stealing with configurable scheduling strategies. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 315–316, 2013. 2.1.3

[39] Martin Wimmer, Francesco Versaci, Jesper Larsson Träff, Daniel Cederman, and Philippas Tsigas. Data structures for task-based priority scheduling. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 379–380, 2014. 2.1.3