A Scheduling Framework for General-purpose Parallel Languages

Matthew Fluet
Toyota Technological Institute at Chicago
fluet@tti-c.org

Mike Rainey John Reppy
University of Chicago
{mrainey,jhr}@cs.uchicago.edu

Abstract

The trend in microprocessor design toward multicore and manycore processors means that future performance gains in software will largely come from harnessing parallelism. To realize such gains, we need languages and implementations that can enable parallelism at many different levels. For example, an application might use both explicit threads to implement course-grain parallelism for independent tasks and implicit threads for fine-grain data-parallel computation over a large array. An important aspect of this requirement is supporting a wide range of different scheduling mechanisms for parallel computation.

In this paper, we describe the scheduling framework that we have designed and implemented for Manticore, a strict parallel functional language. We take a micro-kernel approach in our design: the compiler and runtime support a small collection of scheduling primitives upon which complex scheduling policies can be implemented. This framework is extremely flexible and can support a wide range of different scheduling policies. It also supports the nesting of schedulers, which is key to both supporting multiple scheduling policies in the same application and to hierarchies of speculative parallel computations.

In addition to describing our framework, we also illustrate its expressiveness with several popular scheduling techniques. We present a (mostly) modular approach to extending our schedulers to support cancellation. This mechanism is essential for implementing eager and speculative parallelism. We finally evaluate our framework with a series of benchmarks and an analysis.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages; D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Languages, Performance

Keywords heterogeneous parallel languages, scheduling, compilers, and run-time systems

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

1. Introduction

The laws of physics and the limitations of instruction-level parallelism have forced microprocessor architects to develop new multicore and manycore processor designs. This trend means that future performance gains in software will largely come from harnessing parallelism. Thus we have an urgent need to develop effective programming languages and runtime systems that can enable parallelism for a wide range of applications. The Manticore project is an effort to address this need using strict functional programming as its foundation.

In approaching the problem of parallel computation, our main thesis is that the language and runtime system must support parallelism at a wide range of granularities. The Manticore language design achieves this goal by providing both fine-grain parallelism in the form of nested-data parallelism, medium-grain parallelism in the form of parallel tuples and bindings, and course-grain parallelism in the form of CML-style concurrency primitives [Rep99, Sha07].

While high-level language designs for heterogeneous parallelism are crucial for making parallel programming accessible to programmers, it is nonetheless only one piece of the story. In this paper, we focus on a complementary piece: the design and implementation of a *runtime-system framework*, capable of handling the disparate demands of the various heterogeneous parallelism mechanisms exposed by a high-level language design and capable of supporting a diverse mix of scheduling policies. This runtime framework provides a foundation for rapidly experimenting with both existing parallelism mechanisms and additional mechanisms not yet incorporated into high-level language designs for heterogeneous parallelism.

Our runtime framework consists of a combination of runtime-system and compiler features. It supports a small core of primitive scheduling mechanisms, such as virtual processors, preemption, and computation migration. Our design favors minimal, lightweight representations for computational tasks, borrowing from past work on *continuations*. On top of this substrate, a language implementor can build a wide range of parallelism mechanisms with complex scheduling policies [VR88, BL99, MKH90].

The contributions of this paper are

- We present the design of our scheduling framework, which provides support for mixing different scheduler policies.
- We describe how this scheduling framework can support a variety of different scheduling strategies that have been proposed in the parallel computing literature.
- Thread cancelation is an important mechanism for supporting speculative parallelism. We describe a modular cancelation mechanism using our framework that can be combined other scheduling strategies.

- Our scheduler framework is implemented as part of the Manticore compiler and runtime and we describe various interesting aspects of the implementation.
- We have implemented several scheduling strategies in our system and we report on their performance on a number of benchmark loads.

Although space precludes the presentation of a formal model of this scheduling framework, one has been developed and evaluated using the Redex system [Rai07b].

The remainder of the paper is organized as follows. In Section 2, we give an overview of the Manticore system, covering the compiler technology and runtime-system features that are relevant to this paper. We then introduce our scheduling framework and show how a basic round-robin scheduler is implemented. In Section 4, we describe a more complicated scheduler that supports data parallelism via futures. We then describe a more sophisticated scheduler that implements the Cilk-5 approach to work stealing [FLR98]. In Section 6, we present a modular approach to supporting fiber cancelation in schedulers. This mechanism is crucial for supporting the eager and speculative parallelism mechanisms in Manticore. We then present an empirical evaluation of our framework, covering both a small set of standard parallel benchmarks and a more detailed analysis of the overheads in work stealing and cancelation. We discuss related work in Section 8 and conclude in Section 9.

2. The Manticore system

The main focus of this paper is our framework for implementing schedulers for parallel computation. These schedulers are written in an external representation of our compiler's intermediate representation and loaded at compile time. In this section, we describe the basic architecture of the Manticore system in order to provide a context for the more detailed technical discussions in the remainder of the paper.

2.1 The Manticore compiler

As is typical, our compiler is structured as a sequence of intermediate languages/representations (ILs/IRs) [FFR+07]. For this paper, we are only interested in our normalized direct-style representation called BOM, which is the workhorse of our compiler. In this paper, we use SML syntax to write BOM code, since it is more compact than the actual syntax. An important feature of BOM is what we call *high-level operations* (HLOps). These operations are used to embed higher-level abstractions, such as concurrency and scheduler operations in the BOM representation. Over the course of BOM optimization, the high-level operations are replaced by their definitions, which are loaded from external files (one can think of HLOps as similar to hygienic macros).[1] The BOM IR also includes atomic operations, such as *compare-and-swap* (**cas**).

Another important feature of the BOM IR is the inclusion of first-class continuations, which are a well-known language-level mechanism for expressing concurrency [Wan80, HFW84, Rep89, Ram90, Shi97]. Continuations are supported in the BOM IR by the **cont** binding form for introducing continuations and the **throw** expression form for applying continuations. The **cont** binding:

$$\texttt{let cont } k\ arg = exp\ \texttt{in } body\ \texttt{end}$$

binds k to the first-class continuation

$$\lambda arg.(\texttt{throw}\ k'\ (exp))$$

where k' is the continuation of the whole expression. The scope of k includes both the expression $body$ and the expression exp (*i.e.*, k may be recursive). Continuations have indefinite extent and may be used multiple times.[2]

A couple of examples will help illustrate this mechanism. The traditional `callcc` function can be defined as

```
fun callcc f = let cont k x = x in f k end
```

Here we use the **cont** binding to reify `callcc`'s return continuation. The **cont** binding form is more convenient than `callcc`, since it allows us to avoid the need to nest `callcc`'s in many places. For example, we can create a fiber (unit continuation) from a function as follows:

```
fun fiber (f) = let
    cont k () = ( f(); stop() )
    in k end
```

where `stop` (defined in Section 3) returns control to the scheduler.

2.2 Process abstractions

Our runtime model is based on three distinct notions of process abstraction:

1. *Fibers* are unadorned threads of (sequential) control. A suspended fiber is represented as a unit continuation.
2. *Threads* correspond to the explicit threads of the Manticore language. A thread may consist of multiple fibers running in parallel.
3. *Virtual processors* (vprocs) are an abstraction of a hardware processor. Each vproc has local state, including a local heap and scheduling queue. A vproc runs at most one fiber at a time, and, furthermore, is the only means of running fibers. The vproc that is currently running a fiber is called the *host vproc* of the fiber, and can be obtained obtained by the `host` operator.

2.3 The runtime system

Our runtime system is implemented in C with a small amount of assembly-code glue between the runtime and generated code.

Vprocs Each vproc is hosted by its own POSIX thread (pthread). Typically, there is one vproc for each physical processor core. We use the Linux processor affinity extension to bind pthreads to distinct processors. For each vproc, we allocate a local memory region of size 2^k bytes aligned on a 2^k-byte boundary (currently, $k = 20$ and fixed at compile-time). The runtime representation of a vproc is stored at the base of this memory region and the remaining space is used as the vproc-local heap.

One important design principle that we follow is minimizing the sharing of mutable state between vprocs. By doing so, we minimize the amount of expensive synchronization needed to safely read and write mutable state by multiple vprocs. This improves the parallel performance of a program, because each vproc spends the majority of time executing without needing to coordinate with other vprocs. Secondary effects, such as avoiding cache updates and evictions on the physical processors due to memory writes, further improve the parallel performance and are enabled by minimizing the sharing of mutable state.

We distinguish between three types of vproc state: fiber-local state, which is local to each individual computation; vproc-local state, which is only accessed by code running on the vproc; and global state, which is accessed by other vprocs. The thread-atomic

[1] Our loading infrastructure supports a syntax that is close to what we are using in this paper. After parsing, the loaded code is converted to the BOM representation using the standard normalization algorithm [FSDF93].

[2] The syntax of our continuation mechanism is taken from the Moby compiler's BOL IR [Rep02], but our continuations are first-class, whereas BOL's continuations are a restricted form of one-shot continuations known as *escaping continuations*.

state, such as machine registers, is protected by limiting context switches to "safe-points" (*i.e.*, heap-limit checks).

Fiber-local storage Our system supports dynamically-bound per-fiber storage. This storage is used to provide access to scheduler data structures, thread IDs, and other per-fiber information. Each vproc has a pointer to keep track of the current fiber-local storage (FLS). The following operations are used to allocate FLS and to get and set the host vproc's current FLS pointer.

```
type fls
val newFls : unit -> fls
val setFls : fls -> unit
val getFls : unit -> fls
```

To support flexibility, we provide an association-list-style mechanism for accessing the attributes in FLS:

```
val getFromFls : (fls * 'a tag) -> 'a option ref
```

In keeping with our use of SML syntax, we use phantom types to type these operations. The tag values that are used as lookup keys are globally defined.

Garbage collection Automatic memory management is provided by a garbage collector based on the approach of Doligez, Leroy, and Gonthier [DL93, DG94]. The heap is organized into a fixed-size local heap for each vproc and a shared global heap. The global heap is simply a collection of chunks of memory, each of which may contain many heap objects. Each vproc has a dedicated chunk of memory in the global heap. Heap objects consist of one or more pointer-sized words with a pointer-sized header. The heap has the invariant that there are no pointers into local heap from either the global heap or another vproc's local heap. This invariant allows a vproc to collect its local heap completely independently from the other vprocs. As noted above, avoiding expensive synchronizations with other vprocs improves the parallel performance of a program. Synchronization is only required when a vproc's dedicated chunk of memory is exhausted, and a fresh chunk of memory needs to be allocated and added to the global heap.

Enforcing the heap invariant requires *promoting* objects that might become visible to other vprocs to the global heap. For example, if a thread is going to send a message, then the message must be promoted first, since the receiver may be running on a remote vproc. When objects are promoted, the reachable local data is copied to the global heap, but forward pointers are left so that sharing is reestablished when future promotions or collections encounter pointers to the promoted objects.

Preemption We implement preemption by synchronizing preempt signals with garbage-collection tests as is done in SML/NJ [Rep90]. We dedicate a pthread to periodically send `SIGUSR2` signals to the vproc pthreads. Each vproc has a signal handler that sets the heap-limit register to zero, which causes the next heap-limit check to fail and the garbage collector to be invoked. At that point, the computation is in a safe state, which we capture as a continuation value that is passed to the current scheduler on the vproc. Signals can be masked locally on a vproc, which we do to avoid preemption while holding a spin lock.

Scheduling queues Each vproc maintains a scheduling queue. This queue is actually split into a locally accessible queue that has no synchronization overhead and a globally accessible queue that is protected by a mutex lock. As part of handling preemption, the vproc moves any threads in the global queue into the vproc's local queue. We provide the following operations for operating on a vproc's scheduling queue:

```
val enq : fiber -> unit
val deq : unit -> fiber
val enqOnVP : (vproc * fiber) -> unit
```

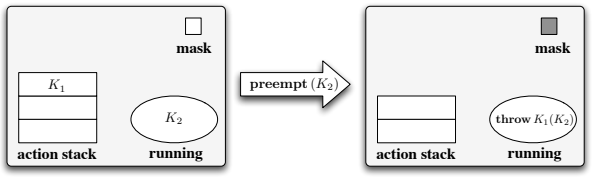

Figure 1. The effect of preemption on a vproc

The first two operations work on the host vproc's local queue and only require that signals be masked locally. The `enqOnVP` operation enqueues a fiber on a remote vproc. Note that there is no way to dequeue a thread from a remote vproc. This asymmetric design complicates the implementation of our load-balancing thread scheduler slightly (see Section 3.3.3), but makes local queue operations (the common case) significantly faster. Also note that, because the enqueued fiber is made visible to another vproc, `enqOnVP` requires the fiber to have been promoted.

3. The scheduling framework

Our scheduling framework takes the philosophy of the micro-kernel architectures for operating systems; we provide a minimum collection of compiler and runtime-system mechanisms to support light-weight scheduling and then build more complex abstractions and the scheduling code on top of that framework. In this section, we describe the abstractions provided by the runtime system and give an informal description of the scheduler operations with some simple examples. As our present work focuses on the design and implementation of a flexible scheduling substrate, we explicitly leave questions of if and how end-programmers can write their own schedulers and how end-programmers express scheduling policies to future work. These are interesting and important questions, but beyond the scope of this paper.

3.1 Scheduling operations

A scheduler action is a continuation that takes a signal and performs the appropriate scheduling activity in response to the signal. At a minimum, we need two signals: `STOP` that signals the termination of the current fiber and `PREEMPT` that is used to asynchronously preempt the current fiber. When the runtime system preempts a fiber it reifies the fiber's state as a continuation that is carried by the preempt signal.

Each vproc has its own stack of scheduler actions. The top of a vproc's stack is called the *current* scheduler action. When a vproc receives a signal, it handles it by setting the signal mask, popping the current action from the stack, and throwing the signal to the current action. The operation is illustrated in Figure 1; here we use dark grey in the mask box to denote when signals are masked.

There are two operations that scheduling code can use to affect a vproc's scheduler stack directly. The operation `run(`k_1`,`k_2`)` pushes the action k_1 onto the host vproc's action stack, clears the vproc's signal mask, and throws to the continuation k_2 (see Figure 2). The `run` operation requires that signals be masked, since it manipulates the vproc's action stack. The other operation is `forward(`sgn`)`, which sets the signal mask and forwards the signal sgn to the current action (see Figure 3). The `forward` operation is used both in scheduling code to propagate signals up the stack of actions and in user code to signal termination, which means that signals may, or may not, be masked when it is executed. For example, a fiber exit function can be defined as

```
fun stop () = forward(STOP)
```

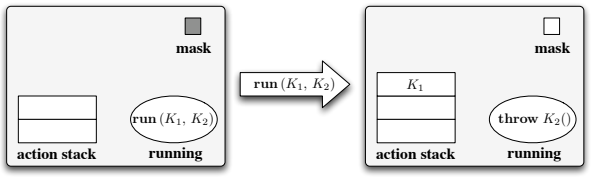

Figure 2. The effect of `run(`k_1`,`k_2`)` on a vproc

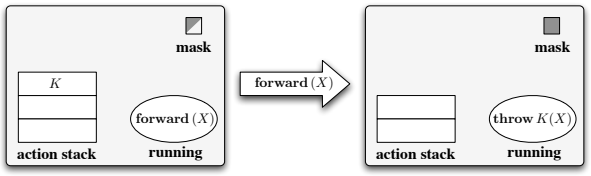

Figure 3. The effect of `forward(`sgn`)` on a vproc

Another example is the implementation of a yield operation that causes the current fiber to yield control of the processor, by forwarding its own continuation:

```
fun preempt (k) = forward(PREEMPT(k))
fun yield () = let
    cont k (x) = x
    in preempt(k) end
```

The last part of our framework is the operations used to map a parallel computation across multiple vprocs. We have already introduced the `enqOnVP` operation to enqueue a fiber on a specific vproc. Using this operation we can implement an explicit migration function that moves the calling computation to a specific vproc:

```
fun migrateTo vp = let
    val fls = getFls()
    cont k (x) = ( setFls(fls); x )
    in
       enqOnVP(vp, k);
       stop()
    end
```

Note that the migrated computation takes its fiber-local storage.

We also provide a mechanism for assigning vprocs to computations. A parallel computation is a group of fibers running on separate vprocs; the scheduling framework uses FLS to distinguish between different parallel computations. Computations request additional vprocs by using the `provision` operation; this operation either returns `SOME vp`, where `vp` is a vproc that is not already assigned to the computation, or returns `NONE`, indicating that no additional vprocs are available for the computation. To balance workload evenly between threads, the runtime system never assigns a vproc to a given group twice and attempts to balance the number of groups assigned to each vproc. When a computation is finished with a vproc, it uses the `release` operation to alert the runtime that it is done with the vproc.

3.2 Scheduler utility functions

On top of the basic scheduling framework described above, we implement a few operations that are common to many schedulers. For example, we have a function for passing preemptions up the action stack:

```
fun atomicYield () = ( yield (); mask() )
```

Note that this function will remask signals when it resumes.

Some of our scheduling code makes use of concurrent queues [MS96], which have the interface below.

```
type 'a queue
val emptyQ : unit -> 'a queue
val addQ   : ('a queue * 'a) -> unit
val remQ   : 'a queue -> 'a option
```

3.3 Language-level threads

Threads in Manticore play the dual role of supporting systems programming and coarse-grained parallelism [FFR+07]. Our threading mechanism is based on a parallel implementation of CML threads with message passing [Rep99, RX07, RX08]. In this section, we briefly describe our scheduling support for threads, including thread creation, uniprocessor scheduling and load-balancing. Our intention is to introduce our basic process mechanisms and thus set the stage for our more sophisticated parallel schedulers in Sections 4–6.

3.3.1 Creating threads

The spawn operation is easy to implement using our fiber-creation facility. We create a fiber for the thread's body, and place the fiber on the local queue of the host vproc.

```
fun spawn (f) =
    enq(fiber(fn () => (setFls(newFls());
                        f())))
```

Note that we provide each fiber with its own FLS, which is installed on the host vproc when the fiber is run.

3.3.2 Uniprocessor scheduling

With our primitives in hand, we can implement a simple round-robin scheduling policy for threads. The scheduler action below embodies this policy. At runtime, each vproc executes its own instance of this scheduler.

```
cont roundRobin (STOP) = dispatch()
   | roundRobin (PREEMPT k) = let
     val fls = getFls()
     cont k' () = (
          setFls(fls);
          throw k () )
     in
        enq k';
        dispatch()
     end
cont dispatch () = run(roundRobin, deq())
```

We have also tested proportional-share policies using engines [HF84] and nested engines [DH89]. Nested engines, although somewhat complicated to implement with first-class continuations alone, have a compact implementation using operators in our runtime framework [Rai07a, Rai07b]. Empirical studies on scheduling behavior of CML-like threads also exist [HJT+93, Rep99].

3.3.3 Load balancing

We conclude this section by briefly describing how we extend our round-robin scheduler with load balancing. The design of this load-balancing scheduler is based on the following observation of typical CML programs. Often, these programs consist of a large number of threads (possibly thousands), but the bulk of these threads are *reactive* (*i.e.*, they spend most of their life waiting for messages). In such a system, only a handful of threads carry enough computational load to benefit from migration. Thus, we have optimized for fast context switching between local threads and have made thread migration bear the synchronization costs. Our implementation of vproc queues reflects this design, as we have no operations for dequeing from remote vprocs.

Our load-balancing scheduler performs thread migration by using the following protocol. When vprocs go idle, they periodically spawn *thief* threads on other vprocs. Thieves (executing on the remote vproc) can safely inspect the remote vproc queues, and if they

observe sufficient load, then they can migrate threads back to the idle vproc. One complication of this protocol, of course, is that if we are not careful, thieves could end up migrating other thieves! We avoid this situation, and similar situations that arise with our schedulers for implicit parallelism, by pinning a thread to a processor. We use fiber-local storage to mark a thread as pinned.

Although we have not performed an empirical evaluation of this scheduler, we have benefited by testing it on synthetic workloads. Since it makes extensive use of our scheduling operations, the scheduler has proved to be a good stress test for the compiler and runtime code. In the future, we plan to incorporate ideas from the growing body of research on OS-level scheduling for multicore processors into our thread-scheduling approach [Fed07].

4. Gang scheduling

In this section, we describe a simple scheduler for data parallelism that is based on futures [Hal84] with gang scheduling. Owing to space limitations, we do not cover our data-parallel array structure — including how arrays are translated into runtime primitives — but instead point the reader to Shaw's Master's paper [Sha07] and a companion paper appearing in these proceedings [FRRS08]. One important aspect, worth mentioning here, however, is that our data-parallel arrays have a specialized tree representation that limits the available parallelism, which in turn, sufficiently coarsens the granularity of the computation so that sophisticated balancing is not necessary. Therefore, we have opted for a simple gang scheduler until we gauge which techniques would be more effective.

For the purposes of our data-parallel arrays, we can use a restricted form of future [Hal84] that we call *one-touch futures*. By fixing the number of fibers touching a future to one, we can utilize a lighter-weight synchronization protocol than in the general case. As is common, our future cell, represented by the `future1` type, contains the current state and the thunk for evaluating the future. We provide operations for creating a future and for obtaining the value of a future.

```
type 'a future1
val future1 : (unit -> 'a) -> 'a future1
val touch1 : 'a future1 -> 'a
```

There are only two opportunities for evaluating a future. If there is insufficient parallelism, then the fiber that originally created the future touches (and evaluates) it using the `touch1` function. Otherwise, an instance of the gang scheduler steals the future using the `future1StealAndEval` function, which evaluates it in parallel.

```
val future1StealAndEval : 'a future1 -> unit
```

The primary difference between touching and stealing a future is that the former blocks if the future is being evaluated (after having been stolen), while the latter is a nop if the future is being evaluated (after having been touched). We elide further discussion of the synchronization protocol, as it is mostly straightforward and irrelevant to our scheduling code.

The entry point for our gang scheduler is the `future1` operation. As can be seen below, it initializes the future cell and adds it to the scheduler's ready queue.

```
fun future1 (thunk) = let
      val fut = newFuture1Cell(thunk)
      in
        addQ (getReadyQueue(),
          fiber (fn () => future1StealAndEval(fut)));
        fut
      end
```

Notice that what we have put on the ready queue is actually a *suspended fiber* for evaluating the future. As we build more advanced scheduling features, the advantage of using this uniform representation will become clear. The operation for getting the ready queue is shown the code below; it returns the queue by querying FLS.

```
fun getReadyQueue () = (
    case !(getFromFls(getFls(), tag(futRdyQ)))
     of NONE => futuresScheduler()
      | SOME gq => gq
    (* end case *))
```

If the queue is not available, then the scheduler needs to be initialized.

```
fun futuresScheduler () = let
    val gq = initGangQueue()
    val fls = getFls()
    cont gsAction (sgn) = let
         cont dispatch () = (case remQ(gq)
               of NONE => (
                  atomicYield();
                  throw dispatch () )
                | SOME k => (
                  setFls(fls);
                  run(gsAction, k) )
              (* end case *))
         in
           case sgn
            of STOP => throw dispatch ()
             | PREEMPT k => (
               addQ(gq, k);
               atomicYield();
               throw dispatch () )
           (* end case *)
         end
    in
       schedulerStartup(gsAction);
       gq
    end
```

Figure 4. The gang scheduler.

Figure 4 shows our initialization and scheduling code, which follows the gang policy. Workers obtain work from the single, shared queue `gq`. The scheduler action `gsAction` embodies a parallel instance of our gang scheduler. This action loops indefinitely, stealing futures from the ready queue. When none are available, the scheduler yields to its parent scheduler, thus giving other schedulers a chance to run. Similarly, when the evaluation of a stolen future is preempted, the gang scheduler returns the in-progress future evaluation to the ready queue (to be stolen by another instance of the scheduler) and yields to its parent scheduler.

Related work on supporting data-parallel arrays in the Data Parallel Haskell, however, has noted deficiencies of the gang policy. Specifically, on NUMA machines, poor data locality can become an issue, degrading performance for memory-intensive computations [CLP+07].

The only remaining piece of this implementation is the code responsible for initializing the scheduler on some collection of vprocs. This operation is important for heterogeneous programs, as deciding how many vprocs should be available to a scheduler affects other running schedulers. Many alternatives are available, including a wide variety of job scheduling techniques [Fei94]. We have prototyped a job scheduler for our framework, but have yet to evaluate its performance. We expect that finding efficient and reliable job scheduling policies will be a significant focus of our future work. In our current implementation, however, this function just spawns the given scheduler on all vprocs in the system.

5. Work stealing

Work stealing is a scheduling policy that has proven effective for a wide range of computations and has been a key component of several parallel systems [BS81, Hal84, MKH90, CHRR95, FLR98]. In addition to performing well empirically, work stealing enjoys a solid theoretical basis [BL99]. Recent languages that support work stealing, such as Cilk and JCilk, have benefited from using this theory to guide their major design choices [FLR98, DLL06]. In this section, we describe a work-stealing scheduler using our framework, which, like Cilk and JCilk, is informed by the theory.

We begin by describing how work stealing computations are expressed in the Manticore language. The classic tree-add example below marks the left recursive call as a parallel computation.

```
fun treeAdd LF = 0
  | treeAdd (ND (x, t1, t2)) = let
    pval l = treeAdd t1
    val  r = treeAdd t2
    in
       l + r + x
    end
```

The left call is evaluated locally, but it's continuation (the right recursive call and body) are placed on the work-stealing queue. The **pval** keyword can be read as specifying a *parallel* binding.[3] Because of the high degree of potential parallelism in tree add, finding an efficient schedule hinges on finding a coarse distribution of the computation among the processors. Unsurprisingly, the same rule applies for other more realistic algorithms.

The work-stealing algorithm finds such efficient schedules dynamically. The algorithm operates as follows. Each processor owns a doubly-ended queue, or *deque*, of fibers. Each of these deques has head and tail pointers. A processor pushes and pops fibers from the tail of its deque in the same way that a C program pushes and pops frames from its stack. When a processor runs out of work, it picks another processor's deque uniformly at random and attempts to steal from the head of that deque. These workers are called the *thief* and the *victim* respectively.

The theory behind work stealing is based on abstract measures of *work*, which is the time necessary to evaluate a computation sequentially, and *depth*, which is the time to evaluate a computation on an infinite number of processors. Implications of this theory have led Frigo *et al.* to propose the *work-first principle*, which states that one should seek to minimize scheduling overheads placed on the work of a computation [FLR98]. Furthermore, scheduling overheads placed on the depth of a computation, which become manifest as overheads of steals, are less important. Inspired by this principle is a compiler transformation that splits a computation into a sequential fast path and a parallel slow path. The computation spends most of its time in the fast path, and switches to the slow path only when a steal occurs. Also inspired by this principle is a light-weight synchronization protocol for deques, which transfers most of the synchronization cost to remote accesses.

We now describe our implementation of work stealing, which incorporates the aforementioned ideas from Cilk-5. We first describe our scheduling primitives and then show our compiler transformation that elaborates **pval**s into these primitives. Below are operations for pushing and popping from the local deque. As with `future1`, our `wsPush` operation is responsible for initializing the scheduler. For our purposes, the pop operation only needs to return a boolean to indicate whether the deque is empty.

[3] Note that the implementation of **pval** that we describe in this paper differs from the version that it is used in the standard Manticore implementation [FFR+07, FRRS08]. We are "hijacking" the syntax to make it easy to experiment with different scheduler frameworks, but the standard implementation uses the gang scheduler described in the previous section.

```
val wsPush : fiber -> unit
val wsPop  : unit -> bool
```

Our synchronization primitives are ivars [ANP89] specialized for our scheduler. In particular, the put operation unblocks waiting processes by putting them on the local deque.

```
type 'a ws_ivar
val wsIVar : unit -> 'a ws_ivar
val wsIGet : 'a ws_ivar -> 'a
val wsIPut : ('a ws_ivar * 'a) -> unit
```

Our underlying deque implementation consists of the operations below.

```
type 'a deque
val makeDeque : unit -> 'a deque
val pushTl : ('a deque * 'a) -> unit
val popTl  : 'a deque -> 'a option
val popHd  : 'a deque -> 'a option
```

Our implementation follows the light-weight synchronization protocol outlined in the Cilk-5 implementation [FLR98].

The interesting part of our scheduling code is the `steal` operation, which attempts to steal from another worker. By temporarily yielding before the steal attempt, our scheduler can evenly distribute load among other schedulers. This technique has been shown to perform well in multiprogrammed workloads both theoretically and empirically [ABP98, BP98]. The remainder of the steal operation is as described earlier; the thief worker picks a victim at random, and attempts to steal a fiber off the victim's deque.

```
cont steal () = (
    atomicYield();
    case popHd(Array.sub(workerDeques,
                         randInt(rand)))
      of NONE => throw steal ()
       | SOME k => throw dispatch k
    (* end case*))
```

As with the gang scheduler, we use a scheduler action to embody parallel instances of the workers. The scheduler handles fiber termination by trying to pop from the local deque. If nothing is available, the scheduler attempts to steal. The scheduler handles preemptions by putting the currently evaluating fiber back on the local queue and yielding temporarily. This process allows other workers to steal the fiber during the preemption and is similar to the *mugging* protocol described by Blumofe et. al. [BLS98].

Figure 5 shows the translation from `pval`s into our runtime primitives. This translation is straightforward; we describe it here, however, to set the stage for our treatment of exceptions and cancelation in Section 6. The majority of our translation deals with splitting the `let` expression into its fast and slow paths. To construct these paths, we move the body of the `let` into the fresh function `bodyFn`. The parameter of this function is a place holder for `x`.

The translated computation pushes the fiber for the slow path onto the local deque, evaluates e_1, and then determines if it can take the fast path. Due to the nature of the work-stealing algorithm, we know that the body has been stolen if and only if the deque is empty. Therefore, a successful pop operation enables the fast path. If the computation takes the fast path, the original fiber can evaluate the body sequentially. Otherwise, the original fiber must seed the ivar and terminate so that the stolen slow path can take over.

6. Exceptions and process cancellation

In Manticore, exceptions follow a *sequential semantics*, which means that for a given expression, exceptions are handled in the order of sequential evaluation. Consider the example below, where the two subexpressions raise different exceptions.

$[\![$ **let pval** x **=** e_1 **in** e_2 **end** $]\!]$
$\Longrightarrow$

```
let
val iv = wsIVar()
fun bodyFn (selFn) = [[ e2[x ↦ selFn()] ]]
cont slowPathK () = bodyFn(fn () => wsIGet(iv))
val _ = wsPush(slowPathK);
val x = [[ e1 ]]
in
  if (wsPop())
    then bodyFn(fn () => x) (* fast path *)
    else ( wsIPut(iv, x); stop() )
end
```

Figure 5. The translation $[\![e]\!]$ from **pval**s to our runtime primitives. We give the interesting case here; the other cases apply structurally in the obvious manner.

```
let pval x = raise Foo
in
  raise Bar
end handle Foo => ... (* handled Foo *)
         | Bar => ... (* impossible *)
```

Since the subexpressions can be evaluated in parallel, either of the exceptions could be raised first. Our sequential semantics, however, requires that `Foo` is always handled instead of `Bar`. Data-parallel arrays also follow a sequential semantics, although supporting this behavior is trickier than for **pval**s and is studied in detail in Shaw's Master's paper [Sha07] and a companion paper appearing in these proceedings [FRRS08].

Despite this rigid ordering for handling exceptions, a significant optimization is still possible. Once an exception has reached its handler, we can often free up the resources of unnecessary parallel computations. For example, once `Foo` is handled, the computation of `f()` can safely be discarded.

```
let pval x = raise Foo
in
  f()
end handle Foo => ... (* handled Foo *)
```

Likewise, any subcomputations that `f()` spawns can be discarded. In our scheduling framework, we call this optimization *process cancellation*. The next two sections focus on building process cancellation into our framework.

Supporting multiple scheduling disciplines, as we do in our framework, makes process cancellation trickier to implement. Consider the example below, where two schedulers are running at once, the work-stealing scheduler for the **pval** and the gang scheduler for the parallel tuple.

```
let pval x = raise Foo
in
  (| f x, g x |)
end handle Foo => ... (* handled Foo *)
```

After handling `Foo`, we can safely cancel the body of the **pval**, including the subcomputation for `g`, which might be evaluating in parallel with the body.

In order to fully support this cancellation semantics, we require that computations perform two jobs. They track parent-child relationships for spawned processes, including those processes spawned on other schedulers, so that cancellations can be passed to all subcomputations. They also poll for cancellation signals. Duplicating this code for each scheduler is not modular, and can result in unnecessary use of shared state for passing cancellation signals.

In the next section, we describe a mechanism for adding cancellation to our framework. Our goals for this mechanism are threefold.

1. It should be flexible enough to work with a variety of scheduling disciplines, yet should be modular.
2. Adding or removing cancellation support into a scheduler should be trivial. This goal is important, since sometimes compiler analysis can determine when cancellation is unnecessary. As shown in Section 7, avoiding cancellation overheads can be beneficial.
3. If, however, a parallel expression needs cancellation, we still want the overhead to be reasonably small. Our evaluation in Section 7 shows that our mechanism meets this goal.

6.1 Cancelable fibers

Since fibers are our most basic form of process, we start by building a mechanism for making them cancelable. A `cancelable` is a communication channel for canceling fibers. Our representation includes a flag for whether the fiber has been canceled, a flag for whether the fiber is running, pointers to the child cancelable, and a pointer to the parent cancelable. These parent-child pointers play a similar role to *try trees* in JCilk [DLL06].

```
datatype cancelable = CANCELABLE of {
    canceled : bool ref,
    inactive : bool ref,
    children : cancelable list ref,
    parent : cancelable option
  }
```

Below are operations for creating cancelables, for canceling them and for making fibers cancelable.

```
val makeCancelable : unit -> cancelable
val cancel : cancelable -> unit
val cancelWrapper :
          (cancelable * fiber) -> fiber
```

The `cancel` operation is synchronous; it waits for its canceled processes and their children to terminate if they are running. Using these functions, we can put fiber cancellation to work. The example program below initiates a long-lived parallel computation and then abruptly cancels the computation, along with any fibers that it spawned in the meantime.

```
let
val k = fiber(fn () => largeComputation())
val c = makeCancelable()
val cancelableK = cancelWrapper(c, k)
in
  enqOnVP(vproc, cancelableK);
  cancel(c)
end
```

Notice that this example is independent of any other scheduling code.

The interesting part of our implementation, specifically how we handle polling, is shown in Figure 6. We use the scheduler action, `wrapper`, to wrap around the fiber and to poll for cancellation. This scheduler is present on the action stack whenever the fiber is evaluating. When either running or stopping the wrapped fiber, the scheduler updates the state properly. The `setInactive` function sets the currently running cancelable to the parent cancelable and marks the fiber as inactive. The `setActive` function sets the current cancelable to the given cancelable and marks the fiber as active.

To track parent-child relationships for fibers, we maintain a pointer to the running cancelable in FLS. When calling `makeCancelable`, we obtain this pointer and update its children list appropriately.

Our `cancel` operator waits for all of its in-flight, canceled fibers to terminate. Our experience is that this synchronous semantics cuts down on the total time to unload the system. Our protocol is as follows. First, we set the cancelable to true and then wait for

```
fun cancelWrapper (c, k) = let
      val CANCELABLE{canceled, ...} = c
      fun terminate () = (setInactive(c); stop())
      fun dispatch (wrapper, k) = if (!canceled)
          then terminate()
          else ( setActive(c);
                 run (wrapper, k) )
      cont wrapper (sgn) = (case sgn
             of STOP => terminate()
              | PREEMPT k =>  (
                setInactive(c);
                atomicYield();
                dispatch(wrapper, k) )
            (* end case *))
      in
        fiber(fn () => (mask();
                        dispatch(wrapper, k)))
      end
```

Figure 6. Cancel wrapper.

the cancelable to become inactive. Once inactive and canceled, we know that the cancelable's children list can no longer change. We then mark the pointer to the children with an empty list and repeat the protocol for all the children.

6.2 Scheduler support

We can add cancellation support to a scheduler by modifying existing code in the three places listed below. The first two of these modifications justify why we structure all of our schedulers to operate on fibers. This design enables the first goal for our cancellation mechanism (namely, that cancellation is modular in the sense that scheduler implementations are able to share common features and code).

1. When spawning, the scheduler wraps the new fiber in a cancelable.
2. When blocking, the scheduler must always re-wrap the resumption fiber.
3. Slightly more subtle, any wrapped fiber must terminate its computation by calling `stop()`. Failing to do so could lead to an unlimited growth of cancel wrappers on the action stack.

In the remainder of this section, we show how to apply these rules to extend our work-stealing scheduler with exceptions. As the process is slightly more involved, we do not describe exceptions for data-parallel arrays, but just assume that they use our cancellation mechanism.

Figure 7 shows our translation of **pval**s, now extended to support our sequential semantics for exceptions. We make modification (1) by wrapping the body before putting it on the work-stealing queue. We make modification (2) by changing the ivar-get operation. This operation differs from in the original only in that it rewraps the return continuation before blocking on the ivar.

```
val wsIGet : ('a ws_ivar * cancelable) -> 'a
```

We make modification (3) by changing the `bodyK` fiber. Instead of returning directly to the outer context, this fiber puts the rest of the computation on the queue and then stops, thus cleaning the cancel wrapper off the action stack.

Our translation supports exceptions as follows. If an exception is raised in e_2, the computation forces e_1 to evaluate. Doing so either blocks on the ivar forever or eventually returns. The former case indicates that e_1 has raised an exception on the original processor. Note that in this case, the garbage collector can reclaim the space for the stolen evaluation of e_2 when it reclaims the ivar. In the latter case, we know that e_1 has finished evaluating, so it is safe

```
⟦ let pval x = e1 in e2 end ⟧
                    ⟹
let
val iv = iVar()
val bodyC = makeCancelable()
fun bodyFn (selFn) =
          (⟦ e2[x ↦ selFn()] ⟧)
            handle exn => ( selFn(); raise exn )
cont slowPathK () = let
     cont retK () =
         bodyFn(fn () => wsIGet(iv, bodyC))
     in
        wsPush(retK);
        stop()
     end
val _ = wsPush(cancelWrapper(bodyC, slowPathK));
val x = ⟦ e1 ⟧
         handle exn => ( cancel(bodyC);
                         raise exn )
in
  if (wsPop())
    then bodyFn(fn () => x) (* fast path *)
    else ( wsIPut(iv, x); stop() )
end
```

Figure 7. The translation $[\![e]\!]$ from **pval**s to our runtime primitives. We only give the interesting case here; he other cases apply structurally in the obvious manner.

to re-raise the exception. If an exception is raised in e_1, we cancel the body computation and re-raise the exception.

6.3 Returning to the example

Using our cancelation mechanism, it is easy to describe how we can handle our motivating example in which an cancelation passes through multiple schedulers.

```
let pval x = raise Foo
in
  (| f x, g x |)
end handle Foo => ... (* handled Foo *)
```

After `Foo` is raised, our translation for the **pval** binding guarantees that we cancel the body computation:

```
(| f x, g x |)
```

By canceling the body, we also implicitly cancel the computation for `g x` (supposing that it is evaluating in parallel), and have thus achieved our goal.

6.4 Parallel-or

Our final parallel mechanism is the *parallel or* combinator defined by Osborne [Osb90]. This combinator has the following type:

```
val por : ((unit -> 'a option) *
           (unit -> 'a option)) -> 'a option
```

Osbourne gives five requirements for evaluating the expression `pOr(`f_1`, `f_2`)`:

1. create a thread to evaluate each f_i in parallel;
2. return the first `SOME(v)` value;
3. return `NONE` if both f_i evaluate to `NONE`;
4. cancel useless computations after the first `SOME(v)` value is returned;
5. schedule the allocation of the resources to the computations and their descendants to minimize the expected time to return a result.

```
fun porBody (f1, f2, retK) = let
    val {markFull, markEmpty} = porCell()
    fun return (vOpt) = (
        wsPush(fiber(fn () => throw retK vOpt));
        stop() )
    fun handlerK (sibling, f) = fiber(fn () =>
        case f()
         of SOME v => ( markFull();
                        cancel(sibling);
                        return(SOME v) )
          | NONE => ( markEmpty();
                      return(NONE) )
        (* end case *))
    val c1 = makeCancelable()
    val c2 = makeCancelable()
    in
       wsPush(cancelWrapper(c2, handlerK(c1, f2)));
       throw (cancelWrapper(c1, handlerK(c2, f1))) ()
    end
```

Figure 8. Parallel-or internals

In this section, we describe our implementation, which meets Osbourne's five requirements. By utilizing our work-stealing scheduler and our cancelation mechanism, our job here is easy. Parallel or initiates by capturing the return continuation and then evaluating body of the computation.

```
fun por (f1, f2) = let
    cont retK (x) = x
    in
       porBody(f1, f2, retK)
    end
```

The function shown in Figure 8 contains the remainder of the implementation. First this function allocates a parallel-or cell that tracks the state of the computation. The cell contains two operations: `markEmpty`, which records that a `NONE` has been computed and finishes the calling thread, and `markFull`, which records that a value has been computed and terminates the second fiber to call it. These functions use atomic compare-and-swap operations on a reference cell that records the current state of the computation. If a computation returns a `SOME(v)` value, it cancels its sibling fiber, and then resumes the outer continuation `retK`.

7. Evaluation

The system described in this paper is implemented and we have conducted some preliminary experiments to test its performance. Our test machine has four dual-core AMD Opteron 870 processors running at 2GHz. Each core has a 1Mb L2 cache. The system has 8Gb of RAM and is running Debian Linux (kernel version 2.6.18-6-amd64).

Figure 9 shows the speedup curves for four standard parallel benchmarks and Table 1 shows our performance measurements for these benchmarks plus one synthetic benchmark that we use to measure overheads. We ran each benchmark five times and report the average time; in all cases the standard deviation was less than 1%. The raytracer benchmark uses nested-data-parallelism to compute its image in parallel and was run with the gang scheduler. The other four benchmarks use our `pval` binding mechanism to express the parallelism and were run using the work-stealing scheduler without cancellation. The first three of these are standard parallel benchmarks, while the `fib` program is a synthetic benchmark used to measure the overheads of the work-stealing implementation. For each benchmark, we report the following numbers:

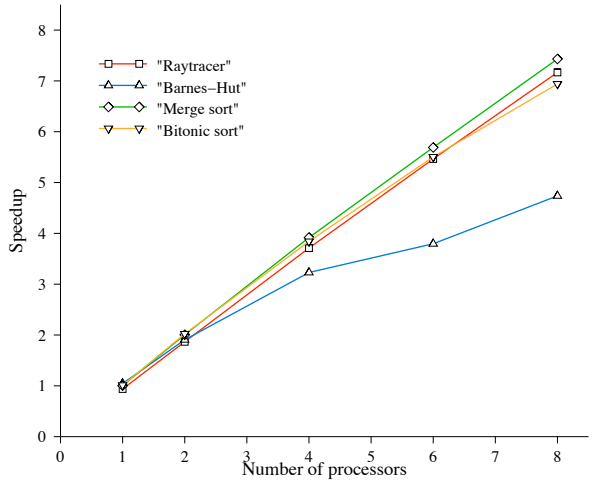

Figure 9. Speedups over the sequential version of several benchmarks.

- T_{seq}, which is the time to run a sequential version of the benchmark compiled with the Manticore compiler, but without any overhead to support parallel execution.[4]
- T_1, which is the time to run the parallel version on a single processor.
- T_8, which is the time to run the parallel version on 8-processors.
- $c_1 = T_1/T_{seq}$, which is a measure of the constant parallelism overhead.
- T_1/T_8, which is the speedup on 8-processors relative to T_1.
- T_{seq}/T_8, which is the speedup on 8-processors relative to T_{seq}.

These numbers show good scalability on several non-synthetic loads. While such a result is not surprising, it demonstrates that our system can effectively deliver parallel speedups on parallel hardware. We discuss the specific benchmarks in more detail below and then present some measurements of the cancellation mechanism in Section 7.4.

7.1 Gang scheduling

The raytracer benchmark is a program translated from Id90 [Nik91], which renders a scene of 10 spheres as an 1024×1024-pixel image. Each pixel is computed in a separate entry of a data parallel array. This program is a brute-force implementation and does not use any spatial data structures. The results show that our example scales well out to 8 processors. Furthermore, the scheduling overhead is close to 1, which suggests that our scheduler implementation is reasonably efficient.

7.2 Work stealing

The Barnes-Hut benchmark is a classic N-body problem solver, which works by first constructing a quadtree (or octree) and then using that data-structure to compute the gravitational force on the bodies in the system. Our particular version is a translation of a Haskell program. Its speedup on 8 processors is less than one might hope for, but we believe that this poor showing is because

[4] It should be noted that our current implementation does little in the way of sequential optimization (*e.g.*, it does not flatten arguments to known functions) and thus our sequential performance tends to be significantly slower than optimizing SML compilers, such as SML/NJ and MLton.

Table 1. Sample benchmarks on one and eight processors.

Benchmark	Scheduler	Times (seconds) T_{seq}	T_1	T_8	c_1	T_1/T_8	T_{seq}/T_8
Raytracer (1024x1024)	Gang scheduling	9.579	10.211	1.337	1.066	7.639	7.167
Barnes Hut (30K)	Work stealing	2.864	2.737	0.605	0.955	4.526	4.737
Merge sort (2^{18})	Work stealing	1.296	1.290	0.174	0.995	7.397	7.431
Bitonic sort (2^{18})	Work stealing	4.751	4.694	0.685	0.988	6.857	6.940
Fib (29)	Work stealing	0.038	0.639	2.556	16.911	0.033	0.015

the tree-building phase is sequential in our implementation. Our measurements show that the force calculation, which is the part that we parallelize, achieves a better speedup of close to 7.

The two sorting benchmarks, however, achieve much better speedups. We believe that some of this speedup arises from the fact that garbage collection is parallel in our system. These programs have high allocation rates, since we use heap-allocated activation records and since our optimizer is not yet highly tuned. A high allocation rate means more frequent local garbage collections, which is work that is effectively parallelized by our implementation.

7.3 Work stealing overhead

The constant-work overheads for our non-synthetic benchmarks are nearly negligible, since in all cases there is sufficient computation to offset the cost of the spawns. The fractional constant overhead for Barnes Hut is a minor anomaly in our data. We believe that this result occurs for two reasons: the overhead for scheduling is negligible to begin with and the **`pval`** translation is having a synergistic effect with our optimizer. Such small work overheads would suggest that our implementation is faithful to the work-first principle but unfortunately our synthetic benchmark, `fib`, tells a different story. This synthetic load is the naïve, exponential version of the Fibonacci computation. Indeed, our work overhead of 17 for the `fib` benchmark is much larger than the value of 3.8 overhead measured for Cilk-5 [FLR98].

To understand this high overhead, we used the methodology of of Frigo *et al.* [FLR98] to isolate the main sources of overhead. We used the `fib(29)` benchmark as it contains little computation relative to the scheduling overhead. We ran a series of tests on a single processor with different parts of the work-stealing implementation included. The difference in execution time between these tests and the sequential time for `fib(29)` was used to quantify the individual overheads. We were able to isolate five components that account for 78% of the overhead (see Figure 10). These components are

- The object promotion that is necessary when pushing a fiber onto the local deque. This promotion is required to maintain our heap invariant, since fibers in the deque are visible to other vprocs, but it accounts for the majority of the overhead.
- The non-synchronization overhead associated with using ivars to hold values.
- The overhead of fiber-local-storage lookups.
- The overhead of maintaining stealable work.
- The overhead of using fibers to represent stealable work.

In addition, 22% of the overhead is unaccounted for.

The main implication of these numbers is that our overriding concern should be reducing the cost and number of promotions. One potential solution is to modify our garbage collection implementation to make a special case for our deques; this approach, however, seems *ad hoc* and tricky. Another is to use static analysis to detect when new objects are going to be promoted and allocate them directly in the global heap. Another more radical solution would be switching to software polling [Fee93] to synchronize deque operations. This approach would make it possible to relegate promotions to successful steals, but would require sophisticated compiler analysis. The unanswered question is whether the overhead of software polling would undermine the benefits of eliminating the promotions. We expect, however, that as we implement to more realistic programs, reducing the amount of promotions will pay off, since doing so will keep data in the local heap and will reduce the frequency of global collections.

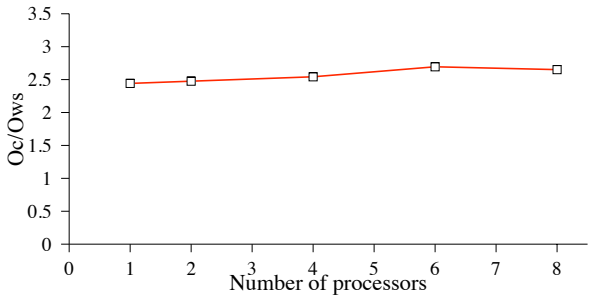

Figure 11. Ratio of cancellation overhead (O_c) to work-stealing overhead (O_{ws}).

7.4 Cancellation and parallel or

The overhead of cancellation has particular significance for our evaluation. Recall that our cancellation mechanism makes heavy use of our action stack, so we expect that the cancellation overhead is partially indicative of the performance we can expect from the primitives of our scheduling operations `run` and `forward`. We used two benchmarks to measure the costs of cancellation.

The first test measures the relative cost of the bookkeeping needed to support cancellation, without any actual cancellation. For this test we used the `fib(29)` program, but included the cancellation support as described in Figure 7. The results are plotted in Figure 11, which shows the ratio of overhead for work-stealing with cancellation to the overhead of work-stealing without cancellation. These results show that cancellation incurs a factor of about 2.5 more overhead than simple work stealing, but that the overhead is independent of the number of processors.

The second test is the classic n-Queens benchmark ($n = 20$ in our case) using parallel or to return the first successful search. Note that a breadth-first work decomposition will not give any speedup for this program on 8 processors, since every initial board position leads to a solution, but this program is useful for measuring the cost of canceling computations. Figure 12 shows the average time that it took to cancel the outstanding computations (*i.e.*, all of the fibers that were either running or in the deque waiting to be stolen). For this program the cancellation time is largely independent of the number of processors. We measured 20 runs each for 1, 2, 4, 6, and

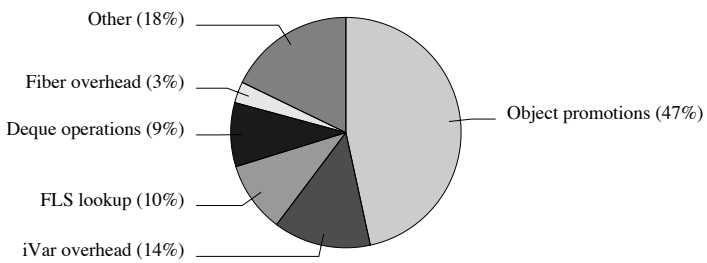

Figure 10. Breakdown of work-stealing overheads. We used `fib(29)` to obtain these numbers.

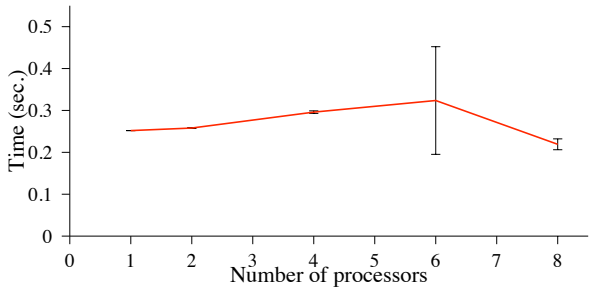

Figure 12. Cancellation time for n-Queens benchmark.

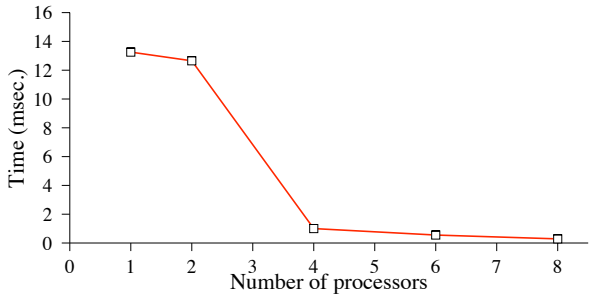

Figure 13. Time per canceled fiber.

8 processors. This plot also shows the standard deviation, which was small except for the case of six processors where two outliers skewed the results. We believe that these points are most likely a result of some OS-level scheduling decision. Similar anomalies have been reported for JCilk [DLL06], although on a different architecture and OS.

We also plotted the average time in milliseconds to cancel a computation in Figure 13. The graph shows that the cancellation phase is benefiting from parallelism.

These two experiments show that the overheads from supporting cancellation are not dependent on the number of processors, which suggests that cancellation will scale well when applied to more realistic workloads.

8. Related work

Our scheduler actions are inspired by Shivers' proposal for exposing hardware concurrency using continuations [Shi97]. We have extended Shivers' proposal to support nested scheduler actions and multiple processors.

The basic design of our scheduling framework was sketched in an earlier workshop paper [FRR+07], but this paper greatly expands and improves on that earlier work. The earlier paper only presented one example scheduler (a simple scheduler for data-parallel computation), whereas this paper presents several more substantial schedulers and an evaluation of the implementation. We have also extended our model to cancelation.

STING [JP92] is a parallel dialect of SCHEME that, like our runtime model, aims to support multiple parallel-language constructs in a unified framework. STING's three layers of process abstraction and more abstract mechanism for implementing scheduling policies contrasts with our approach, which favors minimal process abstractions and a unified infrastructure for implementing schedulers.

Recent work on the Glasgow Haskell Compiler (GHC) runtime is similar to our work [LMPT07]. It supports building modular libraries for concurrency and supports developing nested schedulers, although examples of such schedulers are not yet available in the literature. The GHC work also is intended for a lazy language and makes heavy use of transactional memory, whereas our work is intended for a strict language and uses lower-level concurrency primitives such as *compare-and-swap*.

The JCilk language [DLL06] extends Java with a combination of Cilk-based multithreading and linguistic exceptions. Similar to our approach, their exception semantics tries to mimic the sequential behavior of the host language. Unlike our system, they use the exception mechanism as the primary tool for speculative parallelism. Another difference is that we have endeavored to support cancelation in multiple schedulers, whereas JCilk specializes in work stealing.

9. Conclusion

We believe that in order to take advantage of manycore processors, general-purpose languages will need a mix of schedulers. Furthermore, we believe that in order to be sufficiently general, compilers and runtime systems for these languages will need the flexibility to experiment with scheduling code. We believe that this paper has taken an initial but significant step towards our these goals. This paper outlines the design and implementation of our scheduling framework in Sections 2 and 3. We provide evidence for the effectiveness of our framework in several steps. In Sections 4 and 5, we describe implementations of two schedulers. In Section 6, we present a general cancelation mechanism, which we use to extend our schedulers with speculative parallelism. Through our implementations, we have laid out a modular style for programming our scheduling and synchronization libraries; we believe this style will temper the complexity of our growing implementation, and ultimately hasten the maturity of our compiler. Finally, we have

provided evidence in Section 7 that our design is reasonably efficient, and posit that with some extra tuning our implementation can achieve excellent performance.

References

[ABP98] Arora, N. S., R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors, 1998.

[ANP89] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM TOPLAS*, **11**(4), October 1989, pp. 598–632.

[BL99] Blumofe, R. D. and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, **46**(5), 1999, pp. 720–748.

[BLS98] Blumofe, R. D., C. C. Leiserson, and B. Song. Automatic processor allocation for work-stealing jobs, 1998.

[BP98] Blumofe, R. D. and D. Papadopoulos. The performance of work stealing in multiprogrammed environments (extended abstract). In *Measurement and Modeling of Computer Systems*, 1998, pp. 266–267.

[BS81] Burton, F. W. and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81*, New York, NY, October 1981. ACM, pp. 187–194.

[CHRR95] Carlisle, M., L. J. Hendren, A. Rogers, and J. Reppy. Supporting SPMD execution for dynamic data structures. *ACM TOPLAS*, **17**(2), March 1995, pp. 233–263.

[CLP+07] Chakravarty, M. M. T., R. Leschchinski, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *DAMP '07*, New York, NY, January 2007. ACM, pp. 10–18.

[DG94] Doligez, D. and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL '94*, New York, NY, January 1994. ACM, pp. 70–83.

[DH89] Dybvig, R. K. and R. Hieb. Engines from continuations. *Comp. Lang.*, **14**(2), 1989, pp. 109–123.

[DL93] Doligez, D. and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *POPL '93*, New York, NY, January 1993. ACM, pp. 113–123.

[DLL06] Danaher, J. S., I.-T. A. Lee, and C. E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming*, **63**(2), 2006, pp. 147–171.

[Fed07] Fedorova, A. *Operating System Scheduling for Chip Multithreaded Processors*. Ph.D. dissertation, Department of Computer Science, Harvard University, Boston, MA, 2007.

[Fee93] Feeley, M. Polling efficiently on stock hardware. In *FPCA '93*, New York, NY, June 1993. ACM, pp. 179–187.

[Fei94] Feitelson, D. G. Job scheduling in multiprogrammed parallel systems. *Research Report RC 19790 (87657)*, IBM, October 1994. Second revision, August 1997.

[FFR+07] Fluet, M., N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status report: The Manticore project. In *ML '07*, New York, NY, October 2007. ACM, pp. 15–24.

[FLR98] Frigo, M., C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98*, New York, NY, June 1998. pp. 212–223.

[FRR+07] Fluet, M., M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *DAMP '07*, New York, NY, January 2007. ACM, pp. 37–44.

[FRRS08] Fluet, M., M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. In *ICFP '08*, New York, NY, September 2008. ACM.

[FSDF93] Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI '93*, New York, NY, June 1993. ACM, pp. 237–247.

[Hal84] Halstead Jr., R. H. Implementation of multilisp: Lisp on a multiprocessor. In *LFP '84*, New York, NY, August 1984. ACM, pp. 9–17.

[HF84] Haynes, C. T. and D. P. Friedman. Engines build process abstractions. In *LFP '84*, New York, NY, August 1984. ACM, pp. 18–24.

[HFW84] Haynes, C. T., D. P. Friedman, and M. Wand. Continuations and coroutines. In *LFP '84*, New York, NY, August 1984. ACM, pp. 293–298.

[HJT+93] Hauser, C., C. Jacobi, M. Theimer, B. Welch, and M. Weiser. Using threads in interactive systems: A case study. In *SOSP '93*, December 1993, pp. 94–105.

[JP92] Jagannathan, S. and J. Philbin. A customizable substrate for concurrent languages. In *PLDI '92*, New York, NY, June 1992. ACM, pp. 55–81.

[LMPT07] Li, P., S. Marlow, S. Peyton Jones, and A. Tolmach. Lightweight concurrency primitives for GHC. In *HASKELL '07*, New York, NY, September 2007. ACM, pp. 107–118.

[MKH90] Mohr, E., D. A. Kranz, and R. H. Halstead Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *LFP '90*, New York, NY, June 1990. ACM, pp. 185–197.

[MS96] Michael, M. M. and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96*, New York, NY, May 1996. ACM, pp. 267–275.

[Nik91] Nikhil, R. S. *ID Language Reference Manual*. Laboratory for Computer Science, MIT, Cambridge, MA, July 1991.

[Osb90] Osborne, R. B. Speculative computation in multilisp. In *LFP '90*, New York, NY, June 1990. ACM, pp. 198–208.

[Rai07a] Rainey, M. The Manticore runtime model. Master's dissertation, University of Chicago, January 2007. Available from `http://manticore.cs.uchicago.edu`.

[Rai07b] Rainey, M. Prototyping nested schedulers. In *Redex Workshop*, September 2007.

[Ram90] Ramsey, N. Concurrent programming in ML. *Technical Report CS-TR-262-90*, Dept. of C.S., Princeton University, April 1990.

[Rep89] Reppy, J. H. First-class synchronous operations in Standard ML. *Technical Report TR 89-1068*, Dept. of CS, Cornell University, December 1989.

[Rep90] Reppy, J. H. Asynchronous signals in Standard ML. *Technical Report TR 90-1144*, Dept. of CS, Cornell University, Ithaca, NY, August 1990.

[Rep99] Reppy, J. H. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[Rep02] Reppy, J. Optimizing nested loops using local CPS conversion. *HOSC*, **15**, 2002, pp. 161–180.

[RX07] Reppy, J. and Y. Xiao. Specialization of CML message-passing primitives. In *POPL '07*, New York, NY, January 2007. ACM, pp. 315–326.

[RX08] Reppy, J. and Y. Xiao. Toward a parallel implementation of Concurrent ML. In *DAMP '08*, New York, NY, January 2008. ACM.

[Sha07] Shaw, A. Data parallelism in Manticore. Master's dissertation, University of Chicago, July 2007. Available from `http://manticore.cs.uchicago.edu`.

[Shi97] Shivers, O. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *CW '97*, New York, NY, January 1997. ACM.

[VR88] Vandevoorde, M. T. and E. S. Roberts. Workcrews: an abstraction for controlling parallelism. *IJPP*, **17**(4), August 1988, pp. 347–366.

[Wan80] Wand, M. Continuation-based multiprocessing. In *LFP '80*, New York, NY, August 1980. ACM, pp. 19–28.