Nested schedulers for heterogeneous parallelism

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 rise of commodity multicore processors makes parallel computing available to the masses. Traditional parallel languages focus on large-scale scientific computing and are not well suited to programming the applications one typically finds on desktop systems. Such desktop applications are better supported by heterogeneous parallel languages that provide a spectrum of parallel constructs working at different granularities. In this paper, we focus on the problem of how to support a heterogeneous collection of parallel-programming mechanisms in a compiler and runtime system. 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. Our approach is part of a larger effort to design and implement a parallel functional programming language, but it is flexible enough to support a wide range of possible parallel-programming mechanisms. We give examples of a number of different schedulers, provide a formal specification of the runtime model, and describe our implementation.

1. Introduction

The laws of physics and the limitations of instruction-level parallelism are forcing microprocessor architects to develop new multicore processor designs. As a result, parallel computing is becoming widely available on commodity hardware. Ideal applications for this hardware, such as multimedia processing, computer games, and small-scale simulations, can exhibit parallelism at multiple levels with different granularities, which means that a homogeneous language design will not take full advantage of the hardware resources. For example, a language that provides data parallelism but not explicit concurrency will be inconvenient for the development of the networking and GUI components of a program. On the other hand, a language that provides concurrency but not data parallelism will be ill-suited for the components of a program that demand fine-grain parallelism, such as image processing and particle systems.

Our thesis is that parallel programming languages must provide mechanisms for multiple levels of parallelism, both because applications exhibit parallelism at multiple levels and because hardware requires parallelism at multiple levels to maximize performance. Indeed, a number of research projects are exploring *heterogeneous* parallelism in language designs that combine support for parallel computation at different levels into a common linguistic and execution framework. The Glasgow Haskell Compiler (Glasgow Haskell Compiler Version 6.6) has been extended with support for three different paradigms for parallel programming: explicit concurrency, coordinated with transactional memory (Peyton Jones et al. 1996; Harris et al. 2005), semi-implicit concurrency, based on annotations (Trinder et al. 1998), and data parallelism (Chakravarty et al. 2007), inspired by NESL (Blelloch et al. 1994; Blelloch 1996). The Manticore language design (Fluet et al. 2007) incorporates both coarse-grain parallelism and fine-grain nested parallelism. Manticore's coarse-grain parallelism is based on Concurrent ML (CML) (Reppy 1991), which provides explicit concurrency and synchronous-message passing. Manticore's fine-grain nested parallelism is based on previous nested data-parallel languages, such as NESL (Blelloch et al. 1994; Blelloch 1996; Blelloch and Greiner 1996) and Nepal (Chakravarty and Keller 2000; Chakravarty et al. 2001; Leshchinskiy et al. 2006)), and provides parallel arrays and parallel-array comprehensions.

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 low-level *runtime 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. It is our belief that this runtime framework will provide 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 composition 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. For example, workcrews (Vandevoorde and Roberts 1988), work stealing (Blumofe and Leiserson 1999), and lazy task creation (Mohr et al. 1990) are abstractions providing different scheduling policies for the execution of parallel tasks. By following a few simple rules, these schedulers can be implemented in a modular and nestable way.

The remainder of the paper is organized as follows. In Section 2, we expand on the argument for nested schedulers. The paper presents three main technical contributions. The first of these, in Section 3, is the design of our scheduler framework. We demonstrate the expressiveness of our design with a series of non-trivial examples in Section 4. The second main contribution is a formal model of the scheduler framework, which is given in Section 5.

Submitted for publication; April 2007

This model presents a unified, precise, and clean target architecture to the programmer untethered from any real-world processor architecture. This relieves programmers from optimizing towards exotic and ephemeral multiprocessor designs. The third main contribution is the implementation of our framework, which is described in Section 6. We then review related work and conclude.

2. Nested schedulers

A runtime model should minimally support thread migration and load balancing, but to be a uniform substrate, it should also support policies that provide the parallelism corresponding to coarse-grain explicit concurrency and fine-grain data-parallel computations. Finally, a runtime model should support more complex scheduling policies. For example, workcrews (Vandevoorde and Roberts 1988), work stealing (Blumofe and Leiserson 1999), and lazy task creation (Mohr et al. 1990) are abstractions providing different scheduling policies for the execution of parallel tasks.

We note that these various scheduling policies often need to cooperate in an application to satisfy its high-level semantics (*e.g.*, real-time deadlines in multimedia applications). Furthermore, to best utilize the underlying hardware, these various scheduling policies should be implemented in a distributed manner, whereby a conceptually global scheduler is executed as multiple concrete schedulers on multiple processing units. Programming and composing such policies can be difficult or even impossible under a rigid scheduling regime. A rich notion of scheduler, however, permits both the nesting of schedulers and different schedulers in the same program, thus improving modularity, and protecting the policies of nested schedulers. Such nesting is precisely what is required to efficiently support heterogeneous parallelism.

In this paper, we propose a runtime model that can implement general scheduling (including migration and load balancing), but with significant improvements over previous approaches. Our framework favors a minimal collection of compiler and runtime-system mechanisms to support nested scheduling. This scheduling framework is not tied to any particular high-level language design and we demonstrate that a wide variety of parallelism mechanisms may be expressed in our model.

3. The scheduling framework

The main contribution of this paper is the design of a framework for the modular implementation of nested schedulers that support a variety of scheduling policies.[1] Our approach is similar in philosophy to the microkernel architecture for operating systems; we provide a minimum collection of compiler and runtime-system mechanisms to support nested scheduling and then build the scheduling code on top of that framework. In this section, we describe the abstractions provided by the runtime system, the compiler's intermediate representation, which serves as the scheduler API, and give an informal description of the scheduler operations with some simple examples.

3.1 Process abstractions

Our runtime model has three distinct notions of process abstraction. At the lowest level, a *fiber* is an unadorned thread of control. We use continuations to represent the state of suspended fibers.

Surface-language *threads* are represented as fibers. Depending on the language semantics, threads may have identity and/or thread-local state. We do not model these features in this paper, but they are easy to accommodate by pairing a fiber with its ID/state. Since threads may initiate implicit-parallel computations, a thread may consist of multiple fibers.

[1] Regehr coined the term "*general, heterogeneous schedulers*" for similar scheduler hierarchies (Regehr 2001).

$$
\begin{array}{rcl}
e & ::= & \mathbf{let}\, X = Y\, \mathbf{in}\, e \\
& | & X \\
& | & \mathbf{let}\, X = P(Y_1, \ldots, Y_n)\, \mathbf{in}\, e \\
& | & \mathbf{if}\, X\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \\
& | & \mathbf{fun}\, F(X_1, \ldots, X_n) = e_1\, \mathbf{in}\, e_2 \\
& | & \mathbf{let}\, X = F(Y_1, \ldots, Y_n)\, \mathbf{in}\, e \\
& | & F(X_1, \ldots, X_n) \\
& | & \mathbf{letcont}\, K(X_1, \ldots, X_n) = e_1\, \mathbf{in}\, e_2 \\
& | & \mathbf{throw}\, K(X_1, \ldots, X_n) \\
& | & \mathbf{run}\ (K_1,\, K_2) \\
& | & \mathbf{forward}\ (X) \\
\\
P & = & P^c \cup P^s \cup P^v \cup P^m \\
P^c & = & \{(), \mathbf{true}, \mathbf{false}, \mathbf{int}(n), \mathbf{none}, \mathbf{some}, \\
& & \ \mathbf{stop}, \mathbf{preempt}, \ldots\} \\
P^s & = & \{\mathbf{add}, \mathbf{sub}, \mathbf{alloc}, \mathbf{sel}_i, \cdots\} \\
P^v & = & \{\mathbf{enq}, \mathbf{deq}, \mathbf{host}, \mathbf{mask}, \mathbf{unmask}, \ldots\} \\
P^m & = & \{\mathbf{enqvp}, \mathbf{newgid}, \mathbf{provision}, \mathbf{release}, \\
& & \ \mathbf{ref}, \mathbf{deref}, \mathbf{cas}, \ldots\}
\end{array}
$$

Figure 1. A direct-style intermediate representation

Lastly, a *virtual processor* (vproc) is an abstraction of a hardware processor resource. 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. Each vproc has state that consists of a stack of signal actions, a queue of threads, and a signal mask bit. We discuss these in the sequel.

3.2 The compiler IR

Schedulers in our system are implemented in terms of the compiler's IR, which is a direct-style (ANF) λ-calculus (Flanagan et al. 1993) extended with a collection of scheduler operations. The collected syntax of this language is given in Figure 1. We use uppercase letters (F, K, X, and Y) for variables, with the convention that F is used for variables bound to functions and K denotes variables bound to continuations. This language serves as the intermediate representation (IR) of our compiler and is used to express both language-level concurrency mechanisms, *e.g.*, thread spawning, message passing, or STM, and run-time system scheduler code. For the purposes of this presentation, we extend the syntax of the IR to include sequencing, datatypes, and pattern matching.

The language is largely self-explanatory (formal semantics are given in Section 5), but it has a few constructs that are special to our application. The **run** and **forward** constructs are control-flow operations to support scheduling and are described below. The expression $\mathbf{letcont}\, K(X_1, \ldots, X_n) = e_1\, \mathbf{in}\, e_2$ binds K to a continuation that is the current continuation prepended with $\lambda(X_1, \ldots, X_n).e_1$. The scope of K includes both e_1 and e_2 (*i.e.*, continuations may be recursive) and it is a first-class value that may live beyond the evaluation of e_2. For example, the "*call-with-current-continuation*" function can be implemented as

$$\mathbf{fun}\ \mathit{callcc}(F) = \mathbf{letcont}\, K(X) = X\, \mathbf{in}\, F(K)\, \mathbf{in}\, \cdots$$

Continuations are invoked by the **throw** form.

We distinguish among four kinds of primitive operations: *primitive constants* (P^c), *sequential primitives* (P^s), *vproc primitives* (P^v), and *multiproc primitives* (P^m). Primitive constants correspond to both enumerated and raw types (*e.g.*, unit, booleans, integers) as well as tags in the representation of datatypes (*e.g.*, **preempt** is the tag for a variant of the *signal* datatype). Sequential primitives correspond to primitives that are to be evaluated by the sequential state transition in Section 5. Similarly, vproc primitives and multiproc primitives correspond to primitives that are to

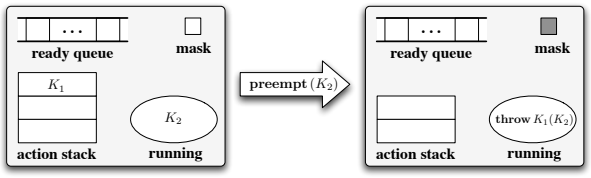

Figure 2. The effect of preemption on a vproc

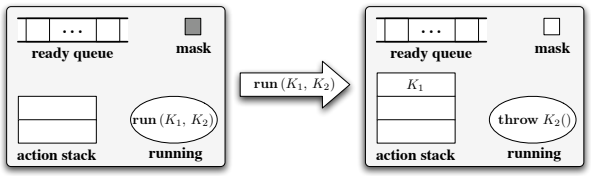

Figure 3. The effect of $\mathbf{run}\ (K_1,\ K_2)$ on a vproc

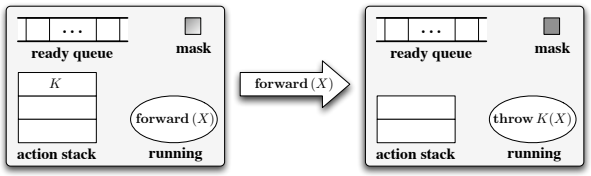

Figure 4. The effect of $\mathbf{forward}\ (X)$ on a vproc

be evaluated by the vproc and multiproc state transitions, respectively. We give informal semantics to these primitives as we encounter them in examples.

3.3 The scheduler-action stack

The heart of our mechanism are scheduler actions. A scheduler action is a continuation that takes a signal and performs the appropriate scheduling activity in response to that 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 popping the current action from the stack, setting the signal mask, and throwing the signal to the current action. The operation is illustrated in Figure 2; here we use dark grey in the mask box to denote when signals are masked.

There are two expression forms in the IR that scheduling code can use to affect a vproc's scheduler stack directly. The expression $\mathbf{run}\ (K_1,\ K_2)$ pushes K_1 onto the host vproc's action stack, clears the vproc's signal mask, and throws to the continuation K_2 (see Figure 3). The run operation requires that signals be masked, since it manipulates the vproc's action stack. The other form is the expression $\mathbf{forward}\ (X)$, which sets the signal mask and forwards the signal X to the current action (see Figure 4). 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 may, or may not, be masked when it is executed. For example, a thread exit function can be defined as

$$\mathbf{fun}\ \mathit{exit}() = \mathbf{forward}\ (\mathbf{stop})\ \mathbf{in}\ \cdots$$

Another example is the implementation of a yield operation that causes the current fiber to yield control of the processor:

$$\begin{array}{l}\mathbf{fun}\ \mathit{yield}() = \\ \quad \mathbf{letcont}\ K() = ()\ \mathbf{in} \\ \quad \mathbf{forward}\ (\mathbf{preempt}\ (K)) \\ \mathbf{in}\ \cdots\end{array}$$

3.4 Scheduling queues

In addition to the scheduler stack, each vproc has a queue of ready fibers that is used for scheduling. The **enq** operation takes a suspended fiber and adds it to the scheduler queue, while the **deq** operation removes the next fiber from the queue. If the queue is empty, then the **deq** operation causes the vproc to go idle until there is work for it.

3.5 Implementing language-level threads

Given that our framework includes first-class continuations, it is easy to implement language-level thread mechanisms (Wand 1980; Ramsey 1990; Reppy 1999). We start with a function for turning a function into a fiber (*i.e.*, continuation).

$$\begin{array}{l}\mathbf{fun}\ \mathit{fiber}(F) = \\ \quad \mathbf{letcont}\ K(X) = \mathbf{let}\ () = F(X)\ \mathbf{in}\ \mathit{exit}()\ \mathbf{in}\ K \\ \mathbf{in}\ \cdots\end{array}$$

This function can be used to implement a spawn primitive.

$$\mathbf{fun}\ \mathit{spawn}(F) = \mathbf{enq}(\mathit{fiber}(F))\ \mathbf{in}\ \cdots$$

Synchronization and communication mechanisms are supported via atomic operations, such as *compare-and-swap* (**cas**), and a concurrent-queue abstraction (Michael and Scott 1996) supported by the compiler and runtime.

For language-level threads we use a simple round-robin scheduler built on top of the per-vproc scheduling queues. This scheduler is installed at the bottom of each action stack and is called the *default scheduler*.

$$\begin{array}{l}\mathbf{letcont}\ \mathit{switch}(\mathit{sig}) = \mathbf{case}\ \mathit{sig} \\ \quad \mathbf{of}\ \mathbf{stop} \Rightarrow \mathbf{run}\ (\mathit{switch}, \mathbf{deq}()) \\ \quad\ |\ \mathbf{preempt}(K) \Rightarrow \\ \quad\quad \mathbf{let}\ () = \mathbf{enq}(K)\ \mathbf{in} \\ \quad\quad \mathbf{run}\ (\mathit{switch}, \mathbf{deq}()) \\ \quad \mathbf{end} \\ \mathbf{in}\ \cdots\end{array}$$

On a **stop** signal, it runs the next fiber in the queue and on a **preempt** signal, it enqueues the preempted fiber and then runs the next fiber.

3.6 Provisioning parallel computations

The last part of our framework are the operations used to map a parallel computation across multiple vprocs. The $\mathbf{enqvp}(VP, K)$ operation enqueues the fiber represented by the continuation K on the vproc named by VP. Using this operation we can implement an explicit migration function that moves the calling computation to a specific vproc.

$$\begin{array}{l}\mathbf{fun}\ \mathit{migrate}(VP) = \\ \quad \mathbf{letcont}\ K() = ()\ \mathbf{in} \\ \quad \mathbf{let}\ () = \mathbf{enqvp}(VP, K)\ \mathbf{in} \\ \quad \mathit{exit}() \\ \mathbf{in}\ \cdots\end{array}$$

We also provide a mechanism for assigning vprocs to computations. The basic parallel computation is a group of fibers running

on separate vprocs; the scheduling framework provides the mechanism of *group IDs* to distinguish between different parallel computations. When initiating a parallel computation, the **newgid** operation is used to create a unique group ID for the computation. This ID is passed to the **provision** operation to request additional vprocs. This operation either returns a vproc that is not already assigned to the computation or else the constant **none** to signal that no additional processing resources are available for the group. When a computation is finished with a vproc, it uses the **release** operation to signal to the runtime that it is done with the vproc.

3.7 Load balancing

Load balancing is important for maximizing the performance of parallel applications. In our framework, there are three ways that load balancing is achieved:

1. The runtime system tracks the distribution of parallel computations across vprocs. When provisioning vprocs for a new computation, we attempt to keep the load even.
2. The implementation of language-level parallel constructs is responsible for balancing the load of a given parallel computation across the vprocs that it is assigned.
3. The runtime system can periodically migrate threads from one vproc to another.

4. Schedulers for parallel computation

The main goal of our work is to support a wide range of parallel programming models and language mechanisms in a unified framework. In this section, we describe schedulers for a number of parallel programming models from the literature. These schedulers demonstrate the flexibility of our approach. Furthermore, they are modular and could easily coexist in the same application.

In the previous section, we described a simple, round-robin, scheduler for language-level threads. Schedulers for parallel computations are more complicated, since they must coordinate the activities of parallel fibers running on multiple vprocs, and they often require auxiliary data structures. For this reason, we switch notation in this section to use SML extended with continuations and the **run** and **forward** primitives. For signals, we used the following datatype

```
datatype signal = STOP | PREEMPT of fiber
```

and we use SML functions to define scheduler actions. Translation of this notation into the IR of Figure 1 is largely a matter of applying the ANF normalization algorithm of Flanagan et al. (1993).

One important characteristic of these schedulers is that they are *distributed* — their implementation consists of scheduler actions installed on multiple vprocs. At any given time, only a fraction of the the vprocs may be executing code specific to any given scheduler. Thus, the interaction between scheduler actions is inherently asynchronous.

4.1 Scheduler utility functions

There are a number of operations that are common to many schedulers. These include

```
val provisionN : int -> vproc list
```

which requests a list of vprocs from the runtime system (note that the result does *not* include the host vproc). We also use a function for passing preemptions up the action stack:

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

Notice that this function will remask signals when it resumes. The `schedFiber` function creates a fiber that will run the function `f` with a given signal action.

```
fun workcrew {nJobs, nWorkers, job} =
      callcc (fn doneK => let
        val gid = newgid()
        val nStarted = ref 1
        val nDone = ref 0
        val vprocs = provisionN (gid, nWorkers-1)
        val n = length vprocs
        fun switch STOP = let
              val nextJob = fetchAndAdd(nStarted,1)
              in
                if (nextJob < nJobs)
                  then run (switch,
                    fiber (fn () => job nextJob))
                else if (fetchAndAdd(nDone,1) = n)
                  then (release (gid, host());
                    throw doneK())
                  else finish gid
              end
          | switch (PREEMPT k) = (
              atomicYield();
              run(switch, k))
        in
          List.app
            (fn vp => dispatchOn (vp, switch))
              vprocs;
          run (switch,
            fiber (fn () => job 0))
        end)
```

Figure 5. Workcrews for data-parallel computations

```
fun schedFiber (switch, f) =
      fiber (fn () => run (switch, fiber f))
```

It is used to run a scheduler action on a remote vproc in the `dispatchOn` function, which takes a vproc and scheduler action, and dispatches the action on the vproc.

```
fun dispatchOn (vproc, switch) =
      enqVP (vproc, schedFiber (switch, exit))
```

By using the `exit` function as the function to run, we cause the `STOP` signal to be sent to the `switch` action on the remote vproc. The `finish` function is used to release a vproc and terminate the computation.

```
fun finish gid = (release(gid, host()); exit())
```

We also need concurrent queues to schedule work between vprocs. These queues have the following interface:

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

4.2 Scheduling data-parallel fibers

Data-parallel computations require multiple fibers running on multiple vprocs. There are a number of different ways to organize this computation, but this example uses the *workcrew* approach (Vandevoorde and Roberts 1988) (also called gang scheduling (Chakravarty et al. 2007)). A workcrew consists of some number of workers, each running on a separate processor, and a global pool of work. Workers iterate getting a job from the work pool and executing it. The function in Figure 5 creates a workcrew of up to `nWorkers` to compute a job that has been partitioned into `nJobs` pieces. The `job` parameter is a function that takes an integer argument i and computes the i^{th} job.

The implementation begins by provisioning a group of processors. It then installs the its scheduler (`switch`) on each of the vprocs. Once initialized on a vproc, `switch` begins running jobs from the work pool. The `STOP` signal indicates the completion of a

```
fun wsSwitch i = let
      fun newWork () = (
        case remQ (Vec.sub(qs, i))
         of NONE =>
              run (wsSwitch i, pickVictim qs)
          | SOME k => run (wsSwitch i, k))
      in
        fn STOP => newWork ()
         | PREEMPT k => (
             addQ(Vec.sub(qs, i), k); atomicYield();
             newWork())
      end
```

Figure 6. The work-stealing scheduler-action function

job. If another job is available, the scheduler creates a fiber for the job, and runs it. Otherwise, it releases the host vproc. The last vproc to finish will return control to `doneK`. Otherwise, there are still other running jobs, so the scheduler calls `finish`, which releases its host vproc and exits. When the scheduler receives a `PREEMPT` signal, it yields control to the current scheduler, which effectively lets us inherit the preemption policy of the parent scheduler. This practice of immediately yielding the vproc allows the parent scheduler to decide whether to resume this scheduler or to perform other computations before resuming this scheduler.

4.3 Work stealing

Language-level threads created by the `spawn` function are not initially run in parallel. In some cases, this choice has advantages such as reduced communication overhead and improved cache locality. But for optimal performance on multiprocessors, the runtime system must balance the load continually amongst all physical processors. A common technique for load balancing is work stealing (Burton and Sleep 1981; Halstead Jr. 1984; Mohr et al. 1990; Carlisle et al. 1995; Blumofe and Leiserson 1999), in which an idle processor (the thief) steals work from another processor (the victim). To illustrate how our framework can support work stealing, we define the implementation of a function `wsSpawn` that spawns a thread, which may be stolen.

As with workcrews, we start by provisioning a number of vprocs and installing the work-stealing scheduler actions on them. We also create a global vector of thread queues with one entry per vproc. The heart of the implementation is the scheduler-action function `wsSwitch`, which is given in Figure 6. This function takes the index of its host vproc and returns the actual scheduler action. The scheduler action handles the `STOP` signal by looking for new work. More interestingly, when the scheduler handles the `PREEMPT` signal, it places the preempted computation back on its queue and then yields control to its parent scheduler. If the parent schedules other computations before resuming the work-stealing scheduler, other schedulers in the work-stealing group that need work can steal the preempted computation. When the work-stealing scheduler action is resumed from the `atomicYield`, it looks for new work.

The scheduler looks for new work by first trying to obtain a fiber from its own ready queue. If its queue is empty, it then picks a victim from which to steal a computation. The process of picking a victim and picking an appropriate fiber to steal requires careful design (Blumofe and Leiserson 1999) and is beyond the scope of this paper. Spawning a unit of work on this scheduler simply equeues a fiber representing the desired computation on one of the queues (for simplicity, we use the first queue).

```
fun wsSpawn f = addQ(Vec.sub(qs, 0), fiber f)
```

```
fun treeSum LEAF = 0
  | treeSum (NODE(trl, i, trr)) =
      callcc (fn retK => let
        val done = ref false
        val result = iVar ()
        fun fCtx () = if TAS(done)
              then exit()
              else let
                val s2 = treeSum trr + i
                val s1 = get result
                in
                  throw retK(s1+s2)
                end
        in
          wsSpawn fCtx;
          put (result, treeSum trl);
          fCtx ()
        end)
```

Figure 7. Lazy task creation for the `treeSum` function

4.4 Futures with Lazy Task Creation

Lazy task creation (Mohr et al. 1990) is a technique for implementing parallel futures that attempts to manage process granularity efficiently. Rather than launching a thread for each future, it provisionally inlines the bodies of futures, but allows idle processors to steal their return continuations. For example, consider the classic `treeSum` example, where the recursive sum of the left tree is marked as a future:

```
fun treeSum LEAF = 0
  | treeSum (NODE (trl, i, trr)) =
    future (treeSum trl) + (treeSum trr + i)
```

Under lazy task creation, the left-recursive call will be evaluated immediately and the return continuation

$$\lambda x.(x \texttt{ + (treeSum trr + i)})$$

is enqueued on a work queue and may be stolen by another processor. Such a mechanism requires compiler support to translate futures into the appropriate lower-level operations, but our scheduling framework is an adequate target for such a translation. We demonstrate this fact by describing an implementation of `treeSum`. For synchronizing on futures, we use *IVars*, a write-once, synchronous memory cell (Arvind et al. 1989). The `iVar` operation creates a new IVar; the `get` operation returns the value of the cell, blocking if the cell is empty; and the `put` operation stores a value into an empty cell, which resumes any fibers blocked on the IVar.

Figure 7 gives our version of `treeSum`. To make the context amenable to either parallel execution or inlining, we reify it as the `fCtx` function. Since this function will be called by both a work-stealing fiber and the original fiber, we use a boolean flag `done` to ensure that it is only executed once (`TAS` is the atomic *test-and-set* operation). To complete the computation, we perform three steps: we launch a thread for the outer context on the work-stealing scheduler; we evaluate the body of the future, writing it into the IVar; and finally we try to run the outer context.

4.5 Speculative parallelism

To make the most out of parallel hardware, one must keep it busy. One technique for doing so is to speculatively execute computations whose results may not be necessary for the final result. For example, search algorithms often sequentially try various different paths though the search space; speculatively trying paths in parallel can often speed up the search. Our framework can support speculative parallelism with only a minor extension.

Supporting speculative computation requires a mechanism to asynchronously signal fibers when they become unnecessary and a

```
fun pOr (f1, f2) = callcc (fn retK => let
      val gid = newgid ()
      val vp1 = host()
      in
        case provision gid
         of NONE => (* sequential evaluation *)
          | SOME vp2 =>
              pOr'(gid, host(), f1, vp2, f2, retK)
      end)
```

Figure 8. Parallel or.

mechanism to handle such signals when they are delivered. Scheduler actions fulfill the latter requirement, as they handle asynchronous preemption signals. But to fulfill the former requirement, we need a mechanism that can deliver an asynchronous signal to another vproc. We introduce the `signal` operator to trigger a preemption on a vproc.

```
val signal : vproc -> unit
```

There are a number of different ways to support speculative parallelism at the language level. One such mechanism is the *parallel or* combinator defined by Osborne (1990). 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. abort 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.

The implementation in Figures 8 and 9 meets these requirements, although in the interest of space, we don't strictly enforce the last requirement. To meet the fourth requirement, the implementation uses the asynchronous signaling mechanisms introduced earlier.

The `pOr` function creates a new group and attempts to acquire a second vproc. If it succeeds, it calls the `pOr'` function that manages the speculative computation; otherwise, the computation is executed sequentially (not shown). The first part of the `pOr'` function (up to and including the `switch` function) manages signal handling. When a preemption comes in, the `termFlg` is checked both before and after the call to `atomicYield`. Checking for termination before yielding allows the scheduler to immediately terminate the computation, while checking for termination after yielding allows the scheduler to compose nicely with other scheduler actions. The remainder of the `pOr'` code deals with merging the results of the subcomputations. First it allocates a parallel-or cell that tracks the state of the computation. It has 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 are implemented using atomic compare-and-swap operations on a reference cell that records the current state of the computation. If a fiber computes a `SOME(v)` value, it kills its sibling using `kill`. Finally, if the speculative fiber proceeds past the marking phase, it resumes the outer continuation `retK`.

```
fun pOr' (gid, vp1, f1, vp2, f2, retK) = let
      val termFlg = ref false
      fun kill vp = (termFlg := true; signal vp)
      fun switch STOP = exit()
        | switch (PREEMPT k) = (
            if termFlg then finish gid else ();
            atomicYield  ();
            if termFlg then finish gid else ())
      val {markFull, markEmpty} = pOrCell gid
      fun wrapper (vp, f) =
            schedFiber (switch, fn () => (
              case f()
               of SOME v => (
                    markFull(); kill vp;
                    throw retK(SOME v))
                | NONE => (
                    markEmpty();
                    throw retK(NONE))))
      in
        enqVP (vp1, wrapper (vp2, f1));
        enqVP (vp2, wrapper (vp1, f2));
        exit()
      end
```

Figure 9. Parallel-or internals

4.6 Scheduler invariants

Our scheduler framework does not provide any explicit mechanisms for guaranteeing liveness or fairness. It is quite easy to write selfish schedulers that monopolize a vproc, but since schedulers are part of the surface-language implementation, such schedulers are considered a bug in the implementation. We do, however, have guidelines for implementing schedulers that allow nested schedulers to share vprocs in a way that is acceptable to all active schedulers.

In order that schedulers be nestable, they must coordinate both *upwards* with their parent scheduler and *downwards* with their children. (Note that a scheduler need not coordinate *across* with its sibling schedulers, as any interaction between sibling schedulers is mediated by the current scheduler.)

A scheduler coordinates upwards by forwarding preemption signals to its parent scheduler in a timely fashion. Schedulers coordinate downwards by eventually running every preempted fiber. These two means of coordination work in concert to ensure that every scheduler (and every scheduled user computation) makes progress. In practice, when coordinating upwards, a scheduler does not forward the same preemption signal to the current scheduler; rather, it forwards a new preemption signal with a new fiber that appropriately resumes the scheduler. Similarly, when coordinating downwards, a scheduler does not immediately run the last preempted fiber; rather, it uses some protocol to select among preempted fibers, eventually running every preempted fiber. When one scheduler fails to coordinate with another (either the current scheduler or one of its children), we say that it *starves* that scheduler. Schedulers that properly coordinate upwards and downwards are composable.

Our definition of upwards coordination gives a guideline for implementing modular and composable schedulers, but there is a more general problem of composing scheduler policies. In his doctoral work, Regehr developed the HLS framework for implementing nested schedulers (Regehr 2001). This framework provides a system of uniprocessor guarantees that specify the needs of a scheduling policy. These guarantees cover a wide range of scheduling requirements, including time-sharing variants, real time, and fixed priority assignments. To allow different schedulers with different guarantee requirements to share processors, HLS provides a run-

time mechanism for negotiating guarantees. When a scheduler enters a hierarchy, it requests a guarantee from the current scheduler. This guarantee, which might be stronger than the one requested, then gets parceled out to subsequent child schedulers. Supporting the HLS guarantee infrastructure in our runtime model entails encoding its syntax and conversion functions and requires mechanisms for negotiating guarantees at run time. We have sketched an implementation of the guarantee mechanism in our runtime model in which the only additional infrastructure is a new signal for requests (Rainey 2007). To request a guarantee from the current scheduler, we simply forward the vproc a request signal that contains our reply continuation.

There are also other guidelines that well-behaved schedulers should follow. For example, schedulers should acquire and release provisioned vprocs in a timely fashion. This ensures that the set of provisioned vprocs approximates the computational load of the application, a useful heuristic for provisioning strategies. Similarly, schedulers should prefer local or decentralized state and avoid expensive synchronization with global shared state. These guideline help promote an application's overall performance, while upwards and downwards coordination promote an application's progress.

5. A formal semantics of the runtime model

In this section, we specify the behavior of our runtime model in terms of an abstract machine and an operational semantics. This formalism serves two primary purposes. First, it plays the rôle of an API for the development of new schedulers. Second, it describes the requisite behavior that must be implemented when porting the runtime model to actual hardware. Furthermore, it demonstrates that an implementation must provide native support for a very small number of scheduling operations; as demonstrated in Section 4, a wide variety of schedulers may be programmed using these scheduling operations.

We formalize the abstract machine and operational semantics of the runtime model in terms of a continuation-passing style language (given below). Although the language we present is small, it is meant to be representative of an expressive sequential language that corresponds to the computational power of a single thread.[2] The evaluation of a sequential program is given by a simple *sequential machine* and a corresponding state transition:

$$\langle e,\ E\rangle \ \hookrightarrow\ \langle e',\ E'\rangle$$

To support the scheduling infrastructure described in Section 3, we lift the sequential machine to a *vproc machine*. The vproc machine includes the state necessary for the management of a single virtual processor (*i.e.*, a stack of scheduler actions, a set of ready fibers, *etc.*) and evaluates according to a simple state transition:

$$\mathit{VP} \ \longmapsto\ \mathit{VP}'$$

Finally, a collection of vproc machines are lifted to a *multiproc machine*. The multiproc machine adds global state that is shared by all vproc machines (*e.g.*, mutable store) and evaluates according to a simple state transition:

$$\mathit{M} \ \Longrightarrow\ \mathit{M}'$$

As noted above, we formalize the operational semantics of the runtime model in terms of a continuation-passing style language, with the following syntax:

$$\begin{array}{rcl} e & ::= & \mathbf{let}\, X = Y\, \mathbf{in}\, e \\ & | & \mathbf{let}\, X = P(Y_1,\ \ldots,\ Y_n)\, \mathbf{in}\, e \\ & | & \mathbf{if}\, X\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \\ & | & \mathbf{fun}\, F(X_1,\ \ldots,\ X_n) = e_1\, \mathbf{in}\, e_2 \\ & | & F(X_1,\ \ldots,\ X_n) \\ & | & \mathbf{run}\ (X,\ F) \\ & | & \mathbf{forward}\ (X) \end{array}$$

The CPS transformation from the direct-style language of Section 3 is given in Figure 10.

Definitions for the sequential machine, vproc machine, and multiproc machine are given in Figure 11. The sequential machine is similar to the $\mathrm{C}_{cps}\mathrm{E}$ machine of Flanagan et al. (1993). A sequential machine state has two components: an active CPS language expression e and an environment E that includes bindings for all the free variables in e. The environment maps CPS language variables to machine values, which include primitive constants, (recursive) closures, tuples (of machine values). Additional machine values (vproc identifiers, group identifiers, and store locations) are treated opaquely by the sequential machine and are manipulated by the vproc and multiproc machines.

A vproc machine state *VP* includes a vproc identifier p and a sequential machine state e and E (for the evaluation of both user computations and scheduler actions), and adds a queue of ready fibers Q, a stack of scheduler actions S, and a signal mask m. The queue of ready fibers maintains a set of continuation closures that are ready to be executed on the vproc. While the queue of ready fibers of one vproc is accessible to other vprocs in the multiproc machine, we expect that a vproc may enqueue and dequeue from its own queue with a minimum of synchronization. The stack of scheduler actions supports the nesting of schedulers. Finally, the signal mask records whether signals are masked ($\mathbf{M}$), in which case preemption and asynchronous signals are disabled, or unmasked ($\mathbf{U}$), in which case preemption and asynchronous signals are enabled.

A multiproc machine state *M* includes a set of vprocs *VPS*, a global provisioning map Φ, and a global store Σ. We require that each vproc in a multiproc machine state has a unique vproc identifier. The provisioning map is used to allocate vprocs to computations. A unique group identifier g is generated for each provisioning task; the provisioning map records which vprocs have *not* been allocated to the group. We discuss the provisioning map in more detail below. Finally, the store models shared mutable state in the multiproc machine.

Figure 11 also shows the initial multiproc machine state for a program e. If the target multiproc machine has n vprocs, then one vproc begins executing the program e, while the remaining vprocs begin executing an initialization expression e_{init}. Note that each vproc is started with an empty queue of ready fibers, an empty stack of scheduler actions, and signals masked. The initialization expression is used to install a default scheduler on each vproc and unmask signals; such a scheduler will wait for work to be installed on the vproc's queue. We expect that a similar initialization expression is also incorporated into the program e. The initial provision map and store are empty.

In the sequel, we make use of the following notational conventions. $E[X \mapsto v]$ denotes the update of the environment E with a binding for X; if X is already bound in E, then the update replaces the previous binding. Similarly, $\Phi[g \mapsto \{q_1, \ldots, q_n\}]$ and $\Sigma[l \mapsto v]$ denote the update of a provision map and store, respectively. Finally, we write $\mathit{VPS} \uplus \mathit{VP}$ for $\mathit{VPS} \uplus \{\mathit{VP}\}$.

Figure 12 gives the state-transition function $\langle e,\ E\rangle \hookrightarrow \langle e',\ E'\rangle$ for the sequential machine, which is entirely standard. Note that all primitive constants and sequential primitives are evaluated by the sequential machine state transition. The evaluation of a

[2] Indeed, such a "sequential" language might even include SIMD parallelism for a single thread to operate on aggregate data.

$$
\begin{array}{rcl}
\mathcal{C}[\![\mathbf{let}\ X = Y\ \mathbf{in}\ e]\!]k & = & \mathbf{let}\ X = Y\ \mathbf{in}\ \mathcal{C}[\![e]\!]k \\
\mathcal{C}[\![X]\!]k & = & k(X) \\
\mathcal{C}[\![\mathbf{let}\ X = P(Y_1,\ \ldots,\ Y_n)\ \mathbf{in}\ e]\!]k & = & \mathbf{let}\ X = P(Y_1,\ \ldots,\ Y_n)\ \mathbf{in}\ \mathcal{C}[\![e]\!]k \\
\mathcal{C}[\![\mathbf{if}\ X\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]k & = & \mathbf{if}\ X\ \mathbf{then}\ \mathcal{C}[\![e_1]\!]k\ \mathbf{else}\ \mathcal{C}[\![e_2]\!]k \\
\mathcal{C}[\![\mathbf{fun}\ F(X_1,\ \ldots,\ X_n) = e_1\ \mathbf{in}\ e_2]\!]k & = & \mathbf{fun}\ F(K,\ X_1,\ \ldots,\ X_n) = \mathcal{C}[\![e_1]\!]K\ \mathbf{in}\ \mathcal{C}[\![e_2]\!]k \quad \text{where } K \text{ is fresh} \\
\mathcal{C}[\![\mathbf{let}\ X = F(Y_1,\ \ldots,\ Y_n)\ \mathbf{in}\ e]\!]k & = & \mathbf{fun}\ K(X) = \mathcal{C}[\![e]\!]k\ \mathbf{in}\ F(K,\ Y_1,\ \ldots,\ Y_n) \quad \text{where } K \text{ is fresh} \\
\mathcal{C}[\![F(X_1,\ \ldots,\ X_n)]\!]k & = & F(k,\ X_1,\ \ldots,\ X_n) \\
\mathcal{C}[\![\mathbf{letcont}\ K(X_1,\ \ldots,\ X_n) = e_1\ \mathbf{in}\ e_2]\!]k & = & \mathbf{fun}\ K(X_1,\ \ldots,\ X_n) = \mathcal{C}[\![e_1]\!]k\ \mathbf{in}\ \mathcal{C}[\![e_2]\!]k \\
\mathcal{C}[\![\mathbf{throw}\ K(X_1,\ \ldots,\ X_n)]\!]k & = & K(X_1,\ \ldots,\ X_n) \\
\mathcal{C}[\![\mathbf{run}\ (X,\ K)]\!]k & = & \mathbf{run}\ (X,\ K) \\
\mathcal{C}[\![\mathbf{forward}\ (X)]\!]k & = & \mathbf{forward}\ (X)
\end{array}
$$

Figure 10. The CPS translation

$$
\begin{array}{rcllll}
\langle e,\ E\rangle & \in & \mathit{SeqMach} & = & \mathit{CPS} \times \mathit{Env} & \text{(sequential machine state)} \\
e & \in & \mathit{CPS} & ::= & \ldots & \text{(CPS language expression)} \\
X, F, K & \in & \mathit{Var} & & & \text{(CPS language variable)} \\
E & \in & \mathit{Env} & = & \mathit{Var} \xrightarrow{\text{fin}} \mathit{Value} & \text{(environment)} \\
v & \in & \mathit{Value} & ::= & P_c \mid \mathit{clos} \mid \mathit{tupl} \mid \mathit{pid} \mid \mathit{gid} \mid \mathit{loc} & \text{(machine value)} \\
\mathit{clos}, f, k & \in & \mathit{Closure} & ::= & [\mathbf{clos}\ F(X_1,\ \ldots,\ X_n) = e,\ E] & \text{(allocated closure)} \\
\mathit{tupl}, t & \in & \mathit{Tuple} & ::= & [v_1,\ \ldots,\ v_n] & \text{(allocated tuple)} \\
\mathit{VP} & \in & \mathit{VirtualProc} & = & \mathit{VProcId} \times \mathit{CPS} \times \mathit{Env} \times \mathit{ReadyFibers} \times \mathit{SchedStack} \times \mathit{SigMask} & \text{(vproc machine state)} \\
\mathit{pid}, p, q & \in & \mathit{VProcId} & & & \text{(vproc identifier)} \\
Q & \in & \mathit{ReadyFibers} & = & \mathcal{P}(\mathit{Closure}) & \text{(set of ready fibers)} \\
S & \in & \mathit{SchedActs} & ::= & [] \mid f :: S & \text{(stack of scheduler actions)} \\
m & \in & \mathit{SigMask} & ::= & \mathbf{M} \mid \mathbf{U} & \text{(signal mask)} \\
M & \in & \mathit{MultiProc} & = & \mathit{VProcSet} \times \mathit{ProvisionMap} \times \mathit{Store} & \text{(multiproc machine state)} \\
\mathit{VPS} & \in & \mathit{VProcSet} & = & \mathcal{P}(\mathit{VirtualProc}) & \text{(set of virtual processors)} \\
\Phi & \in & \mathit{ProvisionMap} & = & \mathit{GroupId} \xrightarrow{\text{fin}} \mathcal{P}(\mathit{VProcId}) & \text{(global provisioning map)} \\
\mathit{gid}, g & \in & \mathit{GroupId} & & & \text{(group identifier)} \\
\Sigma & \in & \mathit{Store} & = & \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Value} & \text{(global store)} \\
\mathit{loc}, l & \in & \mathit{Loc} & & & \text{(store location)}
\end{array}
$$

Initial multiproc machine state for a program e with n vprocs and initialization expression e_{init}:

$$\langle\ \langle p_1,\ e,\ \emptyset,\ [],\ \emptyset,\ \mathbf{M}\rangle,\ \langle p_2,\ e_{init},\ \emptyset,\ \emptyset,\ [],\ \mathbf{M}\rangle,\ \ldots,\ \langle p_n,\ e_{init},\ \emptyset,\ \emptyset,\ [],\ \mathbf{M}\rangle\ ;\ \emptyset\ ;\ \emptyset\ \rangle$$

Figure 11. The abstract machine definitions

function declaration binds the function variable F to a closure, which captures the current environment. Functions are recursive, as can be seen by the behavior of the auxiliary function $\mathit{apply}(\mathit{clos},\ v_1,\ \ldots,\ v_n)$:

$$\mathit{apply}([\mathbf{clos}\ F(X_1,\ \ldots,\ X_n)\ =\ e,\ E],\ v_1,\ \ldots,\ v_n) = \left\langle e,\ E\begin{bmatrix} X_1 \mapsto v_1, \cdots, X_n \mapsto v_n, \\ F \mapsto [\mathbf{clos}\ F(X_1,\ \ldots,\ X_n)\ =\ e,\ E] \end{bmatrix}\right\rangle$$

Note that the formal arguments are bound to actual arguments and the function variable F is bound to its own closure.

More interesting is the state-transition function $\mathit{VP} \longmapsto \mathit{VP}'$ for the vproc machine, which is given in Figure 13. A vproc's primary purpose is to host a sequential machine; the first transition rule evaluates a sequential machine state transition. Note that a sequential machine state transition may be taken under an arbitrary signal mask; this corresponds to the fact that the sequential machine evaluates both user computations (when m equals $\mathbf{U}$) and scheduler actions (when m equals $\mathbf{M}$).

The next five transitions evaluate the five vproc primitives in the expected manner. The **enq** adds a continuation closure to the queue of ready fibers. The **deq** primitive returns an arbitrary element of the queue of ready fibers; hence, the formal semantics places no restrictions on the order of queue elements. Note that the **deq** primitive may only transition when the set of ready fibers is non-empty. In our implementation, attempting to dequeue from an empty set of ready fibers places the vproc in an idle state, which may affect its priority to be selected for provisioning; however, this level of detail is immaterial for the formal semantics. The **host** primitive simply returns the vproc identifier of the vproc to the underlying sequential computation. The **mask** and **unmask** primitives explicitly set the signal mask.

The vproc machine is responsible for evaluating the **run** and **forward** scheduling operations. The **run** $(F,\ K)$ operation installs a new scheduling action on the scheduler stack and begins evaluating a computation. Note that the **run** operation requires masked signals and transitions to a state with unmasked signals.

$$
\begin{array}{rcl}
\langle \mathbf{let}\, X = Y\, \mathbf{in}\, e,\ E\rangle & \hookrightarrow & \langle e,\ E[X \mapsto E(Y)]\rangle \\
\langle \mathbf{let}\, X = P_c()\, \mathbf{in}\, e,\ E\rangle & \hookrightarrow & \langle e,\ E[X \mapsto P_c]\rangle \\
\langle \mathbf{let}\, X = P_s(Y_1,\ \ldots,\ Y_n)\, \mathbf{in}\, e,\ E\rangle & \hookrightarrow & \langle e,\ E[X \mapsto \delta_s(P_s,\ E(Y_1),\ \ldots,\ E(Y_n))]\rangle \\
\multicolumn{3}{r}{\text{where}\quad \delta_s(\mathbf{eq},\ P_1^c,\ P_2^c) \;=\; \begin{cases} \mathbf{true} & \text{if } P_1^c = P_2^c \\ \mathbf{false} & \text{otherwise} \end{cases}} \\
\multicolumn{3}{r}{\delta_s(\mathbf{add},\ \mathbf{int}(n),\ \mathbf{int}(m)) \;=\; \mathbf{int}(n+m)} \\
\multicolumn{3}{r}{\delta_s(\mathbf{sub},\ \mathbf{int}(n),\ \mathbf{int}(m)) \;=\; \mathbf{int}(n+m)} \\
\multicolumn{3}{r}{\delta_s(\mathbf{alloc},\ v_1,\ \ldots,\ v_n) \;=\; [v_1,\ \ldots,\ v_n]} \\
\multicolumn{3}{r}{\delta_s(\mathbf{sel}_i,\ [v_1,\ \ldots,\ v_n]) \;=\; v_i} \\
\multicolumn{3}{r}{\ldots} \\
\langle \mathbf{if}\, X\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2,\ E\rangle & \hookrightarrow & \begin{cases} \langle e_1,\ E\rangle & \text{if } E(X) = \mathbf{true} \\ \langle e_2,\ E\rangle & \text{otherwise} \end{cases} \\
\langle \mathbf{fun}\, F(X_1,\ \ldots,\ X_n) = e_1\, \mathbf{in}\, e_2,\ E\rangle & \hookrightarrow & \langle e_2,\ E[F \mapsto [\mathbf{clos}\ F(X_1,\ \ldots,\ X_n) = e_1,\ E]]\rangle \\
\langle F(X_1,\ \ldots,\ X_n),\ E\rangle & \hookrightarrow & \langle e',\ E'\rangle \quad \text{where } \mathit{apply}(E(F),\ E(X_1),\ \ldots,\ E(X_n)) = \langle e',\ E'\rangle
\end{array}
$$

Figure 12. Sequential state transitions

This corresponds to the fact that **run** is used both to install nested scheduling actions and to initiate the evaluation of user computations.

The **forward** operation is the complement of the **run** operation. The **forward** (X) operation forwards the signal X to the current scheduler action f. Note that the **forward** operation may only transition when the stack of scheduler actions is non-empty; in practice, the initialization expression e_{init} is responsible for installing a default scheduler that always re-installs itself on the stack of scheduler actions. The **forward** operation transitions to a state with masked signals; this ensures that the scheduler action is evaluated without being preempted.

The final transition rule simulates the preemption of the vproc machine. Like the **forward** operation, a preemption invokes the current scheduler action f and transitions to a state with masked signals. The current state of the sequential machine is reified as a closure $[\mathbf{clos}\ K(\_) = e,\ E]$ and tagged with the primitive constant **preempt**; this tuple represents the signal delivered to the scheduler action.

The state-transition function $M \Longrightarrow M'$ for the multiproc machine is given in Figure 14. A multiproc's primary purpose is to host a set of vproc machines; the first transition rule evaluates a single vproc machine state transition.

As noted above, the queue of ready fibers of one vproc is accessible to other vprocs in the multiproc machine. The **enqvp** primitive allows a vproc to enqueue a closure on another vproc's queue. (This primitive also allows a vproc to enqueue on its own queue, although **enq** is expected to be a more efficient means of accomplishing the same task.)

The next three transition rules manipulate the global provision map Φ, which maps group identifiers to the set of vproc identifiers that have *not* been allocated to the group. The **newgid** primitive generates a unique group identifier for a provisioning task; the fresh group identifier ($g \notin \mathrm{dom}(\Phi)$) is returned to the underlying sequential computation and the provision map is updated to map the group identifier g to the complete set of vproc identifiers in the multiproc machine ($\{p_1, \ldots, p_n\}$). The **provision** primitive determines whether there is a vproc that has not yet been allocated to the group. If one exists, then it is returned to the underlying sequential computation in the form of a tuple, tagged with the primitive constant **some**. If none exists, then the primitive constant **none** is returned to the underlying sequential computation. In either case, the provision map is updated in the appropriate manner. Note that the semantics places no restriction on which vproc identifier is returned. In an implementation, the runtime would likely prefer idle vprocs and would attempt to balance the allocation of vprocs.

The **release** primitive returns a vproc identifier to the provision map for a specified group identifier. It is expected that invoking the **release** primitive indicates that the task for which the vproc was provisioned has completed. This ensures that the provision map approximates the computational demand placed on the vprocs; while the formal semantics does not depend on this, an implementation will most likely execute more efficiently if this assumption is met.

The last three transition rules manipulate the global store Σ. The **ref** primitive allocates and initializes a new store location. The **deref** primitive simply reads the machine value at the specified store location. Finally, the **cas** primitive performs a compare-and-swap operation. The first argument to the **cas** primitive is the store location, the second argument is the expected old value at the location, and the third argument is the new value to be stored at the location. The store is updated with the new value if and only if the location's current value is equal to the expected old value. Other atomic operations, such as test-and-set and fetch-and-add can be build using **cas**.

6. Implementation

We have implemented two versions of our runtime framework; both versions have been tested on an 8-way multiprocessor.[3] The first is a prototype that supports writing schedulers and applications in C. We used assembly routines to implement one-shot continuations (essentially like `setjmp`/`longjmp`). The purpose of this first implementation was primarily as a "proof of concept." It allowed us to test our model as a platform for writing schedulers. Most of the schedulers described in Section 4 have been prototyped in this implementation. It also demonstrates that our design carries over to implementations based on traditional stacks, although our experience has been that using heap-allocated continuations greatly simplifies the implementation.

Our second implementation is a direct implementation of the design described in Section 3. It has three major components: a compiler for a functional heterogeneous parallel-programming language, a runtime system that implements our scheduler infrastructure, and schedulers that provide the implementation of language-level parallel constructs on top of the compiler/runtime system. Since the schedulers have been discussed in detail in Sections Section 3 and Section 4, we focus our discussion on the other aspects of the system.

[3] Our test machine has four dual-core AMD Opteron 870 processors running at 2GHz; each with its own 1Mb L2 cache.

$$
\begin{array}{rcl}
\langle p,\, e,\, E,\, S,\, Q,\, m\rangle & \longmapsto & \langle p,\, e',\, E',\, S,\, Q,\, m\rangle \quad \text{where } \langle e,\, E\rangle \hookrightarrow \langle e',\, E'\rangle \\
\langle p,\, \mathbf{let}\, X = \mathbf{deq}()\, \mathbf{in}\, e,\, E,\, S,\, Q \uplus f,\, m\rangle & \longmapsto & \langle p,\, e,\, E[X \mapsto f],\, S,\, Q,\, m\rangle \\
\langle p,\, \mathbf{let}\, X = \mathbf{enq}(K)\, \mathbf{in}\, e,\, E,\, S,\, Q,\, m\rangle & \longmapsto & \langle p,\, e,\, E[X \mapsto ()],\, S \uplus E(K),\, Q,\, m\rangle \\
\langle p,\, \mathbf{let}\, X = \mathbf{host}()\, \mathbf{in}\, e,\, E,\, S,\, Q,\, m\rangle & \longmapsto & \langle p,\, e,\, E[X \mapsto p],\, S,\, Q,\, m\rangle \\
\langle p,\, \mathbf{let}\, X = \mathbf{mask}()\, \mathbf{in}\, e,\, E,\, S,\, Q,\, m\rangle & \longmapsto & \langle p,\, e,\, E[X \mapsto ()],\, S,\, Q,\, \mathbf{M}\rangle \\
\langle p,\, \mathbf{let}\, X = \mathbf{unmask}()\, \mathbf{in}\, e,\, E,\, S,\, Q,\, m\rangle & \longmapsto & \langle p,\, e,\, E[X \mapsto ()],\, S,\, Q,\, \mathbf{U}\rangle \\
\langle p,\, \mathbf{run}\ (F,\, K),\, E,\, S,\, Q,\, \mathbf{M}\rangle & \longmapsto & \langle p,\, e',\, E',\, E(F) :: S,\, Q,\, \mathbf{U}\rangle \quad \text{where } \mathit{apply}(E(K),\, ()) = \langle e',\, E'\rangle \\
\langle p,\, \mathbf{forward}\ (X),\, E,\, f :: S,\, Q,\, m\rangle & \longmapsto & \langle p,\, e',\, E',\, S,\, Q,\, \mathbf{M}\rangle \quad \text{where } \mathit{apply}(f,\, E(X)) = \langle e',\, E'\rangle \\
\langle p,\, e,\, E,\, f :: S,\, Q,\, \mathbf{U}\rangle & \longmapsto & \langle p,\, e',\, E',\, S,\, Q,\, \mathbf{M}\rangle \quad \text{where } K \text{ is fresh} \\
 & & \quad \text{and } \mathit{apply}(f,\, [\mathbf{preempt},\, [\mathbf{clos}\ K(\_) = e,\, E]]) = \langle e',\, E'\rangle
\end{array}
$$

Figure 13. VProc state transitions

$$
\langle\, \mathit{VPS} \uplus \mathit{VP}\, ;\, \Sigma\, ;\, \Phi\, \rangle \Longrightarrow \langle\, \mathit{VPS} \uplus \mathit{VP'}\, ;\, \Sigma\, ;\, \Phi\, \rangle \qquad \text{where } \mathit{VP} \longmapsto \mathit{VP'}
$$

$$
\begin{array}{l}
\langle\, \mathit{VPS} \uplus \langle p,\, \mathbf{let}\, X = \mathbf{enqvp}(Y_p, K)\, \mathbf{in}\, e,\, E,\, S,\, Q,\, m\rangle \uplus \langle p_2,\, e_2,\, E_2,\, S_2,\, Q_2,\, m_2\rangle\, ;\, \Sigma\, ;\, \Phi\, \rangle \Longrightarrow \\
\quad \langle\, \mathit{VPS} \uplus \langle p,\, e,\, E[X \mapsto ()],\, S,\, Q,\, m\rangle \uplus \langle p_2,\, e_2,\, E_2,\, S_2,\, Q_2 \uplus E(K),\, m_2\rangle\, ;\, \Sigma\, ;\, \Phi\, \rangle \qquad \text{where } E(Y_p) = p_2 \text{ and } p \neq p_2
\end{array}
$$

$$
\begin{array}{l}
\langle\, \mathit{VPS} \uplus \langle p,\, \mathbf{let}\, X = \mathbf{newgid}()\, \mathbf{in}\, e,\, E,\, S,\, Q,\, m\rangle\, ;\, \Sigma\, ;\, \Phi\, \rangle \Longrightarrow \\
\quad \langle\, \mathit{VPS} \uplus \langle p,\, e,\, E[X \mapsto g],\, S,\, Q,\, m\rangle\, ;\, \Sigma\, ;\, \Phi[g \mapsto \{p_1, \ldots, p_n\}]\, \rangle \qquad \text{where } g \notin \mathrm{dom}(\Phi)
\end{array}
$$

$$
\begin{array}{l}
\langle\, \mathit{VPS} \uplus \langle p,\, \mathbf{let}\, X = \mathbf{provision}(Y_g)\, \mathbf{in}\, e,\, E,\, S,\, Q,\, m\rangle\, ;\, \Sigma\, ;\, \Phi\, \rangle \Longrightarrow \langle\, \mathit{VPS} \uplus \langle p,\, e,\, E[X \mapsto v_r],\, S,\, Q,\, m\rangle\, ;\, \Sigma\, ;\, \Phi'\, \rangle \\
\quad \text{where } E(Y) = g \text{ and } (v_r, \Phi') = \begin{cases} ([\mathbf{some},\, q_1], \Phi[g \mapsto \{q_2, \ldots, q_m\}]) & \text{if } \Phi(g) = \{q_1, q_2, \ldots, q_m\} \\ (\mathbf{none}, \Phi) & \text{if } \Phi(g) = \emptyset \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\langle\, \mathit{VPS} \uplus \langle p,\, \mathbf{let}\, X = \mathbf{release}(Y_g, Y_p)\, \mathbf{in}\, e,\, E,\, S,\, Q,\, m\rangle\, ;\, \Sigma\, ;\, \Phi\, \rangle \Longrightarrow \\
\quad \langle\, \mathit{VPS} \uplus \langle p,\, e,\, E[X \mapsto ()],\, S,\, Q,\, m\rangle\, ;\, \Sigma\, ;\, \Phi[g \mapsto \Phi(g) \cup \{p\}]\, \rangle \qquad \text{where } E(Y_g) = g \text{ and } E(Y_p) = p
\end{array}
$$

$$
\begin{array}{l}
\langle\, \mathit{VPS} \uplus \langle p,\, \mathbf{let}\, X = \mathbf{ref}(Y)\, \mathbf{in}\, e,\, E,\, S,\, Q,\, m\rangle\, ;\, \Sigma\, ;\, \Phi\, \rangle \Longrightarrow \\
\quad \langle\, \mathit{VPS} \uplus \langle p,\, e,\, E[X \mapsto l],\, S,\, Q,\, m\rangle\, ;\, \Sigma[l \mapsto E(Y)]\, ;\, \Phi\, \rangle \qquad \text{where } l \notin \mathrm{dom}(\Sigma)
\end{array}
$$

$$
\langle\, \mathit{VPS} \uplus \langle p,\, \mathbf{let}\, X = \mathbf{deref}(Y)\, \mathbf{in}\, e,\, E,\, S,\, Q,\, m\rangle\, ;\, \Sigma\, ;\, \Phi\, \rangle \Longrightarrow \langle\, \mathit{VPS} \uplus \langle p,\, e,\, E[X \mapsto \Sigma(E(Y))],\, S,\, Q,\, m\rangle\, ;\, \Sigma\, ;\, \Phi\, \rangle
$$

$$
\begin{array}{l}
\langle\, \mathit{VPS} \uplus \langle p,\, \mathbf{let}\, X = \mathbf{cas}(Y_l, Y_o, Y_n)\, \mathbf{in}\, e,\, E,\, S,\, Q,\, m\rangle\, ;\, \Sigma\, ;\, \Phi\, \rangle \Longrightarrow \langle\, \mathit{VPS} \uplus \langle p,\, e,\, E[X \mapsto v_r],\, S,\, Q,\, m\rangle\, ;\, \Sigma'\, ;\, \Phi\, \rangle \\
\quad \text{where } l = E(Y_l) \text{ and } (v_r, \Sigma') = \begin{cases} (\mathbf{false}, \Sigma) & \text{if } E(Y_o) \neq \Sigma(l) \\ (\mathbf{true}, \Sigma[l \mapsto E(Y_n)]) & \text{if } E(Y_o) = \Sigma(l) \end{cases}
\end{array}
$$

Figure 14. Multiproc state transitions

6.1 The compiler

As is common, our compiler is structured as a sequence of intermediate languages (IRs). The last three IRs prior to code generation are most relevant to this paper. We have a direct-style IR similar to the one presented in Figure 1, which is CPS converted to a higher-order CPS representation. The CPS representation is, in turn, closure converted to a first-order control-flow graph representation from which we generate x86-64 assembly code.

The direct-style representation is the workhorse of our compiler. It is the IR that is used to express optimizations such as fusion (Chakravarty et al. 2007) and analysis and specialization of concurrency primitives (Carlsson et al. 2006; Reppy and Xiao 2007). As a final stage, we expand the scheduling operations (*e.g.*, **enq**, **deq**, **run**, *etc.* into lower-level operations that directly manipulate the runtime system data structures.[4] For example, the expression **run** $(K_1,\, K_2)$ is expanded to the following term that first pushes the signal-action continuation K_1 on the action stack, clears the atomic flag (recall that **run** assumes that signals are masked, and lastly transfers control to K_2:

$$
\begin{array}{l}
\mathbf{let}\ \mathit{VP} = \mathbf{host}\ \mathbf{in} \\
\mathbf{let}\ X = \mathbf{load}(\mathit{VP}, \mathit{StkOffset})\ \mathbf{in} \\
\mathbf{let}\ Y = \mathbf{alloc}(K_1, X)\ \mathbf{in} \\
\mathbf{let}\ () = \mathbf{store}(\mathit{VP}, \mathit{StkOffset}, Y)\ \mathbf{in} \\
\mathbf{let}\ () = \mathbf{store}(\mathit{VP}, \mathit{AtomicOffset}, \mathbf{false})\ \mathbf{in} \\
\quad \mathbf{throw} K_2()
\end{array}
$$

By expanding these operations into lower-level primitives, we simplify the CPS IR and simplify the implementation of the primitives. Some operations are partially supported by the compiler and partially by the runtime. For example, the fast path in **deq** (*i.e.*, when the primary queue is non-empty) is implemented by expansion, while the slow path is implemented by a call to a runtime-system function.

The translation from direct style to CPS eliminates the special handling of continuations and makes control flow explicit. We use the Danvy-Filinski CPS transformation (Danvy and Filinski 1992), but our implementation is simplified by the fact that we start from a normalized direct-style representation.

Lastly, the transformation from CPS to CFG handles the heap allocation of first-class continuations. We analyze the control flow

[4] Our implementation differs from the formal description of Section 5, where the scheduling primitives are preserved into the CPS representation.

of the CFG and specialize calling conventions, and we add heap-allocation checks to the IR. From this CFG we generate assembly code.

6.2 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). 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, $k = 20$). The runtime representation of a vproc is stored in the base of this memory region and the remaining space is used as the vproc-local heap. The **host** primitive is implemented by clearing the low k bits of the allocation pointer.

One important design principle that we follow is minimizing sharing of mutable state between vprocs. We distinguish between three types of vproc state: thread-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 state, such as machine registers, is protected by limiting context switches to "safe-points" (*i.e.*, heap-limit checks).

Scheduling queues Each vproc has two scheduling queues: a primary queue that is vproc local and a secondary queue that is globally accessible. In our framework we distinguish between enqueueing a fiber on the host vproc's scheduling queue (**enq**) and enqueueing it on a remote vproc's queue (**enqvp**). This distinction allows us to keep operations on the primary queue local, which means we can avoid expensive synchronization. The secondary queue is protected with traditional locking and is accessed periodically to move fiber down to the primary queue (or when the primary queue is empty).

Preemption We implement preemption by synchronizing preempt signals with garbage-collection tests as is done by Reppy (1990). 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 wrapped in the **preempt** signal and passed to the topmost signal-action handler. The one downside to this approach is that the compiler must add heap-limit checks to non-allocating loops. An alternative that avoids this extra overhead is to use the atomic-heap transactions of Shivers et al. (1999), but that technique requires substantial compiler support.

Startup and shutdown One challenging part of the implementation is initialization and clean termination. When a program initially starts running, it is single threaded and running on a single vproc. Before executing the user code, it enqueues a thread on every vproc that installs the default scheduler. After initialization, each of the vprocs, except the initial one, will be idle and waiting for a fiber to be added to their secondary queues. If at any point, all of the vprocs go idle, then the system shuts down.

6.3 Implementation status

Our implementation is still a work in progress. So far, our main focus has been on the basic runtime mechanisms and compiler support for schedulers. The runtime system is largely complete, although we have not yet done any performance tuning. Because of this focus, we began the implementation of the compiler from the backend and have been working our way forward. As of this writing (April 2007), we have the basics of the direct-style, CPS, and CFG IRs implemented and have compiled and run parallel workloads written in the IR. We have not implemented any significant optimizations yet, so we are not in a position to benchmark performance.

7. Related work

Our runtime model and scheduling infrastructure is inspired by Shivers' proposal for exposing hardware concurrency using continuations (Shivers 1997). We have extended Shivers' proposal to support nested schedulers and multiple processors.

STING (Jagannathan and Philbin 1992) is a parallel dialect of SCHEME that, like our runtime model, aims to support multiple parallel-language constructs in a unified framework. The language contains three notions of process abstraction: lightweight threads, thread groups, and abstract processors. Application programmers can implement separate scheduling policies that define migration and thread election for thread groups and policies that share an abstract processor amongst different thread groups. The programmer implements these policies by supplying a collection of functions for handling scheduling events. STING's three layers of process abstraction and comparatively heavyweight mechanism for implementing scheduling policies contrasts with our approach, which favors minimal process abstractions and a unified infrastructure for implementing schedulers.

Engines are an elegant mechanism for supporting timed preemption (Haynes and Friedman 1984), and can be used to implement proportional-share thread schedulers and to simulate parallelism for multiple threads. Engines also have nested variants that support more complicated scheduling patterns used by the nested `parallel-or` operator and the self-virtualizing operating system (Dybvig and Hieb 1989). Nested engines, although somewhat complicated to implement with first-class continuations alone, have a straightforward implementation using operators in our runtime model (Rainey 2007).

The basic design of our scheduling framework was sketched in an earlier workshop paper (Fluet et al. 2007), but this paper greatly expands and improves on that earlier work. First, this paper presents a formal specification of the scheduler model. Second, the earlier paper only presented one example scheduler (the data-parallel scheduler of Section 4.2), whereas this paper presents several others, such as work stealing and lazy-task creation. We have also extended our model to handle speculative parallelism. The other major difference is that the description in the previous paper is a design proposal, whereas we now have both a compiler and runtime implementation of the framework, which is described in Section 6. There are also a number of minor tweaks to the design based on our experiences writing schedulers.

8. Conclusion

This paper describes the design and implementation of a runtime framework for heterogeneous parallel languages. We have shown through several examples that this framework is powerful enough to support a wide variety of scheduling policies and parallel-programming mechanisms in a modular and nestable way. We have presented a formal model for this framework that is architecture independent and that specifies precise guidelines for implementation. We have implemented the framework in a combination of a compiler and runtime system. We plan to use this implementation to support a rich surface language for heterogeneous parallel programming.

In addition to continuing our implementation efforts, there are several avenues for future work. The scheduling infrastructure, as it is currently implemented, is only accessible to compiler writers. Increasing demands for application-specific scheduling policies,

however, motivates exposing parts of our infrastructure to the surface language. One clean way to support user-customizable schedulers is to provide programmers with a domain-specific language for schedulers. BOSSA is one such language for developing nested, uniprocessor schedulers in LINUX (Muller et al. 2005). An interesting direction for future work is to target a BOSSA-like language to our runtime framework.

References

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

Guy E. Blelloch. Programming parallel algorithms. *CACM*, 39(3): 85–97, March 1996.

Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ICFP '96*, pages 213–225, New York, NY, May 1996. ACM.

Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *JPDC*, 21(1):4–14, 1994.

Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81*, pages 187–194, New York, NY, October 1981. ACM.

Martin Carlisle, Laurie J. Hendren, Anne Rogers, and John Reppy. Supporting SPMD execution for dynamic data structures. *ACM TOPLAS*, 17(2):233–263, March 1995.

Richard Carlsson, Konstantinos Sagonas, and Jesper Wilhelmsson. Message analysis for concurrent programs using message passing. *ACM TOPLAS*, 28(4):715–746, July 2006.

Manuel M. T. Chakravarty and Gabriele Keller. More types for nested data parallel programming. In *ICFP '00*, pages 94–105, New York, NY, September 2000. ACM.

Manuel M. T. Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Wolf Pfannenstiel. Nepal – Nested Data Parallelism in Haskell. In *Euro-Par '01*, volume 2150 of *LNCS*, pages 524–534, New York, NY, August 2001. Springer-Verlag.

Manuel M. T. Chakravarty, Roman Leschchinski, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: A status report. In *DAMP '07*, pages 10–18, New York, NY, January 2007. ACM.

Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *MSCS*, 2(4):361–391, 1992.

R. Kent Dybvig and Robert Hieb. Engines from continuations. *Comp. Lang.*, 14(2):109–123, 1989.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI '93*, pages 237–247, New York, NY, June 1993. ACM.

Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: A heterogeneous parallel language. In *DAMP '07*, pages 37–44, New York, NY, January 2007. ACM.

Glasgow Haskell Compiler (Version 6.6), 2006. `http://www.haskell.org/ghc`.

Robert H. Halstead Jr. Implementation of multilisp: Lisp on a multiprocessor. In *LFP '84*, pages 9–17, New York, NY, August 1984. ACM.

Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05*, pages 48–60, New York, NY, June 2005. ACM.

Christopher T. Haynes and Daniel P. Friedman. Engines build process abstractions. In *LFP '84*, pages 18–24, New York, NY, August 1984. ACM.

Suresh Jagannathan and Jim Philbin. A customizable substrate for concurrent languages. In *PLDI '92*, pages 55–81, New York, NY, June 1992. ACM.

Roman Leshchinskiy, Manuel M. T. Chakravarty, and Gabriele Keller. Higher order flattening. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, *ICCS '06*, number 3992 in LNCS, pages 920–928, New York, NY, May 2006. Springer-Verlag.

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

Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *LFP '90*, pages 185–197, New York, NY, June 1990. ACM.

Gilles Muller, Julia L. Lawall, and Hervé Duchesne. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *HASE '05*, pages 56–65, October 2005.

Randy B. Osborne. Speculative computation in multilisp. In *LFP '90*, pages 198–208, New York, NY, June 1990. ACM.

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL '96*, pages 295–308, New York, NY, January 1996. ACM.

Mike Rainey. The Manticore runtime model. Master's thesis, University of Chicago, January 2007.

Norman Ramsey. Concurrent programming in ML. Technical Report CS-TR-262-90, Dept. of C.S., Princeton University, April 1990.

John Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, 2001.

John Reppy and Yingqi Xiao. Specialization of CML message-passing primitives. In *POPL '07*, pages 315–326, New York, NY, January 2007. ACM.

John H. Reppy. CML: A higher-order concurrent language. In *PLDI '91*, pages 293–305, New York, NY, June 1991. ACM.

John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

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

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

Olin Shivers, James W. Clark, and Roland McGrath. Atomic heap transactions and fine-grain interrupts. In *ICFP '99*, pages 48–59, New York, NY, September 1999. ACM.

Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + strategy = parallelism. *JFP*, 8(1):23–60, January 1998.

Mark T. Vandevoorde and Eric S. Roberts. Workcrews: an abstraction for controlling parallelism. *IJPP*, 17(4):347–366, August 1988.

Mitchell Wand. Continuation-based multiprocessing. In *LFP '80*, pages 19–28, New York, NY, August 1980. ACM.