THE UNIVERSITY OF CHICAGO

# The Manticore runtime model

A MASTER'S PAPER SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
MICHAEL RAINEY

CHICAGO, ILLINOIS
DRAFT AS OF JANUARY 30, 2007

# TABLE OF CONTENTS

# LIST OF FIGURES

# SECTION 1
# INTRODUCTION

New multicore processor architectures are characterized by their multiple levels of parallelism. At the lowest level, SIMD instructions can fetch a set of operands and perform an operation on them in a single CPU cycle. At the next level, multicore architectures often include simultaneous multi-threading (SMT), which allows multiple threads (usually two) to execute different instructions in the same CPU cycle. At the highest levels of parallelism, multicore architectures have multiple processor cores on a single chip and even multiple chips. Programs, which now must map onto this hardware often, exhibit parallelism at different granularities, roughly categorized as either fine- or coarse-grained. We use the term *heterogeneous parallelism* to refer to a combination of parallelism in multiple hardware levels and at different granularities in programs.

Programming languages need a variety of mechanisms for heterogeneous parallelism, both because applications exhibit parallelism at different granularities and because, for maximum performance, multicore hardware requires parallelism at multiple levels. For example, consider a networked flight simulator. Such an application might use data-parallel computations for particle systems [Ree83] to model natural phenomena such as rain, fog, and clouds. At the same time, it might use parallel threads to preload terrain data and compute level-of-detail refinements, and use SIMD parallelism in its physics simulations. Finally, it might also use explicit concurrency for user interface and network components. Programming these heterogeneous applications will be challenging without language support for heterogeneous parallelism.

Manticore is a research project that explores the design and implementation of programming languages that support heterogeneous parallelism. This project is motivated by the belief that existing general-purpose languages do not provide adequate support for parallel programming, while existing parallel languages, which are largely targeted to scientific applications, do not provide adequate support for general purpose programming. For instance, Erlang [Hed98] has parallel threads with message passing, but has no built-in mechanisms for fine-grained parallelism. NESL [BCH+94, Ble96, BG96] and Nepal [CK00, CKLP01, LCK06], on the other hand, have data-parallel arrays, but support neither coarse-grained parallelism nor explicit threads for systems programming.

The Manticore project proposes a general-purpose language for programming with multiple parallel constructs. The Manticore language is statically-typed and functional, and is rooted in the OCaml [Ler00] and SML [AM91] family of languages. For its initial design, Manticore supports mechanisms for explicit concurrency and coarse-grained parallelism and mechanisms for fine-grained parallelism. Manticore's concurrency mechanisms are based on a parallel version of CML [Rep99], which supports threads and synchronous message passing. Manticore's mechanisms for fine-grained parallelism are based on data-parallel arrays, similar to those in NESL [BCH+94, Ble96, BG96] and Nepal [CK00, CKLP01, LCK06]. Manticore will also include mechanisms for mid-grained parallelism that are based on futures with work stealing [BS81, RHH84, MKH90, CHRR95, BL99].

Supporting heterogeneous parallelism in an evolving language poses new problems for the compiler and runtime system, as different parallel constructs have disparate demands for management and synchronization. For instance, data-parallel computations need a mechanism to keep processors active and to throttle parallelism when it is overabundant, *e.g.*, workcrews [VR88]. On the other hand, threads need load balancing to encourage parallelism, *e.g.*, work stealing [CR95, BL99]. Threads also need timed preemption to simulate concurrency for GUI and network applications. When threads and data-parallel constructs coexist, the language must provide mechanisms for their scheduling policies to coordinate. For example, suppose a thread launches a data-parallel computation across several processors. Some of the data-parallel processes might be subject to timed preemption if they share a processor with other threads.

One can view heterogeneous parallel languages as consisting of two distinct sublanguages: the *computation language* manipulates data for a single thread, while the *coordination language* manages and synchronizes parallel processes. Such a language must encode specialized policies for scheduling parallel computations across processors and for scheduling multiple processes on an individual processor. The coordination language must also be flexible enough to experiment with alternative management policies and rapidly integrate new parallel constructs with new scheduling policies.

The main contribution of this research is a substrate for the Manticore coordination language, called the *runtime model*. The model consists of essential abstractions for hardware concurrency and a collection of operations for managing and synchronizing processes. Using the model, the compiler writer can implement scheduling policies for a wide variety of parallel constructs. But in contrast to other heterogeneous languages, the model supports nested and heterogeneous schedulers [Reg01]. This rich variety of schedulers lets different policies coordinate and share hardware resources in a unified framework. To demonstrate the feasibility of the model, I present several schedulers, including variants for data-parallel arrays and threads. I then develop a formal semantics that sets exact requirements for the implementation, but also acts as an API for rapidly developing new schedulers.

The rest of the paper proceeds as follows. Section 2 introduces the runtime model. Section 3 presents runtime-model support for Manticore's parallel constructs, CML-style threads, and data-parallel arrays. Section 4 implements various time-sharing schedulers to demonstrate how more sophisticated policies are encoded in the model. Section 5 describes general guidelines for programming schedulers. Section 6 presents a formal semantics of the runtime model. Section 7 draws comparisons to related work, and Section 8 concludes with a summary of results and directions for continuing research.

# SECTION 2
# THE MANTICORE RUNTIME MODEL

The computation sublanguage of Manticore is a mutation-free version of SML, a strict, statically-typed applicative language.

The coordination sublanguage of Manticore, the runtime model, consists of three components.

- *First-class continuations* [Wan80, Rey93] represent suspended computations.
- Three essential abstractions express concurrency. *Fibers* are bare threads of control. Language-level *threads* are named collections of fibers. *Virtual processors* (vprocs) represent the physical processors.
- A collection of primitive operations builds a low-level infrastructure for programming schedulers. The infrastructure is inspired by Shivers' proposal for exposing hardware concurrency using continuations [Shi97], but is extended to support nested schedulers and multiple processors.

The model is intended to be a part of the compiler's intermediate representation that immediately precedes CPS conversion. At this stage, the compiler expands parallel constructs in the surface language into the scheduling operations in the runtime model. The compiler infrastructure, in turn, either hooks scheduling operations into runtime system code written in C, or expands the operations into machine code directly.

Although the presentation uses SML notation for convenience, the model is a low-level substrate for schedulers. The compiler writer is responsible for either hooking scheduling code into the compiler, or exposing mechanisms to the surface language to allow programmers to encode their own policies.

## 2.1 Continuations

Continuations are a well-known language-level mechanism for expressing concurrency [Wan80, HFW84, Rep89, Shi97]. Continuations come in a number of different strengths or flavors.

1. *First-class* continuations, such as those provided by SCHEME and SML/NJ, have unconstrained lifetimes and may be used more than once. They are easily implemented in a continuation-passing style compiler using heap-allocated continuations [App92], but map poorly onto stack-based implementations.
2. *One-shot* continuations [BWD96] have unconstrained lifetimes, but may only be used once. The one-shot restriction makes these more amenable for stack-based implementations, but their implementation is still complicated. In practice, most concurrency operations (but not thread creation) can be implemented using one-shot continuations.

3. *Escaping* continuations have a scope-limited lifetime and can only be used once, but they also can be used to implement many concurrency operations [RP00, FR02]. These continuations have a very lightweight implementation in a stack-based framework; they are essentially equivalent to the C library's `setjmp`/`longjmp` operations.

The runtime model uses first-class, heap-allocated continuations *à la* SML/NJ [App92] to express concurrency operations. The `callcc` operator captures the current continuation, and applies it to its function-value argument. The `throw` operator transfers control to the continuation.

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
val throw : 'a cont -> 'a -> 'b
```

Although heap-allocated continuations impose some extra overhead (mostly increased GC load) for sequential execution, they provide a number of advantages for concurrency:

- Creating a continuation merely requires allocating a heap object, so it is fast and imposes little space overhead ($< 100$ bytes).
- Since continuations are *values*, many nasty race conditions in the scheduler can be avoided [FR02].
- Heap-allocated first-class continuations do not have the lifetime limitations of escaping and one-shot continuations, so we avoid prematurely restricting the expressiveness of our intermediate representation.
- By inlining concurrency operations, the compiler can optimize them based on their context of use [FR02].

## 2.2 Fibers, threads, and virtual processors

The runtime model has three distinct notions of process abstraction. At the lowest level, a *fiber* is a basic thread of control. A suspended fiber is represented as a unit continuation.

```
type fiber = unit cont
```

The `fiber` operator takes a function value, and creates a suspended fiber that, when run, calls the function before stopping.

```
val fiber : (unit -> unit) -> fiber
```

The following code implements this operator using first-class continuations.

```
fun fiber f = callcc (fn k => (
                        callcc (fn k' => throw k k');
                        f ();
                        stop () )
```

A surface-language *thread* (*i.e.*, one created by `spawn` in CML) is initially mapped to a fiber paired with a unique thread ID (`tid`).

```
type thread = tid * fiber
```

In addition to having an ID, threads are different from fibers in that they may create multiple fibers to run data-parallel computations. Thus at runtime, a thread consists of a `tid` and one or more fibers.

Lastly, a *virtual processor* (vproc) is an abstraction of a hardware processor. The runtime model represents a vproc with the `vproc` type. Vprocs run at most one fiber at a time, and furthermore are the only means for running fibers. The currently running fiber's vproc is called its *host vproc*, and is obtained by the `hostVP` operator.

```
val hostVP : unit -> vproc
```

The runtime model provides a mechanism, called *provision*, for assigning vprocs to threads. When applied to the desired number of processors, `provision` returns a list of vprocs that are available for a thread (which may be fewer than the number requested). The complementary `release` operator informs the runtime system that a thread is finished with some vprocs.

```
val provision : int -> vproc list
val release   : vproc list -> unit
```

To balance work evenly between threads, the runtime system never assigns a vproc to a given thread twice. The runtime system also considers the load and possibly even the physical-processor topology when assigning vprocs.

The runtime model leaves the representation of vprocs abstract. A runtime-system implementation can, for example, use a POSIX thread to host each vproc. In this case, there would be as many vprocs as there are hardware processors. For some operating systems, this approach has the disadvantage that vprocs are at the mercy of the kernel scheduler, and thus lack control of which physical processor is the actual host. Many operating systems, however, allow applications to explicitly state which hardware processors a given thread can run on. For example, the Linux 2.6 kernel supports the function `sched_setaffinity`, Solaris supports `processor_bind`, and HPUX/TRU64 supports `pthread_processor_bind_np`.

Figure 2.2 shows a possible configuration of the runtime model. Three threads are running on four vprocs. The first two threads each contain multiple fibers running in parallel. In the Manticore language, these threads would represent two different data-parallel computations. The other thread is a common language-level thread, and thus contains a single fiber. All of these threads are mapped onto the vprocs by a cooperative scheduling mechanism. The multi-fiber threads share the second vproc; the language-level thread and one of the multi-fiber threads also share a vproc. The next section describes the infrastructure for supporting this cooperative scheduling.

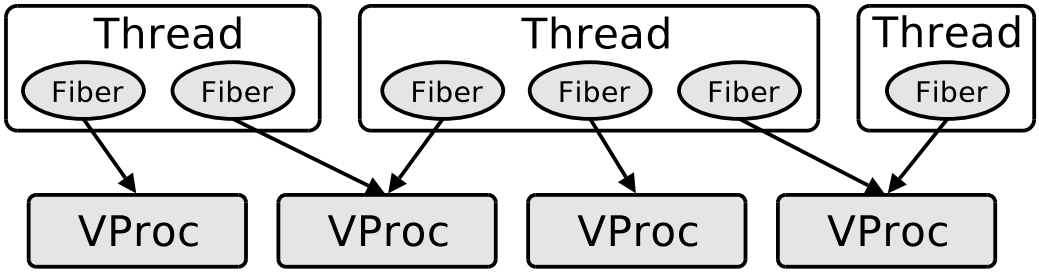


Figure 2.1: Multiple threads running multiple fibers.

## 2.3 Scheduling infrastructure

The scheduling infrastructure is a low-level substrate for writing schedulers. It directly encodes all scheduling that occurs at run time, and does not rely on external or fixed schedulers. To support a wide variety of schedulers, the infrastructure provides mechanisms that divide a vproc's time among multiple fibers and mechanisms that divide and synchronize parallel computations among multiple vprocs. This section focuses on the former mechanisms, and Section 3 describes the latter mechanisms in greater detail.

A *scheduler action* is a function that implements context switching on a vproc. By defining different functions, it is possible to implement different scheduling policies. Scheduler actions have the type

```
datatype signal = STOP | PREEMPT of fiber
type action = signal -> void
```

where the `signal` type represents the events that are handled by schedulers. Here there are only signals for fiber termination and preemption, but this type could be extended to model other forms of asynchronous events, such as asynchronous exceptions [MJMR01]. A scheduler action should never return, so its result type can be the `void` type that contains no values.

The runtime model supports nesting of schedulers (*e.g.*, a data-parallel scheduler runs on top of a language-level thread scheduler) by giving each vproc a stack of scheduler actions. The top of a vproc's stack is called the *parent* scheduler action; it represents the current scheduler for that vproc. When a vproc receives a signal, it handles it by popping the parent from the stack and applying it to the signal. Figure 2.3 gives a pictorial description of operations on a vproc's action stack.

There are two operators that scheduling code can use to affect a vproc's scheduler stack directly.

```
val run     : (action * fiber) -> void
val forward : signal -> void
```

The `run` operator initiates the execution of a fiber. It takes a scheduler action that implements the scheduling policy for the fiber and the fiber itself, pushes the action on the

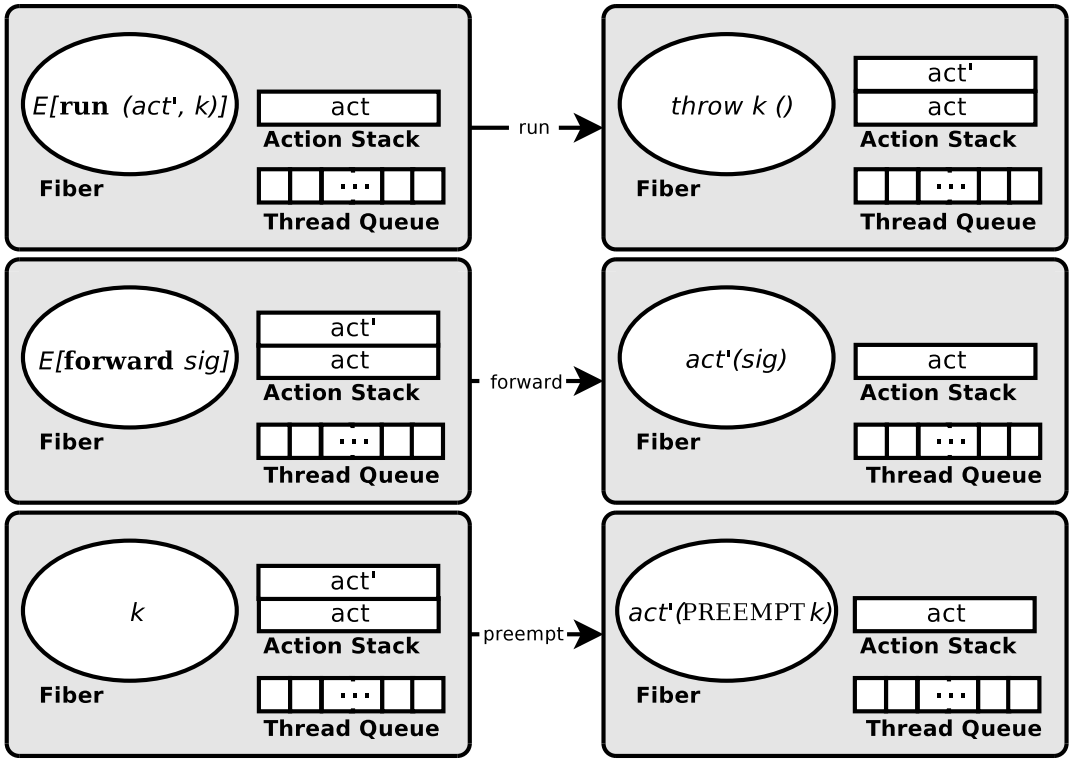


Figure 2.2: How `run`, `forward`, and preemption affect a vproc.

---

scheduler-action stack, and then runs the fiber. The `forward` operator delivers a signal to a vproc; the expression "`forward` *sig*" sends *sig* to the host vproc. The runtime model uses this operator to implement the `stop` function for fiber termination.

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

Preemption is generated by a hardware event, such as as timer interrupt. When a vproc is preempted, it reifies the continuation of the running fiber *k*, and then executes "`preempt` *k*," where the `preempt` function is defined as

```
fun preempt k = forward (PREEMPT k)
```

The vproc then handles the signal as usual; it applies the parent scheduler action to the preemption signal. Using `preempt`, it is possible to define a function that yields the vproc to the parent scheduler.

```
fun yield () = callcc (fn k => preempt k)
```

A scheduler can use this function to pass a preemption signal to its parent, making it useful later for programming nested schedulers.

In addition to the scheduler stack, each vproc has a queue of ready threads. The runtime model uses the queue to schedule threads and as a mechanism to dispatch threads on multiple vprocs. There are three operations on a vproc's queue:

```
val enqueue : thread -> unit
val dequeue : unit -> thread
val enqueueOnProc : (vproc * thread) -> unit
```

The first two operators apply to the host vproc's queue. The second operator blocks a vproc on an empty queue. It can be unblocked when another vproc puts a thread on its queue. The third operator puts a thread on another vproc's queue, and is the only mechanism for parallel dispatch in the runtime model.

To avoid the danger of asynchronous preemption while scheduling code is running, the `forward` operator masks preemption and the `run` operator unmasks preemption on the host vproc. There are also operations for explicitly masking and unmasking preemption on the host vproc.

```
val mask   : unit -> unit
val unmask : unit -> unit
```

Schedulers in this infrastructure must at least implement a scheduler action to be complete. Often though, implementations will need to export a spawn function that can add work to the scheduler's ready queue. Applications can use this function as an entry point to the scheduler. The thread scheduler in Section 2.4 gives an example of a minimal but complete implementation that also provides a spawn function.

## 2.4 Scheduling threads

Thread scheduling is round-robin, and is implemented by the scheduler action `switch`. When a thread terminates, the scheduler action simply runs the next thread. When the running thread is preempted, the scheduler action puts it on the ready queue, and then runs the next thread. The `dispatch` function picks a thread from the ready queue, sets the thread ID, and then runs the thread using `switch` as its scheduler. This scheduler action is always present at the bottom of each vproc's stack, thus precluding a program from popping an empty scheduler action stack.

```
fun switch STOP = dispatch ()
  | switch (PREEMPT k) = (
    enqueue (getTid (), k);
    dispatch () )
and dispatch () =
    let val (tid, k) = dequeue ()
    in
        setTid tid;
        run (switch, k)
    end
```

Applications can initialize threads by calling the `spawn` function, which takes a function value `f`, enqueues its fiber on the host vproc, and returns a fresh thread ID `tid`.

```
fun spawn (f : unit -> unit) =
    let val tid = newTid ()
    in
        enqueue (tid, fiber f);
        tid
    end
```

# SECTION 3
# SUPPORTING HETEROGENEOUS PARALLELISM

This section demonstrates that the runtime model supports different scheduling policies for the parallel constructs in Manticore, and is flexible enough to coordinate those policies on shared processors. The work-stealing scheduler in Section 3.1 introduces explicit parallelism for CML-style threads. The data-parallel scheduler in Section 3.2 supports implicit parallelism in the form of flattened data-parallelism.

## 3.1 Work-stealing scheduler

Surface-language threads created by the `spawn` operator are not initially run in parallel. In some cases, this choice has advantages such as reduced communication and cache affinity between threads. But for optimal performance on multicore architectures, the system must balance work continually amongst all system processors. A common technique that addresses this problem is work stealing [BS81, RHH84, MKH90, CHRR95, BL99], in which a processor (the thief) that is idle picks another processor (the victim) from which to steal work.

The work stealing implementation begins by allocating a group of processors, with one for each worker.

```
val vprocs = provision nVPs
```

Each worker also has its own ready queue that is shared with the other schedulers. The operations are atomic to prevent race conditions.

```
type 'a queue
val mkQueueAtm : unit -> 'a queue
val enqueueAtm : ('a queue * 'a) -> unit
val dequeueAtm : 'a queue -> 'a option
```

References to the queues are stored in a lookup table `qs` that every scheduler can access.

```
val qs = Vector.tabulate (nVPs, fn _ => mkQueueAtm ())
```

When a scheduler goes idle, it tries to run the next thread from its queue. If the local queue is empty, the scheduler picks a victim to fill its queue. Otherwise, the scheduler runs the next thread.

```
fun runNext () =
    (case dequeueAtm q
      of NONE => pickVictim ()
       | SOME thd => runWS thd
    (* esac *))
```

The thief scheduler picks the victim's queue at random, and tries to steal one of its threads. If the queue is empty, the thief waits for a while before trying again; a tight loop might slow down other running processors. Otherwise, the thief runs the stolen thread.

```
fun pickVictim () =
    let val victQ = Vector.sub (qs, randInt () mod nVPs)
    in
        case dequeueAtm victQ
         of NONE => ( backOff (); pickVictim () )
          | SOME thd => runWS thd
    end (* pickVictim *)
```

The scheduler initializes itself by applying the `initOn` function to each of the assigned vprocs, giving each a its own index `i` into the queue vector `qs`.

```
foldl (fn (vproc, i) => (initOn (vproc, i); i+1)) 0 vprocs;
```

This initialization function creates a fiber that, when enqueued on a vproc, immediately invokes the scheduler action `wsSwitch`.

```
fun initOn (vproc, i) =
    let val q = Vector.sub (qs, i)
        fun doit () = run (wsSwitch q, fiber stop)
    in
        enqueueOnProc (vproc, (getTid (), fiber doit))
    end
```

The remaining discussion presents the scheduler action `wsSwitch` informally; Appendix A contains the full code for the work-stealing scheduler. The scheduler action handles the `STOP` signal by simply running the next thread. When the scheduler handles the `PREEMPT` signal, it places the running thread back on the queue, and then yields control to its parent. This practice of yielding the vproc lets the work-stealing scheduler coordinate with other scheduling policies, *e.g.* the thread scheduler, much like cooperative threads. Even though the vproc is temporarily unavailable, other schedulers in the group that need work can resume the pre-empted thread immediately. Once the original scheduler is resumed from the preemption, it tries to find new work.

## 3.2 Data-parallel scheduler

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 [VR88]. A workcrew consists of some number of workers that each run on a separate processor. When idle, a worker picks the next *job* from the global pool of work. Our runtime model implements a workcrew with the `forkN` function.

```
val forkN : {nVProcs : int, nJobs : int, job : int -> unit}
            -> unit
```

where `nVProcs` is the maximum number of vprocs to employ, `nJobs` is the number of parallel jobs, and `job` is the function to be applied in parallel; a job takes an integer between `1` and `nJobs` that indicates its place in the workcrew. This section first presents

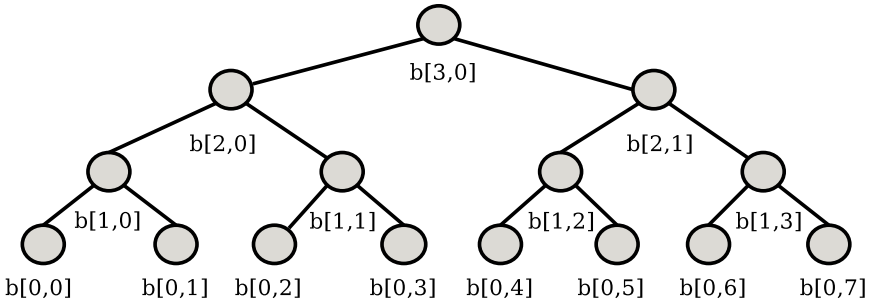


Figure 3.1: The balanced binary tree `b`.

the parallel prefix sums algorithm as an example use of the `forkN` scheduler, and then describes the scheduler in detail.

The parallel prefix sums algorithm illustrates a basic, yet important, data-parallel algorithm. In a simplified formulation, the algorithm takes an array $a$ of $n = 2^k$ integers, and stores the prefix sums in an array $c$ defined by

$$c(i) = \Sigma_{j=1}^{i} a(j)$$

for $1 \leq i \leq n$. The data-parallel algorithm proceeds in two phases. First, it copies the elements of the source array `a` onto the leaves of a balanced binary tree `b`, which the algorithm represents by a upper-triangular matrix.[1] Figure 3.1 gives an example of `b` when $n = 8$. The algorithm then computes the elements of `b` in parallel; it adds sibling nodes, storing their sums into their parent nodes. The algorithm continues up the tree in this way until reaching the root node. The expression `(sumLoop 1)` computes the first phase of the algorithm.

```
fun sumLoop i = if (i <= k) then
    let fun job j = update (b, i, j, sub (b, i-1, 2*j) +
                                     sub (b, i-1, 2*j+1))
    in
        forkN {nVProcs=nVPs, nJobs=shiftR (n, i), job=job};
        sumLoop (i+1)
    end
              else ()
```

When the phase is finished, the root node of `b` contains the sum of all `a`'s elements $c(n)$, and the internal nodes store intermediate results. The second phase proceeds similarly, using the elements of `b` to compute the prefix sums. This phase proceeds from the root

1. In the example, the expression `sub (arr, i, j)` returns the element at the `i`th row and `j`th column of the array `arr`, and the expression `update (arr, i, j, x)` changes the element at that location to `x`.

```
fun prefixSums (a, n, k) =   (* (nRows a) = n = 2^k *)
 let val (b, c) = (array (k+1, n, 0), array (k+1, n, 0))
     fun sumLoop i = if (i <= k) then
         let fun job j = update (b, i, j, sub (b, i-1, 2*j) +
                                          sub (b, i-1, 2*j+1))
         in
             forkN {nVProcs=nVPs, nJobs=shiftR (n, i), job=job};
             sumLoop (i+1)
         end
                     else ()
     fun evenLoop i =
         let fun job j = update (c, i, j*2, sub (c, i+1, j))
         in
             forkN {nVProcs=nVPs, nJobs=shiftR (n, i+1), job=job}
         end
     fun oddLoop i =
         let fun job j = update (c, i, j*2+1, sub (c, i+1, j) +
                                              sub (b, i, j*2+1))
         in
             forkN {nVProcs=nVPs, nJobs=shiftR (n, i+1), job=job}
         end
     fun prefixLoop i = if (i >= 0) then (
         evenLoop i;  update (c, i, 1, sub (b, i, 1));  oddLoop i;
         prefixLoop (i-1) )
                        else ()
 in
     copyFirstRow (b, a);  sumLoop 1;  prefixLoop (k-1);
     c (* The first row of c contains the prefix sums. *)
 end (* prefixSums *)
```

Figure 3.2: Data-parallel prefix sums algorithm.

down to the leaves, and stores the resulting prefix sums in the array `c`. The complete code is given in Figure 3.2.

The code in Figure 3.3 implements the `forkN` scheduler. Similar to the work-stealing scheduler, the data-parallel scheduler initializes itself by allocating a group of vprocs, and then applying each to its `initOn` function. This function enqueues on its vproc a fiber that invokes the scheduler action `dlpSwitch`.

Once initialized on a vproc, the scheduler action acquires jobs from the work pool, and handles preemptions. The `STOP` signal indicates the completion of a job; if there are no more jobs available, the host vproc is relinquished by de-provisioning the vproc and then stopping. The last vproc to complete a job does not stop, but instead returns from the `forkN` function. When the scheduler receives a `PREEMPT` signal, yields control to

```
fun forkN {nVProcs, nJobs, job} = callcc (fn doneK =>
    let val (nStarted, done) = (ref 0, ref 0)
        val vprocs = hostVP () :: provision (min (nVProcs, nJobs))
        val nVProcs = length vprocs
        val tid = getTid ()
        fun dlpSwitch STOP =
            let val nextJob = fetchAndAdd (nStarted, 1)
            in
                if (nextJob < nJobs)
                then dlpDispatch (fiber (fn () => job nextJob))
                else if (fetchAndAdd (done,1) = (nVProcs-1))
                then throw doneK ()
                else ( release [hostVP ()]; stop () )
            end
          | dlpSwitch (PREEMPT k) = (
            yield ();
            dlpDispatch k )
        and dlpDispatch k = ( setTid tid; run (dlpSwitch, k) )
        fun initOn vproc =
            let fun install () = dlpDispatch (fiber stop)
            in
                enqueueOnProc (vproc, (tid, fiber install))
            end
    in
        List.app initOn vprocs;
        stop ()
    end)
```

Figure 3.3: A flat, data-parallel scheduler.

the parent scheduler. At some point in the future, the parent will resume the data-parallel scheduler. If, for example, the parent is the language-level thread scheduler, it will resume the data-parallel scheduler once it cycles through its ready queue. Similar to the work-stealing scheduler, this behavior lets the data-parallel scheduler coordinate with its parent.

# SECTION 4
# A TIME-SHARING SCHEDULER BASED ON ENGINES

An engine [HF84] represents a computation that is subject to timed preemption, and is thus useful for implementing specialized schedulers that distribute an individual processor's time among threads. Compilers such as Chez Scheme have successfully incorporated engines to build flexible threading libraries and support time-sharing policies. To implement engines efficiently, a language only needs first-class continuations and timer interrupts [DH89], both of which are present in the Manticore runtime model.

An engine consists of a processor state and a quantity of *fuel* that measures processor time. When either its fuel runs out or its computation is finished, the active engine is responsible for running the next computation.

Engines come in one of two varieties, either flat or nested. A flat engine only accounts for its own fuel, so it cannot be properly run from within another engine. A nested engine [DH89], on the other hand, can run in a tree of engines. The standard implementation technique is called *fair nesting*, in which every unit of fuel charged to an engine is also charged to its ancestors. Section 4.1 first presents flat engines in the runtime model, and then Section 4.2 presents an extension to nested engines.

## 4.1 Time sharing with flat engines

Time-sharing schedulers keep a ready queue of engines, and periodically switch between them until all are finished computing. In the Manticore runtime model, these schedulers export a function of type `engine_spawn` that, when given an initial function value and some fuel, creates an engine, and adds it to the ready queue.

```
type engine = unit -> unit
type fuel = int
type engine_spawn = engine * fuel -> unit
```

The example below spawns three engines from different functions and amounts of fuel. If the fuel amounts are `t1` = `t2` = `t3` = 1, then the behavior is the same as a round-robin scheduler. But assigning `t1` = 2, `t2` = 3, and `t3` = 5 allocates about 20%, 30%, and 50% of the processor time to `f1`, `f2`, and `f3` respectively.

```
fun f1 () = ...
fun f2 () = ...
fun f3 () = ...

fun f0 (spawnFE : engine_spawn) = (
    spawnFE (f1, 5);
    spawnFE (f2, 2);
    spawnFE (f3, 3) )
```

The `timeSharing` function creates a new instance of this scheduler; it has the type

```
fun timeSharing {f0, fuel0} =
    let val q = mkQueueAtm ()
        fun fill fuel = (fuel, fuel)
        fun teSwitch _ STOP = runNext ()
          | teSwitch (0, fuelCapacity) (PREEMPT k) = (
            enqueueAtm (q, (getTid (), fuelCapacity, k));
            runNext () )
          | teSwitch (fuel, fuelCapacity) (PREEMPT k) =
            run (teSwitch (fuel - 1, fuelCapacity), k)
        and runNext () =
            (case dequeueAtm q
              of SOME (tid, fuel, k) => (
                 setTid tid;
                 run (teSwitch (fill fuel), k) )
               | NONE => stop ()
            (* esac *))
        fun spawnFE (f, fuel) =
            enqueueAtm (q, (newTid (), fuel, fiber f))
        val k = fiber (fn () => f0 spawnFE)
    in
        fiber (fn () => ( setTid (newTid ());
                          run (teSwitch (fill fuel0), k)) )
    end (* timeSharing *)
```

Figure 4.1: A time-sharing scheduler that uses flat engines.

```
val timeSharing : {f0: engine, fuel0 : fuel} -> fiber
```

where `f0` is the initial engine that is given access to the spawn function, `fuel0` is `f0`'s fuel, and the return type is a fiber that starts the engines when run. The following function creates the scheduler fiber, and then enqueues it on its vproc.

```
fun init () =
    let val fbr = timeSharing {f0=f0, fuel0=1}
    in
        enqueue (newTid(), fbr)
    end
```

The code in Figure 4.1 implements a uniprocessor version of the time-sharing scheduler `timeSharing`. The scheduler initializes itself by creating a fiber that, when invoked, applies the initial function `f0` to its `spawnFE` operator. During execution, the scheduler maintains a ready queue `q` of engines that are swapped in and out by the `teSwitch` function. When `teSwitch` receives a `STOP` signal, the scheduler terminates if its queue is empty. Otherwise, it calls `runNext`, which dequeues an engine, reloads its fuel, and then runs the engine. A `PREEMPT` signal, the more interesting case, causes `teSwitch` to

check if the active engine is out of fuel. If so, it is put on the ready queue, and the next engine is run; if not, the active engine is charged a unit of fuel before being resumed. Notice how this behavior precludes proper nesting of engines: the `timeSharing` scheduler does not relinquish control to a parent engine if its fuel runs out.

## 4.2 Time sharing with nested engines

This section describes a scheduler that implements engines with fair nesting. The signature for nested engines is slightly different from before. The spawn function now elides the fuel parameter; instead, fuel is hidden within the `engine` type.

```
type engine
type fuel = int
type nested_engine_spawn = (engine -> unit) -> unit
```

There are two new functions for creating *leaf engines* that run actual computations and *nested engines* that only run other engines.

```
val leaf : ((unit -> unit) * fuel) -> engine
val engine : (nested_engine_spawn * fuel) -> engine
```

The `enginesInit` function takes an initial spawning function, and creates a fiber for the root engine.

```
val enginesInit : nested_engine_spawn -> fiber
```

The following example begins with a nested engine `e1`. The engine is essentially the same as the example in the previous section, except `e1` runs within another engine.

```
fun e1 spawnNE = (
    spawnNE (leaf (f1, 5));
    spawnNE (leaf (f2, 2));
    spawnNE (leaf (f3, 3)) )
```

The root engine contains the engine `e1` and the leaf engine `f4`. The leaf is given 80% of the processor time, but the engines spawned by `e1` share only the remaining 20%.

```
fun e0 spawnNE = (
    spawnNE (engine (e1, 2));
    spawnNE (leaf (f4, 8)) )
```

The `init` function initializes these engines, and enqueues them on the host vproc.

```
fun init () =
    let val k = enginesInit e0
    in
        enqueue (newTid(), k)
    end
```

The spawnNE function is responsible for initializing leaves and engines. When it spawns a leaf, the function initializes it with the leafEngine function. This leaf initializer wraps the leaf fiber in a trivial scheduler that stops when leaf stops, and always yields to the parent scheduler when pre-empted. The spawnNE function spawns an engine by making a recursive call that initializes the subtree of engines.

```
fun spawnNE (LEAF (f, fuelCell)) =
    enqueueAtm (q, (leafEngine f, fuelCell))
  | spawnNE (ENG {engineSpawn, fuelCell}) =
    let val k = enginesInit engineSpawn
    in
        enqueueAtm (q, (k, fuelCell))
    end (* spawnNE *)
```

The remaining discussion presents the scheduler action neSwitch informally; Appendix B contains the complete code for the nested engine scheduler. When it handles the STOP signal, the scheduler action runs the next engine in the queue by invoking runNext. This function then checks to see if the ready queue is empty; if it is, the scheduler terminates. Otherwise, the scheduler picks the next engine, refills its fuel cell, and runs the engine. Preemption, however, is handled differently from the flat scheduler. When it handles the PREEMPT signal, the scheduler action immediately yields to its parent scheduler, initiating a cascade of yields that travels up to the topmost scheduler. Each engine along the way might put the child on its ready queue. But once control returns to the original scheduler, it charges the then active engine a unit of fuel. If the fuel cell is empty, the engine is put back on the ready queue, but it is otherwise resumed.

# SECTION 5
# PROGRAMMING SCHEDULERS

The scheduling infrastructure is low level, and provides no explicit safety mechanisms for enforcing policies. Rather, the infrastructure relies on schedulers to cooperate on behalf of an entire application. Schedulers must coordinate by sharing vprocs, the most crucial resource in the runtime model, in a way that is acceptable to all active schedulers. In order to meet the demands of its threads and to maintain efficiency, a scheduler must get to run its threads with sufficient time to meet their scheduling requirements. Threads could require, for instance, a proportional share of the vproc, or they could have periodic soft real-time deadlines. If schedulers fail to coordinate effectively, threads will likely fail to meet their scheduling requirements.

In the runtime model, there are two ways for schedulers to coordinate. Since schedulers are nested and can thus form a tree, a scheduler must both *coordinate upwards* with its parent, and *coordinate downwards* with its children. Schedulers can coordinate upwards by yielding the vproc immediately upon a preemption. Schedulers can coordinate downwards by scheduling child schedulers and threads so that each thread can meet its scheduling requirements. When one scheduler fails to coordinate with another, it *starves* that scheduler.

The time-sharing schedulers in Section 4 exemplify some extremes of how to coordinate with, or starve, other schedulers. It is easy to see whether these schedulers coordinate upwards. The flat time-sharing scheduler starves its parent because it never yields upon a preemption. On the other hand, the nested time-sharing scheduler coordinates upwards because it immediately yields to its parent scheduler. Both schedulers, however, can coordinate downwards because they periodically run child schedulers according to their amounts of fuel.

Writing schedulers in this low-level, cooperative infrastructure is, by definition, not modular, and is also error prone. Each scheduler must be careful not to starve other schedulers, but at the same time, must enforce its own policies. Although the schedulers presented in this work have simple behavior in isolation, implementations that use several of them simultaneously can potentially interact in ways that cause starvation or poor cache locality for threads. To complicate matters, the Manticore language might, in the future, incorporate other parallel constructs with different scheduling policies. In addition, application-specific scheduling policies may be an important tool in maximizing parallel performance, or in supporting applications with real-time constraints. Such a complicated coordination language, with possibly many active schedulers, would require high-level abstractions and tools for guaranteeing safety.

In his doctoral work, Regehr presents a framework called HLS for safely developing nested schedulers [Reg01]. HLS provides a system of uniprocessor guarantees that specify whether nested schedulers can coordinate safely. A nested scheduler, for instance, might have a soft real-time policy, *i.e.*, it could need 3ms of the CPU every 10ms. Such a scheduler can be only nested correctly if its parent scheduler can always guarantee the required CPU share on time. In turn, the nested scheduler can parcel out its own guarantee to its children.

The guarantees, however, cover a more general range of scheduling requirements than just soft real time, *e.g.*, time-sharing variants and fixed priority assignments. To support heterogeneous schedulers with different guarantees, the framework provides conversions between guarantees. For example, a fixed priority scheduler can always guarantee its highest priority process 100% of CPU time, but can guarantee nothing to its lower-priority processes. Schedulers in HLS can request new guarantees from their parent schedulers during runtime. Thus, the schedulers need a mechanism that can inform the child whether or not the guarantee is acceptable.

The HLS framework might be an important tool for implementing applications with complex scheduling requirements. Supporting the guarantee infrastructure in our runtime model would entail encoding its syntax and conversion functions, and would require mechanisms for negotiating guarantees at run time. Our runtime model could provide these mechanisms with new signals for guarantee success and failure.

```
datatype guarantee = PROPORTIONAL_SHARE of percent | ...
datatype guarantee_notify = SUCCESS of guarantee | FAILURE
datatype signal = ...
        | REQ_GUARANTEE of (guarantee * guarantee_notify cont)
fun requestGuarantee g =
    callcc (fn k => forward (REQ_GUARANTEE (g, k)))
```

The following code would request a 20% share of its vproc, and handle success or failure.

```
        case requestGuarantee (PROPORTIONAL_SHARE 20)
          of SUCCESS g => ...
           | FAIL => ...
```

A high-level language might also be an important tool for developing new schedulers in the runtime model. The Bossa domain-specific language is one such language for developing uniprocessor schedulers [MLD05]. The language is compiled into C code that hooks into a specialized Linux kernel, but could be ported to the Manticore runtime model as well. The Bossa compiler performs several static correctness checks, including standard type correctness and several scheduler invariants. Schedulers are defined by a combination of specialized process queues and signal handlers. When a signal, *e.g.* timed preemption, reaches a scheduler, its signal handler manipulates process queues, and picks the next process to run.

Bossa supports nested schedulers, but unlike the Manticore runtime model, makes a static distinction between the usual process schedulers and *virtual schedulers*, which scheduler other schedulers. In Bossa, the root scheduler handles a signal first, and then propagates the signal down to an eligible process scheduler. This behavior complicates the handling of some signals. For example, schedulers must manually route the signal for blocking on an IO request down to the scheduler responsible for the requesting process. The Manticore runtime model, in contrast, processes signals starting at the current leaf scheduler. The motivation for this approach is to simplify the compiler and runtime system implementations. In most cases, this approach transfers complexity to scheduler implementations.

# SECTION 6
# FORMAL SEMANTICS

The formal semantics serves two purposes. It provides a model for porting the runtime model to actual machines, and it plays the role of an API for developing new schedulers.

The *CMANT* language is the starting point for developing the semantics. *CMANT* is a simple higher-order applicative language with first-class continuations. The context-free grammar below specifies terms in *CMANT*, which are either values or non-values. Values include constants, variables, and functions. Non-values include function application, continuation operators, tuples, and tuple projection. Other primitive operators, conditionals, and types are elided from the presentation to keep focus on the scheduling infrastructure.

$$
\begin{array}{rcl}
e & \in & \textit{CMANT} \\
v & \in & \textit{Values} \\
x,\ k,\ tid & \in & \textit{Variables} \\
c, \bullet & \in & \textit{Constants} \quad \text{(Unit constants are denoted by } \bullet\text{.)}
\end{array}
$$

$$
\begin{array}{rcl}
v & ::= & c \\
 & | & x \\
 & | & \lambda x.\ e
\end{array}
$$

$$
\begin{array}{rcl}
e & ::= & v \\
 & | & e_1\ e_2 \\
 & | & \texttt{letcont}\ k(x)\ =\ e_1\ \texttt{in}\ e_2 \\
 & | & \texttt{throw}\ e_{arg}\ \texttt{to}\ e_k \\
 & | & (e_1, \ldots, e_n) \\
 & | & \#i\ e
\end{array}
$$

*CMANT* has the following context-sensitive properties that are checked by the compiler. In the function $\lambda x.\ e$, the parameter $x$ is bound in the body of $e$. Similarly the expression $\texttt{letcont}\ k(x)\ =\ e_1\ \texttt{in}\ e_2$ binds the parameter $x$ in $e_1$, and it binds the variable $k$ in $e_2$. A variable that is not bound is *free*; the set of free variables in a term $e$ is denoted $FV(e)$. A *program* is a term with no free variables, and an *answer* is a syntactic constant. The semantics of *CMANT* are defined by a partial function from programs to answers.

The operational semantics for *CMANT* is given in an abstract machine called the *sequential machine*. This machine follows the CEK machine [FSDF93] and has the following state transition.

$$St \longmapsto St'$$

To support the scheduling infrastructure described in Section 2, *CMANT* is extended with syntax for signals and scheduling operations. The sequential machine is then packaged into a *vproc machine* that supports these new operations; it has the following state transition.

$$VP \Rightarrow VP'$$

Lastly, the *multiprocessor machine* ties together a group of vprocs with a parallel state transition.

$$M \Rrightarrow M'$$

## 6.1 The sequential machine

The sequential machine has three components: an expression *CMANT*, an environment *E* that binds all free variables in *CMANT*, and a continuation *K*. A sequential machine state *St* consists of one of two forms:

1. The *evaluation* form $\langle e,\ E,\ K\rangle$ evaluates an expression $e$ under an environment *E* with a continuation *K*.

2. The *return* form $\langle K,\ v_d\rangle$ applies the the continuation *K* to the machine value $v_d$.

Appendix C.1 contains the complete state transition function $St \longmapsto St'$. The relation $\longmapsto^*$ is the reflexive transitive closure of the transition function. The auxiliary function $\delta$ takes a syntactic value and an environment, and converts them into a machine value. The notation $E(x)$ refers to an algorithm for finding $x$ in *E*. The notation $[\mathtt{cl}_\lambda\ x.e,\ E]$ is a *function closure*. A closure is a record that contains the code for $e$, and binds free variables of $\lambda x.\ e$.

Many state transitions follow a common pattern: they evaluate a subexpression, and install a continuation for the rest of the computation. For instance, the transition for setting up function application puts $e_1$ into an evaluation form, and installs the continuation **ap1**.

$$\langle e_1\ e_2,\ E,\ K\rangle \quad\longmapsto\quad \langle e_1,\ E,\ \{|\mathbf{ap1}\ e_2,\ E,\ K\ |\}\rangle$$

The state transition for performing function application, however, is more interesting. Given the function closure and its argument value $v_2$, the transition extends the environment $E'$, and puts the function body $e$ into an evaluation form.

$$\langle \{|\mathbf{ap2}\ [\mathtt{cl}_\lambda\ x.e,\ E'],\ E,\ K|\},\ v_2\rangle \quad\longmapsto\quad \langle e,\ E'[x \mapsto v_2],\ K\rangle$$

In the full semantics in Section C.1, these interesting rules are highlighted with a $(*)$.

The sequential machine supports the first-class continuations described in Section 2.1. The expression **letcont** $k(x)\ =\ e_1$ **in** $e_2$ captures the current continuation. To accomplish this, the state transition for **letcont** first packages the continuation *K* into a *continuation closure* $[\mathtt{cl}_k\ x.e_1,\ E,\ K]$, binds the closure to $k$, and then runs $e_2$. If an expression **throw** $v_{arg}$ **to** $k$ is evaluated in the body of $e_2$, the machine applies $v_{arg}$ to the closure, and then runs $e_1$ with the continuation *K*. These control operators later provide the basis for fibers and threads that can run on a vproc.

## 6.2 Virtual processors

This section enriches *CMANT* to operate within a vproc. The syntax below adds support for signals and primitive scheduling operations.

$$
\begin{array}{rcl}
v & ::= & \ldots \\
 & | & \texttt{STOP} \\
 & | & \texttt{PREEMPT}(x_k) \\
e & ::= & \ldots \\
 & | & \texttt{run}(e_{act}, e_k) \\
 & | & \texttt{forward}(e_{sig}) \\
 & | & \texttt{stop}() \\
 & | & \texttt{mask}() \\
 & | & \texttt{unmask}()
\end{array}
$$

Sequential state transitions for these new operations are defined in Section C.1. All of the transitions, excluding the one for **stop**, simply evaluate arguments. The **stop** transition sets up a context that applies the **forward** continuation to the **STOP** signal.

$$\langle \texttt{stop}(),\ E,\ K \rangle \quad \longmapsto \quad \langle \{|\mathbf{forward}|\},\ \texttt{STOP} \rangle$$

Since **stop** is expanded away in the sequential machine, **forward** and **run** are the only control operators in the vproc machine.

Appendix C.2 contains the complete vproc state transition function. A vproc state *VP* consists of a sequential machine state *St*, a mode of operation *VPMd*, a queue of ready threads *Q*, and a stack of scheduler actions *S*.

$$\langle\!| \mathit{St},\ \mathit{VPMd},\ Q,\ S |\!\rangle$$

The vproc mode *VPMd* is either idle on thread $i$, running a thread $i$ with signals unmasked, or running a thread $i$ with signals masked.

$$
\begin{array}{rcll}
\mathit{Md} & \in & \mathit{Mode} & = \ \mathbf{Idle} \\
 & & & | \ \mathbf{U} \\
 & & & | \ \mathbf{M} \\
\mathit{VPMd} & \in & \mathit{VPMode} & = \ \mathit{Md}(i)
\end{array}
$$

The **mask**() and **unmask**() operators allow programs to manually switch modes. Figure 6.1 shows the ways the mode can change during execution.

The vproc ready queue *Q* is accessible to all other vprocs in the system, so operations on it must be atomic. To support this, queue operations are defined later in the multiprocessor machine, but are left opaque to the vproc itself.

The scheduler action stack *S* is either an empty stack **[]** or a frame $act :: S$. If the stack is empty, the $pop$ operation returns a default scheduler action $switch$. In the implementation,

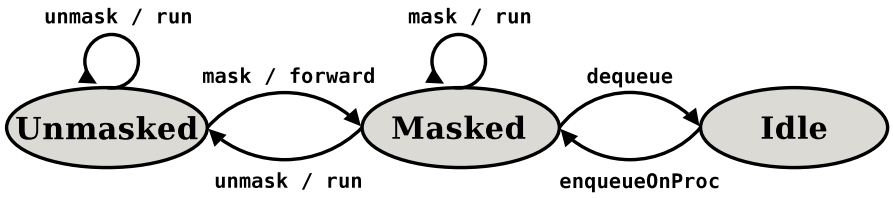


Figure 6.1: Possible vproc mode transitions.

$switch$ is similar to the top-level scheduler given in Section 2.4.

The vproc machine, at the most basic level, hosts a sequential machine: if the thread $i$ has a state transision in the sequential machine, then the vproc machine matches it.

$$\frac{St \longmapsto St'}{\langle\!\cdot St,\ Md(i),\ Q,\ S \cdot\!\rangle \ \Rightarrow\ \langle\!\cdot St',\ Md(i),\ Q,\ S \cdot\!\rangle}$$

The most important job of the vproc machine is to execute the **`run`** and **`forward`** scheduling operations. Consider the expression **`run`**$(switch, k)$ that runs a fiber under a scheduler action. The following vproc state transition accomplishes this by pushing $switch$ on the scheduler stack *S*, and then running $k$.

$$\begin{array}{c}\langle\!\cdot \langle \{\!|\mathbf{run2}\ [\mathtt{cl}_\lambda\ x_{sig}.e',\ E']|\!\},\ k\rangle,\ Md(i),\ Q,\ S \cdot\!\rangle \\ \Rightarrow \\ \langle\!\cdot \langle e,\ E[x \mapsto \bullet],\ K\rangle,\ \mathbf{U}(i),\ Q,\ [\mathtt{cl}_\lambda\ x_{sig}.e',\ E'] :: S \cdot\!\rangle \\ \text{where} \quad switch = [\mathtt{cl}_\lambda\ x_{sig}.e',\ E'] \\ \qquad\qquad\ k = [\mathtt{cl}_k\ x.e,\ E,\ K]\end{array}$$

The **`forward`** operation is complementary to **`run`**. Consider the expression **`forward`**$(sig)$ that forwards the signal $sig$ to the current scheduler action, which in this case is $switch$. The state transition does this by first popping $switch$ off the stack *S*. It then binds $x_{sig}$ to $sig$, and invokes the closure for $switch$.

$$\begin{array}{c}\langle\!\cdot \langle \{\!|\mathbf{forward}|\!\},\ sig\rangle,\ Md(i),\ Q,\ [\mathtt{cl}_\lambda\ x_{sig}.e',\ E'] :: S \cdot\!\rangle \\ \Rightarrow \\ \langle\!\cdot \langle e',\ E'[x_{sig} \mapsto sig],\ \{\!|\mathbf{stop}|\!\}\rangle,\ \mathbf{M}(i),\ Q,\ S \cdot\!\rangle\end{array}$$

Operations in the vproc thus far have been synchronous – the point at which the vproc interrupts execution is always initiated by the sequential machine. The vproc, however, also permits asynchronous operations such as preemption. The following example shows how preemption is generated in the vproc. Suppose a vproc is executing a thread $i$ that is about evaluate the expression $e$. Since the vproc executing the thread is in $\mathbf{U}$ mode, it can safely be preempted. The state transition packages running thread into the continuation $k$,

and then forwards the preemption signal to the vproc.

$$\langle\!: \langle e,\ E,\ K\rangle,\ \mathbf{U}(i),\ Q,\ S :\!\rangle$$
$$\Rightarrow$$
$$\langle\!: \langle \texttt{forward}(\texttt{PREEMPT}(k)),\ [k \mapsto [\texttt{cl}_k\ x.e,\ E,\ K]],\ \{\!|\mathbf{stop}|\!\}\rangle,\ \mathbf{U}(i),\ Q,\ S :\!\rangle$$

## 6.3 The multiprocessor machine

So far, the discussion has centered on sequential evaluation, yet Manticore is intended to support multiple processors running in parallel. The *multiprocessor machine* in this section introduces a notion of parallel evaluation. In addition to modeling parallelism, primitives for allocating vprocs, enqueuing and dequeuing on vprocs, atomically side-effecting memory, and creating thread ids are all part of the multiprocessor semantics. These primitives, although varying in purpose, share the property that they make changes to machine state that are visible to all vprocs in the system. The syntax below extends *CMANT* with these primitives.

$$\begin{array}{rcl} e & ::= & \ldots \\ & | & \texttt{enqueueOnProc}(e_m, e_{tid}, e_k) \mid \texttt{dequeue}() \\ & | & \texttt{provision}(e) \mid \texttt{release}(e) \\ & | & \texttt{ref}(e) \mid e_r\texttt{:=}\ e \mid \texttt{!}e_r \\ & | & \texttt{fetchAndAdd}(e_r, e) \mid \texttt{compareAndSwap}(e_r, e_1, e_2) \\ & | & \texttt{newTid}() \mid \texttt{setTid}(e) \mid \texttt{getTid}() \end{array}$$

Sequential state transitions for evaluating their arguments are given in Section C.1.

A multiprocessor machine state *M* consists of a pool of vprocs *VPM*, with each indexed by a unique integer, a global store *Str* for modeling shared memory, and a vproc provisioning record *PrMp*.

$$\{\!: VPM\ ;\ Str\ ;\ PrMp :\!\}$$

Although the pool of vprocs is a finite map, the presentation often uses the following notational convenience to select the $m$th vproc in the map.

$$\{\!: VPM, VP_m\ ;\ Str\ ;\ PrMp :\!\} \equiv \{\!: \{VP_m\} \cup VPM\ ;\ Str\ ;\ PrMp :\!\}$$

Appendix C.3 contains the complete multiprocessor state transition function. The transitions follow a simple interleaving semantics. If a single vproc (indexed by $m$) steps forward, then the multiprocessor machine steps forward.

$$\frac{\langle\!: St,\ VPMd,\ Q,\ S :\!\rangle\ \Rightarrow\ \langle\!: St',\ VPMd',\ Q,\ S' :\!\rangle}{\begin{array}{l}\{\!: VPM,\ \langle\!: St,\ VPMd,\ Q,\ S :\!\rangle_m\ ;\ Str\ ;\ PrMp :\!\} \\ \quad \Rrightarrow\ \{\!: VPM,\ \langle\!: St',\ VPMd',\ Q,\ S' :\!\rangle_m\ ;\ Str\ ;\ PrMp :\!\}\end{array}}$$

The queue operations form the core of the multiprocessor machine. They have two

different behaviors depending on the vproc's queue $Q$. When the queue is empty, the **dequeue** operation blocks the vproc; a subsequent **enqueueOnProc** operation unblocks it. Otherwise, the operators perform the auxiliary queue functions defined in Section C.3. These queue functions specify no order on queue elements, which allows implementations to experiment with alternative policies that might use priorities or multi-level queues.

The queue state transitions are now described with examples. When a vproc $m$ tries to dequeue from the empty queue $\emptyset$, its mode is switched to $\mathbf{Idle}$. Since the vproc machine cannot make a transition out of this mode, it is effectively blocked.

$$\begin{array}{c}\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \mathbf{dequeue}(),\ E,\ K\rangle,\ \mathit{Md}(i),\ \emptyset,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}\\ \Rrightarrow\\ \{\!:\ \mathit{VPM},\ \langle\!:\ \langle \mathbf{dequeue}(),\ E,\ K\rangle,\ \mathbf{Idle}(i),\ \emptyset,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}\end{array}$$

Suppose that later another vproc $n$ tries to enqueue a thread $(j, k)$ on $m$. The machine now awakens $m$ with a non-empty queue so that it can safely proceed to the next state transition.

$$\begin{array}{c}\{\!:\ \mathit{VPM},\\ \langle\!:\ \langle \{\!|\mathbf{enq3}\ m,\ j,\ K|\!\}\ k\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_n,\\ \langle\!:\ \langle \mathbf{dequeue}(),\ E,\ K\rangle,\ \mathbf{Idle}(i),\ \emptyset,\ S'\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}\\ \Rrightarrow\\ \{\!:\ \mathit{VPM},\\ \langle\!:\ \langle K,\ \bullet\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_n,\\ \langle\!:\ \langle \mathbf{dequeue}(),\ E,\ K\rangle,\ \mathbf{U}(i),\ \{(j,k)\},\ S'\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}\end{array}$$

Continuing the example, the vproc $m$ proceeds to dequeue the new thread. The following state transition returns the thread to the continuation $K$, and lets $m$ resume execution.

$$\begin{array}{c}\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \mathbf{dequeue}(),\ E,\ K\rangle,\ \mathbf{U}(i),\ \{(j,k)\},\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}\\ \Rrightarrow\\ \{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ (k,j)\rangle,\ \mathbf{U}(i),\ \emptyset,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}\end{array}$$

The remaining queue state transitions are described with a new example. If a vproc $m$ tries to enqueue on another vproc $n$ that is not idle, the transition just performs built-in queue operations in the background.

$$\begin{array}{c}\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{enq3}\ n,\ j,\ K|\!\}\ k\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_m,\\ \langle\!:\ \mathit{St},\ \mathbf{U}(i),\ Q',\ S'\ :\!\rangle_n\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}\\ \Rrightarrow\\ \{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ \bullet\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_m,\\ \langle\!:\ \mathit{St},\ \mathbf{U}(i),\ Q' \cup \{(j,k)\},\ S'\ :\!\rangle_n\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}\end{array}$$

The final case is when a vproc $m$ enqueues on its own queue.

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{enq3}\ m,\ j,\ K|\!\}\ k\rangle,\ \mathbf{U}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ \bullet\rangle,\ \mathbf{U}(i),\ Q \cup \{(j,k)\},\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$

Another important job of the multiprocessor machine is to allocate vprocs to threads, and then de-allocate them later. The **`provision`** operator allocates a set of vprocs to the current running thread. Suppose, for instance, the thread $i$ requests 2 vprocs by running the expression **`provision`**$(2)$, and the thread is making its first request; thus, the provisioning map *PrMp* has no entry for $i$. Further, suppose that the vproc $n$ is idle, and thus a good candidate for provisioning. The following state transition then provisions the vproc $n$ along with the host vproc $m$ to the thread $i$.

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{provision}\ K|\!\},\ 2\rangle,\ \mathbf{U}(i),\ Q,\ S\ :\!\rangle_m, \langle\!:\ \mathit{St},\ \mathbf{Idle}(j),\ Q',\ S'\ :\!\rangle_n\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ (m,n)\rangle,\ \mathbf{U}(i),\ Q,\ S\ :\!\rangle_m, \langle\!:\ \mathit{St},\ \mathbf{Idle}(j),\ Q',\ S'\ :\!\rangle_n\ ;\ \mathit{Str}\ ;\ \mathit{PrMp},\ \{m,n\}_i\ :\!\}$$

The thread $i$ can later release $n$ by deleting it from its provisioning set.

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{release}\ K|\!\},\ n\rangle,\ \mathbf{U}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp},\ \{m,n\}_i\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ \bullet\rangle,\ \mathbf{U}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp},\ \{m\}_i\ :\!\}$$

# SECTION 7
# RELATED WORK

The runtime model and, in particular, the scheduling infrastructure in Section 2.3 is inspired by Shivers' proposal for exposing hardware concurrency [Shi97]. His proposal uses continuations for suspended computations and threads for representing both hardware processors and threads of control, but uses different terminology; threads are similar to fibers, and events are similar to signals. There are three operators for managing threads: the `fork-thread`, `advance-thread`, and `event-dispatch` operators are similar to `fiber`, `run`, and `forward` respectively. The primary difference is that `advance-thread` installs just one signal handler per thread, thus precluding built-in support for nested schedulers. Shivers' proposal also does not support multiprocessors, which are of key importance in the Manticore runtime model.

Shivers' proposal also differs in how it encodes event handlers. The Manticore runtime model represents event handlers as functions that take event values, *e.g.* the `switch` function. Alternatively, Shivers uses a desugared representation that encodes handlers as vector of event tag and continuation pairs. For example, the following expression runs a thread `thd` with scheduler actions for `STOP` and `PREEMPT` operations

```
advance-thread ([ (STOP, stopK), (PREEMPT, preemptK) ], thd)
```

where `stopK` is a continuation expecting a unit value, and `preemptK` is a continuation that in turn expects the continuation for the preempted thread. We plan to explore Shivers' representation in the implementation of the runtime model.

The runtime model borrows two components of its design from the MOBY programming language [FR02]. First, the MOBY compiler uses a low-level coordination language for expressing several parallel constructs. This coordination language, however, is limited to a global, fixed scheduling policy. But like Manticore, the MOBY coordination language is a part of compiler intermediate representation, called BOL. BOL is specified by a formal semantics, and it is used as a basis for the semantics in Section 6. The second component is the MOBY runtime system. Similar to Manticore, the system uses OS-level threads to represent physical processors.

STING [JP92] is a dialect of SCHEME that supports multiple forms of parallelism. Like Manticore, the language employs first-class continuations to support multi-threading. STING provides a high-level framework for developing schedulers, which unlike the Manticore runtime model, is intended to let programmers implement different schedulers on-the-fly. This framework consists of several layers of process abstraction: *threads* are similar to fibers, except they contain local storage and operate on a flat stack; *virtual processors* (VPs) are collections of threads with a common scheduling discipline; *abstract physical processors* (APPs) represent the physical processors in the system, and are responsible for running a collection of VPs.

In STING, the programmer must implement separate scheduling policies for VPs and APPs. The former schedules multiple threads on a single VP, and the latter schedules multiple VPs on a single APP. This separation of VPs and APPs provides an explicit mechanism

for creating virtual topologies of VPs [Gre92, HS86]. Such mechanisms complicate the scheduling substrate by introducing an extra layer of abstraction and by requiring separate scheduler implementations for both VPs and APPs. In contrast, the Manticore design is biased towards ease of compiler implementation by unifying scheduling code and towards efficiency of scheduling operations. STING, also in contrast to Manticore, does not provide explicit mechanisms for nested schedulers.

# SECTION 8
# CONCLUSIONS

This paper describes the design of a runtime model for heterogeneous parallel languages. It supports rapid development of scheduling policies, and cooperation between those policies in a unified framework. Even though it is motivated by the Manticore programming language, the runtime model is applicable to the general class of heterogeneous parallel languages.

The scheduler examples given in Sections 3 and 4 demonstrate that the model is general enough to encode a wide variety of parallel constructs and their scheduling policies. In addition, the examples demonstrate how different policies can coordinate to share processor time.

Section 5 gives guidelines for programming new schedulers, and points to related work on developing modular schedulers.

The formal semantics, described informally in Section 6 and given in complete form in the appendix, specifies what it means to be a correct and complete implementation of the runtime model. But the semantics can also be used as an API for developing schedulers.

A prototype C implementation of the runtime model is in development. This prototype serves to evaluate performance of the model, to experiment with alternative stack implementations and interoperability with C, and to plan the Manticore compiler implementation. The prototype represents each vproc by a POSIX thread; POSIX signals are employed to handle timed preemption. Fibers, however, pose a problem since C has a flat stack model. They are currently represented as a stack and context pair.

# REFERENCES


[AM91] Appel, A. W. and D. B. MacQueen. Standard ML of new jersey. In J. Maluszyński and M. Wirsing (eds.), *Proceedings of theThirdInternational Symposium on Programming Language Implementation and Logic Programming*, number 528. Springer Verlag, 1991, pp. 1–13.

[App92] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[BCH+94] Blelloch, G. E., S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *JPDC*, **21**(1), 1994, p. 4=14.

[BG96] Blelloch, G. E. and J. Greiner. A provable time and space efficient implementation of NESL. In *ICFP '96*, New York, NY, May 1996. ACM, pp. 213–225.

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

[Ble96] Blelloch, G. E. Programming parallel algorithms. *CACM*, **39**(3), March 1996, pp. 85–97.

[BS81] Burton, F. W. and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, New York, NY, USA, 1981. ACM Press, pp. 187–194.

[BWD96] Bruggeman, C., O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *PLDI '96*, New York, NY, May 1996. ACM, pp. 99–107.

[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.

[CK00] Chakravarty, M. M. T. and G. Keller. More types for nested data parallel programming. In *ICFP '00*, New York, NY, September 2000. ACM, pp. 94–105.

[CKLP01] Chakravarty, M. M. T., G. Keller, R. Leshchinskiy, and W. Pfannenstiel. Nepal – Nested Data Parallelism in Haskell. In *Euro-Par '01*, vol. 2150 of *LNCS*, New York, NY, August 2001. Springer-Verlag, pp. 524–534.

[CR95] Carlisle, M. C. and A. Rogers. Software caching and computation migration in Olden. In *PPoPP '95*, New York, NY, July 1995. ACM, pp. 29–38.

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

[FR02] Fisher, K. and J. Reppy. Compiler support for lightweight concurrency. *Technical memorandum*, Bell Labs, March 2002. Available from `http://moby.cs.uchicago.edu/`.

[FSDF93] Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. June 1993, pp. 237–247.

[Gre92] Greenberg, D. S. *Full utilization of communication resources*. Ph.D. dissertation, New Haven, CT, USA, 1992.

[Hed98] Hedqvist, P. A parallel and multithreaded ERLANG implementation. Master's dissertation, Computer Science Department, Uppsala University, Uppsala, Sweden, June 1998.

[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.

[HS86] Hudak, P. and L. Smith. Para-functional programming: a paradigm for programming multiprocessor systems. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York, NY, USA, 1986. ACM Press, pp. 243–254.

[JP92] Jagannathan, S. and J. Philbin. A foundation for an efficient multi-threaded scheme system. In *LFP'92*, New York, NY, June 1992. ACM, pp. 345–357.

[LCK06] Leshchinskiy, R., M. M. T. Chakravarty, and G. Keller. Higher order flattening. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra (eds.), *ICCS '06*, number 3992 in LNCS, New York, NY, May 2006. Springer-Verlag, pp. 920–928.

[Ler00] Leroy, X. *The Objective Caml System (release 3.00)*, April 2000. Available from `http://caml.inria.fr`.

[MJMR01] Marlow, S., S. P. Jones, A. Moran, and J. Reppy. Asynchronous exceptions in Haskell. In *PLDI '01*, June 2001, pp. 274–285.

[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.

[MLD05] Muller, G., J. L. Lawall, and H. Duchesne. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *HASE '05*, October 2005, pp. 56–65.

[Ree83] Reeves, W. T. Particle systems — a technique for modeling a class of fuzzy objects. *ACM TOG*, **2**(2), 1983, pp. 91–108.

[Reg01] Regehr, J. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. Ph.D. dissertation, University of Virginia, 2001.

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

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

[Rey93] Reynolds, J. C. The discoveries of continuations. *LASC*, **6**(3-4), 1993, pp. 233–248.

[RHH84] Robert H. Halstead, J. Implementation of multilisp: Lisp on a multiprocessor. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, New York, NY, USA, 1984. ACM Press, pp. 9–17.

[RP00] Ramsey, N. and S. Peyton Jones. Featherweight concurrency in a portable assembly language. Unpublished paper available at `http://www.cminusminus.org/abstracts/c--con.html`, November 2000.

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

[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 *LISP'80*, August 1980, pp. 19–28.

# APPENDIX A
# WORK-STEALING SCHEDULER

```
fun workStealing (nVPs, f) =
        val vprocs = provision nVPs
        val nVPs = length vprocs
        val qs = Vector.tabulate (nVPs, fn _ => mkQueueAtm ())
        fun wsSwitch q s =
            let fun runWS (tid, fbr) = (
                    setTid tid;
                    run (wsSwitch q, fbr) )
                fun pickVictim () =
                    let val victQ = Vector.sub (qs, randInt () mod nVPs)
                    in
                        case dequeueAtm victQ
                         of NONE => ( backOff (); pickVictim () )
                          | SOME thd => runWS thd
                    end (* pickVictim *)
                fun runNext () =
                    (case dequeueAtm q
                      of NONE => pickVictim ()
                       | SOME thd => runWS thd
                    (* esac *))
            in
                case s
                 of STOP => runNext ()
                  | PREEMPT k => (
                    enqueueAtm (q, (getTid (), k));
                    yield ();
                    runNext () )
            end (* wsSwitch *)
        fun initOn (vproc, i) =
            let val q = Vector.sub (qs, i)
                fun doit () = run (wsSwitch q, fiber stop)
            in
                enqueueOnProc (vproc, (getTid (), fiber doit))
            end
    in
        enqueueAtm (Vector.sub (qs, 0), (newTid (), fiber f));
        foldl (fn (vproc, i) => (initOn (vproc, i); i+1)) 0 vprocs;
        ()
    end (* workStealing *)
```

# APPENDIX B
# NESTED ENGINES

Types for nested engines:

```
type fuel = int
type fuel_cell = {fuel : fuel ref, capacity : fuel}

datatype engine = LEAF of (unit -> unit) * fuel_cell
                | ENG of {engineSpawn : (engine -> unit) -> unit,
                          fuelCell : fuel_cell}
datatype fuel_status = EMPTY | HAS_FUEL
```

Fuel management and engine creation functions:

```
fun charge {fuel, capacity} =
    if !fuel >= 0
    then ( fuel := !fuel - 1; HAS_FUEL )
    else EMPTY

fun refillFuelCell {fuel, capacity} = fuel := capacity

fun mkFuelCell capacity = {fuel=ref capacity, capacity=capacity}

fun engine (engineSpawn, fuelCap) =
    ENG {engineSpawn=engineSpawn, fuelCell=mkFuelCell fuelCap}

fun leaf (f, fuelCap) = LEAF (f, mkFuelCell fuelCap)
```

Nested engine scheduler functions:

```
fun leafEngine f =
    let fun leafSwitch STOP = stop ()
          | leafSwitch (PREEMPT k) = (
            yield ();
            run (leafSwitch, k) )
    in
        fiber (fn () => run (leafSwitch, fiber f))
    end (* leafEngine *)

fun enginesInit engineSpawn =
    let val q = mkQueueAtm ()
        fun runNext () =
            (case dequeueAtm q
              of NONE => stop ()
               | SOME (k, fuel) => (
                 refillFuelCell fuel;
                 run (neSwitch fuel, k) )
            (* esac *))
        and neSwitch _ STOP = runNext ()
          | neSwitch currFuel (PREEMPT k) = (
            yield ();
            (case charge currFuel
              of EMPTY => (
                 enqueueAtm (q, (k, currFuel));
                 runNext () )
               | HAS_FUEL => run (neSwitch currFuel, k)
            (* esac *)) )
        fun spawnNE (LEAF (f, fuelCell)) =
            enqueueAtm (q, (leafEngine f, fuelCell))
          | spawnNE (ENG {engineSpawn, fuelCell}) =
            let val k = enginesInit engineSpawn
            in
                enqueueAtm (q, (k, fuelCell))
            end (* spawnNE *)
    in
        fiber (fn () => run (neSwitch (mkFuelCell 1),
                             fiber (fn () => engineSpawn spawnNE)))
    end (* enginesInit *)
```

# APPENDIX C
# FORMAL SEMANTICS

## C.1 Sequential semantics

Semantics of a sequential program $e$:

$$eval(e) = c \quad \text{if} \quad \langle e,\ \emptyset,\ \{\!|\mathbf{stop}|\!\}\rangle \longmapsto^* \langle \{\!|\mathbf{stop}|\!\},\ c\rangle$$

Data specifications:

$$\begin{array}{rcl} e & \in & \textit{CMANT} \\ v & \in & \textit{Values} \\ x,\ k,\ tid & \in & \textit{Variables} \\ c, \bullet & \in & \textit{Constants} \quad \text{(Unit constants are denoted by } \bullet\text{.)} \end{array}$$

Data definitions:

$$\begin{array}{rcll} \textit{St} \in \textit{State} & = & \textit{CMANT} \times \textit{Env} \times \textit{Cont} \ \mid\ \textit{Cont} \times \textit{Value} & \text{(sequential states)} \\ \textit{E} \in \textit{Env} & = & \textit{Variables} \xrightarrow{\text{fin}} \textit{Value}_d & \text{(environments)} \\ v_d \in \textit{Value}_d & = & c & \text{(machine values)} \\ & \mid & [\mathtt{cl}_\lambda\ x.e,\ \textit{E}] & \\ & \mid & [\mathtt{cl}_k\ x.e,\ \textit{E},\ \textit{K}] & \\ & \mid & (v_1, \ldots, v_n) & \\ & \mid & \mathtt{STOP} & \\ & \mid & \mathtt{PREEMPT}(x) & \\ \textit{K} \in \textit{Cont} & = & & \text{(continuations)} \\ & & \{\!|\mathbf{stop}|\!\} \quad \text{– stop thread} & \\ & \mid & \{\!|\mathbf{ap1}\ e,\ \textit{E},\ \textit{K}\ |\!\} \mid \{\!|\mathbf{ap2}\ v,\ \textit{E},\ \textit{K}|\!\} \quad \text{– evaluating function} & \\ & \mid & \{\!|\mathbf{thr1}\ e,\ \textit{E}|\!\} \mid \{\!|\mathbf{thr2}\ v,\ \textit{E}|\!\} \quad \text{– evaluating throw} & \\ & \mid & \{\!|\mathbf{tup}\ i,\ (v_1, \ldots, v_m),\ (e_1, \ldots, e_n),\ \textit{E},\ \textit{K}|\!\} \quad \text{– tuples} & \\ & \mid & \{\!|\mathbf{run1}\ e,\ \textit{E}|\!\} \mid \{\!|\mathbf{run2}\ v|\!\} \quad \text{– run on scheduler} & \\ & \mid & \{\!|\mathbf{forward}|\!\} \quad \text{– forward to parent} & \\ & \mid & \{\!|\mathbf{enq1}\ e_2,\ e_3,\ \textit{E},\ \textit{K}|\!\} \mid \{\!|\mathbf{enq2}\ v_1,\ e_3,\ \textit{E},\ \textit{K}|\!\} \mid & \\ & & \quad \{\!|\mathbf{enq3}\ v_1,\ v_2,\ \textit{K}|\!\} \text{– enqueue} & \\ & \mid & \{\!|\mathbf{provision}\ \textit{K}|\!\} \mid \{\!|\mathbf{release}\ \textit{K}|\!\} \quad \text{– provisioning} & \\ & \mid & \{\!|\mathbf{ref}\ \textit{K}|\!\} \quad \text{– ref} & \\ & \mid & \{\!|\mathbf{assgn1}\ e,\ \textit{E},\ \textit{K}|\!\} \mid \{\!|\mathbf{assgn2}\ v,\ \textit{K}|\!\} \quad \text{– assignment} & \\ & \mid & \{\!|\mathbf{deref}\ \textit{K}|\!\} \quad \text{– deref} & \\ & \mid & \{\!|\mathbf{faa1}\ e,\ \textit{E},\ \textit{K}|\!\} \mid \{\!|\mathbf{faa2}\ v,\ \textit{K}|\!\} \quad \text{– fetch and add} & \\ & \mid & \{\!|\mathbf{cas1}\ e_2,\ e_3,\ \textit{E}\ \textit{K}|\!\} \mid \{\!|\mathbf{cas2}\ v_1,\ e_3,\ \textit{E}\ \textit{K}|\!\} \mid & \\ & & \quad \{\!|\mathbf{cas3}\ v_1,\ v_2,\ \textit{E}|\!\}\textit{K} \text{ – compare and swap} & \\ & \mid & \{\!|\mathbf{setTid}\ \textit{E}|\!\}\textit{K} \quad \text{– set tid} & \end{array}$$

Syntax:

$$
\begin{array}{rcl}
v & ::= & c \\
& | & x \\
& | & \lambda x.\, e \\
& | & \texttt{STOP} \\
& | & \texttt{PREEMPT}(x_k)
\end{array}
$$

$$
\begin{array}{rcl}
e & ::= & v \\
& | & e_1\ e_2 \\
& | & \texttt{letcont}\ k(x)\ =\ e_1\ \texttt{in}\ e_2 \\
& | & \texttt{throw}\ e_{arg}\ \texttt{to}\ e_k \\
& | & (e_1, \ldots, e_n) \\
& | & \#i\ e \\
& | & \texttt{run}(e_{act}, e_k) \mid \texttt{forward}(e_{sig}) \mid \texttt{stop}() \\
& | & \texttt{mask}() \mid \texttt{unmask}() \\
& | & \texttt{enqueueOnProc}(e_m, e_{tid}, e_k) \mid \texttt{dequeue}() \\
& | & \texttt{provision}(e) \mid \texttt{release}(e) \\
& | & \texttt{ref}(e) \mid e_r\texttt{:=}\ e \mid \texttt{!}e_r \\
& | & \texttt{fetchAndAdd}(e_r, e) \mid \texttt{compareAndSwap}(e_r, e_1, e_2) \\
& | & \texttt{newTid}() \mid \texttt{setTid}(e) \mid \texttt{getTid}()
\end{array}
$$

Converting syntactic values to machine values:

$$
\begin{array}{rcl}
\delta(c,\ E) & = & c \\
\delta(x,\ E) & = & E(x) \\
\delta(\lambda x.\ e,\ E) & = & [\texttt{cl}_\lambda\ x.e,\ E] \\
\delta((v_1, \ldots, v_n),\ E) & = & (v_1, \ldots, v_n) \\
\delta(\texttt{STOP},\ E) & = & \texttt{STOP} \\
\delta(\texttt{PREEMPT}(x_k),\ E) & = & \texttt{PREEMPT}(x_k)
\end{array}
$$

Machine transition rules (those marked with a $(*)$ are semantically interesting, and the rest build activation records):

$$
\begin{array}{rcll}
\langle v,\ E,\ K\rangle & \longmapsto & \langle K,\ \delta(v,\ E)\rangle & (*)\\
\langle e_1\ e_2,\ E,\ K\rangle & \longmapsto & \langle e_1,\ E,\ \{\!|\mathbf{ap1}\ e2,\ E,\ K\ |\!\}\rangle & \\
\langle \mathtt{letcont}\ k(x)\ =\ e_1\ \mathtt{in}\ e_2,\ E,\ K\rangle & \longmapsto & \langle e_2,\ E[k \mapsto [\mathtt{cl}_k\ x.e_1,\ E,\ K]],\ K\rangle & (*)\\
\langle \mathtt{throw}\ e_{arg}\ \mathtt{to}\ e_k,\ E,\ K\rangle & \longmapsto & \langle e_k,\ E,\ \{\!|\mathbf{thr1}\ e_{arg},\ E|\!\}\rangle & \\
\langle \#i\ (e_1,\ldots,e_n),\ E,\ K\rangle & \longmapsto & \langle e_1,\ E,\ \{\!|\mathbf{tup}\ i,\ (),\ (e_2,\ldots,e_n),\ E,\ K|\!\}\rangle & \\
\langle \mathtt{run}(e_{act}, e_k),\ E,\ K\rangle & \longmapsto & \langle e_{act},\ E,\ \{\!|\mathbf{run1}\ e_k,\ E|\!\}\rangle & \\
\langle \mathtt{forward}(e_{sig}),\ E,\ K\rangle & \longmapsto & \langle e_{sig},\ E,\ \{\!|\mathbf{forward}|\!\}\rangle & \\
\langle \mathtt{stop}(),\ E,\ K\rangle & \longmapsto & \langle \{\!|\mathbf{forward}|\!\},\ \mathtt{STOP}\rangle & (*)\\
\langle \mathtt{enqueueOnProc}(e_m, e_{tid}, e_k),\ E,\ K\rangle & \longmapsto & \langle e_m,\ E,\ \{\!|\mathbf{enq1}\ e_{tid},\ e_k,\ E,\ K|\!\}\rangle & \\
\langle \mathtt{provision}(e_m),\ E,\ K\rangle & \longmapsto & \langle e_m,\ E,\ \{\!|\mathbf{provision}\ K|\!\}\rangle & \\
\langle \mathtt{release}(e_m),\ E,\ K\rangle & \longmapsto & \langle e_m,\ E,\ \{\!|\mathbf{release}\ K|\!\}\rangle & \\
\langle \mathtt{ref}(e),\ E,\ K\rangle & \longmapsto & \langle e,\ E,\ \{\!|\mathbf{ref}\ K|\!\}\rangle & \\
\langle e_1\ \mathtt{:=}\ e_2,\ E,\ K\rangle & \longmapsto & \langle e_1,\ E,\ \{\!|\mathbf{assgn1}\ e_2,\ E,\ K|\!\}\rangle & \\
\langle\ \mathtt{!}e,\ E,\ K\rangle & \longmapsto & \langle e,\ E,\ \{\!|\mathbf{deref}\ K|\!\}\rangle & \\
\langle \mathtt{fetchAndAdd}(e_1, e_2),\ E,\ K\rangle & \longmapsto & \langle e_1,\ E,\ \{\!|\mathbf{faa1}\ e_2,\ E,\ K|\!\}\rangle & \\
\langle \mathtt{compareAndSwap}(e_1, e_2, e_3),\ E,\ K\rangle & \longmapsto & \langle e_1,\ E,\ \{\!|\mathbf{cas1}\ e_2,\ e_3,\ E\,K|\!\}\rangle & \\
\langle \mathtt{setTid}(e),\ E,\ K\rangle & \longmapsto & \langle e,\ E,\ \{\!|\mathbf{setTid}\ K|\!\}\rangle &
\end{array}
$$

$$
\begin{array}{rcll}
\langle \{\!|\mathbf{ap1}\ e_2,\ E,\ K\ |\!\},\ v_1\rangle & \longmapsto & \langle e_2,\ E,\ \{\!|\mathbf{ap2}\ v_1,\ E,\ K|\!\}\rangle & \\
\langle \{\!|\mathbf{ap2}\ [\mathtt{cl}_\lambda\ x.e,\ E']\ E,\ K|\!\},\ v_2\rangle & \longmapsto & \langle e,\ E'[x \mapsto v_2],\ K\rangle & (*)\\
\langle \{\!|\mathbf{thr1}\ e_{arg},\ E|\!\},\ v_k\rangle & \longmapsto & \langle e_{arg},\ E,\ \{\!|\mathbf{thr2}\ v_k,\ E|\!\}\rangle & \\
\langle \{\!|\mathbf{thr2}\ [\mathtt{cl}_k\ x.e,\ E',\ K],\ E|\!\},\ v_{arg}\rangle & \longmapsto & \langle e,\ E'[x \mapsto v_{arg}],\ K\rangle & (*)\\
\langle \{\!|\mathbf{tup}\ i,\ (v_1,\ldots,v_{n-1}),\ (),\ E,\ K|\!\},\ v_n\rangle & \longmapsto & \langle K,\ v_i\rangle & (*)\\
\multicolumn{4}{c}{\langle \{\!|\mathbf{tup}\ i,\ (v_1,\ldots,v_{j-2}),\ (e_j,\ldots,e_n),\ E,\ K|\!\},\ v_{j-1}\rangle}\\
\multicolumn{4}{c}{\longmapsto}\\
\multicolumn{4}{c}{\langle e_j,\ E,\ \{\!|\mathbf{tup}\ i,\ (v_1,\ldots,v_{j-2},v_{j-1}),\ (e_j,\ldots,e_n),\ E,\ K|\!\}\rangle}\\
\langle \{\!|\mathbf{run1}\ e_k,\ E|\!\},\ v_{act}\rangle & \longmapsto & \langle e_k,\ E,\ \{\!|\mathbf{run2}\ v_{act}|\!\}\rangle & \\
\langle \{\!|\mathbf{enq1}\ e_{tid},\ e_k,\ E,\ K|\!\},\ v_m\rangle & \longmapsto & \langle e_{tid},\ E,\ \{\!|\mathbf{enq2}\ v_m,\ e_k,\ E,\ K|\!\}\rangle & \\
\langle \{\!|\mathbf{enq2}\ v_m,\ e_k,\ E,\ K|\!\},\ v_{tid}\rangle & \longmapsto & \langle e_k,\ E,\ \{\!|\mathbf{enq3}\ v_m,\ v_{tid},\ K|\!\} & \\
\langle \{\!|\mathbf{assgn1}\ e_2,\ E,\ K|\!\},\ v_1\rangle & \longmapsto & \langle e_2,\ E,\ \{\!|\mathbf{assgn2}\ v_1,\ K|\!\}\rangle & \\
\langle \{\!|\mathbf{faa1}\ e_2,\ E,\ K|\!\},\ v_1\rangle & \longmapsto & \langle e_2,\ E,\ \{\!|\mathbf{faa2}\ v_1,\ K|\!\}\rangle & \\
\langle \{\!|\mathbf{cas1}\ e_2,\ e_3,\ E\,K|\!\},\ v_1\rangle & \longmapsto & \langle e_2,\ E,\ \{\!|\mathbf{cas2}\ v_1,\ e_3,\ K\ |\!\}\!\} & \\
\langle \{\!|\mathbf{cas2}\ v_1,\ e_3,\ E\,K|\!\},\ v_2\rangle & \longmapsto & \langle e_3,\ E,\ \{\!|\mathbf{cas3}\ v_1,\ v_2,\ K|\!\}\rangle &
\end{array}
$$

## C.2 VProc semantics

Semantics of a vproc executing a program $e$:

$$eval(e) = c \quad \text{if}$$
$$\{\!:\ \langle e,\ \emptyset,\ \{\!|\mathbf{stop}|\!\}\rangle,\ \mathbf{U}(0),\ \emptyset,\ switch :: [] \ :\!\} \Rightarrow^* \{\!:\ \langle \{\!|\mathbf{stop}|\!\},\ c\rangle,\ \mathit{VPMd},\ \mathit{Q},\ \mathit{S}\ :\!\}$$
where $switch$ is the top-level scheduler action.

Data definitions:

| | | | | | |
|---|---|---|---|---|---|
| $\mathit{VP}$ | $\in$ | $\mathit{VirtualProc}$ | $=$ | $\{\!:\ \mathit{St},\ \mathit{VPMd},\ \mathit{Q},\ \mathit{S}\ :\!\}$ | (vproc machine state) |
| $i,\ j$ | $\in$ | $\mathit{ThreadId}$ | | | (thread identification number) |
| $\mathit{Md}$ | $\in$ | $\mathit{Mode}$ | $=$ | $\mathbf{Idle}$ | (vproc mode) |
| | | | $\mid$ | $\mathbf{U}$ | |
| | | | $\mid$ | $\mathbf{M}$ | |
| $\mathit{VPMd}$ | $\in$ | $\mathit{VPMode}$ | $=$ | $\mathit{Md}(i)$ | (vproc state) |
| $\mathit{Q}$ | $\in$ | $\mathit{ReadyThreads}$ | $=$ | $\mathcal{P}(\mathit{ThreadID} \times \mathit{Cont})$ | (set of ready threads) |
| $\mathit{S}$ | $\in$ | $\mathit{SchedStack}$ | $=$ | $\mathit{List}(\mathit{Value}_d)$ | (scheduler action stack) |

Auxiliary functions:

| | | | |
|---|---|---|---|
| $push(S, act)$ | $=$ | $act :: S$ | (push scheduler action on the stack) |
| $pop([])$ | $=$ | $switch$ | (pop an empty stack) |
| $pop(act :: S)$ | $=$ | $(S, act)$ | (pop a scheduler action) |

Machine transition rules:

$$\frac{\mathit{St} \longmapsto \mathit{St}'}{\{\!:\ \mathit{St},\ \mathit{Md}(i),\ \mathit{Q},\ \mathit{S}\ :\!\} \Rightarrow \{\!:\ \mathit{St}',\ \mathit{Md}(i),\ \mathit{Q},\ \mathit{S}\ :\!\}}$$

**run**

$$\{\!:\ \langle \{\!|\mathbf{run2}\ v_{act}|\!\},\ v_k\rangle,\ \mathit{Md}(i),\ \mathit{Q},\ \mathit{S}\ :\!\} \quad \Rightarrow \quad \{\!:\ \langle e,\ E[x \mapsto \bullet],\ K\rangle,\ \mathbf{U}(i),\ \mathit{Q},\ push(S, v_{act})\ :\!\}$$
where $v_k = [\mathtt{cl}_k\ x.e,\ E,\ K]$

**forward**

$$\{\!:\ \langle \{\!|\mathbf{forward}|\!\},\ v_{sig}\rangle,\ \mathit{Md}(i),\ \mathit{Q},\ \mathit{S}\ :\!\} \quad \Rightarrow \quad \{\!:\ \langle e,\ E[x \mapsto v_{sig}],\ \{\!|\mathbf{stop}|\!\}\rangle,\ \mathbf{M}(i),\ \mathit{Q},\ \mathit{S}'\ :\!\}$$
where $(S', [\mathtt{cl}_\lambda\ x.e,\ E]) = pop(S)$

**preempt**

$$\{\!:\ \langle e,\ E,\ K\rangle,\ \mathbf{U}(i),\ \mathit{Q},\ \mathit{S}\ :\!\}$$
$$\Rightarrow$$
$$\{\!:\ \langle \mathtt{forward}(\mathtt{PREEMPT}(k)),\ [k \mapsto [\mathtt{cl}_k\ x.e,\ E,\ K]],\ \{\!|\mathbf{stop}|\!\}\rangle,\ \mathbf{U}(i),\ \mathit{Q},\ \mathit{S}\ :\!\}$$

**mask preemption**

$$\langle\!: \langle \texttt{mask}(),\ E,\ K\rangle,\ Md(i),\ Q,\ S :\!\rangle \quad \Rightarrow \quad \langle\!: \langle K,\ \bullet\rangle,\ \mathbf{M}(i),\ Q,\ S :\!\rangle$$

**unmask preemption**

$$\langle\!: \langle \texttt{unmask}(),\ E,\ K\rangle,\ Md(i),\ Q,\ S :\!\rangle \quad \Rightarrow \quad \langle\!: \langle K,\ \bullet\rangle,\ \mathbf{U}(i),\ Q,\ S :\!\rangle$$

## C.3 Multiprocessor semantics

Initial multiprocessor configuration for a program $e$:

$$\{\!: \{\langle\!: \langle e,\ \emptyset,\ \{\!|\mathbf{stop}|\!\}\rangle,\ \mathbf{U}(i),\ \emptyset,\ switch :: [] :\!\rangle_0\} \ ;\ \emptyset \ ;\ \emptyset :\!\}$$

where $switch$ = the top-level scheduler action.
$i$ = $freshTID()$

Data definitions:

| | | | | |
|---|---|---|---|---|
| $M$ | $\in$ | *Multiprocessor* | $=$ $\{\!: VPM\ ;\ Str\ ;\ PrMp :\!\}$ | (multiprocessor state) |
| $VPM$ | $\in$ | *VProcPool* | $=$ $VProcId \xrightarrow{\text{fin}} VirtualProc$ | (set of virtual processors) |
| $m, n$ | $\in$ | *VProcId* | | (vproc identification number) |
| $PrMp$ | $\in$ | *ProvisionMap* | $=$ $ThreadId \xrightarrow{\text{fin}} \mathcal{P}(VProcId)$ | (vproc provisioning map) |
| $l$ | $\in$ | *Loc* | $=$ $Nat$ | (store locations) |
| $Str$ | $\in$ | *Store* | $=$ $Loc \xrightarrow{\text{fin}} Value_d$ | (global store) |
| $v_d$ | $\in$ | $Value_d$ | $=$ $\ldots \mid l$ | (machine values) |

Auxiliary functions:

| | | |
|---|---|---|
| $enqueue(Q, thd)$ | $=$ $\{thd\} \cup Q$ | (enqueue a thread) |
| $dequeue(\emptyset)$ | $=$ $\emptyset$ | (dequeue on empty queue) |
| $dequeue(\{thd\} \cup Q)$ | $=$ $(Q, thd)$ | (dequeue a thread) |
| $freshTID()$ | $=$ $i$ | (create a fresh thread id) |

Machine transition rules:

$$\frac{\langle\!: St,\ VPMd,\ Q,\ S :\!\rangle \ \Rightarrow\ \langle\!: St',\ VPMd',\ Q,\ S' :\!\rangle}{\begin{array}{l}\{\!: VPM,\ \langle\!: St,\ VPMd,\ Q,\ S :\!\rangle_m\ ;\ Str\ ;\ PrMp :\!\} \\ \quad \Rrightarrow\ \{\!: VPM,\ \langle\!: St',\ VPMd',\ Q,\ S' :\!\rangle_m\ ;\ Str\ ;\ PrMp :\!\}\end{array}}$$

**block on empty queue**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \mathbf{dequeue}(),\ E,\ K\rangle,\ \mathit{Md}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \mathbf{dequeue}(),\ E,\ K\rangle,\ \mathbf{Idle}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\text{where}\quad \emptyset \ =\ \mathit{dequeue}(Q)$$

**dispatch another vproc on enqueue**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{enq3}\ n,\ v_{tid},\ K|\!\}\ v_k\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_m,$$
$$\langle\!:\ \mathit{St},\ \mathbf{Idle}(i),\ Q',\ S'\ :\!\rangle_n\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ \bullet\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_m,\ \langle\!:\ \mathit{St},\ \mathbf{U}(i),\ Q'',\ S'\ :\!\rangle_n\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\text{where } Q'' = \mathit{enqueue}(Q', (v_{tid}, v_k))$$

**enqueue thread on self**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{enq3}\ m,\ v_{tid},\ K|\!\}\ v_k\rangle,\ \mathit{Md},\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ \bullet\rangle,\ \mathit{Md},\ Q',\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\text{where } Q' = \mathit{enqueue}(Q, (v_{tid}, v_k))$$

**dequeue thread**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \mathbf{dequeue}(),\ E,\ K\rangle,\ \mathit{Md}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ (v_k, i)\rangle,\ \mathit{Md}(i),\ Q',\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\text{where}\quad (Q', (v_k, i)) \ =\ \mathit{dequeue}(Q)$$

**enqueue thread on another vproc**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{enq3}\ n,\ v_{tid},\ K|\!\}\ v_k\rangle,\ \mathit{Md},\ Q,\ S\ :\!\rangle_m,$$
$$\langle\!:\ \mathit{St},\ \mathit{Md}',\ Q',\ S'\ :\!\rangle_n\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ \bullet\rangle,\ \mathit{Md},\ Q,\ S\ :\!\rangle_m,\ \langle\!:\ \mathit{St},\ \mathit{Md}',\ Q'',\ S'\ :\!\rangle_n\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\text{where } Q'' = \mathit{enqueue}(Q', (v_{tid}, v_k))$$

**provision vprocs**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{provision}\ K|\!\},\ k\rangle,\ \mathit{Md}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp},\ \mathit{vprocs}_i\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ (\mathit{avail}_1, \ldots, \mathit{avail}_{\min(k,|\mathit{avail}|)})\rangle,\ \mathit{Md}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp},\ \mathit{vprocs}'_i\ :\!\}$$
$$\text{where}\quad \begin{array}{rcl} \mathit{vprocs}'_i & = & \mathit{vprocs}_i \cup \mathit{avail} \\ \mathit{avail} & = & \mathrm{dom}(\mathit{VPM}) \setminus \mathit{vprocs}_i \setminus \{m\} \end{array}$$

**release vprocs**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{release}\ K|\!\},\ n\rangle,\ \mathit{Md}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp},\ \mathit{vprocs}_i\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ \bullet\rangle,\ \mathit{Md}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp},\ \mathit{vprocs}'_i\ :\!\}$$
$$\text{where}\quad \mathit{vprocs}'_i = \mathit{vprocs}_i \setminus \{n\}$$

**add a vproc**

$$\{\!:\ \mathit{VPM}\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \mathit{dispatch}(),\ \emptyset,\ \{\!|\mathbf{stop}|\!\}\rangle,\ \mathbf{U}(i),\ \emptyset,\ \mathit{switch} :: [] \ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\text{where}\quad \begin{array}{rcl} i & = & \mathit{freshTID}() \\ m & \notin & \mathrm{dom}(\mathit{VPM}) \end{array}$$

**remove a vproc**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \mathit{St},\ \mathbf{Idle}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM}\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$

**ref**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{ref}\ K|\!\},\ v\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ l\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}[l \mapsto v]\ ;\ \mathit{PrMp}\ :\!\}$$
$$\text{where}\quad l \notin \mathrm{dom}(\mathit{Str})$$

**assign**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{assgn2}\ l,\ K|\!\},\ v\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ \bullet\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}[l \mapsto v]\ ;\ \mathit{PrMp}\ :\!\}$$

**deref**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{deref}\ K|\!\},\ l\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ \mathit{Str}(l)\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$

**fetch and add**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{faa2}\ l,\ K|\!\},\ v\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ \mathit{Str}(l)\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}[l \mapsto \mathit{Str}(l) + v]\ ;\ \mathit{PrMp}\ :\!\}$$

**compare and swap**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{cas3}\ l,\ v_1,\ K|\!\},\ v_2\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ \mathit{Str}(l)\rangle,\ \mathit{VPMd},\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}'\ ;\ \mathit{PrMp}\ :\!\}$$
$$\text{where}\quad \mathit{Str}' \;=\; \begin{cases} \mathit{Str} & \text{if } \mathit{Str}(l) \neq v_1 \\ \mathit{Str}[l \mapsto v_2] & \text{otherwise} \end{cases}$$

**get thread ID**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \mathtt{getTid}(),\ E,\ K\rangle,\ \mathit{Md}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ i\rangle,\ \mathit{Md}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$

**new thread ID**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \mathtt{newTid}(),\ E,\ K\rangle,\ \mathit{Md}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ freshTID()\rangle,\ \mathit{Md}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$

**set thread ID**

$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle \{\!|\mathbf{setTid}\ K|\!\},\ j\rangle,\ \mathit{Md}(i),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$
$$\Rrightarrow$$
$$\{\!:\ \mathit{VPM},\ \langle\!:\ \langle K,\ \bullet\rangle,\ \mathit{Md}(j),\ Q,\ S\ :\!\rangle_m\ ;\ \mathit{Str}\ ;\ \mathit{PrMp}\ :\!\}$$