A Foundation for an Efficient Multi-Threaded Scheme System

Suresh Jagannathan Jim Philbin
NEC Research Institute
4 Independence Way
Princeton, NJ 08540
`<suresh | philbin>@research.nj.nec.com`

Abstract

We have built a parallel dialect of Scheme called STING that differs from its contemporaries in a number of important respects. STING is intended to be used as an operating system substrate for modern parallel programming languages.

The basic concurrency management objects in STING are first-class lightweight threads of control and virtual processors (VPs). Unlike high-level concurrency structures, STING threads and VPs are not encumbered by complex synchronization protocols. Threads and VPs are manipulated in the same way as any other Scheme structure.

STING separates thread policy decisions from thread implementation ones. Implementations of different parallel languages built on top of STING can define their own scheduling and migration policies without requiring modification to the runtime system or the provided interface. Process migration and scheduling can be customized by applications on a per-VP basis.

The semantics and implementation of threads minimizes the cost of thread creation, and puts a premium on storage locality. The storage management policies in STING lead to better cache and page utilization, and allows users to experiment with a variety of different execution regimes – from *fully delayed* to *completely eager* evaluation.

1 Introduction

The growing availability of general-purpose multiprocessors has led to increased interest in building efficient and expressive platforms for concurrent programming. Most efforts to incorporate concurrency into high-level symbolic programming languages such as Scheme[1, 23] or ML[18] involve the addition of special-purpose primitives (*e.g.*, parallel `let` operations[9], futures[11], events[24], etc.). These primitives are typically implemented using a dedicated runtime system sensitive to the particular semantics of these primitives.

While reasonably efficient, these systems have nonetheless proven to be difficult to use as a substrate or foundation for a concurrent (symbolic) programming environment. This is because (a) the high-level semantics of the concurrency primitives they support lead to complex or inefficient implementations of other concurrency structures or paradigms; (b) the inaccessibility of the language's runtime structures make it cumbersome for applications to tailor the implementation based on their particular requirements; and, (c) the reliance on operating system services for process management and control found in many of these languages incurs high overhead in the presence of fine-grained, interactive, or realtime concurrency.

This paper describes an alternative approach to implementing concurrent symbolic programming languages. We have built a parallel dialect of Scheme called STING that differs from its contemporaries in a number of important respects. We outline the salient properties of the system below and elaborate upon these points throughout the paper:

Generality: The fundamental concurrency objects in STING are first-class lightweight threads of control and virtual processors (VPs). Unlike high-level concurrency structures, STING threads and VPs are not encumbered by complex synchronization protocols. Threads define a representation for a *process* or *task* and execute concurrently with other threads. Virtual processors are an abstraction of a physical computing device. VPs can be tailored to implement specialized process migration and scheduling protocols. Threads and VPs are manipulated in the same way as any other Scheme structure.

Efficiency: Because threads are fully integrated into the base language, and not provided as part of a library package[6], it is straightforward to optimize their implementation and use.

Thread operations are performed by a thread controller implemented as a procedure that allocates no storage. In addition, the semantics of threads and the design of the thread controller

---

Appears in the Proceedings of the 1992 ACM Symposium on Lisp and Functional Programming.

minimizes the cost of thread creation, and puts a premium on storage locality. Thus, the execution context for a newly terminated thread (*e.g.,* its stack and heap) is recycled and used immediately by other newly created threads, storage allocation is delayed until the thread actually executes, and different threads share the same stack and heap if data dependencies warrant. The storage management policies in STING lead to better cache and page utilization, and allows users to experiment with a variety of different execution regimes – from *fully delayed* to *completely eager* evaluation.

Programmability: Beyond providing mechanisms for managing concurrency, STING also handles exceptions across thread boundaries, preemption, non-blocking I/O, asynchronous storage management, and maintains extensive thread genealogy information. In addition, STING provides an infra-structure for implementing multiple address spaces, and long-lived persistent objects. As a result, we envision STING as an expressive operating system substrate upon which an advanced programming environment for parallel symbolic computing can be built.

Customizability: STING separates thread policy decisions from thread implementation ones. Although all threads conform to the same basic structure, implementations of different parallel languages built on top of STING may define their own scheduling, migration, and load-balancing policies without requiring modification to the thread controller or the provided interface. Process migration and scheduling concerns typically handled internally by a runtime library or an operating system in other languages can be customized by applications on a per-VP basis.

Unlike other systems that implement application dependent scheduling policies[28], STING does not incur a performance penalty for this flexibility; scheduling policy decisions are implemented entirely in STING and thus do not require a trap into a low-level system kernel.

The focus of this paper is on the design and implementation of the system. The implications of its design and its utility in expressing high-level paradigms for parallel symbolic computing are discussed in [13].

The remainder of the paper is structured as follows. Section 2 gives a brief description of the STING compiler and runtime system. Section 3 describes the structure of threads, and outlines the design of the thread manager. Section 4 provides a brief discussion the structure of virtual processors and support for process scheduling and thread migration. Section 5 presents details on thread creation, and runtime optimizations are discussed in Section 6. Section 7 gives an outline of the garbage collector. We present performance figures in Section 8. Section 9 gives comparison to related work, and Section 10 presents future directions.

2 The STING Abstract Machine

2.1 The Abstraction Hierarchy

The STING runtime environment is organized as a layer of abstractions. The lowest level consists of a physical machine containing a number of physical processors. Physical machines implement *virtual machines*; there may be many virtual machines executing on a single physical one. Each virtual machine may contain a number of *virtual processors.*

Each physical processor corresponds to a computing engine in a multiprocessor environment. Associated with every physical processor P is a physical policy manager (PPM) that handles scheduling decisions for the virtual processors that execute on P. A PPM will context switch among virtual processors because of preemption, or because it is explicitly requested to do so[1]. A physical processor may execute virtual processors associated with different virtual machines.

Virtual machines encapsulate a single address space managed exclusively by its virtual processors. Virtual machines may share global information (*e.g.,* libraries, file systems, etc.), and are responsible for mapping global objects (*i.e.,* objects resident in a global address space) into their local address space. Virtual machines also contain the root of a live object object graph (*i.e.,* the root environment) that is used to trace all live objects in its address space.

Virtual processors are responsible for executing and scheduling lightweight threads of control. In addition, they manage software interrupts (page faults, preemption, etc.). Different VPs within the same virtual machine may be closed over different scheduling policy managers. State transition operations on threads are implemented by a thread controller common to all VPs in the system.

All STING objects (including threads, VPs, and virtual machines) reside in a persistent memory. The memory is organized in terms of a collection of disjoint areas[4]. Objects are garbage collected within an area using a generational collector[17, 25]. An object can reference any other object found in its address space. Objects initially reside in short-lived thread-local areas. Objects that persist percolate upwards in the generation hierarchy. This functionality is completely transparent to the user.

2.2 Extensions to Scheme

The STING compiler is a modified version of Orbit[15]. The consing routine implemented by the compiler uses write-protected guard pages found at the top of every thread heap; an attempt to write onto this page triggers garbage collection. The garbage collector itself is significantly different from Orbit's because heaps and stacks are local to threads, and because inter-area references are supported.

---

[1] For example, a virtual processor may relinquish control of its physical processor if no threads are executing on it, and none are migratable from other VPs.

The target machine seen by the compiler also includes a dedicated thread register to hold a reference to the currently running thread object. Moreover, time critical operations such as those that save and restore registers on a context switch, allocate thread storage (*i.e.,* stacks and heaps), and execute test-and-set functions are provided as primops. Sequential Scheme programs will compile and execute without modification. STING implementations of futures, distributed data structures[5], and speculative concurrency operations also exist; Scheme programs can be freely augmented with the concurrency operations supported by any of these paradigms.

3 Threads

A STING thread defines a separate locus of control. A thread has a static and dynamic component (see Fig. 1). The static portion of a thread T contains state information indicating T's status (*e.g.*, delayed, scheduled, evaluating, etc.), a lock, the procedure P to be run, waiting threads blocked on the completion of T, P's dynamic and exception environment, and genealogy information indicating T's parent, children and sibling threads. Genealogy information is used for debugging and analysis. The value yielded by P's application is also stored in the thread upon completion.

Currently evaluating threads are associated with a *thread control block* (or TCB). The TCB contains information about the thread's dynamic context; a TCB is a generalized representation of a continuation and is closed over its own stack and heap. Besides storage objects, a TCB includes an associated lock, values of all live registers extant at the time the thread last performed a context switch, the current dynamic state of the running thread (*e.g.,* initialized, ready, evaluating, blocked, suspended, etc.), the next transition state for the thread, the VP on which the thread was last executing, thread priority, and time quantum. We introduce the operations allowable on threads and TCBs during the course of the paper.

The relationship between thread states and TCB states is given in Fig. 2. A thread in state *delayed* corresponds to a delayed or lazy object. A delayed thread will never be run unless the value of the thread's procedure is explicitly demanded. A *scheduled* thread is a thread known to a virtual processor, but which has not yet started evaluating. An *evaluating* thread is a thread which has been allocated a TCB and has started executing on some processor. Stolen threads are an important optimization of evaluating threads and are discussed in Section 6. A thread is *determined* if its associated procedure has been executed.

TCB states reflect the operations allowed on running threads. If running thread T has TCB T_b, the state field of T_b indicates one of the following:

initialized: The stack and heap associated with T_b have been initialized, but no code has yet been executed.

ready: T can execute on any available VP, but is currently not.

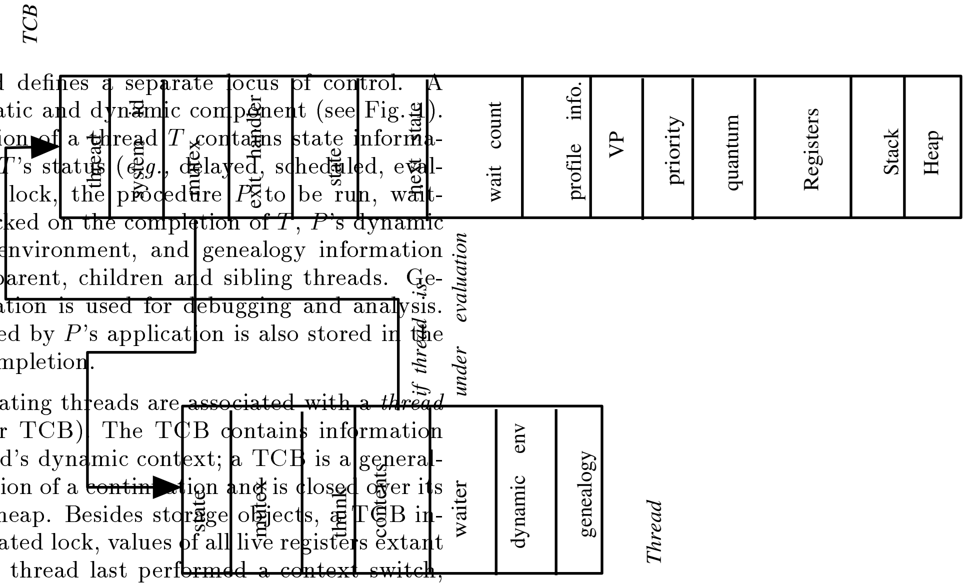

Figure 1: Organization of threads and TCBs.

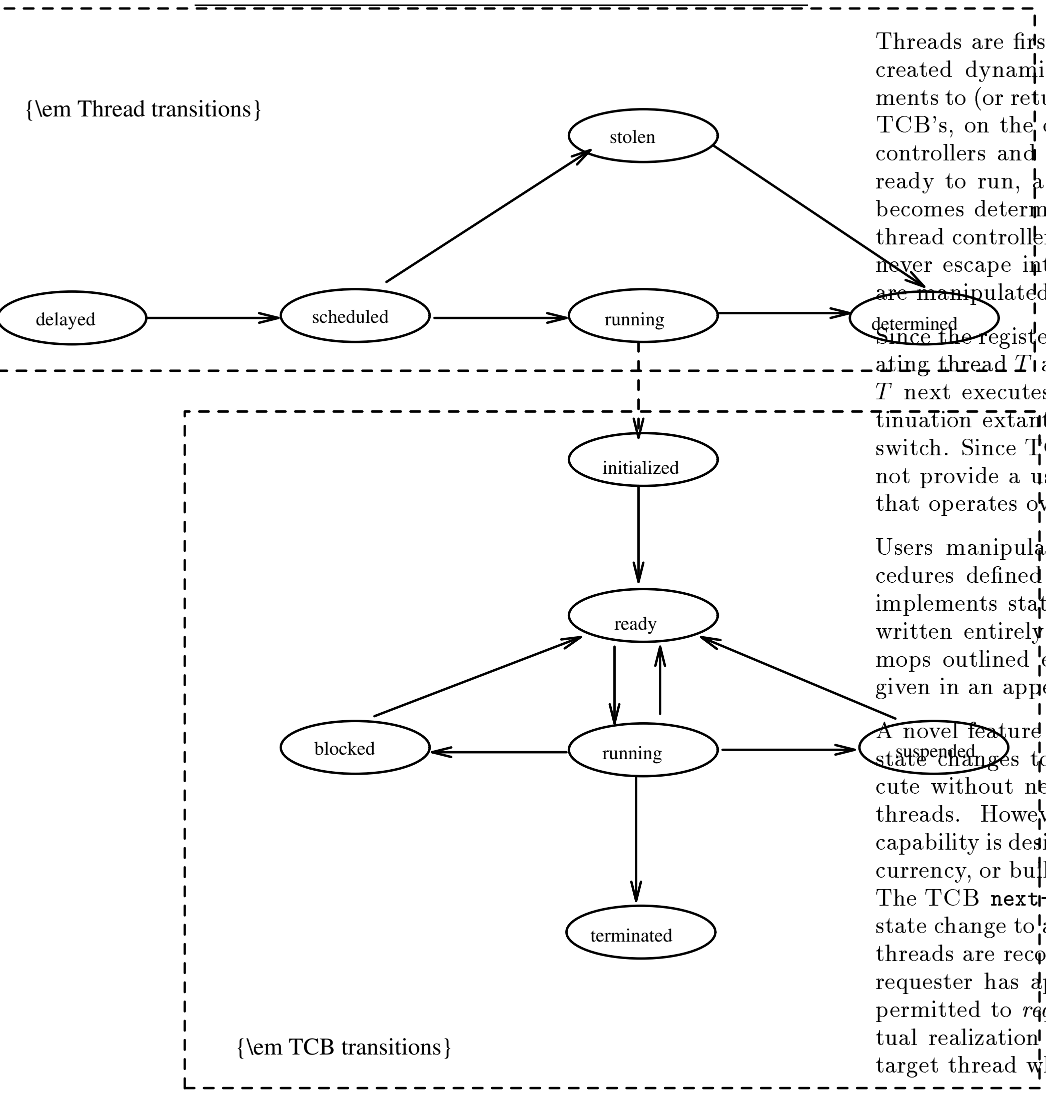

Figure 2: Transition diagram for threads and TCBs.

running: T is currently executing on some VP.

blocked: T is currently blocked on some thread or condition.

suspended: T is suspended for some potentially infinite duration, and exists on a suspend queue of some VP.

terminated: T has finished executing.

Threads are first-class objects in STING and thus may be created dynamically, stored in objects, passed as arguments to (or returned as results from) procedures. Thread TCB's, on the other hand, are accessible only to thread controllers and policy managers. When a new thread is ready to run, a TCB is allocated for it; when a thread becomes determined, its TCB is available for use by the thread controller for threads created subsequently. TCBs never escape into user maintained data structures; they are manipulated exclusively by system-level procedures.

Since the register values recorded in the TCB of an evaluating thread T are used to establish thread context when T next executes, T's TCB represents the thread's continuation extant at the time T last performed a context switch. Since TCBs are not released to users, STING does not provide a user-level operation analogous to `call/cc` that operates over TCBs.

Users manipulate threads and TCBs via a set of procedures defined by the thread controller (TC); the TC implements state transition operations on threads. It is written entirely in STING with the exception of the primops outlined earlier. The core of the TC interface is given in an appendix.

A novel feature of the design permits threads to request state changes to other threads. In general, threads execute without need for requesting state changes to other threads. However, in certain important instances, such capability is desirable (*e.g.,* implementing speculative concurrency, or building common operating system services). The TCB `next-state` field provides this functionality. A state change to an evaluating thread T requested by other threads are record in T's `next-state` field (provided the requester has appropriate authority). Threads are only permitted to *request* a change to another thread; the actual realization of this change can only be done by the target thread when it next enters the thread controller.

The actual next state of a thread is determined by comparing its desired next state with its `next-state` field. This comparison is based on a simple total ordering:

`terminate` $\geq$ `suspend` $\geq$ `block` $\geq$ `ready` $\geq$ `running`

Because the ordering on state transitions is total and because only a thread can change its own state, the thread controller does not lock a TCB in order to determine the TCB's next state; if the next state requested is less than the current state, it can be ignored.

4 The Virtual Processor

A virtual processor is closed over a thread controller, a policy manager that defines the scheduling policy for the VP, a local "cache" of TCBs, and interrupt handlers that service page faults and preemption. Policy managers are closed over a set of queues holding threads and TCBs in various states of execution (*e.g.,* scheduled, ready, suspended, etc.).

The TCB cache is used by a VP to efficiently assign a dynamic context to a new thread. When a thread terminates, its TCB is reinitialized and kept in the VP TCB cache; a new thread created immediately thereafter on this VP is assigned this TCB. The overhead of thread/TCB assignment is reduced given the high likelihood that pieces of the terminated thread's TCB will still be in the VP's working set. All VPs within a virtual machine also share access to a global TCB pool (initialized at system startup) that is used if their cache is exhausted. TCBs are never created dynamically; the overhead in using TCBs is limited to their initialization and assignment to threads.

Virtual processors and virtual machines are first-class objects; part of the virtual machine state is a vector closed over the virtual processors defined by the machine. The system provides an interface to directly map virtual processors onto physical processors[22]. Thus, it is possible for applications to explicitly map computations onto specific processors as dictated by the parallel algorithm being implemented. For example, a process P known to communicate closely with Q (which is executing on VP V) should also run on a VP topologically near V[13]. As another example, first-class VPs and virtual machines make it possible to map arbitrary abstract topologies onto a concrete architectural surface[10].

The scheduling and migration routines executed by a virtual processor are customizable. Thus, different VPs may be closed over different scheduling regimes. The ability to customize policy managers has some significant implications. First, applications are free to optimize the scheduling of threads based on expected runtime dynamics of thread creation, longevity, and communication. Second, tasks found within a given application may be mapped onto virtual processors that implement the scheduling policy best suited for them. Different VPs can be dedicated to implement different schedulers; applications then map tasks onto VPs based on the scheduler required.

The policy manager implements a scheduling policy for the threads found on its VP, establishes a priority and quantum for the currently executing thread, and a load balancing policy that dictates how threads get mapped onto VPs. Load balancing in this context addresses two concerns: (1) how do threads initially get associated with a VP?, and (2) what are the criteria by which threads migrate across VPs?

The policy manager may implement any one of a number of scheduling policies (*e.g.,* FIFO, LIFO, realtime, interactive, round-robin, etc.). The overhead in accessing and updating thread queues can be customized as well; thus, a policy manager can classify threads as local (non-migratable) or global (migratable); local threads are stored on queues that involve no lock manipulation. There may be several ready queues holding threads with various priorities; long-lived threads may be stored on queues with lower priority than newly created threads. VPs may share access to global queues or can maintain their own local (private) images. The STING runtime environment provides a number of default policy managers; these managers (all written in STING) perform no dynamic storage allocation. Thus, users need not be aware of the policy manager to write STING programs. Applications that rely exclusively on these schedulers are guaranteed not to have threads trigger garbage collection on a state transition. We describe the semantics of some of the policy manager operations in the following sections.

5 Thread Creation and Execution

The implementation of the STING thread controller highlights a number of interesting issues. The state transition procedure is shown in Fig. 3. Note that operations on TCBs found in this procedure are not available to user applications.

The procedure takes two arguments – the desired next state for the current thread (*i.e.,* the thread which has entered the TC), and an argument field containing auxiliary information for threads that block or suspend. The procedure `get-tcb-next-state` returns the actual next state of the current TCB by comparing the value recorded in its `next-state` field with the `desired-next-state`.

Since the thread controller is written in STING, all synchronous calls to TC procedures are treated as ordinary procedure calls; thus, live registers used by the procedure running in the current thread are saved automatically in the thread's TCB. The procedure first attempts to acquire a new thread to execute from the ready queue of this VP. The policy manager procedure `pm-get-next-thread` is used for this purpose. Note that `pm-get-next-thread` enqueues the current thread's TCB if the TCB is in a ready state; this permits the thread to be rerun at some future point.

If the queue is empty, the root TCB of the current VP is invoked (via the call to `vp.root-tcb vp`). This procedure may (a) perform housekeeping operations, (b) simply reinvoke the state transition procedure, or (c) request the PPM to switch to a new VP.

If the queue is not empty, a new thread (or TCB) is returned. A new thread object is returned only if the thread is *not* evaluating. Such an object (call it T) is created and scheduled as a consequence of evaluating the expression:

`(fork-thread` *expr vp*`)`

This operation enqueues T on *vp*'s ready queue. It becomes eligible for evaluation when the TC removes it from the queue.

Once a thread begins evaluation, it is never directly stored in any queue maintained by a VP. Its TCB is stored

```
(define (state-transition desired-next-state arg)

  (receive (actual-next-state arg)
     (get-tcb-next-state desired-next-state arg)
    (set-new-state actual-next-state arg)
    (let ((vp (current-vp)))
      (let ((next
             (let ((next-thread (pm-get-next-thread vp)))
               (if (false? next-thread)
                   (vp.root-tcb vp)
                   next-thread))))
        (cond ((eq? next (current-tcb)))
              ((tcb? next)
               (set-tcb.state next tcb-state/running)
               (if (eq? desired-next-state tcb-state/dead)
                   (return-current-tcb-to-vp-pool vp))
               (set-vp.current-tcb vp next)
               (set-thread.vp (tcb.thread next) vp)
               (save-current-tcb-registers)
               (restore-tcb-and-registers next))
              ((thread? next)
               (if (thread-stolen? thread)
                   (state-transition desired-next-state arg)
                   (let ((tcb (setup-new-thread next-state next vp)))
                     (save-current-tcb-registers)
                     (start-new-tcb tcb start-new-thread)))))))))
```

Figure 3: Implementation of the state transition procedure. We use "." notation to express structure selection.

instead[2]. Thus, a TCB returned by `pm-get-next-thread` is always associated with an `evaluating` thread.

The outermost conditional in `state-transition` performs the actual context switch. If the current TCB happens to be the TCB returned by `pm-get-next-thread`, it is simply rerun – no extra register saves or restores need be performed (other than those needed to execute the call/return sequence to/from this procedure).

If the object returned by `pm-get-next-thread` is another (distinct) TCB, all live TCB registers of the current TCB are saved, and the continuation encapsulated in the TCB returned is restored. If the current TCB is dead (because it has terminated), it is recycled in the current VP's TCB pool. Because `restore-tcb-and-registers` is a primop, the compiler treats `save-current-tcb-registers` as if it were in a tail call position; thus, when the saved thread resumes execution (*i.e.,* assuming its TCB has not terminated), it simply returns from `state-transition`.

If the object returned is a thread, a TCB is allocated for it via the call `setup-new-thread` *provided* the thread has not been stolen (see Section 6). The current thread state is saved (via the call `save-current-tcb-registers`, and the new thread begin execution via the call to the primop `start-new-tcb`. This primop sets up a new continuation encapsulating the evaluation of `start-new-thread` and commences its evaluation using `tcb` as its dynamic context. Here again, the compiler treats the register save operation as if it were in a tail call position; thus, when the saved thread resumes execution it simply falls through the conditional and returns to the caller.

The code for `start-new-thread` is shown in Fig. 4.

A thread object with thunk E_t can begin evaluation once a TCB is allocated for it, and it becomes associated with a default error handler and appropriate cleanup code. Throws to `exit` (the default error handler) from E_t cause the thread stack to be unwound properly, thereby permitting resources such as locks held by the thread to be properly released. The exit code following the evaluation of E_t garbage collects the thread stack and heap, stores the value yielded by E_t as part of the thread state, wakes up all threads waiting for this value, reinitializes the TCB state and stores it back into the TCB pool (via the call to `re-initialize-tcb`), and finally makes a tail recursive call to the top-level transition procedure to choose a new thread to run. Because E_t is wrapped within a dynamic wind form, we guarantee that thread storage will be garbage collected and initialized even if a thread terminates abnormally.

Garbage collection must take place before the thread's waiters are awakened because objects that outlive the thread (including the object returned by the thread's thunk) found on its heap must be migrated to another live heap. Failure to do so would allow other threads to obtain references to the newly terminated thread's storage.

Preemption is disabled within the TC. Disabling preemp-

---

[2] The thread object associated with a non-terminated TCB is accessible via the thread slot found in that TCB.

```
(define (start-new-thread)

  (let ((z (error-value "Thread has no value")))
    (unwind-protect (set z (catch exit
                               (set-tcb.exit-handler (current-tcb) exit)
                               ((thread.thunk (current-thread)))))
      (let ((thread (current-thread)))
        (set-tcb.state (current-tcb) tcb-state/terminated))
        (thread-gc thread)
        (set-thread.value thread z)
        (wakeup-waiters thread)
        (re-initialize-tcb (current-tcb))
        (record-current-thread-death)
        (state-transition tcb-state/dead '#f))
```

Figure 4: Starting a new thread.

tion guarantees that the TC will not be interrupted as a result of timer expiration. Other interrupts need not be disabled since TC-maintained data structures are not manipulated by VP interrupt handlers. Each VP has a separate area (implemented using small stacks and heaps) used by interrupts and the garbage collector for servicing any storage requirements they may have.

6 Stealing Threads

The STING implementation defers the allocation of storage for a thread until necessary. In many thread packages, the act of creating a thread involves not merely setting up the environment for the process to be forked, but also allocating and initializing storage. This approach lowers efficiency in two important respects: first, in the presence of fine-grained parallelism, the TC may spend more time creating and initializing threads than actually running them. Second, since stacks and process control blocks are immediately allocated upon thread creation, context switches among threads often cannot take advantage of cache and page locality.

Page locality in thread create operations is obviously important. Consider a virtual processor P executing thread T_1. Suppose T_1 spawns a new thread T_2. Under a fully eager thread creation strategy, storage for T_2 will be allocated from a new TCB. T_2's heap and stack will be mapped to a new set of physical pages in P's virtual machine. Thus, assume T_2 runs on P only after T_1 completes (*e.g.,* because there is no other free processor available or because T_1 is strict in the value yielded by applying T_2's thunk). Setting up T_2 now involves a fairly costly context switch. The working set valid when T_1 was executing is now possibly invalid; in particular, the contents of the data and instruction cache containing T_1's physical pages will probably need to be flushed when T_2 starts executing.

STING reduces the cost of eager thread allocation by deferring the acquisition of TCB stack and heap areas for a thread until it is actually about to be run. If an evaluating thread terminates, the next ready thread to be executed can *reuse* the TCB stack and heap space previously allocated to the terminated thread[16]. When a thread exits, its stack and heap storage are recycled for use by subsequent evaluating threads. Reusing thread storage increases locality since the contents of the processor cache and page mapping tables will contain pieces of the terminated process stack and heap. Just as significantly, delaying the allocation of thread storage minimizes the amount of housekeeping required; threads that are scheduled but which are never run (*e.g.,* because of unnecessary speculative parallelism) need never be allocated storage. Moreover, one processor can migrate such threads from another without inducing significant data movement since scheduled or delayed threads have small storage requirements.

The distinction between threads that are scheduled and those that are evaluating is used in STING to optimize the implementation of non-strict structures such as *futures*[11]; it's also used to throttle the unfolding of the process call tree in fine-grained parallel programs. In a naive implementation, a process that accesses a future (or which initiates a new process whose value it requires) blocks until the future becomes determined (or the newly instantiated process yields a value). This behavior is sub-optimal for the reasons described above – cache and page locality is compromised, bookkeeping information for context switching increases, and processor utilization is not increased since the original process must block until the new process completes.

STING implements the following optimization: a thunk t associated with a thread T that is in a *scheduled* or *delayed* state is applied using the dynamic context of a thread S if S demands T's value. The semantics of future/touch (or any similar producer/consumer protocol) is not violated. The rationale is straightforward: since T has not been allocated storage and has no dynamic state information associated with it, t can be treated as an ordinary procedure and evaluated using the TCB already allocated for S. In effect, S and T share the same dynamic storage. T's state is set to *stolen* as a consequence. A thread is stolen if its thunk executes using the TCB associated with another running thread.

Thus, in the STING implementation of futures[3], a process can run the code associated with an unevaluated future it touches within its own dynamic context – no TCB is allocated in this case and the accessing process does not block. A consumer blocks only when the producer has already started evaluating. The producer is already associated with a TCB in this case, and no opportunity for stealing presents itself.

Applying the closure of a stolen thread incurs some overheads not present in an ordinary procedure call. (Let T be a scheduled or delayed thread and let S be an evaluating thread that demands the value of T's thunk.) To steal S, the implementation performs the following actions:

1. The thread slot in S's TCB and the TCB slot in T is updated.
2. The dynamic state of S is stored in T; dynamic state information is used in case T raises an exception or throws to a catch point stored in S's TCB stack.
3. A catch point is wrapped around the application of S's thunk guaranteeing the stack and heap are in a consistent state if control throws out of the procedure being applied.
4. Thread related bookkeeping (*e.g.,* waking blocked threads, recording termination, etc.) is performed once S becomes determined.

These overheads are not significant, involving mostly stores to data structures (stacks, heaps, and records). No garbage collection is required in order to implement thread stealing.

Thread stealing is distinguished from the lazy task creation mechanism (also known as lazy futures) found in Mul-T[19] or Gambit[8], and load based-inlining as implemented in WorkCrews[26]. Whereas stealing permits the work needed to start a thread running to be deferred as long as possible, Mul-T lazy task creation is a technique intended to increase process granularity by permitting inlined tasks to be retroactively forked based on runtime conditions; load-based inlining in WorkCrews permits programs to notify free processors that a task is available for execution. Unlike either of these two approaches, thread stealing is not intended to be an *implementation* of a load balancing policy, although it does simplify the task of building such implementations; instead, the stealing operation prevents saturating the virtual machine with storage allocation requests, and improves cache and page performance of newly created threads. This is especially useful in result parallel programs in which processes typically exhibit significant data dependencies among one another. Neither lazy task creation nor load based inlining are intended to improve storage locality.

6.1 Synchronization

The primitive synchronization operation on threads is `thread-wait`. Given a thread T as its argument, this procedure first checks if T is determined. If T is under evaluation, the procedure stores the current thread in a queue, q, accessed via the `waiters` slot in T, and blocks the current thread[4]. When T becomes determined, all waiting threads on q are rescheduled using the procedure `wakeup-waiters`.

The *value* of a thread is the result yielded by applying its thunk. Threads can ask for the value of other threads using the procedure `thread-value` shown in Fig. 6.

If the argument thread T is determined, and its value is another thread, the procedure is applied recursively to this new thread; otherwise, the value is returned.

If T is either scheduled or delayed, T's state is set to stolen and it's thunk is evaluated using the current thread's TCB. `Run-stolen-thread` implements this behavior. Otherwise, the current thread blocks (via the `thread-wait` procedure); when it is rescheduled, its continuation makes a tail-recursive call to find T's value.

7 Garbage Collection

Each thread has a stack, a private heap and a public heap. Objects located in the stack are always private to the thread and are never collected. Objects in the private heap are accessible only to the thread that created them, while objects in the public heap are accessible to any thread in the system. A private object can become public by allowing a reference to it to escape the dynamic context of the thread in which it's found. Such objects are copied from the thread's private heap to its public heap. Both the private and public heaps use a generational style copying garbage collector. Any heap in the system can be collected independently of any other, but during the collection, objects in the heap are inaccessible to threads.

Our investigations and those of others[17, 25] indicate that most objects are created in the private heap nursery and die there. Because it is private, this heap can be collected by the thread at any time without affecting any other thread in the system. The root set for this collection is contained in the thread's stack and is thus usually quite small.

While any garbage collection is synchronous with respect to the thread doing the collecting, it is *asynchronous* with respect to other threads in the system. This type of asynchrony offers several benefits over completely synchronous garbage collectors[14]: (1) useful computation can proceed in the system even if some threads are garbage collecting, (2) the cost of garbage collection is not distributed evenly across all threads, but is charged proportionately to the amount of storage allocated by each thread; and (3) data locality is enhanced since objects are allocated in an area managed exclusively by the thread that creates them.

---

[3] Threads are a natural representation structure for futures. A future is created in sting using fork-thread; a touch on a future is equivalent to applying thread-value (see Section 6.1) on its thread argument.

[4] The policy manager does not store a blocked thread in any structure it maintains. Blocking is therefore tantamount to a context-switch in which the current thread is not recorded by the thread controller.

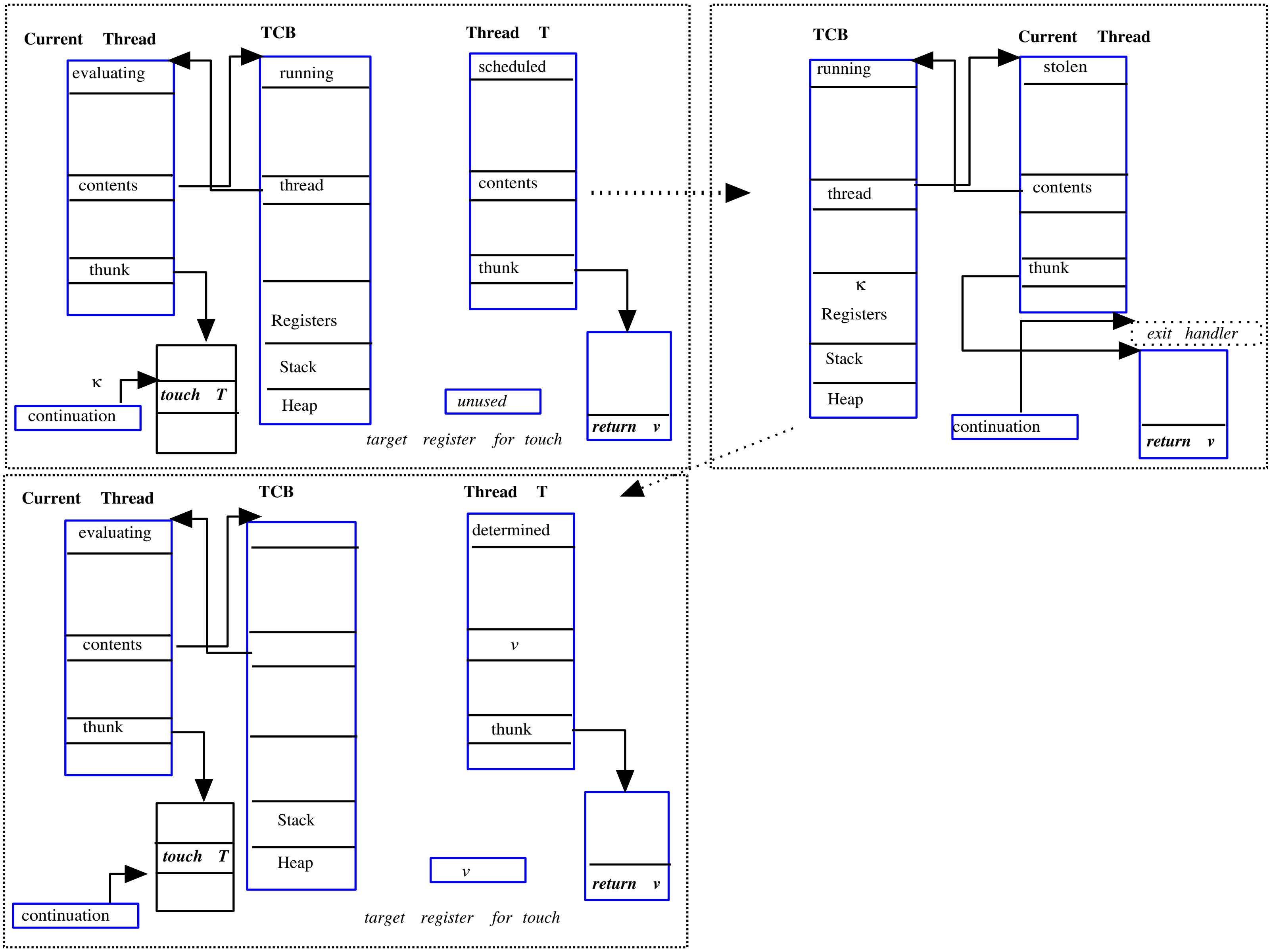

Figure 5: The dynamics of thread stealing.

```
(define (thread-value thread)

  (cond ((thread-determined? thread)
         (let ((val (thread.value thread)))
           (if (thread? val) (thread-value val) val)))
        (else
          (mutex-acquire (thread.mutex thread))
          (cond ((or (thread-scheduled? thread)
                     (thread-delayed? thread))
                 (set-thread.state thread thread-state/stolen)
                 (mutex-release (thread.mutex thread))
                 (run-stolen-thread thread))
                (else
                  (mutex-release (thread.mutex thread))
                  (thread-wait thread)))
                  (thread-value thread))))
```

Figure 6: Determining the value of a thread. A mutex is a binary semaphore.

Because of these properties, we also believe that our collector will be more efficient and maintain better data locality then concurrent garbage collection systems[2, 3, 20]. We are currently in the process of validating our assumptions.

8 Performance

Sting is currently implemented on an 8 processor Silicon Graphics MIPS R3000 shared-memory (cache-coherent) multiprocessor. The physical machine configuration maps physical processors to lightweight Unix threads; each node in the machine runs one such thread. The benchmarks shown in Fig. 7 were implemented using sting's implementation of futures, and used a virtual machine configuration in which each physical processor implements at most one virtual processor. The policy manager implemented a single global LIFO queue. All timings are in seconds. None of the benchmarks trigger garbage collection.

The benchmarks were not coded to maximize execution efficiency, but to highlight and exercise different aspects of the sting implementation. "Matrix Multiply" is a fine-grained program that multiplies two 50×50 integer matrices. In this program, a thread is created for each element in the result matrix. Threads share no data dependencies with one another. "Primes" is a naive prime finder program that computes the first 3000 primes. Threads exhibit significant data dependencies with one another, and thus present opportunities for stealing (at least for small processor ensembles). "Random Tree" creates a tree of 2000 nodes in a random shape using a branching probability of 50%; like "Primes", threads created by "Random Tree" also share strong data dependencies with one another. "Boyer" is a consing intensive theorem prover whose threads are created speculatively; the input in our particular benchmark is the tautological formula:

$$((x \rightarrow y) \wedge ((y \rightarrow z) \wedge ((z \rightarrow u) \wedge (u \rightarrow w))) \rightarrow (x \rightarrow w)$$

There are five statistics associated with each benchmark. "Threads" and "Threads stolen" indicate the number of threads created and stolen respectively. "TCBs created" is the number of new dynamic contexts created during the execution of the benchmark. "TCBs allocated" is the number of TCBs assigned to evaluating threads taken from the VP TCB pool. "TCBs reused" is the number of TCBs that were associated with determined threads and which were immediately recycled for use by other newly created (runnable) threads. These last two statistics reflects VP caching policy efficiency. The drop in stolen threads in both "Primes" and "Boyer" as the number of VPs increase is most likely due to the fact that the sequential component of the threads generated is not sufficient to keep processors busy, and the ready queue relatively full; thus, VPs are able to remove scheduled threads from the ready queue and commence their evaluation of threads before they can become stolen.

9 Comparison to Related Work

There have been several efforts to implement concurrency in Lisp-based languages. QLambda, MultiLisp and Mul-T are some notable examples; continuation-based implementations (*e.g.,* [12, 27]) are another. Despite the common base language used, sting differs in important ways from these efforts as we have described earlier. In summary, the semantics of threads and the design of the abstract virtual machine makes sting amenable for use as a high-level systems programming language and, in particular, as an OS kernel for modern programming languages. Other parallel Lisps that provide mechanisms for creating processes do not permit flexible thread management and customization of their runtime environment essential to implementing a robust systems programming environment. The sting design was furthermore intended to provide a platform on which different concurrency paradigms could be competitively evaluated; the high-level concurrency primitives and opaque runtime environment found in other concurrent symbolic languages make such exper-

	1	2	4	8
Matrix Multiply	3.32	1.66	.94	.51
Threads	2501	2501	2501	2501
Threads stolen	0	0	0	0
TCBs created	1	2	4	8
TCBs allocated	1	536	1375	2052
TCBs reused	2502	1968	1131	457
Primes	1.84	.96	.63	.28
Threads	1501	1501	1501	1501
Threads stolen	1499	19	1	0
TCBs created	1	2	4	8
TCBs allocated	1	305	737	1120
TCBs reused	3	1180	768	381
Random Tree	.91	.52	.23	.17
Threads	581	581	581	581
Threads Stolen	580	579	539	502
TCBs created	1	2	13	20
TCBs allocated	1	2	13	18
TCBs reused	1	3	21	40
Boyer	.3	.18	.08	.042
Threads	32	32	32	32
Threads stolen	13	8	1	1
TCBs created	1	2	4	8
TCBs allocated	1	2	7	10
TCBs reused	20	25	29	30

Figure 7: STING benchmarks. The columns indicate number of virtual processors.

iments problematic.

STING shares many of the same properties as asynchronous dialects of ML[7]. The ML thread system built on top of Mach, for example, provides support for lightweight threads of control similar in many essential respects to the threads described here; *e.g.,* context switching manipulates a continuation structure, preemption is supported, and there is a flexible exception handling mechanism. STING differs from this effort, however, in several significant respects. Most importantly, policy management decisions using ML threads (scheduling, migration, etc.) are managed by the underlying Mach thread, not by the thread system. Moreover thread operations that require functionality provided by the C runtime system (*e.g.,* allocating a new thread to a processor) involve a context switch operation that saves the current state of the thread, restores registers needed by the runtime system and invokes a C routine that will provide the appropriate service. Another important point of difference is the distinction maintained between scheduled and running threads in our system. Finally, STING's support for data locality and process throttling via thread stealing and local TCB caches, and the presence of first-class virtual processors and machines are novel design features[5].

The tight coupling of the Scheme compiler and the runtime system to implement threads is also similar in spirit to dataflow programming systems[21] in which threads are extracted from the source program by the compiler and thread scheduling decisions are shared by the compiler and the runtime system. STING differs obviously from these systems insofar as threads are manifest data objects and parallel programs make explicit reference to them. Nonetheless, the idea of involving the compiler in the management of threads was influenced by these contemporary efforts.

10 Future Work

While the thread system takes advantage of several compiler optimizations, there are a few other significant optimizations we still have not put in place. Most notably, storage and flow analysis routines to minimize storage allocation for threads that do not require stacks or heaps have not yet been fully integrated into the compiler. As another example, interprocedural flow analysis would permit us to determine statically whether a scheduled thread has been evaluated (or already stolen) at a touch point. The cost of preemption might be reduced by maintaining per-procedure information on register use. We also expect that the implementation of threads can be better customized in the presence of a more aggressive type system. Our system currently does not support `call/cc` across thread boundaries; an implementation that provides this functionality is underway.

We eventually intend to extend this work to a full blown OS running on bare hardware. The full kernel will support multiple address spaces, *i.e.,* multiple virtual machines. Thus, page faults, file mapping, and many device interrupts can all be handled at the user-level by the appropriate address space.

References

[1] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., 1985.

[2] Andrew Appel, John Ellis, and Kai Li. Real-time Concurrent Collection on Stock Multiprocessors. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–21, 1988.

[3] Henry Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[4] Peter Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection.* PhD thesis, MIT Laboratory for Computer Science, 1977.

[5] Nick Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444 – 458, April 1989.

[6] Eric Cooper and Richard Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie-Mellon University, June.

[7] Eric Cooper and J.Gregory Morrisett. Adding Threads to Standard ML. Technical Report CMU-CS-90-186, Carnegie-Mellon University, 1990.

[8] Mark Feeley and James Miller. A Parallel Virtual Machine for Efficient Scheme Compilation. In *Proceedings of the 1990 Conference on Lisp and Functional Programming*, pages 119–131, 1990.

[9] R. Gabriel and J. McCarthy. Queue-Based Multi-Processing Lisp. In *Proceedings of the 1984 Conference on Lisp and Functional Programming*, pages 25–44, August 1984.

[10] David Greenberg. *Full Utilization of Communication Resources.* PhD thesis, Yale University, 1991. Also published as Yale University Computer Science Technical Report, YALEU/DCS/TR-860.

[11] Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[12] Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 128–137, March 1990.

[13] Suresh Jagannathan and James Philbin. A Customizable Substrate for Concurrent Languages. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1992.

---

[5] A version of ML threads running on a SGI multiprocessor apparently addresses many of these issues (Personal communication, Andrew Tolmach, Feb. 1992).

[14] David Kranz, Robert Halstead, and Eric Mohr. Mul-T: A High Performance Parallel Lisp. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation*, pages 81–91, June 1989.

[15] David Kranz, R. Kelsey, Jonathan Rees, Paul Hudak, J. Philbin, and N. Adams. ORBIT: An Optimizing Compiler for Scheme. *ACM SIGPLAN Notices*, 21(7):219–233, July 1986.

[16] Butler Lampson and D. Redell. Experiences with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):104–117, February 1980.

[17] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetime of Objects. *Communications of the ACM*, 26(6):419–429, June 1973.

[18] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[19] Rick Mohr, David Kranz, and Robert Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990.

[20] David Moon. Garbage Collection in a Large Lisp System. In *Proceedings of the 1984 Conference on Lisp and Functional Programming*, pages 235 – 246, 1984.

[21] Greg Papadopolus and David Culler. Monsoon: An Explicit Token-Store Architecture. In *Proceedings of the 1990 Conference on Computer Architecture*, pages 82–92, 1990.

[22] James Philbin. STING *Users Manual.* NEC Research Institute, 1992. Forthcoming.

[23] Jonathan Rees and William Clinger, editors. The Revised^3 Report on the Algorithmic Language Scheme. *ACM Sigplan Notices*, 21(12), 1986.

[24] John Reppy. CML: A Higher-Order Concurrent Language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–306, June 1991.

[25] David Ungar. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.

[26] M. Vandevoorde and E. Roberts. WorkCrews: An Abstraction for Controlling Parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

[27] Mitch Wand. Continuation-Based MultiProcessing. In *Proceedings of the 1980 ACM Lisp and Functional Programming Conference*, pages 19–28, 1980.

[28] William Wulf, Roy Levin, and Samuel Harbison. *HYDRA/C.mmp: An Experimental Computer System.* McGraw-Hill, 1981.

Appendix: The Core of the Thread and Policy Manager Interface

Thread Operations

`(fork-thread` *expr vp*`)` creates a thread to evaluate *expr*, and schedules it to run on *vp*.

`(create-thread` *expr*`)` creates a *delayed* thread that when demanded evaluates *expr*.

`(thread-run` *thread vp*`)` schedules a delayed, blocked or suspended *thread* to run on *vp*.

`(thread-wait` *thread*`)` causes the thread executing this operation to block until *thread*'s state becomes *determined*.

`(thread-value` *thread*`)` returns the value of *thread*'s thunk. If *thread* is evaluating, the operation blocks until *thread* become determined. If *thread* is scheduled or delayed, the operation may choose to either steal it or block[6].

`(thread-block` *thread condition*`)` requests *thread* to block on *condition*. *Condition* is presumably the object on which *thread* is enqueued. *Condition* is used only for debugging purposes.

`(thread-suspend` *thread* `.` *quantum*`)` requests *thread* to suspend execution. If the *quantum* argument is provided, the thread is resumed when the period specified has elapsed; otherwise, the thread is suspended indefinitely until it is explicitly resumed using `thread-run`.

`(thread-terminate` *thread* `.` *values*`)` requests *thread* to terminate with *values* as its result.[7]

`(yield-processor)` causes the current thread to relinquish control of its VP and place itself on the ready queue of its current virtual processor.

`(current-thread)` returns the thread executing this operation.

`(current-vp)` returns the VP on which this operation executes.

Policy Operations

`(pm-get-next-thread` *vp*`)` returns the next ready TCB or thread to run on *vp*. If a TCB is returned, its associated thread is *evaluating*; if a thread is returned, its state is *scheduled*, and a new TCB must be allocated for it.

---

[6] Currently, this choice is dictated by the value of a global flag set by the application.

[7] As in some other Scheme dialects, expressions can yield multiple values.

`(pm-enqueue-ready-thread` *obj vp state*`)` enqueues *obj* which may be either a thread or a TCB into the ready queue of the policy manager associated with *vp*. The state argument indicates the current state of the state in which the call to the procedure is made: *delayed*, *kernel-block*, *user-block*, or *suspended*. *Kernel-block* and *user-block* indicates that the thread was previously blocked on a kernel operation (*e.g.,* I/O) or a user-specified condition. *State* is used only for debugging.

`(pm-allocate-vp)` returns an appropriate virtual processor on the current virtual machine; for example, this procedure may be defined as returning the least loaded VP in the machine.

`(pm-need-thread)` migrates a runnable thread from another VP onto its VP and returns the thread as its result. If no threads are available for migration, the procedure returns false.

`(pm-vp-idle` *vp*`)` is called by the thread controller if there are no evaluating threads on *vp*. This procedure can migrate a thread from another virtual processor (using `pm-need-thread`), do bookkeeping, or call the physical processor policy manager to have the processor switch itself to another VP.