A Customizable Substrate for Concurrent Languages

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

Abstract

We describe an approach to implementing a wide-range of concurrency paradigms in high-level (symbolic) programming languages. The focus of our discussion is STING, a dialect of Scheme, that supports lightweight threads of control and virtual processors as first-class objects. Given the significant degree to which the behavior of these objects may be customized, we can easily express a variety of concurrency paradigms and linguistic structures within a common framework without loss of efficiency.

Unlike parallel systems that rely on operating system services for managing concurrency, STING implements concurrency management entirely in terms of Scheme objects and procedures. It, therefore, permits users to optimize the runtime behavior of their applications without requiring knowledge of the underlying runtime system.

This paper concentrates on (a) the implications of the design for building asynchronous concurrency structures, (b) organizing large-scale concurrent computations, and (c) implementing robust programming environments for symbolic computing.

1 Introduction

The growing interest in parallel computing has led to the creation of a number of parallel programming languages that define explicit high-level program and data structures for expressing concurrency. Parallel languages targeted for non-numerical application domains typically support (in varying degrees of efficiency) concurrency structures that realize dynamic lightweight process creation[13, 15] high-level synchronization primitives[28, 29], distributed data structures[6], and speculative concurrency[8, 25]. In effect, all these parallel languages may be viewed as consisting of two sublanguages – a coordination language responsible for managing and synchronizing the activities of a collection of processes, and a computation language responsible for manipulating data objects local to a given process.

In this paper, we describe the implementation of a coordination substrate that permits the expression of a wide range of concurrency structures within the context of a symbolic computation language. Our intention is to define a general-purpose coordination model on top of which a number of specialized coordination languages can be efficiently implemented. We use Scheme[27] as our computation base. We emphasize, however, that the design of the substrate could be incorporated into any high-level symbolic programming language.

One obvious way of implementing a high-level parallel language is to build a dedicated (user-level) virtual machine. The virtual machine serves primarily as a substrate that implements the high-level concurrency primitives found in the coordination sublanguage. Given a coordination language L supporting concurrency primitive P, the role of L's virtual machine (L_P) is to handle all implementation aspects related to P; this often requires that the machine manage process scheduling, storage management, synchronization, etc. Because L_P is tailored only towards efficient implementation of P, however, it is often unsuitable for implementing significantly different concurrency primitives. Thus, to build a dialect of L with concurrency primitive P' usually requires either building a new virtual machine or expressing the semantics of P' using P. Both approaches have their obvious drawbacks: the first is costly to implement given the complexity of implementing a new virtual machine; the second is inefficient given the high-level semantics of P and L_P's restricted functionality.

Rather than building a dedicated virtual machine for implementing concurrency, a language implementation may use low-level operating system services[5, 30]. Process creation and scheduling is implemented by creating a heavy- or lightweight OS-managed thread of control; synchronization is handled using low-level OS-managed structures. These implementations tend to be more portable and extensible than systems built around a dedicated runtime system, but they necessarily sacrifice efficiency[2] since every (low-level) kernel call requires a context switch between the application and the operating system. Moreover, generic OS facilities perform little or no optimization at either compile time or runtime since they are usually

A slightly modified version of this paper appears in the Proceedings of the 1992 ACM Symposium on Programming Language Design and Implementation.

insensitive to the semantics of the concurrency operators of interest.

The dialect of Scheme described here (called STING) includes a coordination language (implemented via a dedicated virtual machine) for expressing asynchronous lightweight concurrency that combines the best of both approaches. In contrast to other parallel Scheme systems[12, 13, 19] and parallel dialects of similar high-level languages[10, 28], the basic concurrency objects in STING (threads and virtual processors) are streamlined data structures with no complex synchronization or value transmission semantics. Unlike parallel systems that rely on OS services for managing concurrency, STING implements all concurrency management issues in terms of Scheme objects and procedures, permitting users to optimize the runtime behavior of their applications without requiring knowledge of underlying OS services. We argue that STING supports the features essential to creating and managing various forms of asynchronous parallelism within a conceptually unified, very general framework.

Our results show that it is possible to build an efficient substrate upon which various parallel dialects of high-level symbolic languages can be built. STING is not intended merely to be a vehicle that implements stand-alone short-lived programs, however. We envision this system as providing a framework for building a rich programming environment for parallel symbolic computing. In this regard, the system provides support for thread preemption, per-thread asynchronous garbage collection, exception handling across thread boundaries, and application dependent scheduling policies. In addition, it contains the necessary functionality to handle persistent long-lived objects, multiple address spaces and other features commonly associated with advanced programming environments.

This paper concentrates on the implications of the STING design for building asynchronous concurrency structures, organizing large-scale concurrent computations, and implementing robust programming environments. A detailed description of its implementation is given in [18]. The paper is structured as follows. In the next section, we give an overview of STING focusing primarily on the structure of the coordination model. Section 3 describes the thread and virtual processor abstractions. Section 4 describes the dynamics of thread execution and synchronization in the context of implementating of several well-known concurrency paradigms (*e.g.,* result-parallel (fine-grained) parallelism[3], master-slave computations[11], speculative concurrency, and barrier synchronization). We argue that despite obvious syntactic and methodological differences, these paradigms all impose common requirements on the underlying runtime system: they require throttling of dynamically generated processes, cheap synchronization, efficient storage management, and the ability to treat processes as *bona fide* data objects. Section 5 presents some performance figures, and comparison to related work is given in Section 6.

2 The Context

Four features of the STING design, when taken as a whole, distinguish the system from many other symbolic parallel languages:

1. *The Concurrency Abstraction:* Concurrency is expressed in STING is via a lightweight thread of control. A thread is a non-strict first-class data structure that superficially resembles the object created by a MultiLisp *future*[13], for example. We elaborate upon the differences in the following section.

2. *The Processor and Policy Abstractions:* Threads execute on a virtual processor (VP) that represents an abstraction of a physical computing device. There may be many more virtual processors than the actual physical processors available. Like threads, virtual processors are also first-class objects. A VP is closed over a policy manager that determines the scheduling and migration regime for the threads that it executes. Different VPs can be closed over different policy managers without incurring any performance penalty.

   A collection of virtual processors can be combined to form a virtual machine. A virtual machine is closed over an address space managed by its virtual processors. Multiple virtual machines can execute on a single physical machine consisting of a set of physical processors. Virtual machines are also denotable Scheme objects and may be manipulated as such.

3. *Storage Model:* A thread allocates data on a stack and heap that it manages exclusively. Thus, threads garbage collect their state independently of one another; no global synchronization is necessary in order for a thread to initiate a garbage collection. Data may be referenced across threads. Inter-area reference information maintained by areas is used to garbage collect objects across thread boundaries[4, 26]. Storage is managed via a generational scavenging collector[21, 32]; long-lived or persistent data allocated by a thread thread is accessible to other threads in the same virtual machine.

   The design is sensitive to storage locality concerns; for example, storage for running threads are cached on VPs and are recycled for immediate reuse when a thread terminates. Moreover, multiple threads may share the same dynamic context whenever data dependencies warrant.

4. *The Program Model:* STING permits exceptions to be handled across threads, supports non-blocking I/O, permits the scheduling of virtual processors on physical processors to be customizable in the same way that the scheduling of threads on a virtual processor is customizable, and provides an infra-structure for implementing multiple address spaces and long-lived shared persistent objects.

3 Threads and Virtual Processors

The computation sublanguage of STING is Scheme[27], a higher-order lexically scoped dialect of Lisp. The compiler used is a modified version of Orbit[20].

The main components in the coordination sublanguage of STING are lightweight threads of control and virtual processors. Threads are simple data structures that encapsulate local storage (*e.g.,* registers, stack and heap organized into areas), code, and relevant state information (*e.g.,* status, priorities, preemption bits, locks, etc.). They define a separate locus of control. The code associated with a thread is executed for effect, not value; thus, threads do not adhere to any particular synchronization protocol. The system imposes no constraints on the code encapsulated by a thread: any valid Scheme expression can be treated as a distinct process.

Each virtual processor (VP) is closed over a (1) a thread controller that implements a state transition function on threads, and (2) a policy manager that implements both a thread scheduling and thread migration policy. A VP is also closed over a set of registers and interrupt handlers. Most of the dedicated registers found in the VP are managed by Orbit; these include a register containing the return address of the current procedure, registers to hold the arguments to a procedure, and registers that refer to the top-of-stack and top-of-heap. Another register points to the currently executing thread. Besides various handlers and a policy manager, a VP also retains the identity of its virtual machine, and the physical processor on which it is executing.

Virtual processors are multiplexed on physical processors in the same way that threads are multiplexed on virtual processors; associated with each physical processor is a policy manager that dictates the scheduling of the virtual processors which execute on it.

3.1 Threads

Threads are first-class objects in STING. Thus, they may be passed as arguments to procedures, returned as results, and stored in data structures. Threads can outlive the objects that create them. A thread is closed over a thunk, a nullary procedure, that is applied when the thread executes. The value of the application is stored in the thread on completion.

The static component of a thread also contains state information on the thread's status. A thread can be either *delayed*, *scheduled*, *evaluating*, *stolen* or *determined.* A *delayed* thread will never be run unless the value of the thread is explicitly demanded. A *scheduled* thread is a thread scheduled to evaluate on some virtual processor but which has not yet started executing. An *evaluating* thread is a thread that has started running. A thread remains in this state until the application of its thunk yields a result. At this point, the thread's state is set to *determined.* Stolen threads are discussed in Section 4.1.

In addition to state information and the code to be evaluated, a thread also holds references to

1. other threads waiting for it to complete,
2. references to the thunk's dynamic and exception environment, and
3. genealogy information indicating the thread's parent, siblings, and children.

Dynamic and exception environments are used to implement fluid bindings and inter-process exceptions. Genealogy information serves as a useful debugging and profiling tool that allows applications to monitor the dynamic unfolding of a process tree.

Evaluating threads are associated with a dynamic context called a *thread control block* (TCB). Besides encapsulating thread storage (stacks and heaps), the TCB contains information about the current state of the active thread (*e.g.,* is the thread currently running on some VP, is it blocked, suspended, terminated, etc?), requested state transitions on this thread made by other threads, the current quantum for the thread, and the virtual processor on which the thread is running. TCBs hold some other information for implementing speculative and barrier synchronization that we discuss in Section 4.3.

The implementation of threads requires no alteration to the implementation of other primitive operations in the language. The synchronization semantics of a thread is a more general (albeit lower-level) form of the synchronization facility available via *e.g.,* MultiLisp's "**touch**", Linda's tuple-space[7], or CML's "**sync**"[28]. The application completely control the condition under which blocked threads may be resumed. However, there is explicit system support for dataflow (*i.e.,* future-touch), nondeterministic choice, and constraint-based or barrier synchronization.

Users manipulate threads via a set of procedures (listed below) defined by a thread controller (TC) that implements synchronous state transitions on thread state. The TC is written entirely in Scheme with the exception of a few primitive operations to save and restore registers. The thread controller allocates no storage; thus, a TC call never triggers garbage collection. Besides these operations, a thread can enter the controller because of preemption.

`(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*`)` inserts a delayed, blocked or suspended *thread* into the ready queue of the policy manager for *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 the application associated with *thread*.

`(thread-block` *thread* `.` *blocker*`)` requests *thread* to block; *blocker* is the condition on which the thread is blocking.

Figure 1: Threads and Virtual Processors

(`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.[1]

`(yield-processor)` causes the current thread to relinquish control of its VP. The thread is inserted into a suitable ready queue.

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

3.1.1 A Simple Example

To illustrate how users might program with threads, consider the program shown in Fig. 2 that defines a Sieve of Erasthosenes prime finder implementation.

Note that the definition makes no reference to any particular concurrency paradigm; such issues are abstracted by its `op` argument.

This implementation relies on a user-defined synchronizing stream abstraction that provides a blocking operation

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

```
(define (filter op n input)
  (let loop ((input input)
             (output  (make-stream))
             (last? true))
    (let ((x (hd input)))
       (cond ((terminate? x)
              (attach (terminate-token) output)
              output)
             ((zero? (mod x n))
              (loop (rest input)
                    output
                    last?))
             (last?
                (op (lambda ()
                      (filter op x output)))
                (loop (rest input)
                      (attach x output)
                      false))
             (else (loop (rest input)
                         (attach x output)
                         last?)))))

(define (sieve op n)
  (let ((input (make-integer-stream n)))
    (op (lambda ()
             (filter op 2 input)))))
```

Figure 2: An abstraction of a concurrent prime finder.

on stream access (`hd`) and an atomic operation for appending to the end of a stream (`attach`).

We can define various implementations of a prime number finder that exhibit different degrees of asynchronous behavior. For example,

```
(let ((filter-list (list)))
  (sieve (lambda (thunk)
            (set filter-list
                 (cons (create-thread (thunk))
                       filter-list)))
          n))
```

defines an implementation in which filters are generated lazily; once a demanded, a filter repeatedly removes elements off its input stream, and generates potential primes onto its output stream. To initiate a new filter scheduled on a VP using a round-robin thread placement discipline, we might write:

```
(thread-run
  (car filter-list)
  (mod (1+ (vm.vp-vector (current-vp).vm))
       n))
```

`(Current-vp)` returns the `vp` on which the expression is evaluated; `((current-vp).vm)` defines the virtual machine of which the current VP is a part. A virtual machine's public state includes a vector containing its virtual processors.

By slightly rewriting the above call to sieve, we can express a more lazy implementation:

```
(let ((filter-list (list)))
  (sieve (lambda (thunk)
           (let ((new-thread
                   (create-thread
                          (begin
                             (map thread-run
                                  filter-list)
                             (thunk)))))
              (map thread-block filter-list))
              (set filter-list
                   (cons new-thread filter-list))
              new-thread)
          n))
```

In this definition, a filter that encounters a potential prime p, creates a lazy thread object L and requests all other filters in the chain to block. When L's value is demanded, it unblocks all the elements in the chain, and proceeds to filter all multiples of p on its input stream. This implementation throttles the extension of the sieve and the consumption of input based on demand.

We can also define an eager version of the sieve as follows:

```
(sieve
   (lambda (thunk)
      (fork-thread (thunk)))
    n)
```

Evaluating this application schedules a new thread responsible for filtering all multiples of a prime.

This simple exercise highlights some interesting points about the system. First, STING treats thread operations as ordinary procedures, and manipulates the objects referenced by them just as any other Scheme object; if two filters attached via a common stream are terminated, the storage occupied by the stream may be reclaimed. STING imposes no *a priori* synchronization protocol on thread access – application programs are expected to build abstractions that regulate the coordination of threads.

The threads created by `filter` maybe terminated in one of two ways. The top-level call to `sieve` may be structured so that it has an explicit handle on these threads; the `filter-list` data structure used to create a lazy sieve is such an example. One can then evaluate:

```
(map thread-terminate filter-list)
```

to terminate all threads found in the sieve. STING also provides *thread groups* as a means of gaining control over a related collection of threads[19]. A thread group is closed over debugging and thread operations that may be applied *en masse* to all of its members. Every thread has a thread group identifier that associates it with a given group. Thread groups provide operations analogous to ordinary thread operations (*e.g.,* termination, suspension, etc.) as well as operations for debugging and monitoring (*e.g.,* resetting, listing all threads in a given group, listing all groups, profiling genealogy information, etc..) Thus, when the thread T under which the call to `sieve` is terminated, users can request all of T's children (which are defined to be part of T's group to be terminated) thus:

`(kill-group (thread.group` T`))`

Second, lazy threads are distinguished from scheduled ones. A lazy thread defines a thread object closed over a thunk and dynamic state (but which is unknown to any virtual processor). A scheduled thread is also a lightweight data structure, but is known to some VP and will eventually be assigned a TCB. Applications can choose the degree of laziness (or eagerness) desired. Only the thread controller can initiate a thread transition to *evaluating* – the interface does not permit applications to insist that any specific thread immediately run on some virtual processor. All default policy managers implement a fair scheduling policy, but STING imposes no constraints on user-defined policy managers in this regard.

Third, threads can request state changes to other threads; the change itself takes place only when the target thread next makes a TC call (either synchronously or because of preemption). Requested state changes to a thread T made by another T' are recorded as part of T's next state in its TCB. State changes are recorded only if they do not violate the state transition semantics (*e.g., evaluating* threads cannot be subsequently *scheduled*; *terminated threads* cannot become subsequently *blocked*, etc.), and the requesting thread has appropriate authority.

Only threads can actually effect a change to their own state. This invariant implies that a TCB can perform a state transition without acquiring locks.

3.2 Virtual Processors

Virtual processors (and by extension, virtual machines) are first-class objects in STING. According first-class status to VPs has several important implications that distinguish STING from other high-level thread systems[9, 10] or other asynchronous parallel languages. First, one can organize parallel computations by explicitly mapping processes onto specific virtual processors. For example, a process P known to communicate closely with Q should execute on a VP topologically near V. Such considerations can be expressed in STING since VPs can be directly enumerated. Systolic style programs for example can be expressed by using self-relative addressing off the current VP (*e.g.,* `left-VP, right-VP, up-VP`, etc.). The system provides a number of default addressing modes for many common topologies (*e.g.,* hypercubes, meshes, systolic arrays, etc.). Furthermore, since VPs can be mapped onto specific physical processors, the ability to manipulate virtual processors as first-class data values gives STING programmers a great deal of flexibility in expressing different parallel algorithms that are defined in terms of specific processor topologies[16].

More significantly, since VPs can be closed over different virtual policy managers, different groups of threads created by an application may be subject to different scheduling regimes. Virtual machines or VPs can be tailored to handle different scheduling protocols or policies. We discuss the implications of customizable schedulers in the following section.

3.3 The Policy Manager

The STING thread controller defines a thread state transition procedure, but does not define *a priori* scheduling or migration policies. These policies can be application dependent. Although several default policies are provided as part of the overall STING runtime environment, users are free to write their own. In fact, each virtual processor is closed over its own policy manager (PM); thus, different VPs in a given virtual machine may implement different policies. The PM handles thread scheduling, processor/thread mapping, and thread migration.

The ability to partition an application into distinct scheduling groups is important for long-lived parallel (or interactive) programs. Threads executing I/O bound procedures have different scheduling requirements than those executing compute bound routines; applications with real-time constraints should be implemented using different scheduling protocols than those that require only a simple FIFO scheduling policy.

Tree-structured parallel programs may realize best runtime performance using a LIFO-based scheduler; applications running master/slave or worker farm algorithms may do better using a round-robin preemptive scheduler for fairness. Since all of these applications may be components of a larger program structure or environment, the flexibility afforded by having them evaluate with different policy managers is significant. Distinct applications can exist as independent executing threads evaluating on the same virtual machine. Moreover, each distinct scheduler is realized by a policy manager with different performance characteristics and implementation concerns.

Our design seeks to provide a flexible framework able to incorporate and experiment with different scheduling regimes transparently without requiring modification to the thread controller itself. To this end, all PMs provide the same interface although no constraints are imposed on the implementations themselves. The interface shown below provides operations for choosing a new thread to run, enqueuing an evaluating thread, setting thread priorities, and migrating threads. These procedures are expected to be used exclusively by the TC; in general, user applications need not be aware of the policy/thread manager interface.

`(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 not *evaluating*, and a new TCB must be allocated for it.

`(pm-enqueue-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 state in which the the call to the procedure is made: *delayed*, *kernel-block*, *user-block*, or *suspended*.

`(pm-priority` *priority*`)` and `(pm-quantum` *quantum*`)` use their *priority* and *quantum* arguments as hints to establish a new priority and quantum for the currently executing thread.

`(pm-allocate-vp)` returns a new virtual processor on the current virtual machine.

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

Besides determining a scheduling order for evaluating threads, the PM implements two basic load-balancing decisions: (1) it may choose a VP on which a newly created thread should be run, and (2) it determines which threads on its VP can be migrated, and which threads it will choose for migration from other VPs.

The first decision point is important to handle initial load-balancing; the second is important to support dynamic load-balancing protocols. Determining the initial placement of a newly evaluating thread is often based on priorities different from those used to determine the migration of currently evaluating threads. The PM interface preserves this distinction.

Scheduling policies can be classified along several important dimensions:

Locality: Is there a single global queue of threads in this system, or does each PM maintain its own local queues?

Granularity: Are threads distinguished based on their current state or are all threads viewed as equals by the PM? For example, an application might choose an implementation in which all threads occupy a single queue regardless of their current state. Alternatively, it might choose to classify threads into different queues based on whether they are evaluating, scheduled, previously suspended etc.

Structure: Are the queues implemented as FIFO's, LIFO's, round-robin, priority, or realtime structures (among others)?

Serialization: What kind of locking structure does an application impose on various policy manager queues?

Choosing different alternatives in this classification scheme leads to different performance characteristics. For example, if we adopt a granularity structure that distinguishes evaluating threads (*i.e.,* threads with TCBs) from scheduled ones, and we impose the constraint that only scheduled threads can be migrated, then no locks are required to access the evaluating thread queue; this queue is local to the VP on which it was created. Queues holding scheduled and suspended threads however must be locked because they are targets for migration by PMs on other VPs. This kind of scheduling regimen is useful if dynamic load-balancing is not an issue. Thus, when there exist many long-lived non-blocking threads (of roughly equal duration), most VPs will be busy most of the time executing threads on their own local ready queue. Eliminating locks on this queue in such applications is therefore beneficial. On the other hand, applications that generate threads of varying duration may exhibit better performance when used with a policy manager that permits migration of both scheduled and evaluating threads even if there is an added cost associated with locking the runnable ready queue.

Global queues imply contention among policy managers whenever they need to execute a new thread, but such an implementation is useful in implementing many kinds of parallel algorithms. For example, in master/slave (or worker-farm) programs, the master initially creates a pool of threads; these threads are long-lived structures that do not spawn any new threads themselves. Once running on a VP, they rarely block. Thus, a PM executing such a thread has no need to support the overhead of maintaining a local thread queue. Local queues are useful, however, in implementing result-parallel programs in which the process structure takes the form of a tree or graph; these queues can be used in such applications to load balance threads fairly among a set of virtual processors.

4 The Dynamics of Thread Execution and Synchronization

Obvious differences exist in program methodology, syntax, etc. among the numerous proposals for incorporating concurrency structures into high-level symbolic programming languages. STING supports the functionality required by the semantics of many of these proposals: (a) threads may be dynamically instantiated and require runtime scheduling, (a) communication among threads takes place via concurrent data structures that may be shared by many readers and writers, (c) communicating threads execute within a single address space, and (d) threads synchronize either by waiting for values generated by other processes, or by waiting at explicit barrier points.

```
(define (primes limit)
  (let loop ((i 3)
             (primes (future (list 2))))
    (cond ((> i limit)
           (touch primes))
          (else
            (loop
              (+ i 2)
              (future
                (filter i primes)))))))


(define (filter n primes)
  (let loop ((j 3))
    (cond ((> (* j j) n)
           (cons n (touch primes)))
          ((zero? (mod n j)) primes)
          (else (loop (+ j 2))))))
```

Figure 3: An implementation of primes using futures. A future must be explicitly touched to access its value in this implementation.

4.1 Support for Result (Fine-Grained) Parallelism

In a result parallel program, each concurrently executing process contributes to the value of a complex data structure (*e.g.,* an array or list). Process communication is via this result structure. Expressions that attempt to access a component of the result whose contributing process is still evaluating block until the process completes.

Futures[13] are a good example of an operation well-suited for implementing result parallel algorithms. The object created by the MultiLisp or Mul-T expression, `(future` E`)`, creates a thread responsible for computing E; the object returned is known as a future. When E finishes, yielding v as its result, the future is said to be *determined.* An expression that *touches* a future either blocks if E is still being computed or yields v if the future is determined. Threads are a natural representation for futures.

To motivate the implementation of result parallelism in STING, Figure 3 is an implementation of a parallel prime number finder using futures.

In this program, a future is created for each odd element between 2 and `limit`. A number is added onto a current prime list if `filter` determines it to be a prime number. In a naive implementation, each instantiation of a future will entail the creation of a new thread; thus, the number of threads allocated in executing this program (under this implementation) is proportional to `limit`. This behavior is undesirable because a future computing the primality of i has an implicit dependence with the future created to compute the primality of $i-2$ and so on. Poor processor and storage utilization will results given the data dependencies found in this program. This is because many of the lightweight processes that are created will either:

1. need to block when they request the value of other yet-unevaluated futures or,
2. in the case of processes computing small primes, do a small amount of computation relative to the cost incurred in creating them.

Because the dynamic state of a thread consists of large objects (*e.g.,* stacks and heaps), cache and page locality is compromised if process blocking occurs frequently or if process granularity is too small.

The semantics of touch and future dictate that a future F which touches another future G must block on G if G is not yet determined. Assume T_F and T_G are the thread representation of F and G, respectively. The runtime dynamics of the touch operation on G can entail accessing T_G either when T_G is (a) delayed or scheduled, (b) evaluating, or (c) determined. In the latter case, no synchronization between these threads is necessary. Case (b) requires T_F to block until T_G completes. STING performs an important optimization for case (a), however, which we discuss below.

4.1.1 Thread Stealing

T_F can evaluate the closure encapsulated within T_G (call it E) using its own stack and heap, rather than blocking a forcing a context switch if T_G is delayed or scheduled. In effect, this implementation treats E as an ordinary procedure, and the touch of G as a simple procedure call; we say that T_F *steals* T_G in this case. The correctness of this optimization lies in the observation that that T_F would necessarily block otherwise; by applying E using T_F's dynamic context, the VP on which T_F executes does not incur the overhead of executing a context switch. In addition, no TCB need be allocated for T_G since T_F's TCB is used instead.

The optimization may only lead to observably different results if used in instances where the calling thread need not necessarily block. For example, suppose T_G was an element of a speculative call by T_F. Furthermore, assume T_G diverges, but another speculative thread (call it T_H) does not. In the absence of stealing, both T_G and T_H would spawn separate thread contexts. T_H returns a value to T_F. In the presence of stealing, however, T_F will also loop because T_G does. Users can parameterize thread state to inform the TC if a thread can steal or not; STING provides interface procedures for this purpose.

Figure 4: Dynamics of thread stealing. Dashed lines indicate dataflow constraints, solid lines specify thread transitions.

Like load-based inlining[33] or lazy task creation[24], stealing throttles process creation. Unlike these other techniques, however, stealing also improves locality. Locality is increased because a stolen thread is run using the TCB of a currently evaluating thread; consequently, the stack and heap of this TCB remains in the virtual machine's working set.

Because of stealing, STING reduces the overhead of context switching, and increases process granularity for programs in which processes (a) exhibit strong data dependencies among one another, and (b) block only when they require data from other processes. Of course, for the operation to be most effective, appropriate scheduling policies must be adopted. For example, a preemptible FIFO scheduler in the prime number code would not take full advantage of stealing since processes computing small primes would be run before processes that compute large ones. Stealing operations will be minimal in this case: processes exhibit few data dependencies with processes instantiated earlier, and threads computing small primes must necessarily be determined before threads computing large primes can proceed. On the other hand, a LIFO scheduling policy will cause processes computing large primes (*i.e.,* primes close to `limit`) to be run first. Stealing will occur much more frequently here since processes will demand the results of other processes computing smaller primes which have not yet run; the process call graph will, therefore, unfold more effectively.

4.2 Master-Slave Programs: Blocking and Synchronization

The master-slave paradigm is a popular parallel program structuring technique. In this approach, the collection of processes generated is bounded *a priori*; a master process generates a number of worker processes and collates their results. Process communication typically occurs via shared concurrent data structures or variables. Master-slave programs often are more efficient than result parallel ones on stock multiprocessor platforms because workers rarely need to communicate with one another except to publish their results, and process granularity can be better tailored for performance.

We have used STING to build an optimizing implementation of first-class *tuple-spaces* in Scheme. A tuple-space is an object that serves as an abstraction of a synchronizing content-addressable memory[6]; tuple-spaces are a natural implementation choice for many master/slave-based algorithms.

The semantics of tuple-spaces in our system differ significantly from their definition in C.Linda, for example. Besides the added modularity brought about by denotable tuple-space objects, our system also treats tuples as objects, and tuple operations as binding expressions, not statements. We have built a customized type inference procedure to specialize the representation of tuple-spaces whenever possible[17]. In our current implementation, tuple-spaces can be specialized as synchronized vectors, queues, streams, sets, shared variables, semaphores, or bags; the operations permitted on tuple-spaces remains invariant over their representation. In addition, applications can specify an inheritance hierarchy among tuple-spaces if so desired.

Processes can read, remove or deposit new tuples into a tuple-space. The tuple argument in a read or remove operation is called a *template* and may contain variables prefixed with a "?". Such variables are referred to as *formals* and acquire a binding-value as a consequence of the match operation. The bindings acquired by these formals are used in the evaluation of a subordinate expression: thus, we can write:

```
(get TS [?x]
     (put TS [(+ x 1)]))
```

to remove atomically a singleton tuple from `TS`, increment it by one, and deposit it back into `TS`.

Our implementation, in the general case, uses two hash-tables (call them H_R and H_P) as the representation structures for a fully associative tuple-space. Processes that attempt to read or remove a tuple first hash on their non-formal tuple elements in H_P. If at least one match exists, the proper bindings for the formals are established, the retrieved tuple is marked as deleted in the case of a remove operation, and the executing process proceeds. When a match does not exist, the process hashes on its non-formal tuple elements in H_R, deposits a structure that indicates its identity. and blocks.

A depositing process is defined symmetrically – any processes waiting for its tuple in H_R are unblocked and rescheduled. Otherwise, the tuple is deposited into H_P using its fields as hash keys. The implementation minimizes synchronization overhead by associating a mutex with every hash bin rather than having a global mutex on the entire hash table. This permits multiple producers and consumers of a tuple-space to concurrently access its hash tables.

The implementation also takes advantage of stealing to permit the construction of fine-grained parallel programs that synchronize on tuple-spaces. We use threads as *bona fide* elements in a tuple. Consider a process P that executes the following expression:

`(rd TS [ x1 x2 ]` E`)`

where `x1` and `x2` are non-formals. Assume furthermore that a tuple in `TS` is deposited as a consequence of the operation:

`(spawn TS [` E_1 E_2 `])`

This operation schedules two threads (call them T_{E_1} and T_{E_2}) responsible for computing E_1 and E_2. If both T_{E_1} and T_{E_2} complete, the resulting (passive) tuple contains two determined threads; the matching procedure applies `thread-value` when it encounters a thread in a tuple; this operation retrieve the thread's value.

If T_{E_1} is still scheduled at the time P executes, however, P is free to steal it, and then determine if its result matches `x1`. If a match does not exist, P may proceed to search for another tuple, leaving T_{E_2} still in a scheduled state. Another process may subsequently examine this same tuple and steal T_{E_2} if warranted. Similarly, if T_{E_1}'s result matches `x1`, P is then free to steal T_{E_2}. If either T_{E_1} or T_{E_2} are already evaluating, P may choose to either block on one (or both) thread(s), or examine other potentially matching tuples in `TS`. The semantics of tuple-spaces impose no constraints on the implementation in this regard.

STING's combination of first-class threads and stealing allows us to write quasi-demand driven fine-grained (result) parallel programs using shared data structures. In this sense, the thread system attempts to minimizes any significant distinction between structure-based (*e.g.,* tuple-space) and dataflow style (*e.g.,* future/touch) synchronization.

4.2.1 Mutexes

Operations on tuple-spaces or similar high-level synchronization structures make use of mutex operations, `mutex-acquire` and `mutex-release`.

Mutexes are created by the mutex operation, `(make-mutex` *active passive*`)`. `Mutex-acquire` attempts to acquire a mutex. If the mutex is locked, the executing thread actively spins for the period specified by *active*; active spinning causes the thread to retain control of its virtual processor during the period that it is blocked waiting for the mutex to be released. When the active spin count becomes zero, the thread relinquishes control of its VP, and inserts itself into an appropriate ready queue. When next run, it attempts to re-acquire the mutex, yielding its processor if unsucessful. This operation is repeated *passive* number of times. If the passive spin count is exhausted, and the mutex has not yet been acquired, the executing thread blocks on the mutex. When the mutex is ultimately released, (via `mutex-release`) all threads blocked on this mutex are restored onto some ready queue.

Using mutex primitives, macros, and Scheme's support for exception handling, one can easily build a "safe" version of a mutex acquire operation, `(with-mutex` *mutex body*`)` . This operation ensures that *mutex* is released if *body* raises an exception during its evaluation that causes control to exit the current dynamic environment.

4.2.2 Preemption and Interrupts

A preemptive round-robin or FIFO scheduling policy is best suited for master-slave applications in which the master performs relatively little processing after the initial set of spawned workers complete. A round-robin policy allocates a specified quantum for each worker in the worker pool. Support for preemption is important because workers rarely block; in its absence, long-running workers might occupy all available VPs at the expense of other enqueued ready threads.

Preemption is sometimes best disabled in master/slave programs that make significant use of barrier synchronization. In these applications, the master generates a new set of worker processes after all previously created workers complete. If the time to execute a particular set of workers is small relative to the total time needed to complete the application, enabling preemption may degrade performance[31]. Threads can disable an initial preemption by setting a flag in their TCB; if preemption takes place when this quantum flag is false, another bit in the TCB state is set indicating that a subsequent preemption should not be ignored. Users can encapsulated time critical code using the the syntactic form, `(without-preemption` *body*`)` that evaluates *body* with preemption disabled. The `without-preemption` form is in fact a specialized version of a more general construct, `without-interrupts` , that disables all interrupts during the evaluation of its body.

4.3 Speculative Parallelism and Barrier Synchronization

Speculative parallelism is an important programming technique that often cannot be effectively utilized because of runtime overheads incurred in its implementation. The two features most often associated with systems that support a speculative programming model are the ability to favor certain (more promising) tasks over others, and the means to abort, reclaim (and possibly undo) unnecessary computation.

STING permits programmers to write speculative applications by:

1. allowing users to explicitly program thread priorities,
2. permitting a thread to wait on the completion of other threads, and
3. allowing threads to terminate other threads.

Promising tasks can execute before unlikely one because priorities are programmable. A task α that completes first in a set of tasks can awaken any thread blocked on its completetion; this functionality permits STING to support a simple form of OR-parallelism[8]. α can terminate all other tasks in its task set once it has been determined that their results are unnecessary. Speculative computation using STING, however, will not be able to undo non-local side-effects induced by useless tasks; the system does not provide a primitive backtracking mechanism[2].

Consider the implementation of a `wait-for-one` construct. This operator evaluates its list of arguments concurrently, returning the value yielded by the first of its arguments to complete. Thus, if a_i yields v in the expression:

`(wait-for-one` $a_1\ a_2\ \ldots\ a_i\ \ldots\ a_n$`)`

the expression returns v, and, if desired by the programmer, terminates the evaluation of all the remaining a_j, $j \neq i$.

The specification of a `wait-for-all` construct that implements an AND-parallel operation is similar; it also evaluates its arguments concurrently, but returns true only when all its arguments complete. Thus, the expression:

`(wait-for-all` $a_1\ a_2\ \ldots\ a_i\ \ldots\ a_n$`)`

acts as a barrier synchronization point since the thread executing this expression is blocked until all the a_i complete. The implementation of this operation is very similar to the implementation of the speculative wait-for-one operation.

The TC implements these operations using a common procedure, `block-on-group` . Threads and TCBs are defined to support this functionality. For example, associated with a TCB structure is information on the number of threads in the group that must complete before the TCB's associated thread can resume.

`Block-on-group` takes a list of threads and a count. These threads correspond to the arguments of the `wait-for-one` and `wait-for-all` operations shown above; the count argument represents the number of threads that must complete before the current thread (*i.e.*, the thread executing this procedure) is allowed to resume. If the count is one, we get an implementation of `wait-for-one` ; if the count is equal to n, we get an implementation of `wait-for-all` .

The relationship between a thread T_g in the group and the current thread (T_w) that is to wait on T is maintained in a data structure (called a thread barrier (TB)) that contains references to:

1. T_w's TCB.
2. the TB of another waiter blocked on T_g (if one exists).

[2] Sting does not support first-class continuations across thread boundaries.

```
(define (block-on-group count group)
  (let loop ((i count)
             (threads group))
    (cond
      ((zero? i))
      ((null? group)
       (set-TCB.wait-count (current-TCB) i)
       (thread-block (current-thread)))
      (else
        (let ((thread (car threads)))
          (mutex-acquire thread.mutex)
          (cond
            ((determined? thread)
             (mutex-release thread.mutex)
             (loop (1- i) (cdr threads)))
            (else
              (let ((tb (make-tb)))
                (set-tb.tcb tb
                            (current-tcb))
                (set-tb.thread tb thread)
                (set-tb.next tb
                     (thread.waiters thread))
                (set-thread.waiters thread tb))
                (mutex-release mutex))
                (loop (1- i)
                      (cdr threads))))))))
```

Figure 5: Definition of `block-on-group`.

3. T_g – this is used only for debugging purposes.

We give a definition for `block-on-group` in Fig. 5.

The call:

`(block-on-group` m T_1 T_2 ... T_n`)`

causes the current thread (call it T) to block on the completion of m of the T_i, $m \leq n$. Each of these T_i have a reference to T in their chain of waiters. The procedure checks if a thread in the thread group has already been determined; in the case, the `wait-count` is decremented, but no thread barrier is constructed. Otherwise, a TB is constructed as well. When all threads in `group` have been examined, the procedure sets the current thread's `wait-count` field to the extant count, and issues a `thread-block` operation.

Applications use `Block-on-group` in conjunction with a `wakeup-waiters` procedure that is invoked by the a_i when they complete. `Wakeup-waiters` examines the list of waiters chained from the `waiters` slot in its thread argument. A waiter whose `wait-count` becomes zero is enqueued on the ready queue of some VP. The TC invokes `wakeup-waiters` whenever a thread T completes (*i.e.,* whenever it terminates or abnormally exits). All threads waiting on T's completion are thus rescheduled.

Given these two procedures `wait-for-one` can be defined simply:

```
(define (wait-for-one block-group)
  (block-on-group 1 block-group)
  (map thread-terminate block-group)
```

Case	*Timings(in μseconds)*
Thread Creation	8.9
Thread Fork and Value	44.9
Scheduling a Thread	18.9
Synchronous Context Switch	3.77
Stealing	7.7
Thread Block and Resume	27.9
Tuple-Space	170
Speculative Fork (2 threads)	68.9
Barrier Synchronization (2 threads)	144.8

Figure 6: Baseline timings.

If T executes `wait-for-one`, it blocks on all the threads in its `block-group` argument. When T is resumed, it is placed on a queue of ready threads in the policy manager of some available virtual processor. The `map` procedure executed upon T's resumption terminates all threads in its group.

STING's `wait-for-all` procedure can omit this operation since all threads in its `block-group` are guaranteed to have completed before the thread executing this operation is resumed.

5 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. We ran the benchmarks shown below using a virtual machine in which each physical processor implements a single virtual processor.

Fig. 6 gives baseline figures for various thread operations; these timings were derived using a single LIFO queue.

The "Thread Creation" timing is the cost to create a thread not placed in the genealogy tree, and which has no dynamic state. "Thread Fork and Value" measures the cost to create a thread that evaluates the null procedure and returns. "Scheduling a Thread" is the cost of inserting a thread into the ready queue of the current VP. A "Synchronous Context Switch" is the cost to make a `yield-processor` call in which the calling thread is resumed immediately. The cost for "Stealing" does not include the time to schedule the thread being stolen. "Thread Block and Resume" is the cost to block and resume a null thread. "Tuple Space" is the cost to create a tuple-space, insert and then remove a singleton tuple. The speculative synchronization timings reflects the cost to compute two null threads speculatively; the barrier synchronization is the cost to build a barrier synchronization point on two threads both computing the null procedure. We present detailed benchmarks of several application programs in a companion paper[18].

6 Related Work and Conclusions

Insofar as STING is a programming system that permits the creation and management of lightweight threads of control, it shares several common traits with thread package systems developed for other high-level languages[9, 10, 23]. These systems also view threads as a manifest datatype, support preemption in varying degrees, and in certain restricted cases, permit programmers to specify a specialized scheduling regimen. The thread abstraction defines the coordination sublanguage in these systems.

There are some important differences however that clearly distinguish STING from these other systems. First, the scheduling and migration protocol STING uses is *completely* customizable; different applications can run different schedulers without modifying the thread manager or the virtual processor abstraction; such customization can be applied to the organization of the virtual machine itself. Second, STING's support for data locality, storage optimization, and process throttling via stealing is absent in other systems. Moreover, all thread operations are implemented directly within the STING virtual machine: there is no context switch to a lower level kernel that must be performed in order to execute a thread operation. STING is built on abstract machine intended to support long-lived applications, persistent objects, and multiple address spaces. Thread packages provide none of this functionality since (by definition) they do not define a complete program environment.

STING also differs from programming languages that provide high-level abstractions (*e.g.,* continuations[34, 14] to model concurrency. Because we designed STING as a systems programming language, it provides low-level concurrency abstractions – application libraries can directly create thread objects, and can define their own scheduling and thread migration strategies. High-level concurrency constructs are realizable using threads, but the system does not prohibit users from directly using thread operations in the ways described above if efficiency considerations warrant. In particular, the same application may define concurrency abstractions with different semantics and efficiency concerns within the same runtime environment.

In certain respects, STING resembles other advanced multithreaded operating system environments[1, 22]: for example, it supports non-blocking I/O calls with call-back, user control over interrupts, and local address space management as user-level operations. It cleanly separates user-level and kernel-level concerns: physical processors handle (privileged) system operations and operations across virtual machines; virtual processors implement all user-level thread and local address-space functionality. However, because STING is an extended dialect of Scheme, it provides the functionality and expressivity of a high-level programming language (*e.g.,* first-class procedures, general exception handling, and rich data abstractions) that typical operating system environments do not offer.

STING is a platform for building asynchronous programming primitives and experimenting with new parallel programming paradigms. In addition, the design also allows different concurrency models to be evaluated competitively. Scheme offers an especially rich environment in which to undertake such experiments because of its well-defined semantics, its overall simplicity, and its efficiency. However, the STING design itself is language independent; we believe it could be incorporated fairly easily into any high-level programming language.

STING does not merely provide hooks for each concurrency paradigm and primitive we considered interesting. We focussed instead on basic structures and functionality common to a broad range of parallel programming structures; thus, the implementation of blocking is easily used to support speculative computation, the "stealing" optimization used to throttle the execution of threads is well-suited for implementing futures and tuple-space synchronization, and, finally, customizable policy managers make it possible to build fair and efficient schedulers for a variety of other paradigms.

References

[1] Thomas Anderson, Edward Lazowska, and Henry Levy. The Performance Implications of Thread Management Alternatives for Shared Memory MultiProcessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 95–109. Association for Computing Machinery SIGOPS, October 1991.

[3] Arvind, Rishiyur Nikhil, and Keshav Pingali. I-Structures: Data Structures for Parallel Computing. *Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

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

[5] David Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, May 1990.

[6] Nick Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3), September 1989.

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

[8] K.L Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.

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

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

[11] J. Dongarra, D. Sorenson, and P. Brewer. Tools and Methodology for Programming Parallel Processors. In *Aspects of Computation on Asynchronous Processors*, pages 125–138. North-Holland, 1988.

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

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

[14] 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.

[15] Waldemar Horwat, Andrew Chien, and William Dally. Experience with CST: Programming and Implementation. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 101–109, June 1989.

[16] Paul Hudak. Para-Functional Programming. *IEEE Computer*, 19(8):60–70, August 1986.

[17] Suresh Jagannathan. Optimizing Analysis for First-Class Tuple-Spaces. In *Third Workshop on Parallel Languages and Compilers*. MIT Press, August 1990.

[18] Suresh Jagannathan and James Philbin. A Foundation for an Efficient Multi-Threaded Scheme System. In *Proceedings of the 1992 Conf. on Lisp and Functional Programming*, June 1992.

[19] 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.

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

[21] 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.

[22] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 110–21. Association for Computing Machinery SIGOPS, October 1991.

[23] Sun Microsystems. *Lightweight Processes*, 1990. In SunOS Programming Utilities and Libraries.

[24] 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.

[25] Randy Osborne. Speculative Computation in MultiLisp. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 198–208, 1990.

[26] James Philbin. *STING: An Operating System Kernel for Highly Parallel Computing*. PhD thesis, Dept. of Computer Science, Yale University, 1992. Forthcoming.

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

[28] 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.

[29] Vijay Saraswat and Martin Rinard. Concurrent Constraint Programming. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 232–246, 1990.

[30] A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young. Mach Treads and the UNIX Kernel: The Battle for Control. In *1987 USENIX Summer Conference*, pages 185–197, 1987.

[31] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Machines. In *Proceedings of the 12th Annual ACM Symposium on Operating Systems Principles*, pages 114–122, 1989.

[32] 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.

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

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