High-Level Abstractions for Efficient Concurrent Systems

Suresh Jagannathan and James Philbin

NEC Research Institute, 4 Independence Way, Princeton, 08540, NJ, USA.
{*suresh,philbin*}@*research.nj.nec.com*

1 Introduction

Parallel symbolic algorithms exhibit characteristics that make their efficient implementation on current multiprocessor platforms difficult: data is generated dynamically and often have irregular shape and density, data sets typically consist of objects of many different types and structure, the natural unit of concurrency is often much smaller than can be efficiently supported on stock hardware, efficient scheduling, migration and load-balancing strategies vary widely among different algorithms, and sensible decomposition of the program into parallel threads of control often cannot be achieved by mere examination of the source text.

Many of these challenges are also faced by implementors of multi-threaded operating systems and kernels[2, 23, 29]. In both cases, the utility of an implementation is determined by how well it supports a diverse range of applications in terms of performance and programmability.

Implementations that hard-wire decisions about process granularity, scheduling, and data allocation often perform poorly when executing parallel symbolic algorithms. An alternative approach to building concurrent systems would permit a system's internals to be customized by users *without* sacrificing performance or security. The generality and expressivity of the abstractions used by an implementation dictate the extent to which it may be customized.

Because different applications will exercise different operations in different ways, systems which permit customization of these operations on a per-application basis offer the promise of greater flexibility and programmability. First-class procedures[1, 21] and continuations[13, 28] are two abstractions we have found to be extremely useful in implementing a variety of environment and control operations. Languages such as Scheme[10] or ML[25] have demonstrated that these abstractions are effective building blocks for expressing a number of interesting data, program and control structures.

It is often claimed, however, that such high-level program constructs are too inefficient to serve as a foundation for building efficient high-performance concurrent systems. In this paper, we present evidence to the contrary. Sting is a highly efficient and flexible operating system implemented in Scheme. It provides mechanisms to (a) create

⋆ Appeared in the Proceedings of the International Conference on Programming Languages and Systems Architecture, March 1994.

lightweight asynchronous threads of control, (b) build customized scheduling, migration, and load-balancing protocols, (c) support a range of execution strategies from *fully eager* to *completely lazy* evaluation, (d) experiment with diverse storage allocation policies, and (e) handle multiple persistent address spaces.

The design of the Sting implementation relies heavily on both first-class procedures and continuations. Higher-order procedures are used to implement thread creation and synchronization operations, message passing, and customizable thread schedulers. Continuations[2] are used to implement state transition operations, exception handling, and important storage optimizations.

Earlier reports on Sting [17, 18] focussed primarily on program methodology and paradigms, discussing how one might express and use lightweight threads of control in a high-level programming language such as Scheme. In this paper, we discuss systems-level concerns within the Sting context, concentrating on the role of continuations and first-class procedures in the implementation of an efficient and general-purpose multi-threaded operating system and programming environment. In particular, we discuss three components of the Sting architecture not explicated elsewhere; these features are influenced heavily by the system's pervasive use of first-class procedures and continuations:

1. *First-class Ports:* Sting allows message-passing abstractions to be integrated within a shared-memory environment. A port is a first-class data object that serves as a receptacle for messages that maybe sent by other threads. Because Sting uses a shared virtual memory model [22], any complex data structure (including closures) can be sent along a port. This flexibility permits Sting applications to implement user-level message-passing protocols transparently and to combine the best features of shared memory and message passing within a unified environment.
2. *Memory Management:* A Sting virtual address space consists of a collection of *areas*; areas are used for organizing data that exhibit strong temporal or spatial locality. Sting supports a variety of areas: thread control blocks (or continuations), stacks, thread private heaps, thread shared heaps, etc.. Data is allocated to areas based on its intended use and lifetime; different areas can thus have different garbage collectors associated with them.
3. *Exception Handling:* As is the case with a thread-level context-switch, exceptions and interrupts are always handled in the execution context of some thread. Exception handlers are implemented as ordinary Scheme procedures, and dispatching an exception primarily involves manipulating continuations.

The remainder of this paper is structured as follows. Section 2 gives a brief overview of the system. Section 3 describes Sting's first-class user-level threads. Section 4 outlines the interaction of message-passing primitives and first-class procedures. Section 5 provides details of Sting's memory model. Section 6 describes *virtual processors*, an

[2] A continuation is an abstraction of a program point. It is typically represented as a procedure of one argument that defines the remaining computation needed to be performed from the program point it denotes [13].

abstraction that permit significant programmer-level control over mapping, scheduling and load-balancing of lightweight threads. Section 7 describes abstract physical machines, and Sting's exception handling mechanism. Section 8 provides benchmark results. Conclusions and comparison to related work is given in Section 9.

2 Sting Overview

Sting is a parallel dialect of Scheme[10] designed to serve as a high-level operating system for modern symbolic parallel programming languages.

The abstract architecture of Sting is organized hierarchically as layers of abstractions (see Fig. 1). The lowest level in this hierarchy is an *Abstract Physical Machine* (APM). An APM consists of a collection of *Abstract Physical Processors* (APP). An APP is an abstraction of an actual computing engine. An abstract physical machine contains as many abstract physical processors as there are real processors in a multiprocessor environment.

A *Virtual Machine* (VM) is an abstraction that is mapped onto a APM or a portion thereof. Virtual machines manage a single address space. They are also responsible for mapping global objects into local address spaces. In addition, each virtual machine contains the root of a graph of objects (*i.e.,* root environment) that is used to trace the set of live objects in its associated address space. A virtual machine is closed over a set of *Virtual Processors* (VPs). Virtual processors execute on abstract physical processors.

Abstract physical machines are responsible for managing virtual address spaces, and shared objects, handling hardware device interrupts and coordinating physical processors. Associated with each physical processor P is a virtual processor policy manager that implements policy decisions for the virtual processors that execute on P.

Virtual processors are responsible for managing user-created *threads*. A thread defines a separate locus of control. In addition, virtual processors also handle non-blocking I/O, synchronous exceptions (*e.g.,* invalid instructions, memory access violations), and interrupts (page refill, thread quantum expiration, etc.). Each virtual processor V is closed over a Thread Policy Manager (TPM) that (a) schedules threads executing within V, (b) migrates threads to and from other VPs, and (c) performs initial thread placement on the VPs defined within a given virtual machine.

Virtual processors are multiplexed on a physical processor in the same way that threads are multiplexed on a virtual processor. Separating the virtual machine abstraction from a specific hardware configuration allows us to map virtual topologies (in terms of virtual processors) onto any concrete architectural surface[12, 16].

Physical processors context-switch virtual processors because of preemption, or because a VP specifically requests a context switch (*e.g.,* because of an I/O call initiated by its current thread). We discuss virtual processors in greater detail in the following sections. State transitions on threads are implemented by a thread controller that implements the thread interface and thread state transitions.

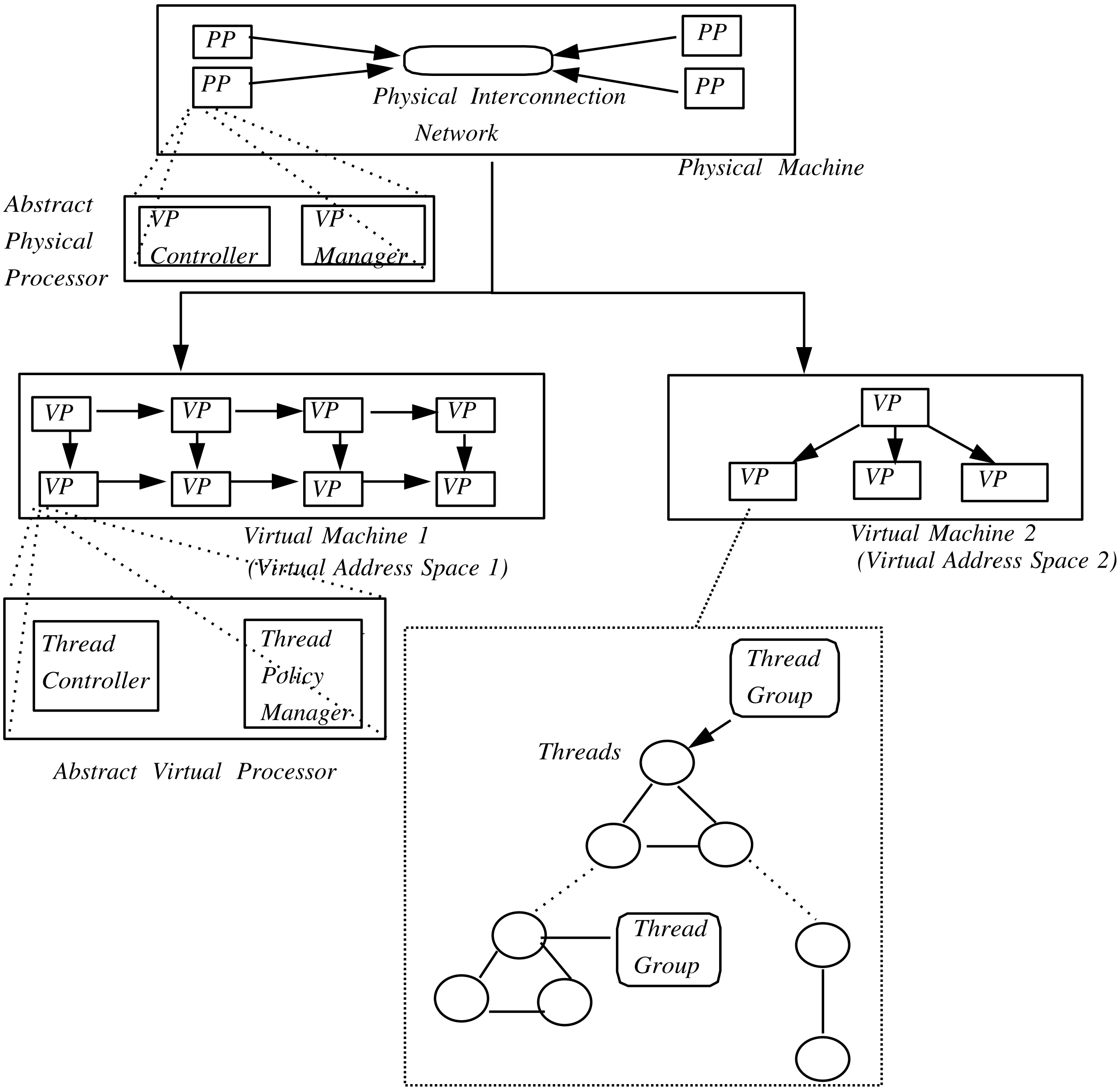

Fig. 1. The Sting Abstract Architecture.

Abstract physical machines, abstract physical processors, virtual machines, virtual processors and threads are all first-class objects in Sting. Since all these abstractions are implemented in terms of Scheme objects, the policy decisions of a processor (both physical and virtual) can be fully customized; for example, different virtual processors (even in the same virtual machine) may be closed over different policy managers. A processor is closed over a given policy manager by simply assigning a procedure to the appropriate slot of processor's data structure; this procedure defines an implementation of a scheduling or migration policy. Thus, processor abstractions serve to separate issues of policy (*i.e.,* scheduling, migration, etc.) from control (*i.e.,* blocking, suspension, termination etc.). This separation of concerns leads to important expressivity gains, and the use of first-class procedures ensures no performance penalty is incurred as a result of this flexibility.

3 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's state contains a thunk, *i.e.,* a nullary procedure, that is invoked when the thread executes. The value of the application is stored in the thread on completion. For example, evaluating the expression:

```
(*fork-thread (lambda () (+ y (* x z))) (current-vp))
```

creates a lightweight thread of control which is enqueued for execution on its evaluating virtual processor. When run, the thread applies the procedure,

```
(lambda () (+ y (* x z)))
```

The evaluation environment of this thunk is its lexical environment. We can avoid writing explicit thunks by using the syntactic form, `fork-thread` ; the expression,

(`fork-thread` E V)

is equivalent to,

(`*fork-thread (lambda ()` E) V)

A thread can be either *delayed*, *scheduled*, *evaluating*, *absorbed* or *determined*. A *delayed* thread will never be run unless the value of the thread is explicitly demanded. A *scheduled* thread is a thread that is scheduled to be evaluated on some VP, but which has not yet been allocated storage resources. An *evaluating* thread is a thread that has started running. A thread remains in this state until the invocation of its thunk yields a result, at which point its state becomes *determined*. Absorbed threads are an important specialization of evaluating threads used to optimize fine-grained programs in which threads exhibit significant data dependencies among one another[17].

Users manipulate threads via a set of procedures and syntactic forms defined by a thread controller (TC) which implements synchronous state transitions on a thread's state. The TC is written entirely in Scheme with the exception of two primitive operations to save and restore registers.

Because Scheme supports first-class procedures seamlessly, it is easy to build variants on the functionality provided by the TC *without* altering the implementation. For example, consider the following expression:

```
(define (P op)
  (let ((v  initial value))
    (op (lambda () (f v)))))
```

If `P` is applied thus,

```
(P (lambda (proc) (fork-thread (proc) (current-vp))))
```

the application of `f` to `v` occurs eagerly in a separate thread; in a slightly different call to `P`,

```
(P (lambda (proc) (delay-thread proc)))
```

the application is evaluated lazily. It is straightforward to construct several other alternatives, none of which involve altering `P`'s definition. Concurrency abstraction follows naturally from procedural abstraction in this framework.

3.1 Control State

Control state in Sting is managed via *thread control blocks* (TCBs). A TCB is a generalized representation of a *continuation*[13]; it records its associated thread's current state (registers, program counter, etc.). A TCB also includes its own stack and heaps.

The Sting implementation delays the allocation of a TCB to a thread until necessary. In most thread systems, the act of creating a thread involves not merely setting up the environment for the thread to be forked, but also allocating and initializing storage. This approach lowers efficiency in three important respects: first, in the presence of fine-grained parallelism, the thread controller 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; in fact, locality is often significantly reduced in such an implementation. Third, if threads T_1 and T_2 are both scheduled, but T_1 evaluates fully before T_2, T_2 will not reuse the storage allocated to T_1; thus, the aggregate storage requirements for such a program will be greater than necessary. Sting permits storage allocated to a one thread to be recycled and used by other threads that execute subsequently when possible [17, 18].

Thread control blocks are allocated from a pool local to each VP. This pool is organized as a LIFO queue. When a thread terminates, its TCB is recycled on the TCB pool of the VP on which it was executing; the LIFO organization of this pool guarantees that this TCB will be allocated to the next new thread chosen for execution on this VP. Since it is likely that the most recently used pieces of the TCB will be available in the physical processor's working set, cache and page locality is not compromised as a consequence of initiating evaluation of a new thread. Sting incorporates one further optimization on this basic theme: if a thread T terminates on VP V, and V is next scheduled to begin evaluation of a new thread (*i.e.,* a thread that is not yet associated with a dynamic context), T's TCB is immediately allocated to the new thread; no pool management costs are incurred in this case.

Besides local VP pools, VPs share access to a global TCB pool. Every local VP pool maintains an overflow and underflow threshold. When a pool overflows, its VP moves half the TCBs in the pool to the global pool; when the pool underflows, a certain number of TCBs are moved from the global pool to the VP-local one. Global pools serve two purposes: (1) they minimize the impact of program behavior on TCB allocation and reuse, and (2) they ensure a fair distribution of TCBs to all virtual processors. Since new TCBs are created only if both the global and the VP local pool are empty, the number of TCBs actually created during the evaluation of a Sting program is determined collectively by all VPs.

4 Message-Passing Communication

Sting uses a shared virtual memory model. Implementations of Sting on distributed memory platforms must be built on top of a distributed shared virtual memory substrate[5, 22]. Thus, the meaning of a reference does not depend on where the reference is generated, or where the object is physically located.

Message-passing has been argued to be an efficient communication mechanism on disjoint memory architectures [20], especially for parallel applications that are coarse-grained, or have known communication patterns. A *port* is a data abstraction provided in Sting to minimize the overheads of implementing shared memory on disjoint memory architectures. First-class procedures and ports exhibit an interesting and elegant synergy in this context.

Ports are first-class data structures. There are two basic operations provided over ports:

1. `(put` *obj port*`)` copies *obj* to *port*. The operation is asynchronous with respect to the sender.
2. `(get` *port*`)` removes the first message in *port*, and blocks if *port* is empty.

Objects read from a port P are copies of objects written to P. This copy is a shallow copy, *i.e.,* only the top-level structure of the object is copied, the substructure is shared. We have designed ports with copying semantics because they are designed to be used when shared memory would be inefficient. While the standard version of `put` does a shallow copy, there is also a version available that does a deep copy; this version not only copies the top-level object, but also all its substructures.

For example, sending a closure in a message using shallow copying involves constructing a copy of the closure representation, but preserving references to objects bound within the environment defined by the closure. The choice of copy mechanisms used clearly is influenced by the underlying physical architecture, and the application domain. There are a range of message transmission implementations that can be tailored to the particular physical substrate on which the Sting implementation resides.

Thus, evaluating the expression,

```
(put (lambda () E) port)
```

transmits the closure of the procedure `(lambda () E)` to *port*. If we a define a receiver on *port* thus,

```
(define (receiver port)
  (let ((msg (get port)))
    (fork-thread (msg) (current-vp))
    (receiver port)))
```

the procedural object sent is evaluated on the virtual processor of the receiver. By creating a new thread to evaluate messages, the receiver can accept new requests concurrently with the processing of old ones.

This style of communication has been referred to as *active messages* [31] since the action that should be taken upon message receipt is not encoded as part of the underlying implementation but determined by the message itself. There is a great deal of flexibility and simplicity afforded by such a model since the virtual processor/thread interface does not require any alteration in order to support message communication. Two aspects of the Sting design are crucial for realizing this functionality: (1) the fact that objects reside in a shared virtual memory[3] allow all objects (including those containing references to other objects, *e.g.,* closures) to be transmitted among virtual processors freely, and (2) first-class procedures permit complex user-defined message handlers to be constructed; these handlers can execute in a separate thread on any virtual processor. To illustrate, in the above example, E may be a complex query of a database. If a receiver is instantiated on the processor on which the database resides, such queries do not involve expensive migration of the database itself. Communication costs are reduced because queries are directly copied to the processor on which the database resides; the database itself does need not to migrate to processors executing queries. The ability to send procedures to data rather than more traditional RPC-style communication leads to a number of potentially significant performance and expressivity gains [14].

First-class procedures and lightweight threads make active message passing an attractive high-level communication abstraction. In systems that support active messages without the benefit of these abstraction, this functionality is typically realized in terms of low-level support protocols. First-class procedures make it possible to implement active message trivially; an active message is simply a procedure sent to a port. We also note that first-class ports have obvious and important utility in distributed computing environments as well and lead to a simpler and cleaner programming model than traditional RPC.

5 Memory Management

In Sting, there are three storage areas associated with every TCB. The first, a stack, is used to allocate objects created by the thread whose lifetime does not exceed the

[3] On distributed memory machines, objects would reside in a distributed shared virtual memory.

dynamic extent of its creator. More precisely, objects allocated on a stack may only refer to other objects that are allocated in a current (or earlier) stack frame, or which are allocated on some heap. Stack allocated objects can refer to objects in heaps because the thread associated with the stack is suspended while the heap is garbage collected; references contained in stacks are part of the root set traced by the garbage collector.

5.1 Private Heaps

Thread private heaps are used to allocate non-shared objects whose lifetimes might exceed the lifetime of the procedure that created them. We say might exceed because it is not always possible for the compiler to determine the lifetime of an object in higher order programming languages such as Scheme or ML. Furthermore, it may not be possible to determine the lifetimes of objects in languages which allow calls to higher-order unknown procedures. References contained in a private heap can refer to other objects in the same private heap, or objects in shared heaps, but they cannot refer to objects in the stack. References in the stack may refer to objects in the private heap, but references in the shared heap may not. Private heaps lead to greater locality since data allocated on them are used exclusively by a single thread of control; the absence of interleaving allocation among multiple threads means that objects close to together in the heap are likely to be logically related to one another.

No other thread can access objects that are contained in a thread's stack or private heap. Thus, both thread stacks and private heaps can be implemented in local memory on the processor without any concern for synchronization or memory coherency. Thread private heaps are actually a series of heaps organized in a generational manner. Storage allocation is always done in the youngest generation in a manner similar to other generational collectors [30, 4]. As objects age they are moved to older generations. All garbage collection of the private heap is done by the thread itself. In most thread systems that support garbage collection all threads in the system must be suspended during a garbage collection [19]. In contrast, Sting's threads garbage collect their private heaps independently and asynchronously with respect to other threads. Thus, other threads can continue their computation while any particular thread collects its private heap; this leads to better load balancing and higher throughput. A second advantage of this garbage collection strategy is that the cost of garbage collecting a private heap is charged only to the thread that allocates the storage, rather than to all threads in the system.

5.2 Thread Groups and Shared Heaps

Sting provides *thread groups* as a means of gaining control over a related collection of threads. Every thread is associated with some thread group. A child thread is in the same group as its parent unless it is created as part of a new 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.,* listing all threads in a given group, listing all groups, profiling, genealogy information, etc..) In addition, a thread group also includes a *shared heap* accessible to all its members.

A thread group's shared heap is allocated when the thread group is created. The shared heap like the private heap is actually a series of heaps organized in a generational manner. References in shared heaps may only refer to other objects in shared heaps. This is because any object that is referenced from a shared object is also a shared object and, therefore must reside in a shared heap. This constraint on shared heaps is enforced by ensuring that references stored in shared heaps refer to objects that are (a) either in a shared heap, or (b) allocated in a private heap and garbage collected into a shared one. That is, the graph of objects reachable from the referenced object must be copied into or located in the shared heap. The overheads of this memory model depend on how frequently references to objects allocated on private heaps escape; in our experience in implementing fine-grained parallel programs, we have found that most objects allocated on a private heap remain local to the associated thread, and are not shared. Those objects that are shared among threads often are easily detected either via language abstractions or by compile-time analysis.

To summarize, the reference discipline observed between the three areas associated with a thread are as follows:

1. References in the stack refer to objects in its current or previous stack frame, its private heap, or its shared heap.
2. References in the private heap refer to objects on that heap or to objects allocated on some shared heap.
3. References in the shared heap refer to objects allocated on its shared heap (or some other shared heap).

Like private heaps shared heaps are organized in a generational manner, but garbage collection of shared heaps is more complicated than that for private heaps because many different threads can simultaneously access objects contained in the shared heap. Note that as a result, shared heap allocation requires locking the heap.

In order to garbage collect a shared heap, all threads in the associated thread group (and its inferiors) are suspended. This is because any of these threads can access data in the shared heap. However, other threads in the system, *i.e.,* those not inferior to the group associated with the heap being collected, continue execution independent of the garbage collection.

Each shared heap has a set of incoming references associated with it. These sets are maintained by checking for stores of references that cross area boundaries. After the threads associated with the shared heap have been suspended, the garbage collector uses the set of incoming references as the roots for the garbage collection. Any objects reachable from the incoming reference set are copied to the new heap. When the garbage collection is complete the threads associated with the shared heap are resumed.

6 Virtual Processors and Thread Policy Managers

Virtual processors define scheduling, migration, and load-balancing decisions for the threads they execute. A virtual processor is closed over a thread policy manager (TPM) which defines a set of procedures that collectively determine a thread scheduling regime for this VP. Virtual machines or VPs can thus be tailored to handle different scheduling protocols or policies. The implementation relies on first-class procedures to attain this flexibility; VPs can execute different scheduling protocols simply by closing themselves over different TPMs.

The Sting design seeks to provide a flexible framework that is able to incorporate different scheduling regimes transparently without requiring modification to the thread controller itself. To this end, all TPMs must conform to the same interface although no constraints are imposed on the implementations themselves. The interface provides operations for choosing a new thread to run, enqueuing a scheduled or evaluating thread, setting thread priorities, and migrating threads. These procedures are expected to be used exclusively by the thread controller (TC); in general, user applications need not be aware of the thread policy manager/thread controller interface. A detailed description of policy managers and virtual processors is given in [17, 26].

7 Physical Processors

Sting is intended to serve as an operating system for modern programming languages. Like other contemporary operating systems (*e.g.,* Mach[7, 29], Chorus[27], or Psyche[23]), Sting's lowest-level abstraction is a micro-kernel called the *Abstract Physical Machine* (*APM*).

The APM plays three important roles in the Sting software architecture:

1. It provides a secure and efficient foundation for supporting multiple virtual machines.
2. It isolates all other components in the system from hardware dependent features and idiosyncrasies.
3. It controls access to the physical hardware of the system.

Physical processors multiplex virtual processors just as virtual processors multiplex threads. Each physical processor in an APM includes a virtual processor controller (VPC), and a virtual processor policy manager (VPM). In this sense, physical processors are structurally identical to virtual processors. The VPC effects state changes on virtual processors in the same way that the TC effects state changes on threads. Like threads, virtual processors may be running, ready, blocked, terminating or dead. A running VP is currently executing on a physical processor; a ready VP is capable of running, but is currently not. A blocked VP is executing one or more threads waiting on some external event (*e.g.,* I/O). The VPM is responsible for scheduling VPs on a physical processor; its structure is similar to a TPM, although the scheduling policies it defines will obviously be quite different. The VPM presents a well-defined interface to the VP controller; different Sting systems can contain different VP policy managers (*e.g.,* time-sharing systems have different scheduling requirements from real-time ones).

In current micro-kernels, program code in the micro-kernel is significantly different from that found in user programs. This occurs because many of the facilities available to user programs are not available at the kernel level. Sting addresses this problem by implementing the abstract physical machine in the *root virtual machine*. The root virtual machine has all the facilities that are available to any other program (or subsystem) running in a virtual machine including, a virtual address space, virtual processors, and threads. In addition, it has abstract physical processors, device drivers, and a virtual memory manager. A Sting abstract physical machine therefore has several important characteristics that distinguish it from other operating systems:

1. There are no heavyweight threads in the system. All threads are lightweight.
2. There are no kernel threads or stacks for implementing system calls. All system calls use the execution context of the thread making the system call. This is possible because portions of the abstract physical machine are mapped into every virtual machine in the system.
3. Asynchronous programming constructs in the abstract physical machine are implemented using threads as in any other virtual machine. Threads in the abstract physical machine can be created, terminated, and controlled in the same manner that threads in any virtual machine can.
4. When a thread blocks in the kernel it can inform its virtual processor that it has blocked. The virtual processor can then choose to execute some other thread. This is true for inter-thread communication as well as for I/O (*e.g.,* page faults).

A primary responsibility for APMs is to create, destroy and manage virtual machines. A virtual machine loosely corresponds to a Unix kernel process, a Mach task, or a Topaz address space [6]. Each of these entities define a virtual address space and a thread, but in these other systems the thread is a kernel thread, and therefore, heavyweight. Furthermore, these systems have no concept of customizable virtual processors or machines; thus, constructing scheduling, migration and load-balancing protocols in user-space is not possible. Schedular activations [3] and Psyche's processor abstraction [23] address one of the problems that Sting's virtual machines are intended to solve, *i.e.,* user space threads blocking in the kernel and informing the kernel when no thread is available to run in a kernel process or task; however, scheduler activations do not address customizable virtual processors, and do not leverage off high-level language mechanisms such as continuations or first-class procedures.

Psyche kernel processes are used to implement the virtual processors that execute user level threads. When a user thread blocks in the kernel, the kernel calls the software interrupt handler for this condition. The handler may decide to block the virtual processor, run another thread, or do whatever else is appropriate for the particular thread system being implemented. Software interrupts allow the kernel to notify the virtual processor whenever an event which might be of interest to it occurs in the kernel. Given Psyche's virtual processors it is possible to implement many different thread semantics and many different scheduling policies, and thus, Psyche is fully customizable. However, Psyche does not separate control and policy mechanisms in the virtual processor and thus each thread package must implement both of these mechanisms. In Psyche, user threads are distinct from kernel threads. In Sting, lightweight threads are integrated into the kernel design so that no kernel threads are necessary. We believe the Sting approach provides a significant increase in programmer efficiency, while at the same time providing an increase in program efficiency compared to that of more traditional operating systems.

To create a new virtual machine, an abstract physical machine must:

- create a new virtual address space,
- map itself into the virtual address space,
- create a root virtual processor in the virtual address space,
- and (possibly) create other virtual processors.

In addition, it must also allocate necessary abstract physical processors and map the virtual processors of the new machine onto them. It must also schedule the root virtual processor to run on its corresponding abstract physical processor. The root virtual processor executes the root thread for the virtual machine.

Destroying a virtual machine terminates all non-root threads executing on any virtual processors in the virtual machine. All devices are closed, all persistent areas in the address space are unmapped, and the root thread of the root virtual processor notifies its abstract physical processor which de-allocates the virtual address space associated with the virtual machine.

7.1 Exceptions

Of particular interest is Sting's treatment of exceptions (both synchronous and asynchronous) within the APM. Exception handling in Sting relies fundamentally on first-class procedures and continuations, and offers a good domain to highlight the utility of these abstractions in an important systems programming application.

Associated with every exception is a handler responsible for performing a set of actions to deal with the exception. Handlers are procedures that execute within a thread. An exception raised on processor P executes using the context of P's current thread. There are no special exception stacks in the Sting micro-kernel.

When an exception (*e.g.,* invalid instruction, memory protection violation, etc) is raised on processor P, P's current continuation (*i.e.,* program counter, heap frontier, top-of-stack, etc.) is first saved. The exception dispatcher then proceeds to find the target of the exception, interrupting it if the thread is *running*, and pushing the continuation of the handler and its arguments onto the target thread's stack. Having done so, the dispatcher may choose to (a) resume the current thread by simply returning into it, (b) resume the target thread, or (c) call the thread controller to resume some other thread on this processor. When the target thread is resumed, it will execute the continuation found on the top of its stack; this is the continuation of the exception handler.

The implementation of exceptions in Sting is novel in several respects:

1. Handling an exception simply involves calling it since it is a procedure.
2. Exceptions are handled in the execution context of the thread receiving it.
3. Exceptions are dispatched in the context of the current thread.
4. Exceptions once dispatched become the current continuation of the target thread and are executed automatically when the thread is resumed.
5. An exception is handled only when the target thread is resumed.
6. Exception handling code is written in Scheme and manipulates continuations and procedures to achieve the desired effect.

The target thread of a synchronous exception is always the current thread. Asynchronous exceptions or interrupts are treated slightly differently. Since interrupts can be directed at any thread (not just the currently executing one), handling such exceptions requires the handler to either process the exception immediately, interrupt the currently running thread to handle the exception, or create a new handler thread. Since interrupt handlers are also Scheme procedures, establishing a thread to execute the handler or using a current thread for that purpose merely involves setting the current continuation of the appropriate thread to call the handler.

```
 1: (define (exception-dispatcher type . args)
 2:   (save-current-continuation)
 3:   (let ((target handler (get-target&handler type args)))
 4:     (cond ((eq? target (current-thread))
 5:             (apply handler args))
 6:            (else
 7:              (signal target handler args)
 8:              (case ((exception-priority type))
 9:                ((continue) (return))
10:                ((immediate) (switch-to-thread target))
11:                ((reschedule) (yield-processor)))))))
```

Fig. 2. Pseudo Code for the Sting Exception Dispatcher.

To illustrate, Figure 2 shows pseudo-code for the Sting exception dispatcher. In line 2, the current continuation is saved on the stack of the current thread. The continuation can be saved on the stack because it cannot escape and it will only be called once. On line 3 the dispatcher finds the thread for which the exception is intended and the handler for the exception type. Line 4 checks to see if the target of the exception is the current thread and if so does not push the exception continuation (line 5). Rather, the dispatcher simply applies the handler to its arguments. This is valid since the dispatcher is already running in the context of the exception target, i.e. the current thread. If the target of the exception is not the current thread, the dispatcher sends the exception to the target thread (line 7). Sending a thread a signal is equivalent to interrupting the thread and pushing a continuation containing the signal handler and its arguments onto the thread's stack, and resuming the thread which causes the signal handler to be executed. After signaling the target thread, the handler decides which thread to run next on the processor (line 8). It may be itself (line 9), the target thread (line 10), or the thread with the highest priority (line 11).

There is one other important distinction between Sting's exception handling facilities and those found in other operating systems. Since threads that handle exceptions are no different from other user-level threads in the system (*e.g.,* they have their own stack and heap), and since exception handlers are ordinary first-class procedures, handlers are free to allocate storage dynamically. Data generated by a handler will be reclaimed by a garbage collector in the same way that any other datum is recovered. The uniformity between the exception handling mechanism and higher-level Sting abstractions allows device driver implementors expressivity and efficiency not otherwise available in parallel languages or operating systems.

Sting is able to provide this model of exceptions because first-class procedures and threads, manifest continuations, dynamic storage allocation, and a uniform addressing mechanism are all central features of its design.

8 Benchmarks

Sting is currently implemented on Silicon Graphics MIPS R3000 shared-memory multiprocessor (cache-coherent) machines. 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 runs exactly one virtual processor. The timings shown below were taken on an 8 processor configuration.

Fig. 3 gives baseline figures for various thread operations; these timings were derived using a single global FIFO queue. Our processor platform was an eight node 40MHz SGI cache-coherent shared-memory machine.

Case	*Timings(in μseconds)*
Thread Creation	40
Thread Fork and Value	86.9
Thread Enqueue/Dequeue	14.5
Synchronous Context Switch	12.8
Thread Block and Resume	27.9

Fig. 3. Baseline timings.

The "Thread Creation" timing is the cost to create a thread. "Thread Fork and Value" measures the cost to create a thread that evaluates the null procedure and returns. "Thread

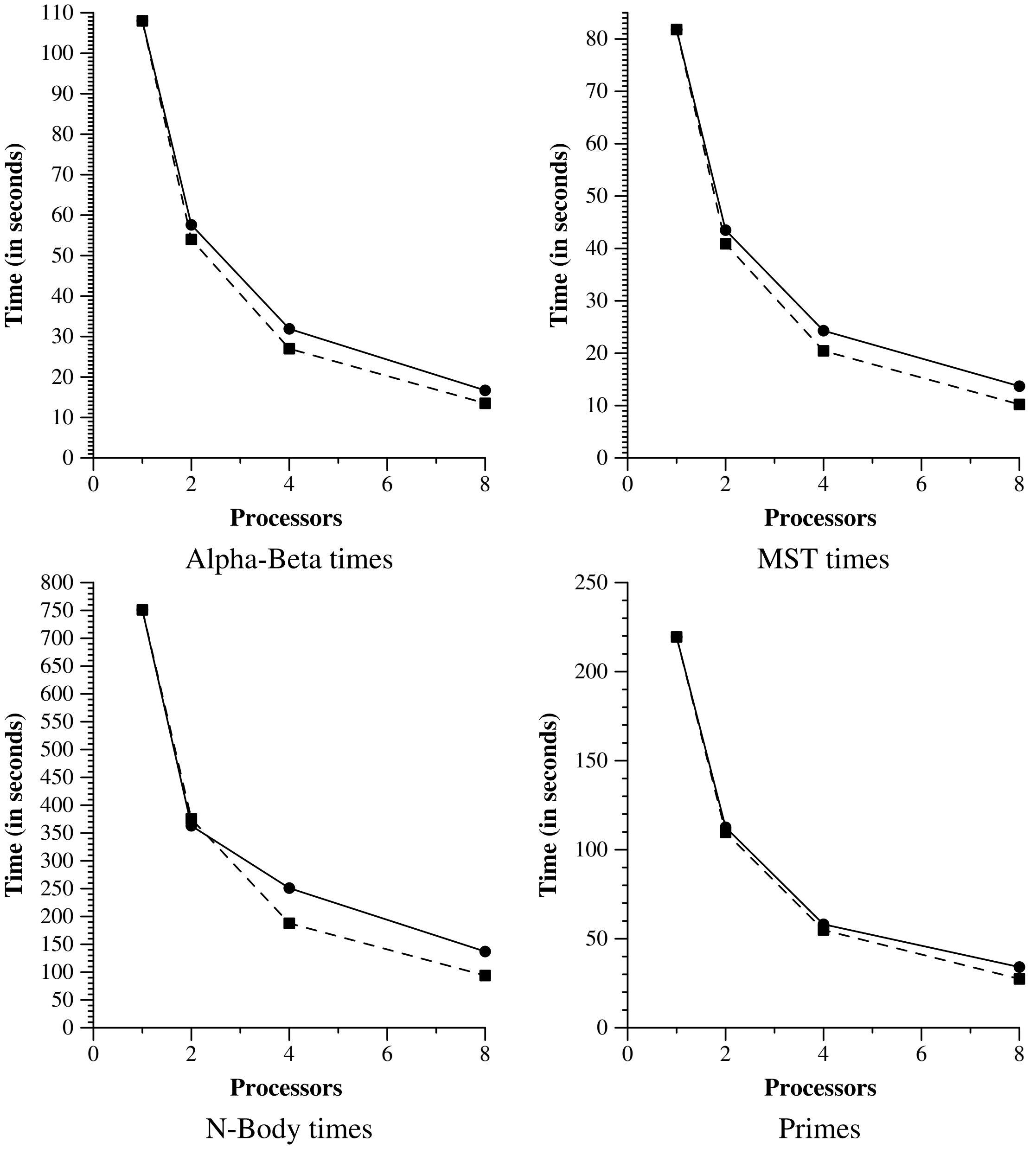

Fig. 4. Wallclock times on benchmark suite.

Enqueue/Dequeue" is the cost of inserting a thread into the ready queue of the current VP, and immediately removing it; it is thus one measure of scheduling overhead. A "Synchronous Context Switch" is the cost of context-switching a thread that resumes immediately. "Thread Block and Resume" is the cost of blocking and resuming a null thread.

Fig. 4 shows benchmark times for four applications (solid lines indicate actual wallclock times; dashed lines indicate ideal performance relative to single processor times):

1. **Primes:** This program computes the first million primes. It uses a master-slave algorithm; each slave computes all primes within a given range.

2. **Minimum Spanning Tree:** This program implements a geometric minimum spanning tree algorithm using a version of Prim's algorithm. The input data is a fully connected graph of 5000 points with edge lengths determined by Euclidean distance. The input space is divided among a fixed number of threads. To achieve better load balance, each node not in the tree does a parallel sort to find its minimum distance among all nodes in the tree. Although 46,766 threads are created with this input, only 16 TCBs are actually generated. This is again due to Sting's aggressive treatment of storage locality, reuse and context sharing.
3. **Alpha-Beta:** This program defines a parallel implementation of a game tree traversal algorithm. The program does α/β pruning[11, 15] on this tree to minimize the amount of search performed. The program creates a fixed number of threads; each of these threads communicate their α/β values via distributed data structures. The input used consisted of a tree of depth 10 with fanout 8; the depth cutoff for communicating α and β values was 3. To make the program realistic, we introduced a 90% skew in the input that favored finding the best node along the leftmost branch in the tree.
4. **N-body:** This program simulates the evolution of a system of bodies under the influence of gravitational forces. Each body is modeled as a point mass and exerts forces on all other bodies in the system. The simulation proceeds over time-steps, each step computing the net force on every body and thereby updating that body's position and other attributes. This benchmark ran the simulation on 5000 bodies over six time-steps.

Speedup and efficiency ratios for these problems are shown in Fig. 5.

Benchmark	Speedup	Efficiency
Primes	6.4	81
MST	5.9	75
N-Body	5.5	69
Alpha-Beta	6.5	81

Fig. 5. Efficiency and speedup rations on 8 processors.

9 Conclusions

The utility of high-level language abstractions such as first-class procedures and continuations is well-appreciated in sequential symbolic application domains. Sting is an attempt to generalize the purview of these abstractions to systems software. The need for efficiency often compromises generality and elegance in operating system implementations. Our investigations support the hypothesis that with careful engineering and thoughtful design, useful high-level abstractions can be profitably applied to such performance critical domains. The free use of these abstractions also leads to smaller and simpler systems and data structures.

Besides providing an efficient multi-threaded programming system similar to other lightweight thread systems[8, 9, 24], the liberal use of procedural and control abstractions provide added flexibility. First, advanced memory management techniques are facilitated. Second, Sting's pervasive use of continuation objects in implementing thread storage permits a variety of other storage optimizations not available otherwise[18]. Third, first-class procedures provide an expressive basis for implementing flexible message-passing protocols. Finally, Sting is built on an 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.

In certain respects, Sting resembles other advanced multi-threaded operating systems[3, 23]: for example, it supports user control over interrupts, and exception handling at the virtual processor level. 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, continuations, general exception handling, and rich data abstractions) that typical operating system environments do not offer.

References

1. Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985.
2. 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.
3. 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.
4. Andrew Appel. A Runtime System. *Journal of Lisp and Symbolic Computation*, 3(4):343–380, November 1990.
5. John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type–Specific Memory Coherence. In *Symposium on Principles and Practice of Parallel Programming*, March 1990.
6. A.D. Birrell, J.V. Guttag, J.J. Horning, and R. Levi. Synchronization Primitives for a Multiprocessor: A Formal Specification. In *Proceedings of the 11^{th} Symposium on Operating Systems Principles*, pages 94–102, November 1987.
7. David L. Black, David B. Golub, Daniel P. Julin, Richard Rashid, Richard P. Draves, Randall W. Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel Operating System Architecture and Mach. In *Workshop Proceedings Micro-Kenels and other Kernel Architectures*, pages 11–30, April 1992.

8. Eric Cooper and Richard Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie-Mellon University, June.
9. Eric Cooper and J.Gregory Morrisett. Adding Threads to Standard ML. Technical Report CMU-CS-90-186, Carnegie-Mellon University, 1990.
10. William Clinger *et. al*. The Revised Revised Revised Report on Scheme or An UnCommon Lisp. Technical Report AI-TM 848, MIT Artificial Intelligence Laboratory, 1985.
11. Raphael Finkel and John Fishburn. Parallelism in Alpha-Beta Search. *Artificial Intelligence*, 19(1):89–106, 1982.
12. David Saks Greenberg. *Full Utilization of Communication Resources*. PhD thesis, Yale University, June 1991.
13. Christopher Haynes and Daniel Friedman. Embedding Continuations in Procedural Objects. *ACM Transactions on Programming Languages and Systems*, 9(4):582–598, 1987.
14. Wilson Hsieh, Paul Wang, and William Weihl. Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 239–249, May 1993. Also appears as ACM SIGPLAN Notices, Vol. 28, number 7, July, 1993.
15. Feng hsiung Hsu. *Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess*. PhD thesis, Carnegie-Mellon University, 1990. Published as Technical Report CMU-CS-90-108.
16. Paul Hudak. Para-Functional Programming. *IEEE Computer*, 19(8):60–70, August 1986.
17. Suresh Jagannathan and James Philbin. A Customizable Substrate for Concurrent Languages. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1992.
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, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hon Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–63, 1993.
21. Peter J. Landin. The Mechanical Evaluation of Languages. *Computer Journal*, 6(4):308–320, January 1964.
22. Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
23. 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.
24. Sun Microsystems. *Lightweight Processes*, 1990. In SunOS Programming Utilities and Libraries.
25. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
26. James Philbin. *An Operating System for Modern Languages*. PhD thesis, Dept. of Computer Science, Yale University, 1993.
27. M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Newhauser. Overview of the Chorus Distributed Operating System. In *Workshop Proceedings Micro-Kenels and other Kernel Architectures*, pages 39–69, April 1992.

28. Guy Steele Jr. Rabbit: A Compiler for Scheme. Master's thesis, Massachusetts Institute of Technology, 1978.
29. 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.
30. 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.
31. Thorsten Von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. Active Messages: A Mechanism for Integrated Communication and Compuation. In *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.