A Portable Multiprocessor Interface for Standard ML of New Jersey

J. Gregory Morrisett[1] Andrew Tolmach[2]
Carnegie Mellon University Princeton University
jgmorris@cs.cmu.edu apt@cs.princeton.edu

June 1992
CMU-CS-92-155

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We have designed a portable interface between shared-memory multiprocessors and Standard ML of New Jersey. The interface is based on the conventional kernel thread model and provides facilities that can be used to implement user-level thread packages. The interface supports experimentation with different thread scheduling policies and synchronization constructs. It has been ported to three different multiprocessors and used to construct a general purpose, user-level thread package. In this paper, we discuss the interface and its implementation and performance, with emphasis on the Silicon Graphics 4D/380S multiprocessor.

[1]Supported in part by a National Science Foundation Graduate Fellowship.
[2]Supported in part by NSF grant CCR-9002786.

This research was sponsored in part by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168. The research was conducted in part while the authors were visiting AT&T Bell Laboratories, Murray Hill, NJ. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or AT&T.

Keywords: concurrency, threads, Standard ML, continuations, shared memory, multiprocessor

1 Introduction

Many applications, such as window systems, operating systems, transaction systems, interactive games, *etc.*, provide multiple services to one or more clients. To provide low latency service, such programs are often structured in a *multi-threaded* fashion. Many researchers [11, 13, 14, 19, 25, 31, 39, 41, 48] are interested in adding concurrency primitives to "higher-order" languages such as Standard ML [33] so that applications can be written in a multi-threaded fashion and still take advantage of other high-level features, such as strong typing, garbage collection, modules, *etc.*

Standard ML of New Jersey [9] (SML/NJ) provides an elegant set of extensions to SML that makes writing an efficient thread package for uniprocessors quite easy. Uniprocessor implementations *simulate* concurrent thread execution, distributing access to the processor among the currently active threads, typically with the aid of a clock-driven preemption mechanism. At least four such packages currently exist [14, 31, 37, 39].

Many multi-threaded applications offer better latency if they are executed in truly concurrent fashion, using multiple processors. Moreover, a truly concurrent thread model can be used to express parallel algorithms designed to maximize throughput. We have designed and implemented a shared-memory multiprocessor interface for SML/NJ, called **MP**, so that thread packages can take advantage of parallel hardware. Our major design goals were portability, simplicity, and efficiency. To achieve these goals, we have kept the **MP** interface as small as possible, so that it can be easily ported to a variety of machines and operating systems. Porting **MP** to a new machine involves implementing only a small number of system-dependent routines. The interface provides enough functionality so that thread packages may be portably implemented in ML. This gives us flexibility to experiment with different sets of thread primitives, scheduling policies, *etc.*, using the language's powerful abstraction mechanisms. We use a range of mechanisms to interface ML code with the SML/NJ runtime system both efficiently and cleanly.

The **MP** interface has been ported to the Silicon Graphics (SGI) 4D/380S, the Omron Luna 88k, and the Sequent Symmetry multiprocessors.[1] Both the Omron and Sequent implementations were completed within a week—a testament to the portability of the interface. Trivial uniprocessor implementations of **MP** exist for all systems that run SML/NJ. A portable version of ML Threads [14] extended with "safe refs" as suggested by Appel and Tolmach [45], has been built on top of the **MP** interface and can be used as a basis for experimenting with multiprocessing.

In this paper, we first show how SML/NJ provides appropriate mechanisms for building uniprocessor thread packages. We then present our multiprocessor interface and show how a parallel thread package can be implemented on top of it. We discuss the system-independent implementation routines and the system-dependent routines for the SGI multiprocessor. We close with performance results and some open problems.

We will assume that the reader has a basic knowledge of SML/NJ. In particular, the reader should understand first-class continuations in order to follow the examples. Some knowledge of operating systems and their concurrency features will also be helpful.

[1] The Sequent port was done by Lorenz Huelsbergen of the University of Wisconsin using the `sml2c` compiler [44].

2 Terminology

A *machine* consists of some number of independent physical *processors* that share a common memory. The shared memory might not be sequentially consistent; i.e., for a processor to guarantee that the result of a memory operation is visible to other processors, it may need to issue explicit instructions in addition to the memory operation itself. A machine that has only one processor is called a *uniprocessor*, while machines with more than one processor are called *multiprocessors*.

A *task* (sometimes referred to as an *address space*), is an abstract unit of resource allocation exported by the operating system. A task typically contains a view of a portion of the machine's memory, some number of *kernel threads*, and other resources (*e.g.*, file descriptors) that the OS has granted the task.

A *thread* is an abstract agent of computation that provides the illusion of a *virtual* processor. We say that the threads of a program *multiplex the control* of the program. Threads and their associated operations may be provided by the operating system (kernel threads) or by a set of user-level routines (user-level threads).

A *kernel thread* is a thread that is provided and managed by the OS. Each kernel thread belongs to at most one task and can be scheduled to execute on any of the processors at any time. In particular, the OS may choose to run the threads of a task in *parallel*. Additionally, the OS has the ability to *preempt* (*i.e.*, suspend) a kernel thread at any time. Each of a task's kernel threads may access any memory location of which the task has a view.

A *user-level thread* is a thread that is provided and managed by a user-level set of routines. User-level threads can provide much better performance and flexibility in scheduling than kernel threads [2, 16, 30, 42, 47], because thread operations do not require expensive system calls. However, kernel intervention is typically still required to obtain access to multiple processors and to manage I/O. Therefore, user-level threads are often multiplexed on top of kernel threads; typically, there is one kernel thread per physical processor.

Figure 1 shows the typical relationship between processors, tasks, kernel threads, and user-level threads. In SML/NJ, operating system services are provided via a runtime system written in C; most services are exported to the ML environment as functions (see Section 5.1). We have a choice between implementing user-level threads in C and exporting them to ML, or exporting an interface to kernel threads and implementing user-level threads directly in ML. Since ML provides excellent facilities for implementing threads in a uniprocessor environment, as described in Section 3, it is natural to choose the latter alternative; the resulting interface is described in Section 4.

3 An SML/NJ User-Level Thread Package

Standard ML is a mostly functional programming language that provides first-class functions (closures), compile-time typing, polymorphism, exceptions, garbage collection, and a powerful module facility. In addition, the language has a complete formal semantic specification [33].

The Standard ML of New Jersey implementation [9, 7] supports type-safe, first-class continuations [17] and provides asynchronous exception handling facilities in the form of signal

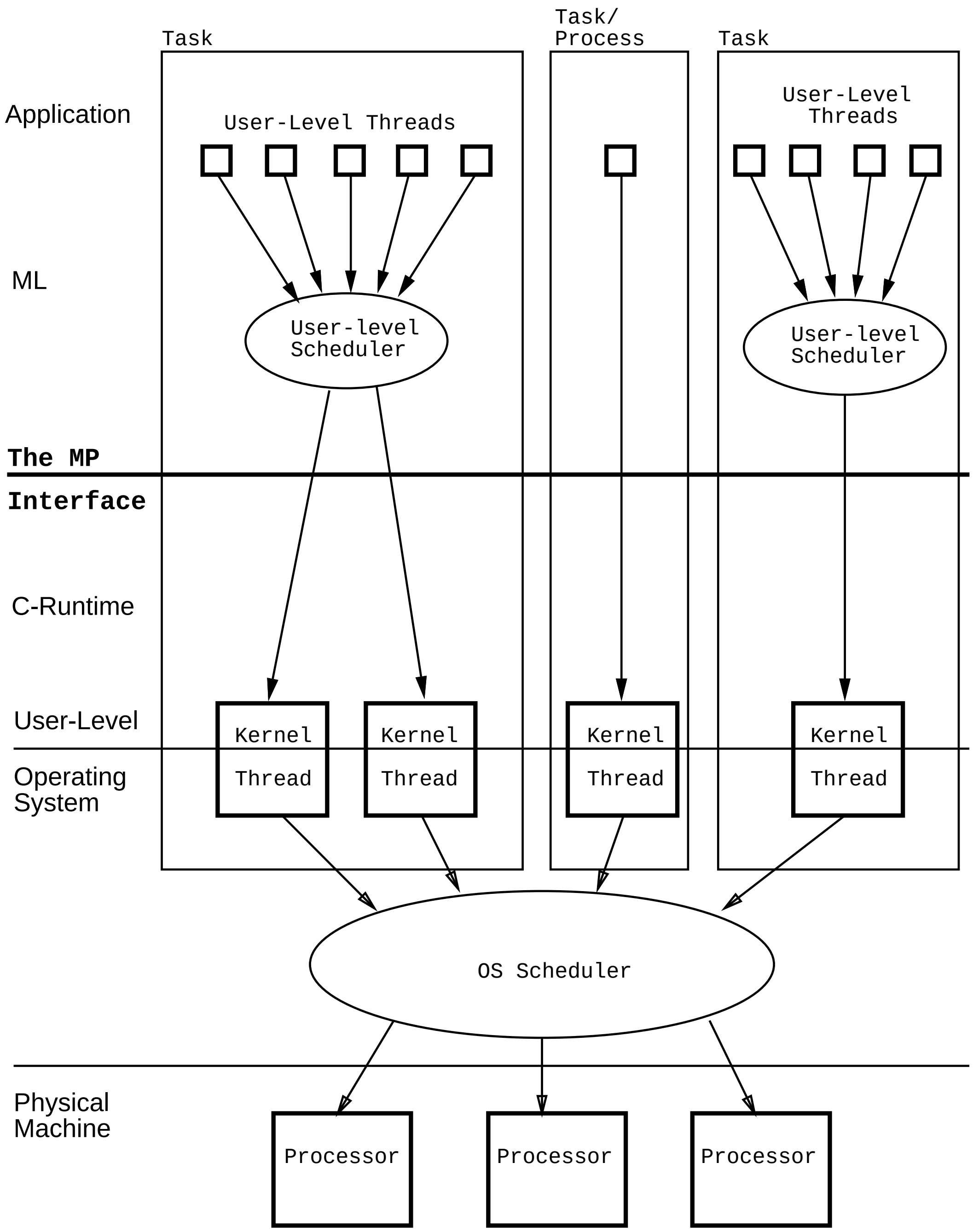

Figure 1: Relationship of Tasks, Kernel Threads, and User-Level Threads

```
signature QUEUE =
    sig
        type 'a queue
        val create : unit -> '1a queue
        val enq : 'a queue -> 'a -> unit
        val deq : 'a queue -> 'a
        exception Deq
    end

signature THREAD =
    sig
        val fork : (unit -> unit) -> unit
    end
```

Figure 2: Queue and Thread Signatures

handlers [38]. These extensions provide everything that is needed to build complete, uniprocessor, user-level thread packages *in* the language.

This section describes the implementation of a simple thread package, similar to the core of existing systems such as CML [39] and ML Threads [14].

3.1 Thread Management

Wand [46] is credited with showing how concurrency can be simulated using continuations. Figure 3 shows how a very simple user-level thread mechanism can be implemented in SML/NJ. The thread module is presented as a functor, parameterized by a `QUEUE` structure, whose implementation is straightforward and is not shown here. The `THREAD` signature exports just the function `fork` to create new threads. The `QUEUE` and `THREAD` signatures are shown in Figure 2.

Each waiting thread is represented by a unit-accepting continuation. A queue (`ready`) of continuations is used to store threads that are not currently running. When given a function `child`, `fork` captures the current continuation using `callcc` and binds it to `parent`. The `parent` continuation is placed on the `ready` queue, and `child` is executed. When `child` completes, a continuation is dequeued from `ready` and invoked using `throw`, causing execution to continue in the context of another thread.

Threads inherit the environment of their parent, which may contain both mutable and immutable identifiers. The mutable identifiers form a shared memory: any `ref` cell or array accessible from two threads is implicitly shared by them. A variety of synchronization and communication mechanisms can be implemented on top of this shared memory. Two such mechanisms, mutex locks and pipes, are described in Appendices A and B.

To prevent a compute-bound thread from monopolizing the processor, timer alarm signals are used to trigger thread preemption. The call to `setitimer` causes the signal `SIGALRM` to be delivered to the process every 50 *msec*. The call to `setHandler` installs the function `switch` as a handler for the signal. When an alarm signal is delivered to the process, `switch`

```
functor CoThread (Queue : QUEUE) : THREAD =
    struct
        val ready : (unit cont) Queue.queue = Queue.create ()
        val atomic_flag = ref false
        fun atomic () = !atomic_flag
        fun enter_atomic () = atomic_flag := true
        fun leave_atomic () = atomic_flag := false
        fun atomically f x =
             (enter_atomic ();
              let val r = f x
              in
                  leave_atomic ();
                  r
              end) handle exn => (leave_atomic ();
                                  raise exn)
        val reschedule = atomically (Queue.enq ready)
        fun getnext () = atomically (Queue.deq ready)

        fun fork (child : unit -> unit) =
               callcc (fn parent =>
                       (reschedule parent;
                        child ();
                        throw (getnext()) ()))

        fun switch(_,k : unit cont) =
            if atomic () then
              k
            else (reschedule k;
                  getnext ())
        local
            open System.Signals System.Timer
                 System.Unsafe.CInterface
            val t = TIME {sec = 0, usec=50000}
        in
            val _ = (setHandler(SIGALRM,SOME switch);
                     setitimer(0,t,t))
        end
    end
```

Figure 3: Implementing Threads in SML/NJ

is given the current continuation (`k`) of the process. The handler is expected to return a continuation to be invoked. Ordinarily, `switch` enqueues `k` onto the `ready` queue, and dequeues and returns another thread's continuation to be invoked.

The `enq` and `deq` operations must be atomic with respect to preemption; otherwise, the `ready` queue might end up in an inconsistent state. A flag (`atomic_flag`) is used to indicate when an "atomic" operation is in progress; `switch` preempts the current function only if `atomic_flag` is clear. If `atomic_flag` is set when the alarm goes off, `switch` simply returns the continuation that it was given, and the computation proceeds as if it had not been interrupted.

3.2 Performance Considerations

The efficiency of this thread package implementation depends directly on that of SML/NJ's continuation primitives. Fortunately, the cost of creating and invoking continuations in SML/NJ is low relative to overall execution speed, comparable with the cost of invoking an ML library function. There are several reasons for this efficiency. First, the SML/NJ compiler uses an intermediate form known as *continuation-passing style* that makes all continuations in a program explicit, so it is simple for the compiler to expose continuations to the user. Second, all closures (*i.e.*, procedure-frames) are allocated on the heap instead of on the stack. Thus, continuation creation (via `callcc`) consists merely of allocating and initializing a closure. No copying from a stack to the heap needs to take place. Conversely, when a continuation is invoked (via `throw`), no copying from the heap back to the stack is necessary.

On the other hand, SML/NJ's use of heap allocation for closures may make ordinary execution slower than it would be were conventional stack allocation used. Although Appel has shown that heap allocation can compete with stack allocation for certain uniform memory architectures [4], this is less likely to be true for systems with small first-level caches [24]. Therefore, while thread operations built on top of SML/NJ's continuations should be efficient relative to the rest of the ML computation, thread applications may not perform as well overall under SML/NJ as under competing systems.

In fact, first-class continuations are an overly general mechanism for saving thread state. Since any waiting thread can be resumed at most once, it is permissible to destroy the thread's saved state in the process of reinstating it, which can reduce the amount of copying required by a stack-based implementation [22].

4 The MP Interface

Extending SML/NJ to support multiprocessing involves both explicitly adding to the set of language primitives and implicitly adapting existing primitives to the multiprocessor environment. Figure 4 shows the explicit **MP** multiprocessor interface in the form of an ML signature. This signature describes the abstract set of services available to an **MP** *client*, e.g., a particular thread package. The main abstraction presented by the **MP** interface is a *proc*—a language-level view of a kernel thread executing on a physical processor. The interface provides operations for managing procs and their state. Mutual exclusion among procs is provided by *spin locks*.

```
signature MP =
  sig
      type proc_datum
      datatype proc_state = PS of (unit cont * proc_datum)

      exception Acquire_Proc
      val acquire_proc: proc_state -> unit
      val release_proc: unit -> 'a
      val active_procs: unit -> int

      val initial_datum : proc_datum
      val get_datum     : unit -> proc_datum
      val set_datum     : proc_datum -> unit

      type spin_lock
      exception Spin_Lock
      val spin_lock: unit -> spin_lock
      val try_lock : spin_lock -> bool
      val lock     : spin_lock -> unit
      val unlock   : spin_lock -> unit
  end
```

Figure 4: The **MP** Interface

4.1 Proc Management

The `proc_state` datatype represents the state of a proc. It consists of a continuation and a client-defined `proc_datum`. `acquire_proc` attempts to acquire a new proc for the task. It takes as an argument a `proc_state` and immediately returns. If the operation is successful, the new proc is given the continuation of the `proc_state` to execute in parallel with the proc that called `acquire_proc`. In addition, the `proc_datum` portion of the `proc_state` is installed, as described below.

Typically, **MP** is implemented so that the number of available procs is equal to the number of physical processors on the machine; after this limit is reached, calls to `acquire_proc` will raise the exception `Acquire_Proc`. However, the number of physical processors available to an ML program typically changes during a computation as a result of activity by other users and by the operating system itself. Most operating systems lack a mechanism to inform user space code about dynamic changes in processor availability. Thus, as with kernel threads, the correspondence between procs and available physical processors is only an approximate one.

The operation `release_proc` can be used to give the calling proc back to **MP**. If there is only one proc currently running and it calls `release_proc`, then the entire task exits, which might be undesirable. To help the client avoid releasing the last proc, function `active_procs` returns an integer indicating how many active procs the task currently has. `release_proc` is typically implemented by releasing the current physical processor to the operating system, by either blocking in or exiting from the current kernel thread.

```
signature MP_ARG =
  sig
      type proc_datum
      val initial_datum : proc_datum
  end

functor MP (MP_Arg : MP_ARG) : MP =
  struct
      ...
  end
```

Figure 5: The MP Functor

Since `acquire_proc` and `release_proc` require communication with the operating system, the client may wish to invoke them sparingly. To obtain good performance (at the expense of other system users) a client typically calls `acquire_proc` repeatedly when it starts up, acquiring as many procs as possible, and holds on to them for the duration of the computation.

4.2 Per-Proc Data

Often, a proc needs some small amount of *private* state. For example, consider the flag `atomic_flag` of Figure 3. This flag indicates whether the single proc can be interrupted or not. If we have multiple procs, then we need to have multiple flags, since some procs might be executing an atomic operation at the same time that other procs are not.

The **MP** interface provides a single, programmer-defined `proc_datum` for each proc. The operations `get_datum` and `set_datum` allow a proc to read and write its private datum. The datum value for the initial proc is given by `initial_datum`. As an example, the `atomic_flag` for a proc can be stored directly in its `proc_datum`. If clients need to store more complex pieces of state, they can treat `proc_datum` as a record or array. To support arbitrary datum values in a type-safe manner, type `proc_datum` and the `initial_datum` value are defined as arguments to the **MP** functor, as shown in Figure 5.

4.3 Memory Management and Mutual Exclusion

All live heap memory is implicitly shared among all procs; in particular, a proc can freely read or write into heap locations allocated by another proc. In ML, only specially declared `ref` and `array` variables can be updated after being allocated, so all communication between procs must be via such variables. If two procs perform conflicting operations on the same data, the result is unpredictable—it depends on details of scheduling, relative execution speed, compiler code generation, and hardware architecture. One way to solve these race condition problems is to use some form of *mutual exclusion locks.*

We provide *spin locks* to solve race condition problems at the **MP** level. Spin locks are meant to be held for very short periods of time—so short, in fact, that it is more efficient to spin (*i.e.*, wait) for a lock than to do some sort of context-switch. Every potentially shared, mutable memory location should be protected by such a lock, to guarantee that only one proc at a time can access it.

The current **MP** specification does not address the underlying memory consistency model provided by the hardware architecture; maintaining the desired degree of consistency is the responsibility of the **MP** client. For the platforms on which it is currently implemented, **MP** does expose enough of the low-level architecture to allow clients to control consistency; on the SGI, for example, wrapping all shared memory accesses—including aligned single-word writes—with spin lock operations will suffice to give the illusion of sequential consistency. Defining a uniform higher-level model remains an important goal for future work.

We chose not to provide "heavier-weight" synchronization primitives for two reasons. First, most hardware today provides primitives that directly support spin locks, such as an atomic test-and-set instruction. Second, other synchronization constructs such as mutex locks, reader/writer locks, semaphores, pipes, channels, *etc.,* can be easily implemented using spin locks in conjunction with first-class continuations; see Appendices A and B for examples.

The operation `spin_lock` attempts to return a fresh `spin_lock` in unlocked state. Implementations of **MP** are expected to implement these locks as efficiently as possible, e.g., using hardware support. Since some machines may provide only a limited number of hardware locks,[2] the **MP** implementation is also permitted to run out of locks, in which case exception `Spin_Lock` is raised by the next `spin_lock` operation. Thus, thread packages are required to cope with the possibility of running out of locks by multiplexing them as necessary.

The operation `try_lock` attempts to lock the specified spin lock and immediately returns a `bool` indicating the success of the operation. If the operation is successful, the proc is said to "own the lock". At most one proc can own a given spin lock at any time. The spin lock can be released by calling the `unlock` operation; this need not be done by the proc that set the lock.

The `lock` operation is similar to `try_lock` except that it does not return until the lock is successfully obtained. It is functionally equivalent to the following routine:

```
fun lock sl = while not(try_lock sl) do () (* spin *)
```

`lock` is provided in the interface since some operating systems may provide a more efficient spin than the one shown above (*e.g.*, by using backoff techniques [1]).

4.4 I/O

I/O presents at least two problems for user-level thread packages. The first affects both uniprocessor and multiprocessor implementations. When a user-level thread performs some blocking I/O operation, it should not block the proc on which it is running indefinitely if there are other user-level threads that need to run. A thread blocked while attempting I/O is similar to a compute-bound thread, and it can be dealt with in the same way, by using UNIX timer alarm signals to preempt the thread. In the existing SML/NJ implementation, I/O and signal handling are coordinated to support this preemption technique [38]. Before

[2] In fact, the SGI multiprocessor is the only machine we have worked with that limits the number of hardware locks.

a blocking I/O operation is performed (*e.g.*, read from the keyboard), a UNIX `select` call is made. The `select` may be interrupted by a signal. If the `select` completes without being interrupted, indicating that the operation may be completed without blocking, signals are masked and the actual operation is done. If a signal occurs during the `select`, the current continuation is "backed-up" to retry the entire operation, so it appears to the signal handler as though the signal occurred *before* the blocking I/O operation was initiated. Thus, no proc will be blocked indefinitely on an I/O operation.

A second problem affects multiprocessor implementations: two processors may perform I/O operations simultaneously, possibly accessing the same runtime data structures. At present, **MP** takes no specific steps to prevent such conflicts; the client is responsible for protecting the primitives with suitable mutual exclusion locks.

4.5 Signals

We use the existing SML/NJ signal interface [38], adding suitable conventions for multiprocessing. SML/NJ exposes only a subset of the UNIX signals to the ML user; of these, we are primarily concerned with providing suitable treatment for SIGALRM (timer alarm) and SIGINT (keyboard interrupt) signals.

Signal handlers are installed via a call to `setHandler`. The signal handlers are per-task resources. That is, each proc within a task shares the same signal handling function. However, each proc may choose to mask or unmask signals independently via the `maskSignals` operation. When a signal occurs, it is delivered to each proc. If the proc has masked its signals, then the delivery will be delayed until the proc unmasks signals.

There is no facility in the interface for procs to signal one another, since they can communicate through their shared address space. **MP** does not support asynchronous alerts or control operations by one proc acting on another, although these may be simulated using polling in the target proc. The timer alarm can be used to issue the poll at appropriate intervals.

4.6 An MP Thread Package

In this section, we show how to extend the implementation of the simple thread package from Section 3 so that user-level threads can run in parallel using the **MP** interface.

The `ready` queue must be protected by spin locks to avoid race conditions between competing procs. Figure 6 shows a functor that takes a queue implementation, and provides queues that are safe with respect to multiple procs.

Figure 7 shows a functor that takes a safe queue implementation, an **MP** implementation where the per-proc data are `bool` values and the `initial_object` is `false`, and produces a structure that matches `THREAD`. The `ready` queue is a safe unit continuation queue. Thus, enqueuing or dequeuing a thread is an atomic operation with respect to multiple procs. The `atomic_flag` is replaced by the per-proc datum. As before, the `switch` signal handler will only allow a context switch to occur when the flag is not set.

For simplicity, the implementation shown here handles proc acquisition and release naively; it holds procs only as long as they're immediately useful rather than stockpiling them as suggested in Section 4.1. The `fork` function, when given a function to fork, attempts to

```
functor Safe_Queue (structure MP : MP
                    structure Queue : QUEUE) : QUEUE =
struct
    type 'a queue = ('a Queue.queue * MP.spin_lock)
    fun create () = (Queue.create (), MP.spin_lock ())
    fun with_lock l f =
        (fn x => (MP.lock l;
                  let val r = f x
                  in
                      MP.unlock l;
                      r
                  end) handle exn => (MP.unlock l;
                                      raise exn))

    fun enq (q,l) = with_lock l (Queue.enq q)
    exception Deq = Queue.Deq
    fun deq (q,l) = (with_lock l Queue.deq) q
end
```

Figure 6: Safe Queues

acquire a proc to run the parent's continuation. If the call to `acquire_proc` fails, the parent is rescheduled onto the `ready` queue and the child function is evaluated. When the child completes, a continuation is atomically dequeued from `ready` using `getnext` and invoked. If the exception `Deq` is raised, there are no more threads to run, so the calling proc releases itself.

Figure 8 shows how the system may be linked, given the **MP** functor and a structure `Queue`.

5 MP Generic Implementation Details

The implementation of the **MP** interface is divided into a generic system-independent layer, and a system-dependent layer. The generic layer makes up the bulk of an implementation, allowing the interface to be ported easily.

5.1 Interfacing ML code to the Outside World

In order to track the changes that are being made to the SML/NJ system's development, we tried to cause as little disturbance as possible while adding **MP** support. Internally, SML/NJ supports a range of mechanisms for access to the underlying hardware and operating system [6, 7]. Each mechanism involves a different tradeoff between execution overhead and portability.

Primops: The compiler generates generic machine code, which is then translated into machine-specific instruction sequences. The generic machine model includes general-purpose

```
functor MP_Thread(structure MP    : MP
                  structure SafeQ : QUEUE
                  sharing type MP.proc_datum = bool) =
struct
    structure SafeQ = SafeQ
    val ready : (unit cont) SafeQ.queue = SafeQ.create ()
    fun atomic () = MP.get_datum ()
    fun enter_atomic () = MP.set_datum true
    fun leave_atomic () = MP.set_datum false
    fun atomically f =
        (fn x => (enter_atomic ();
                  let val r = f x
                  in
                      leave_atomic ();
                      r
                  end) handle exn => (leave_atomic ();
                                      raise exn))
    val reschedule = atomically (SafeQ.enq ready)
    fun getnext () = (atomically SafeQ.deq) ready
                         handle SafeQ.Deq => MP.release_proc ()

    fun fork child =
        (callcc (fn parent =>
                   (MP.acquire_proc (MP.PS(parent,true))
                       handle MP.Acquire_Proc =>
                                  reschedule parent;
                    child ();
                    throw (getnext ()) ())))

    fun switch (_,k : unit cont) =
         if atomic () then k
         else (reschedule k;
               getnext ())
    local
        open System.Signals System.Timer
            System.Unsafe.CInterface
        val t = TIME {sec = 0, usec=50000}
    in
        val _ = (setHandler(SIGALRM,SOME switch);
                 setitimer(0,t,t))
    end
end
```

Figure 7: **MP** Threads

```
structure MP_Arg : MP_ARG =
struct
  type proc_datum = bool
  val initial_datum = false
end

structure MP = MP(MP_Arg)

structure SafeQ = Safe_Queue (structure MP = MP
                              structure Queue = Queue)

structure Thread : THREAD = MP_Thread(structure MP = MP
                                      structure SafeQ = SafeQ)
```

Figure 8: Linking the System

registers and transfer operations, and also a set of primitive operators (*primops*) for performing specific arithmetic and logical operations and other specialized tasks, such as `callcc`. Typically, each primop translates into a different sequence of native machine instructions for each supported architecture. It is relatively easy to add new primops, provided a suitable implementation is available for each architecture, and depends only on the architecture (and not, *e.g.*, the operating system).

Assembly functions: The code sequences for some common operations, such as string allocation and certain arithmetic functions, are too lengthy to generate in line. Instead, they are provided as assembly-level functions that can be invoked directly from generated ML code. Invoking such a function is similar in cost to calling an ML library function; in particular, a return closure for the function call must be constructed. Assembly functions can have variant implementations for different architectures and different operating systems.

C functions: The runtime system for SML/NJ is written in C and provides a coroutine interface to ML code. When ML requires a runtime service, such as garbage collection, it sets a global variable `request` to a value indicating which service is desired, saves its register set in a global state vector, loads the C registers from the C stack and begins to execute the C code that will provide the service. When the runtime service is complete, the C registers are saved on the C stack, the ML registers re-loaded, and execution of ML code continues. Two assembly language routines, `saveregs` and `restoreregs`, handle the machine-specific task of crossing this ML/C boundary.

The most important runtime services are garbage collection (see Section 5.5) and signal processing, which have their own `request` values. Other services, such as I/O, are provided via named C functions. New functions can be added easily at linktime; in fact, function names are not bound to code addresses until runtime.

5.2 Proc Implementation

Each proc is allowed to execute both ML and C code. The obvious alternative—restricting C execution to a single server thread—would introduce unnecessary synchronization. Each proc is given an `MLState` vector that holds per-proc copies of runtime flags and values. While executing ML code, a pointer to the state vector sits on top of the C-stack. When crossing from ML to C, the pointer is popped off the stack and passed as an argument to `saveregs`. This routine passes the state pointer to the runtime service routine as well. When the service is complete, the pointer is passed to `restoreregs`, which places it on top of the stack.[3]

In a C signal handler, it is difficult to recover the the state pointer from the stack, since a new frame is pushed on the stack. Thus, each state vector contains the kernel thread id of the proc that owns it. While executing a C signal handler, the proc calls the system-dependent routine `ml_getpid` to determine its identity. It then walks through the list of state vectors until it finds the one with its id. While slow, this method is portable.

Implementing `acquire_proc` and `release_proc` requires access to the operating system, so these routines are implemented as C functions. Since the ML runtime system must allocate data structures for each process, a compile-time constant determines the maximum number of procs that the runtime system can provide. When a client calls `acquire_proc`, the runtime system first attempts to find an available `MLState` vector and initializes it appropriately. If the blocked pool, described below, is empty, the runtime system invokes a system-dependent routine, `new_proc`, which in turns calls the operating system, passing the continuation to be executed and the state vector as arguments. The operating system creates a new kernel thread and starts it executing the continuation.

When a proc calls `release_proc`, the runtime system puts the current kernel thread to sleep using the system-dependent routine `block`, making the current physical processor available for other system users. The blocked kernel thread is placed in a pool for subsequent reuse. If the pool is non-empty when `acquire_proc` is called, the runtime system will remove a sleeping thread from the pool and wake it, using the system-dependent routine `unblock`, instead of asking the operating system for a new kernel thread. Under some operating systems, if the runtime system acquired and released kernel threads whenever `acquire_proc` and `release_proc` were called, it would rapidly run out of available thread (process) ids.

5.3 Per-Proc Data Implementation

Since it is very important to access the `proc_datum` as efficiently as possible, the SML/NJ generic machine model was modified to include a dedicated virtual register to hold the `proc_datum`. Two primops corresponding to `get_datum` and `set_datum` were added to read and write the register. On RISC machines that have 32 or more registers, like the MIPS-based SGI, dedicating a register for per-proc data does not affect performance [7, page 189], so the virtual register is implemented by an actual register. On architectures with fewer registers (*e.g.*, VAX), the datum is stored on the stack and accessed indirectly through the stack pointer.

[3]The idea of passing the state pointer around in the runtime system is due to Andrew Appel. An OS-dependent solution to this problem was used in the SML/Mach runtime system [14].

5.4 Spin Lock Implementation

Each spin lock is represented by one or more memory locations. On some machines, any location can be used; exclusive access is guaranteed by using specially interlocked exchange or test-and-set instructions. In this case, spin locks can be allocated in the heap, and garbage collected in the usual way. Other machines provide a special bank of memory for the locks, and only a fixed maximum number of locks are available. In this case, we do not bother garbage collecting the lock memory, since the **MP** client must already cope with the possibility of a restricted number of locks; if maximizing the number of available locks proves desirable, modifying the collector to reclaim unreachable locks should not be too difficult.

The spin lock operations `try_lock` and `unlock` should be as fast as possible. This suggests adding new primops to the compiler. However, while primop implementation may depend on the target architecture, SML/NJ currently has no provision for generating code based on the target operating system. Therefore, we implement these operations using assembly language routines.

5.5 Storage Management

SML/NJ performs heap allocation very frequently (approximately one word per every 3-7 instructions [7, page 196]). Allocation is in-lined by the compiler, making it quite fast. An allocation consists of incrementing a top-of-heap pointer and writing the values. A check for heap overflow is made at the entrance of each *code tree*;[4] if there is insufficient free space to contain the maximum allocation that the code tree might perform, a trap instruction is used to enter the C runtime system and initiate garbage collection. SML/NJ uses a two-generation, copying garbage collector [5].

In adapting this system to a multiprocessor, it is important to avoid proc synchronization during allocation; this requirement precludes allocating into a global heap. Instead, the entire allocation region is divided into per-proc regions at startup; each proc allocates into its own region. All regions can be read or updated by the other procs, however. When the allocation region is filled and a garbage collection (GC) is required, the procs are synchronized, the collection is performed, and the allocation region is redivided. Procs in the blocked pool are given a small, fixed-size region. The rest of the allocation area is divided evenly among the active procs.

In practice, procs, and hence processors, tend to allocate at different rates. When redividing the allocation region after a GC, we could try to give each proc a region roughly proportional to its allocation rate. However, the previous allocation rate of a proc can be a poor predictor of the future allocation rate, especially if procs frequently switch context. Since garbage collections could be a serious performance bottleneck, we want to make sure that all of the available space in the allocation area is used before initiating a GC.

To address this problem, we further divide each proc's region into *chunks*. Initially each proc is given the first chunk in its region as its allocation area. When a proc's current chunk does not have enough space left, the proc attempts to extend its current chunk by acquiring the next contiguous chunk in its region. This prevents the proc from wasting space that *is* left in the current chunk. If, however, there are no more chunks in the proc's region, the

[4] A *code tree* is an acyclic set of blocks with one entry point and one or more exits.

proc attempts to find and "steal" some free chunks at the *end* of another proc's region. If no more chunks are available, then the proc initiates a garbage collection.

Setting the chunk size appropriately involves some tradeoffs. Setting it too large could keep a fast allocator from finding available chunks to steal, causing garbage collection to be initiated more frequently. Setting the chunk size too small forces the procs to trap and look for chunks more often.

Our current implementation does sequential garbage collection. When one proc finds that no more chunks are available, it sends a signal (`SIGUSR1`) to the other procs, using the system-dependent routine `signal_proc`. These procs note the signal, set their heap-limit pointer to a special value, and continue what they were doing. When they enter the next code tree, and are thus in a clean state, the special value in their heap-limit pointer will cause them to fault and enter the GC routine. When all the procs have checked in, one of them collects all of the roots and does the actual garbage collection while the others block (using the `block` routine). When the collection is done, the new allocation area is divided among the procs and the blocked procs are resumed (using the `unblock` routine).

The synchronization code for garbage collection is by far the most complex change we made to the SML/NJ system. Fortunately, the bulk of this code is implemented at the generic layer.

6 The SGI System-Dependent Layer

This section describes the machine dependent layer of the **MP** implementation for the Silicon Graphics 4D/380S multiprocessor running version 3.3 of the IRIX System V operating system. This machine uses one to eight 40MHz MIPS R3000 processors, and supports up to 256 MB of main memory. Each processor has a 64KB direct-mapped primary instruction cache, a 64KB direct-mapped write-through primary data cache, and a 1MB secondary snoopy data cache. Test-and-set locks are provided through special hardware, described in more detail below.

6.1 Procs and Processes

IRIX does not provide kernel threads *per se*, but does provide the ability for processes (single-threaded tasks) to share the same address space. Processes that share an address space form a *process group*. We map each **MP** proc to a distinct IRIX process. The *sproc(2)* system call is used to implement `new_proc`. The *blockproc(2)* and *unblockproc(2)* system calls are used to implement `block` and `unblock`, respectively.

One advantage of using processes (as opposed to kernel threads) as procs is that the interface to IRIX signals is uniform; thus, no SML/NJ runtime changes were needed to support **MP** signals at the system-dependent layer for the SGI.

6.2 Spin Locks

The MIPS R3000 does not have a test-and-set instruction. However, the SGI provides a limited number of hardware locks, which are implemented by a separate lock memory and bus, which is mapped to a special page of memory called a synchronization arena. Reading from a location in the arena performs a test-and-set operation and returns the result of the test. Writing an arbitrary value to the location clears the lock. A system call (*usinit(3)*) is needed to place the page in the process group's address space.

At most 4096 locks may be allocated from a synchronization arena. The runtime system reserves a small number of locks for its own use and exports the rest to **MP**. The runtime system keeps an array of pointers to the exportable locks. When a `spin_lock` call is made, the runtime system returns one such pointer as the result of the operation. This extra indirection is needed to support stand-alone or exported images of programs, since the operating system does not gurantee to place the synchronization arena in the same virtual address range for each invocation of a program.

It is possible to "trick" the SML/NJ compiler to generate the proper sequence of instructions for the `try_lock` and `unlock` operations, since they are just reads and writes:

```
fun try_lock (x : spin_lock) =
      System.Unsafe.boxed(!(!(System.Unsafe.cast x)))

fun unlock (x : spin_lock) = (!(System.Unsafe.cast x)) := 0
```

The `boxed` predicate returns true if its argument is 0 (considered as a machine word). This approach allows us to gain the benefits of inlining without having to add a new primop, at the expense of introducing machine dependencies at the ML level.

6.3 Performance

In this section, we present preliminary measurements for the SGI implementation of **MP** and a simple thread package built on top of it. A proper performance analysis of **MP** should compare it with other, perhaps less portable, mechanisms for implementing threads on multiprocessors.

Figure 9 gives times (in μsec) for selected **MP** and thread primitive operations. Thread tests used an ML Threads package implemented on top of **MP**. The package's implementation is similar to the one described in Figures 7 and 12; mutex locks are enhanced to record the id of the owning thread and check that the unlock is performed by the owner, but association of mutexes to references remains informal. Spin locks are implemented using the in-line ML code shown in Section 6.2. Thread package code and the ML portion of **MP** code are placed in the same compilation unit as the test code to encourage in-lining. Tests used 100 MB of main memory; all times include garbage collection. Figures presented are the averages over thousands of trials; loops were unrolled to minimize overhead.

Figure 10 gives running times (in seconds) for five applications running on an 8 processor SGI using 100 Mbytes of main memory. The same thread package was used as in Figure 9. Each reported figure is the average of at least three trials.

The benchmarks were:

- `allpairs`: Floyd's algorithm for computing all shortest paths between two nodes of a 100 node graph [34].
- `mst`: Computes the minimum spanning tree on 500 randomly distributed points using Prim's algorithm [34].

Primitive operation	Time (μsec)
Get and set `proc_datum`.	0.07
Lock and unlock spin lock.	5.7
Unsuccessfully try to lock spin lock.	1.12
Lock and unlock mutex.	40.7
Unsuccessfully try to lock mutex.	10.6
Fork null thread and wait for it to terminate.	287
Synchronize with another thread ("pingpong").	232
Voluntarily yield control to another thread.	32.5
Acquire and release proc (using 8 processors).	80.2

Figure 9: Primitive Timings

	Processors Used							
Prog	1	2	3	4	5	6	7	8
`allpairs`	22.38	12.60	10.25	9.53	9.51	9.57	9.69	9.71
`mst`	47.30	30.25	23.11	22.79	22.12	21.18	21.39	22.80
`abisort`	20.53	12.06	9.44	8.44	8.39	8.17	8.35	8.69
`simple`	58.34	37.12	32.61	32.17	32.91	33.40	33.65	34.24
`matrix`	26.64	13.51	8.96	6.76	5.40	4.50	3.89	3.40

Figure 10: Thread Benchmarks

- `abisort`: An adaptive bitonic sorting algorithm [12] of 2^{13} integers [34].
- `simple`: The Simple hydrodynamics benchmark [15], which solves a set of differential equations across a grid of size 100×100, run for one time step.
- `matrix`: Matrix multiply of two 250×250 integer matrices.

Except for `matrix`, these applications show very poor relative speedup, being unable to make productive use of more than 3 processors. We conjecture that this poor result is due almost entirely to main-memory bus contention. Independent measurements indicate that the bus has a maximum achievable bandwidth of about 30 MB/sec; most of these benchmarks (with the notable exception of `matrix`) generate 15–20 MB/sec of bus traffic in allocation alone! We suspect that a large performance improvement can be made only by moving to a different memory allocation strategy (see Section 8).

7 Related Work

There has been a lot of interest in adding threads to ML, but we are only aware of two projects that have addressed running the threads in parallel on multiprocessors. Cooper and Morrisett [14] describe a Mach-based multiprocessor thread implementation for SML/NJ, which was the basis for our **MP** work. The SML/Mach interface provides fork, mutex locks, condition variables, and per-thread ref cells (`vars`). At the time SML/Mach was developed, the VAX-6300 was the only multiprocessor running Mach for which an SML/NJ code generator existed. To increase the availability of the ML Threads package, we decided to rethink the interface to acheive portability across operating systems and machines. Another shortcoming of the SML/Mach interface is that it does not provide a mechanism for preemptively scheduling user-level continuation threads. Also, the interface is designed to support only one thread package (ML Threads) directly and we wanted to support other thread packages, notably CML.

Matthews [31] describes several implementations of a concurrent extension of Poly/ML using CSP-style communication operators [11]. The concurrency primitives are implemented as hardwired extensions to the Poly/ML runtime system. Matthews describes a uniprocessor implementation using timer preemption, an implementation for the DEC Firefly multiprocessor, and the outline for a distributed workstation implementation. The Firefly implementation is comparable to our **MP** approach; in particular, garbage collection is arranged in very similar fashion. However, support for multiple threads and for communication between them is buried deeper in the runtime system and is not designed for portability among different thread models or hardware platforms.

The Lisp community has a great deal of experience with multiprocessor symbolic computing. Multilisp [21] and Qlisp [18] are two languages that provide concurrency extensions to Lisp and have been implemented for various multiprocessors [20, 28, 32, 43]. Multilisp provides *futures* as its primary concurrency mechanism. Evaluation of the expression `(future X)` forks a thread to evaluate `X`. A capability for `X`'s value is immediately returned to the caller. When the value of `X` is needed, the capability can be *touched*, at which point the caller is delayed until the evaluation of `X` is complete. Qlisp also provides futures along with a variety of mechanisms including `Qlet` and mutex locks. `Qlet` evaluates the right-hand sides of its bindings in parallel.

These versions of Lisp only support one thread model (*e.g.*, futures) and are not extensible. A great deal of work has gone into determining how to schedule the threads and how to bound the number of threads in the system. For instance, some implementations allow the user to select from a fixed set of scheduling policies (FIFO or LIFO). Other implementations turn parallel code into sequential code when the compiler can determine that doing so will improve performance. Our approach provides more flexibility; **MP** clients can tailor a thread package's scheduling policy and mechanisms and its synchronization constructs to meet the needs of particular applications.

Some existing concurrent systems do stress flexibility and portability at the runtime-system level. The primary goals of the Portable Common Runtime system (PCR) are to provide portability and language inter-operability [47]. PCR offers storage management, symbol binding (*i.e.*, dynamic loading), threads, and low-level I/O to a language implementor. The implementation of threads under PCR is similar to ours—user-threads are multiplexed on top of kernel threads. However, PCR does not allow the thread package or its scheduling policy to be customized. Since PCR must work with a variety of languages, it uses a conservative, non-copying garbage collector.

The goals of Jagannathan and Philbin's STING system [26, 27] closely resemble ours. STING is a dialect of Scheme enhanced with primitive concurrency operators to form a substrate for custom design of concurrent symbolic computing environments. The basic data types are threads and virtual processors; the system provides flexibility in scheduling, storage allocation and thread migration policies without compromising efficiency. A major difference from our work is that we rely on SML/NJ's first-class continuations to represent threads; STING's thread data type is less elegant but more carefully crafted for performance. Also, STING's design does not emphasize portability among different hardware and operating system platforms.

8 Discussion and Future Work

There is much work to be done regarding the **MP** interface. Most importantly, to determine the adequacy of the interface, we need to build additional complete thread packages, such as CML, on top of it. In the following paragraphs, we briefly discuss other issues and open problems.

Smarter memory management: Bus contention resulting from poor locality of reference is a major problem with the current implementations. We suspect that the major cause of this problem is SML/NJ's allocation strategy. Recall that in SML/NJ all procedure-frames are allocated on the heap. This has the advantage that `callcc` and `throw` become relatively cheap. However, it has the disadvantage that space is not re-used for allocation until a garbage collection occurs. If the size of the *from* and *to* spaces of the copying collector exceed the size of a processor's cache, this strategy insures a cache-miss on almost every allocation.

There are two basic approaches to alleviating this problem. One approach is to try to size the *from* and *to* spaces (or, more generally, the allocation space and youngest generation of a multi-generation collector) for each processor so that they fit within the processor's cache. This approach has the advantage of leaving the fundamental SML/NJ heap allocation strategy unchanged. However, it is unattractive on machines with small caches, such as the Omron, which has a 16KB cache. The other approach is to use a stack to allocate procedure frames that do not "escape." Stack allocation of frames would allow memory to be re-used, thus improving locality. To keep the cost of `callcc` and `throw` relatively cheap, we could use techniques described by Hieb, *et al.* [24].

Concurrent GC: As mentioned in Section 5.5, concurrent garbage collection is an important goal. There have been many proposals for concurrent and/or parallel garbage collection [8, 10, 23, 29, 36]. Parallel GC on a shared-bus machine might be unattractive since GC is a memory intensive operation. We expect that a bus would rapidly become saturated as more processors "helped" with the collection. Collecting garbage while the computation proceeds seems more appealing, but most of the algorithms proposed have drawbacks. Some require non-standard hardware support [23], while others require more efficient traps and more flexible virtual memory management than today's operating systems provide [8]. As well, most of the algorithms (with the notable exception of the one described by Herlihy and Moss [23]) rely on some global synchronization of the processes. If, as in the SML/NJ system, processes cannot be stopped at arbitrary points to initiate a GC, the cost of the synchronization could overwhelm the benefits of a concurrent GC.

Smarter memory model: At present, the SML/NJ compiler takes a very conservative approach to optimizing accesses to references and arrays. It performs all operations specified in the source code even when simple dataflow analysis would show some of them to be superfluous. This conservatism has been helpful in developing a multiprocessor interface, since the compiler makes no assumptions that might be violated in a concurrent setting. It might be worthwhile, however, to make the compiler smarter about performing such optimizations, which could be particularly valuable for decreasing the cost of runtime safety checks on shared accesses.

Such optimizations require making a more explicit connection between shared memory locations and the mutual exclusion locks that govern them, and exposing that connection to the compiler. As more complex cache coherence protocols come into common use, the compiler will be required to take on this knowledge in any case, in order to produce correct code even in a conservative fashion.

Smarter I/O: Our current I/O implementation is portable, but not wholly satisfactory. When an **MP** client thread blocks, we would ideally like the current proc (and physical processor) to be made available to run other threads; under most operating systems, however, **MP** loses control of the processor until the I/O completes or is aborted by a timer signal.

One option would be to use extra kernel threads to do the blocking I/O operations. Unfortunately, "handing off" the I/O operation is problematic. The extra kernel threads can either busy-wait for an I/O operation or block. If they busy-wait, they waste cycles when I/O operations are infrequent. If they block, then an extra kernel call is required to wake them.

Scheduler activations [2] are an elegant solution to this problem. In the scheduler activation model, each task is given a virtual multiprocessor. However, the number of processors assigned to a task at any point in time is controlled by the kernel. There is a one-to-one mapping of a task's non-blocked kernel threads (scheduler activations) to its processors. When a scheduler activation blocks for I/O, the kernel constructs a new activation to run on the processor. The new activation makes an upcall to the application informing it that the original activation blocked. The new activation is free to continue executing other user-level threads. When the I/O completes, the kernel makes another upcall to inform the application.

We are currently investigating different approaches to integrating user-level threads and I/O under the Mach 3.0 operating system. We are also investigating a scheduler activation-based multiprocessor interface for SML [35].

Semantics: A formal specification of the SML/**MP** system would be valuable. Without a specification, one of the most important advantages of using SML as a base language is lost. Recently, some progress has been made in the exploration of a semantics for a multiprocess SML [11, 40]. We hope to be able to use these results to give a somewhat more formal semantics for the **MP** interface.

9 Acknowledgments

Most of the **MP** interface was developed while the authors were visiting or consulting for AT&T. We would like to thank Suresh Jagannathan and Jim Philbin of the NEC Research Institute, Princeton, NJ, for graciously providing access to their SGI machine for benchmarking and profiling. We would also like to thank David MacQueen, Eric Grosse, and Tom Szymanski for their help and support with the SGI at AT&T. Andrew Appel, David MacQueen, and David Tarditi all helped with their knowledge of SML and the New Jersey system. Andrew Appel also helped with many low-level implementation details. Jeannette Wing, Andrzej Filinski, Scott Nettles and John Reppy offered many insightful comments on earlier versions of this report. Several of our benchmarks were ported from Scheme programs written by Eric Mohr. Lal George translated Simple from Id into ML and later parallelized it. Finally, a special thanks to Lorenz Huelsbergen for doing the Sequent port.

A Implementing Mutex Locks in SML

In the ML Threads paradigm [14], a mutex lock is informally associated with each piece of shared, mutable data (*e.g.*, a `ref` cell.) When a thread wants to perform an operation on some set of data, it should first *acquire* the associated mutex locks. After it is through with the operation, it *releases* the mutexes. We say that a thread "holds" a mutex if it has acquired but not released the mutex. Only one thread is allowed to hold a given mutex at any point in time.

Mutex locks are typically implemented by blocking a thread that attempts to acquire a mutex that is already held. On a uniprocessor, this approach is essential since some other thread must run in order for the mutex to become available. On a multiprocessor, it makes better use of processor resources than busy-waiting.

```
signature MUTEX =
  sig
    type mutex
    val mutex : unit -> mutex
    val acquire : mutex -> unit
    val release : mutex -> unit
  end

signature PIPE =
  sig
    type '1a pipe
    val pipe : unit -> '1a pipe
    val put : '1a pipe -> '1a -> unit
    val get : '1a pipe -> '1a
  end

signature THREAD_INTERNALS =
  sig
    structure SafeQ : QUEUE
    val ready : (unit cont) SafeQ.queue
    val enter_atomic : unit -> unit
    val leave_atomic : unit -> unit
  end
```

Figure 11: Signatures for Mutex Locks and Pipes

A signature for Mutex locks is shown in Figure 11. Mutex locks can be added to the **MP** thread package of Section 4.6 by using the functor `Par_Mutex` shown in Figure 12. The functor has access to "internal" components of the `MP_Thread` functor of Figure 7 via parameter `T`; the `THREAD_INTERNALS` signature is shown in Figure 11. Each mutex consists of a flag (`held`) that indicates whether the mutex is owned by some thread, and a queue (`waiters`) of waiting threads. The `acquire` operation examines the `held` flag. If the mutex is not held, then the flag is simply set to `true`. If the mutex is held, the current thread's continuation is enqueued on `waiters` and a new continuation is dequeued from `ready` and invoked. If `ready` is empty, exception `Deadlock` is raised with the system in a dirty state. The `release` operation attempts to hand off ownership of the mutex to some other thread. It does so by dequeuing a waiting thread and rescheduling it. If no threads are waiting for the mutex, the `held` flag is cleared.

To prevent a mutex data structure from entering an inconsistent state, preemption is prevented during `acquire` and `release` operations by wrapping them with `enter_atomic` and `leave_atomic` calls (see Figure 3). To make mutex locks work in a multiprocessor environment, the mutex data structure must also be protected against simultaneous update by two different processors, using an **MP** `spin_lock` (see Section 4.6). A single global `spin_lock` could be used to protect all mutexes, but it is more efficient to introduce per-mutex locks, as shown in Figure 12. We simply add a call to lock the `spin_lock` before doing an operation and a call to unlock it after the operation is complete.

```
functor Par_Mutex (structure MP : MP
                   structure UnsafeQ : QUEUE
                   structure T : THREAD_INTERNALS) : MUTEX =
struct
  exception Deadlock
  datatype mutex = MUTEX of {lock : MP.spin_lock,
                             held : bool ref,
                             waiters : unit cont UnsafeQ.queue}

  fun mutex () = MUTEX {lock = MP.spin_lock (),
                        held = ref false,
                        waiters = UnsafeQ.create ()}

  fun acquire (MUTEX{lock,held,waiters}) =
      (T.enter_atomic ();
       MP.lock lock;
       if (!held) then
           callcc (fn k => (UnsafeQ.enq waiters k;
                            let val k' = T.SafeQ.deq T.ready
                                   handle T.SafeQ.Deq => raise Deadlock
                            in
                                MP.unlock lock;
                                T.leave_atomic ();
                                throw k' ()
                            end))
       else
           (held := true;
            MP.unlock lock;
            T.leave_atomic ()))

  fun release (MUTEX{lock,held,waiters}) =
      (T.enter_atomic ();
       MP.lock lock;
       (T.SafeQ.enq T.ready (UnsafeQ.deq waiters))
           handle UnsafeQ.Deq => (held := false);
       MP.unlock lock;
       T.leave_atomic ())
end
```

Figure 12: **MP** Mutex Locks

Figure 14 shows how the system can be linked to provide a `Mutex` structure.

B Implementing Pipes in SML

An alternative to the shared memory paradigm for communication and synchronization is message passing. In this paradigm, threads are allowed to share only non-mutable data and abstract communication channels. The operations provided for channels and their synchronization semantics can vary a great deal [3]; at a minimum, operations for sending values to channels and receiving values from channels are needed. The CML language [39] extends SML by providing threads, channels, and powerful communication operations on channels.

Pipes are a special case of channels characterized by their asymmetric synchronization. A pipe can be thought of (and is usually implemented) as a queue. Senders *put* values into the pipe and never block. Receivers *get* values from the pipe, blocking if necessary until a value is available.

A signature for pipes is shown in Figure 11. Pipes can be added to the **MP** thread package described in Section 4.6 using the functor `Par_Pipe` shown in Figure 13. A `'1a pipe` value consists of a queue of `'1a` values (`values`) and a queue of `'1a`-accepting continuations (`waiters`). The `get` operation attempts to dequeue a value and return it. If no value is available, the thread is blocked on the waiters queue as an `'1a`-accepting continuation. The `put` operation attempts to "hand" the given value to a waiting thread and reschedule the thread. The "handing" of the value is accomplished by the `make_unit` function. `make_unit` takes a `'1a`-accepting continuation, `k`, and a `'1a` value, `x`, and returns a `unit`-accepting continuation which, when invoked, throws `x` to `k`. If `put` discovers that there are no waiters, it enqueues the given value onto the `values` queue.

The pipe's data structures are protected by wrapping pipe operations with `enter_atomic` and `leave_atomic`, and adding a `spin_lock` to the `pipe` datatype with appropriate calls to `lock` and `unlock` it.

Figure 14 shows how the system can be linked to provide a `Pipe` structure.

```
functor Par_Pipe (structure MP : MP
                  structure UnsafeQ : QUEUE
                  structure T : THREAD_INTERNALS) : PIPE =
struct
  exception Deadlock
  datatype '1a pipe = PIPE of {values : '1a UnsafeQ.queue,
                               waiters: ('1a cont) UnsafeQ.queue,
                               lock   : MP.spin_lock}

  fun pipe () = PIPE {values  = UnsafeQ.create (),
                      waiters = UnsafeQ.create (),
                      lock    = MP.spin_lock ()}

  fun make_unit (k : 'a cont) (x : 'a) : unit cont =
      callcc (fn c1 => (callcc (fn c2 => (throw c1 c2));
                        throw k x))

  fun put (PIPE{values,waiters,lock}) x =
      (T.enter_atomic ();
       MP.lock lock;
       (T.SafeQ.enq T.ready (make_unit (UnsafeQ.deq waiters) x))
         handle UnsafeQ.Deq => (UnsafeQ.enq values x);
       MP.unlock lock;
       T.leave_atomic ())

  fun get (PIPE{values,waiters,lock}) =
      (T.enter_atomic ();
       MP.lock lock;
       (let val x = UnsafeQ.deq values
        in
            MP.unlock lock;
            T.leave_atomic ();
            x
        end) handle UnsafeQ.Deq =>
            (callcc (fn k => (UnsafeQ.enq waiters k;
                              let val c = T.SafeQ.deq T.ready
                                    handle T.SafeQ.Deq => raise Deadlock
                              in
                                  MP.unlock lock;
                                  T.leave_atomic ();
                                  throw c ()
                              end))))
end
```

Figure 13: **MP** Pipes

```
structure Thread0 = MP_Thread (structure MP = MP
                               structure SafeQ = SafeQ)
structure Thread : THREAD = Thread0
structure ThreadInternals : THREAD_INTERNALS = Thread0

structure Mutex = Par_Mutex(structure MP = MP
                            structure UnsafeQ = Queue
                            structure T = ThreadInternals)

structure Pipe = Par_Pipe(structure MP = MP
                          structure UnsafeQ = Queue
                          structure T = ThreadInternals)
```

Figure 14: Linking Mutexes and Pipes

References

[1] T. Anderson. The performance of spin lock alternatives for shared memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proc. 13th ACM Symposium on Operating Systems Principles*, pages 95–109, Oct. 1991.

[3] G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, Mar. 1983.

[4] A. W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[5] A. W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice & Experience*, 19(2):171–183, Feb. 1989.

[6] A. W. Appel. A runtime system. *Journal of Lisp and Symbolic Computation*, 3(4):343–380, Nov. 1990.

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

[8] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988. Published as *SIGPLAN Notices*, 23(7), July 1988.

[9] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In M. Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, Aug. 1991. Springer-Verlag.

[10] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, Apr. 1978.

[11] D. Berry, R. Milner, and D. Turner. A semantics for ML concurrency primitives. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, pages 119–129, Jan. 1992.

[12] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal of Computing*, 18(2):216–228, Apr. 1989.

[13] E. Cooper, R. Harper, and P. Lee. The Fox project: Advanced development of systems software. Technical Report CMU-CS-91-178, School of Computer Science, Carnegie Mellon University, Aug. 1991.

[14] E. C. Cooper and J. G. Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, Dec. 1990.

[15] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE code. Technical Report UCID 17715, Lawrence Livermore Laboratory, Livermore, CA, Feb. 1978.

[16] T. W. Doeppner Jr. Threads: A system for the support of concurrent programming. Technical Report CS-87-11, Department of Computer Science, Brown University, June 1987.

[17] B. F. Duba, R. W. Harper, and D. B. MacQueen. Typing first-class continuations in ML. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Jan. 1991.

[18] R. Gabriel and J. McCarthy. Qlisp. In J. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, pages 63–89. Kluwer Academic Publishers, Boston, 1988.

[19] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, Apr. 1989.

[20] R. Goldman and R. P. Gabriel. Preliminary results with the initial implementation of Qlisp. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 143–152, July 1988.

[21] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Prog. Lang. Syst.*, 7(4):501–538, Oct. 1985.

[22] R. H. Halstead, Jr. New ideas in parallel lisp: Language design, implementation, and programming tools. In T. Ito and R. H. Halstead, Jr., editors, *Parallel Lisp: Languages and Systems: Proc. US/Japan Workshop on Parallel Lisp, June 1989*, volume 441 of *Lecture Notes in Computer Science*, pages 2–57. Springer, 1990.

[23] M. Herlihy and J. E. B. Moss. Lock–free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, May 1992.

[24] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, July 1990.

[25] L. Huelsbergen and J. Larus. Dynamic program parallelization. In *Proc. of the 1992 ACM Conference on Lisp and Functional Programming*, pages 311–323, June 1992.

[26] S. Jagannathan and J. Philbin. A customizable substrate for concurrent languages. In *Proc. ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 55–67, June 1992. Published as *SIGPLAN Notices*, 27(7), July 1992.

[27] S. Jagannathan and J. Philbin. A foundation for an efficient multi-threaded scheme system. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, pages 345–357, June 1992.

[28] D. A. Kranz, J. Robert H. Halstead, and E. Mohr. Mul-T: A high-performance parallel Lisp. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 81–90, June 1989. Published as *SIGPLAN Notices*, 24(7), July 1989.

[29] H. Kung and S. Song. An efficient parallel garbage collection system and its correctness proof. In *18th Symposium on Foundations of Computer Science*, pages 120–131, Oct. 1977.

[30] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proc. 13th ACM Symposium on Operating Systems Principles*, pages 110–121, Oct. 1991. Published as *Operating Systems Review*, 25(5), Oct. 1991.

[31] D. C. J. Matthews. A distributed concurrent implementation of standard ML. Technical Report ECS-LFCS-91-174, Laboratory for Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh, Aug. 1991.

[32] J. Miller. *MultiScheme: A parallel processing system based on MIT Scheme.* PhD thesis, Massachusetts Institute of Technology, Aug. 1987.

[33] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[34] E. Mohr. *Dynamic Partitioning of Parallel Lisp Programs.* PhD thesis, Yale University, Aug. 1991.

[35] J. G. Morrisett. Running your continuation threads in parallel. In *Proceedings of the Third International Workshop on Standard ML*, Pittsburgh, PA, Sept. 1991.

[36] C. Pixley. An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing*, 3(1):41–49, Dec. 1988.

[37] N. Ramsey. Concurrent programming in ML. Technical Report CS-TR-262-90, Department of Computer Science, Princeton University, Apr. 1990.

[38] J. H. Reppy. Asynchronous signals in Standard ML. Technical Report TR 90-1144, Department of Computer Science, Cornell University, Aug. 1990.

[39] J. H. Reppy. Concurrent programming with events. Department of Computer Science, Cornell University, July 1991.

[40] J. H. Reppy. *Higher-order Concurrency.* PhD thesis, Computer Science Department, Cornell University, Ithaca, NY, Jan. 1992. Also Cornell Univ. Computer Science Dept. Tech. Report 92-1285.

[41] J. H. Reppy and E. R. Gansner. The eXene library manual. Cornell University, Dept. of Computer Science, Ithaca, NY, 1990.

[42] J. Sansilo and M. Squillante. An RPC/LWP system for interconnecting heterogeneous systems. In *Proceedings of the USENIX Tech. Conf., Dallas, TX*, Feb. 1988.

[43] M. R. Swanson, R. R. Kessler, and G. Lindstrom. An implementation of Portable Standard Lisp on the BBN Butterfly. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 132–141, July 1988.

[44] D. Tarditi, A. Acharya, and P. Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[45] A. P. Tolmach and A. W. Appel. Debuggable concurrency extensions for Standard ML. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 120–131, May 1991. Published as *SIGPLAN Notices* 26(12), December 1991. Also Princeton Univ. Dept. of Computer Science Tech. Rep. CS-TR-352-91.

[46] M. Wand. Continuation-based multiprocessing. In *Proceedings of the 1980 LISP Conference*, pages 19–28, Aug. 1980.

[47] M. Weiser, A. Demers, and C. Hauser. The portable common runtime approach to interoperability. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 114–122, Dec. 1989.

[48] J. M. Wing. The Venari project: Goals and plans. Venari Note 1, School of Computer Science, Carnegie Mellon University, 1990, internal note.