Procs and Locks: A Portable Multiprocessing Platform for Standard ML of New Jersey*

J. Gregory Morrisett
Carnegie Mellon University
jgmorris@cs.cmu.edu

Andrew Tolmach
Portland State University
apt@cs.pdx.edu

Abstract

We have built a portable platform for running Standard ML of New Jersey programs on multiprocessors. It can be used to implement user-level thread packages for multiprocessors *within* the ML language with first-class continuations. The platform supports experimentation with different thread scheduling policies and synchronization constructs. It has been used to construct a Modula-3 style thread package and a version of Concurrent ML, and has been ported to three different multiprocessors running variants of Unix. This paper describes the platform's design, implementation, and performance.

1 Introduction

Many concurrent and parallel computations can be expressed elegantly and efficiently using collections of lightweight threads. Both kernel-level and user-level thread packages have become a common part of computing environments. User-level packages can offer substantially better performance than kernel facilities, because thread operations do not require expensive system calls [2, 15, 23].

A user-level thread package can also provide flexibility in choosing scheduling mechanisms and synchronization primitives, especially if the package is implemented within a powerful higher-level language. Wand [35] enumerated three essential mechanisms for language-based multithreaded computation on a uniprocessor: process saving (i.e., the ability to remember the state of a paused thread), elementary exclusion, and data protection. He showed that first-class continuations can be used to implement process saving elegantly and simply. Elementary exclusion is trivial to achieve on a uniprocessor, and the thread package's internal data can be protected if the implementation language provides an abstract data type mechanism. Wand used Scheme [10] as his implementation language.

Standard ML of New Jersey (SML/NJ) is another suitable language system for building continuation-based uniprocessor thread packages [13, 28, 30]. Standard ML [26] is a mostly functional programming language that provides first-class functions (closures), compile-time typing, polymorphism, exceptions, garbage collection, and a powerful module facility. The SML/NJ implementation [6, 7] supports type-safe, first-class continuations [16] and provides asynchronous exception handling facilities in the form of signal handlers [29].

Continuation-based mechanisms can also be used to build language-based *multi*processor thread packages that support true parallelism. To make this possible in ML, we have extended the SML/NJ runtime system to include a multiprocessing platform, called **MP**, that makes parallel hardware resources directly available to ML code. Our primary design goal was to provide a platform that could be easily ported to different architectures and/or operating systems. To this end, we kept the interface as simple as possible. Efficiency and elegance were also important goals.

From the point of view of a thread system, or *client*, **MP** consists of a processor abstraction `Proc` and a mutex lock abstraction `Lock`. Together with first-class continuations, these facilities suffice to implement multiprocessor thread packages in a machine-independent fashion. **MP** has been used to build an enhanced and portable version of ML Threads [13], a Modula-3 style thread package, and this in turn has served as a basis for experimentation with concurrent debugging [33, 34], a transaction system [37], and general systems programming [11]. **MP** has also been used to construct a multiprocessor prototype of Concurrent ML (CML) [30, 31], an ML dialect supporting threads, channels, synchronous communication events (e.g., send, receive), and powerful event combinators (e.g., CSP-style nondeterministic choice).

MP is intended for shared-memory multiprocessors running Unix-like operating systems. Porting **MP** to a new machine involves implementing only a small number of system-dependent routines. **MP** has been implemented for the Mips R3000-based Silicon Graphics (SGI) 4D/380S running Irix, the Motorola 88100-based Omron Luna88k running Mach, and the Intel 80386-based Sequent Symmetry running Dynix. In addition, a trivial uniprocessor implementation works on all processors that run SML/NJ, including VAX, SPARC, and Motorola 68000.

*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 first author was supported in part by a NSF Graduate Fellowship. The work of the second author was conducted in part at Princeton University with support from NSF grant CCR-9002786.

```
signature THREAD =
sig
    val fork : (unit -> unit) -> unit   (* start new thread *)
    val yield : unit -> unit            (* yield processor to another thread *)
    val id : unit -> int                (* return own thread id *)
end

signature QUEUE =
sig
    type 'a queue
    val create : unit -> '1a queue
    val enq : 'a queue -> 'a -> unit  (* enqueue *)
    val deq : 'a queue -> 'a          (* dequeue *)
    exception Empty                   (* raised on dequeue when empty *)
end

functor UniThread (Queue : QUEUE) : THREAD =
struct
    val ready : (unit cont * int) Queue.queue = Queue.create ()
    val current_id = ref 0
    val next_id = ref 1

    fun reschedule (cont,id) = Queue.enq ready (cont,id)
    fun dispatch () = let val (cont,id) = Queue.deq ready
                      in current_id := id;
                          throw cont ()
                      end

    fun fork (child : unit -> unit) =
        callcc (fn parent =>
                (reschedule (parent,!current_id);
                 current_id := !next_id;
                 next_id := !next_id + 1;
                 child ();
                 dispatch()))

    fun yield () =
        callcc(fn cont =>
               (reschedule (cont,!current_id);
                dispatch()))

    fun id () = !current_id
end
```

Figure 1: Implementing Threads in Uniprocessor SML/NJ. Some ML syntax: The keyword `signature` declares a module interface specification; `functor` declares a parameterized module. `A.x` refers to element `x` of module `A`. `unit` denotes the void type; `unit->unit` is the type of a procedure that takes no arguments and returns no results, and is thus executed only for its effects. `ref x` constructs a mutable data object with initial contents `x`; its contents are fetched by the explicit dereferencing operator `!` and changed by the assignment operator `:=`. `callcc(fn c =>` *body*`)` binds the "current continuation" (i.e., the continuation of the `callcc` invocation) to `c` and executes *body*; `throw c ()` invokes continuation `c`. `'a` and `'1a` are type variables; they appear in the types of polymorphic functions.

2 User-Level Threads in SML/NJ

Figure 1 shows the specification and implementation of a simple user-level thread package in uniprocessor SML/NJ. An ML module interface is called a `signature`. A simple module implementation is a `structure`; an implementation parameterized by another interface is a `functor`. Our `THREAD` signature exports three functions: `fork` creates a thread to execute a specified function and assigns it a new integer thread identifier, and sets it running in parallel with its parent; `yield` temporarily yields control of the processor to another thread; and `id` returns the integer identifier of the current thread. A thread inherits the environment of its parent, which may contain both mutable and immutable identifiers. This environment forms a shared memory; any variable accessible from two threads is implicitly shared by them. Shared access to mutable variables (`ref` cells) requires a suitable mutual exclusion protocol (not shown here).

The key aspect of the implementation shown in Figure 1 is the representation of waiting threads by a queue of first-class continuations. Making the queue explicit gives us great flexibility in managing threads. The thread module we present is a functor parameterized by a `QUEUE` structure whose implementation is straightforward and is not shown here. The `QUEUE` signature does not specify the queuing discipline; for example, FIFO and randomized queue implementations will both match the signature. Thus, thread scheduling policy can be changed simply by varying the functor's argument.[1]

A more realistic implementation would use timer alarm signals to preempt compute-bound threads periodically, preventing them from monopolizing the processor. To implement preemption, we can set up an alarm signal handler to invoke `yield` asynchronously. For brevity, preemption has been omitted from the examples in this paper.

This continuation-based implementation is particularly efficient in SML/NJ because creating and invoking continuations is as fast as function invocation. In a system where closures (i.e., procedure-frames) are stack allocated, capturing continuations (via `callcc`) generally requires that the stack be copied, since continuations might be invoked (via `throw`) multiple times. SML/NJ allocates all closures on the heap instead of on a stack and the closures are reclaimed by the garbage collector. Thus, `callcc` simply allocates and initializes a new closure without having to copy anything; the same work is required to call an arbitrary procedure.[2]

On the other hand, allocating closures on the heap may slow down ordinary execution compared to a stack-based implementation. Although Appel has shown that heap allocation can be as fast as stack allocation with respect to instruction counts [3], he ignores memory effects. This allocation scheme is likely to be slower than a stack-based implementation for systems with small first-level caches. Thus, 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 stack-based systems.

The heap-based approach has another significant advantage over systems, such as C threads [12], that require a stack to be allocated explicitly for each thread. Since SML/NJ threads do not use a stack, no stack space needs to be reserved for them. Consequently, we support the use of hundreds or even thousands of continuation-based threads. Since closures are garbage collected as heap objects, threads that are unreachable can also be garbage collected.

[1] Many useful scheduling policies would require minor changes to the signature; for example, priority queues would need a priority to be passed to the enqueue operation.

[2] Functions whose call sites are all statically known can sometimes be called with less overhead.

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

    val acquire_proc: proc_state -> unit
    exception No_More_Procs
    val release_proc: unit -> 'a

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

signature LOCK =
sig
    type mutex_lock
    val mutex_lock: unit -> mutex_lock
    val try_lock : mutex_lock -> bool
    val lock     : mutex_lock -> unit
    val unlock   : mutex_lock -> unit
end
```

Figure 2: The `PROC` and `LOCK` Interfaces

3 MP Specification

We have extended SML/NJ to support multiprocessing by adding new primitive facilities for processor management and for locking. Figure 2 shows ML signatures for these facilities. A structure `Proc` that implements the signature `PROC` provides operations for managing processors and their state. A structure `Lock` that implements `LOCK` provides mutual exclusion among processors.

3.1 Procs

A *proc* is a language-level view of a kernel thread executing on a physical processor. `Proc` does not define a proc datatype; rather, it contains functions that an executing proc can call to alter itself, or to start a new proc. Initially, a single root proc is executing on a single processor. An existing proc can start a new proc executing in parallel by invoking `acquire_proc` with the continuation to be executed and a client-defined `proc_datum`, described below.

Typically, `Proc` 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 `No_More_Procs`. However, the number of physical processors available to an SML/NJ image can change without warning during a computation, as a result of activity by other users and by the operating system itself. Thus, as is usual with kernel threads, the correspondence between procs and available physical processors is only an approximate one.

Function `release_proc` causes the calling proc to stop executing, and releases the current physical processor to the operating system. If the client wishes to save the execution state of the proc before releasing it, it does so by capturing a continuation using `callcc`. Since `acquire_proc` and `release_proc` require communication with the operating system, clients will wish to invoke them sparingly. To obtain good performance (at the expense of other system users) a client can call `acquire_proc` repeatedly when it starts up, acquiring as many procs as possible, and hold on to them for the duration of the computation.

3.2 Per-Proc Data

Often, a proc needs some small amount of private state. For example, consider the `ref` cell containing the `current_id` of the executing thread in Figure 1. In a multiprocessor setting in which all `ref` cells are shared by various processors, it is not possible to store a unique id in a single `ref`; instead, each processor requires a private copy of `current_id`.

`Proc` 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 initial datum value for the root proc is specified by `initial_datum`.

Specific requirements for private data vary from client to client. For the thread system of Figure 1, a single integer thread identifier suffices; this integer can be stored directly in the `proc_datum` slot. Clients needing to store more complex state information can define `proc_datum` to be a record or array.

3.3 Memory Management and Mutual Exclusion

All heap memory is implicitly shared among all procs; in particular, a proc can freely read or write into heap locations allocated by another proc. To prevent access conflicts on shared mutable variables, **MP** provides *mutex locks* to achieve mutual exclusion. Mutex locks are one-bit shared memory locations that can be atomically tested and set. They are often used as spin locks. More elaborate synchronization constructs such as reader/writer locks, semaphores, channels, etc., can be synthesized from mutex locks, `refs`, and first-class continuations.

Function `mutex_lock` returns a fresh lock in unlocked state. `try_lock` attempts to lock the specified mutex lock and immediately returns a `bool` indicating whether the operation was successful. `unlock` releases a mutex lock; it may be called by any proc (not necessarily the one that set the lock). `lock` is equivalent to the following function:

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

It is included in the interface because some operating systems may provide a more efficient spin than the one shown above (e.g., by using backoff techniques [1]).

Mutex locks are directly supported in hardware on most current multiprocessors, typically in the form of a test-and-set primitive. Implementations of `Lock` use hardware support where available to make locks as efficient as possible.

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 machines on which it is currently implemented, **MP** does expose enough of the low-level architecture to allow clients to control consistency. However, lack of a common consistency model can be a major problem in designing efficient, portable client code.

3.4 I/O and Signals

In addition to the explicit addition of `Proc` and `Lock` facilities, we must specify the behavior of existing SML/NJ I/O and signal handling facilities on multiprocessors.

The major I/O problem posed by multiprocessing is that two procs may may perform I/O operations simultaneously, possibly accessing the same runtime-system data structures. **MP** takes no specific steps to prevent such conflicts since different clients may have different locking needs. For instance, our CML implementation protects the data structures by a single global lock. Other clients may wish to use finer-grained locking.

We use the existing SML/NJ signal interface [29], adding suitable conventions for multiprocessing. Signal handlers are installed on a global basis, i.e., all procs share the same signal-handling functions, and all procs receive each delivered signal. However, masking and unmasking of signals is controlled on a per-proc basis. There is no **MP** facility for procs to signal, control, or alert one another, primarily because we found it difficult to provide a semantically uniform interface that could be ported easily. Fortunately, these operations may be simulated using timer-driven polling in the target proc.

4 MP Applications

This section illustrates the use of **MP** facilities by describing some simple client packages.

4.1 A Thread Package

Figure 3 shows a multiprocessor implementation of the simple thread signature from Figure 1.

Changes from the uniprocessor version are few and simple. When a thread is forked, the multiprocessor kernel first attempts to allocate a new proc on which to continue running the parent thread; only if this attempt fails is the parent blocked on the `ready` queue. Conversely, a proc calling `dispatch` releases itself if there are no threads available in the `ready` queue. The `ready` queue and `next_id` reference must be protected by mutex locks to avoid race conditions between competing procs. The `current_id` number is stored in the per-proc datum, as suggested above.

4.2 A Selective Communication Facility

A real thread package also requires a mechanism for communication and synchronization between threads. A fairly complicated example of such a mechanism is CSP-style selective communication, described by the `SELECT` signature of Figure 4. It specifies dynamically created, polymorphic channels (`'a chan`), a `send` operation that sends a value to a channel (blocking until the value is received), and a `receive` operation that takes a list of channels and "nondeterministically" chooses to receive a value from one of them.

Figure 5 shows how `SELECT` can be implemented using **MP**, `refs`, and continuations. The protocol is similar to one used in our multiprocessor CML prototype. A channel consists of a queue `sndrs` of sender states and a queue `rcvrs` of receiver states, jointly protected by a mutex lock `ch_lock`. A sender's state consists of its continuation, its id, and the value that it is sending. A receiver's state consists of its continuation, its id, and a mutex lock, `committed`, that serves as a flag to indicate when a sender has been determined.[3]

[3] This mechanism is similar to the one proposed by Ramsey [28].

```
functor MPThread(structure Proc  : PROC
                 structure Lock  : LOCK
                 structure Queue : QUEUE
                 sharing type Proc.proc_datum = int) : THREAD =
struct
    val ready : (unit cont * int) Queue.queue = Queue.create ()
    val ready_lock = Lock.mutex_lock()
    val next_id = ref 1
    val next_id_lock = Lock.mutex_lock()

    fun reschedule (cont,id) = (Lock.lock ready_lock;
                                Queue.enq ready (cont,id);
                                Lock.unlock ready_lock)

    fun dispatch () = (Lock.lock ready_lock;
                       let val (cont,id) = Queue.deq ready
                       in Lock.unlock ready_lock;
                          Proc.set_datum id;
                          throw cont ()
                       end handle Queue.Empty => (Lock.unlock ready_lock;
                                                  Proc.release_proc()))

    fun fork child =
        callcc (fn parent =>
                 (let val current_id = Proc.get_datum()
                  in Proc.acquire_proc (Proc.PS(parent,current_id))
                         handle Proc.No_More_Procs =>
                                 reschedule (parent,current_id)
                  end;
                  Lock.lock next_id_lock;
                  Proc.set_datum (!next_id);
                  next_id := !next_id + 1;
                  Lock.unlock next_id_lock;
                  child ();
                  dispatch()))

    fun yield () =
        callcc (fn cont =>
                 (reschedule (cont,Proc.get_datum());
                  dispatch()))

    fun id () = Proc.get_datum ()
end
```

Figure 3: **MP** Threads. More ML syntax: *expr* `handle` *exn* `=>` *body* installs *body* as the handler for exception *exn* during the execution of *expr*.

```
signature SELECT =
sig
  type 'a chan

  val chan : unit -> '1a chan          (* create a channel *)
  val send : ('a chan * 'a) -> unit    (* send a value to a channel *)
  val receive : '1a chan list -> '1a   (* receive a value from one channel *)
end
```

Figure 4: Selective Communication

```
type 'a sndr = {kont : unit cont, id : Proc.proc_datum, value: 'a}

type 'a rcvr = {kont : 'a cont, id : Proc.proc_datum, committed : Lock.mutex_lock}

type 'a chan = {ch_lock : Lock.mutex_lock,
                sndrs : 'a sndr Queue.queue,
                rcvrs : 'a rcvr Queue.queue}

fun send ({ch_lock, sndrs, rcvrs}, v) =
   (Lock.lock ch_lock;
    let fun loop () =
            let val {kont,id,committed} = Queue.deq rcvrs
            in if (Lock.try_lock committed) then
                   (Lock.unlock ch_lock;
                    reschedule_thread (kont,v,id))
                 else
                   loop()
            end
    in loop()
    end handle Queue.Empty =>
          callcc (fn c =>
                  (Queue.enq sndrs {kont = c, id = Proc.get_datum(), value = v};
                   Lock.unlock ch_lock;
                   dispatch ())))

fun receive (chans: '1a chan list) =
    callcc (fn c  =>
      let val committed = Lock.mutex_lock ()
          val r = {kont = c, id = Proc.get_datum(), committed = committed}
          fun loop [] = dispatch ()
            | loop ({ch_lock, sndrs, rcvrs}::rest) =
               (Lock.lock ch_lock;
                let val {kont,id,value} = Queue.deq sndrs
                in if Lock.try_lock committed then
                       (Lock.unlock ch_lock;
                        reschedule (kont,id);
                        value)
                     else
                       (Lock.unlock ch_lock;
                        dispatch ())
                end handle Queue.Empty =>
                       (Queue.enq rcvrs r;
                        Lock.unlock ch_lock;
                        loop rest))
      in
          loop (randomize chans)
      end)
```

Figure 5: Implementing Send and Receive. Function `reschedule_thread`, not shown here, converts the receiver's `'1a cont` and `'1a` value to a `unit cont` that can be rescheduled as in Figure 3.

When a thread wants to send a value to a channel, it first checks to see if a receiver is available for that channel. If so, the sender must check that no other sender has already delivered a value to that receiver; it does so by trying to lock the receiver's `committed` lock. If the sender is successful in obtaining the lock, it effects the communication by rescheduling the receiver's continuation and id along with the value. If the sender fails to lock the receiver's `committed` lock, it looks for another receiver; if there are no more, the sender enqueues its current state and the value it is sending onto the channel's `sndrs` queue, and passes control of the proc to another thread by calling `dispatch`.

When a thread wants to receive a value from a list of channels, it creates a single receiver state, including a `committed` lock. It then loops through the channels in pseudo-random order, trying to find a blocked sender. If it finds a sender, the receiver attempts to lock its own `committed` lock; if successful, it effects the communication by reading the sender's value and rescheduling the sender for execution. If the receiver cannot acquire its own `committed` lock, some sender must have already effected the communication and re-scheduled the receiver's continuation, so this invocation of the receiver abandons its computation and passes control of the proc to another thread by calling `dispatch`. If the receiver cannot find a sender on the channel, it inserts its state into the channel's `rcvr` queue, and tries another channel on its list; when all channels have been processed, the receiver calls `dispatch`.

5 MP Implementation

The implementation of the **MP** platform is divided into a generic system-independent layer, and a system-dependent layer. The generic layer makes up the bulk of the implementation, allowing the platform to be ported easily. In adding **MP** support, we attempted to minimize changes to the existing SML/NJ implementation. Internally, SML/NJ supports a range of mechanisms for access to the underlying hardware and operating system [5, 6]. The compiler generates generic machine code, which is then translated into machine-specific instruction sequences. The generic machine model includes general-purpose registers and transfer operations, and a set of primitive operators (primops) for arithmetic and logic functions and specialized tasks such as `callcc`; primops are translated to in-line machine-specific code. Assembly-level functions are used for common code sequences, such as string allocation and built-in arithmetic functions, that are too lengthy to generate in line. The SML/NJ runtime system is written in C and uses a coroutine interface with machine code generated from ML. ML code obtains runtime services, such as garbage collection or I/O, by passing control to C code. This code executes on the ordinary C stack, accesses C global variables containing runtime flags and values, and can invoke the underlying operating system using standard system calls.

In our **MP** 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 delays. Most of the runtime system's global variables have become per-proc variables; a few remaining globals are shared under protection of (internal) mutex locks.

`acquire_proc` and `release_proc` are implemented as C functions. Since the ML runtime system must statically 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` (and the proc limit has not yet been reached), the runtime system normally obtains a new kernel thread from the operating system and sets it executing the specified continuation. Alternatively, the runtime system may choose to re-use a previously released kernel thread and avoid creating a new one.

The Luna port maps procs onto kernel threads, which are provided directly by Mach. Irix and Dynix do not provide kernel threads *per se*, but do provide the ability for Unix-style processes to share a common address space (though by somewhat different mechanisms). Thus, for the SGI and Sequent, each **MP** proc is mapped onto a distinct process.

To implement the `proc_datum`, we modified the SML/NJ generic machine model to include a new dedicated virtual register. 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 scarcely affects performance [6, page 189], so the virtual register is implemented by an actual register. On architectures with fewer registers, the datum is stored on the stack and accessed indirectly through the stack pointer.

Mutex lock implementation is heavily dependent on underlying hardware support. The Motorola 88100 provides an atomic exchange instruction on any word in memory. Thus, on the Luna, mutex locks are implemented as ML boolean `ref`s and an exchange instruction is used to implement the `try_lock` routine. The Sequent has a similar facility. 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. The runtime system uses these to control an extensible set of software locks implemented as ML `ref` cells. On all machines, the mutex lock operations `try_lock` and `unlock` are implemented as assembly language subroutines.

Our present **MP** implementation uses a modification of SML/NJ's existing two-generation, copying garbage collector [4]. SML/NJ allocates heap storage very frequently (approximately one word per every 3-7 instructions [6, page 196]), and thus depends on allocation being fast. Allocation is performed by in-line code; heap overflow checks are made at "clean" points where the location of all live roots is known. In adapting this system to a multiprocessor, it is important to avoid proc synchronization during allocation. This means that each proc must allocate into a separate part of the total allocation region; all regions can be read or updated by the other procs, however. When one proc fills its share of the allocation region, it can "steal" spare memory from other procs. When the allocation region is completely filled and a garbage collection (GC) is required, the procs are synchronized at clean points, the collection is performed by one of them, and the allocation region is redivided.

6 Evaluation

To judge **MP**'s portability, we counted the number of lines of code (including whitespace and comments) that make up the system-dependent routines of each **MP** implementation. The SGI system-dependent code consists of 144 lines of C code and 15 lines of assembly. The Sequent system-dependent code consists of 267 lines of C code and 10 lines of assembly; 115 of these lines implement restart of exported images. The Luna system-dependent code consists of 630 lines of C code and 34 lines of assembly. However, 214 lines of the Luna code, which implement routines needed

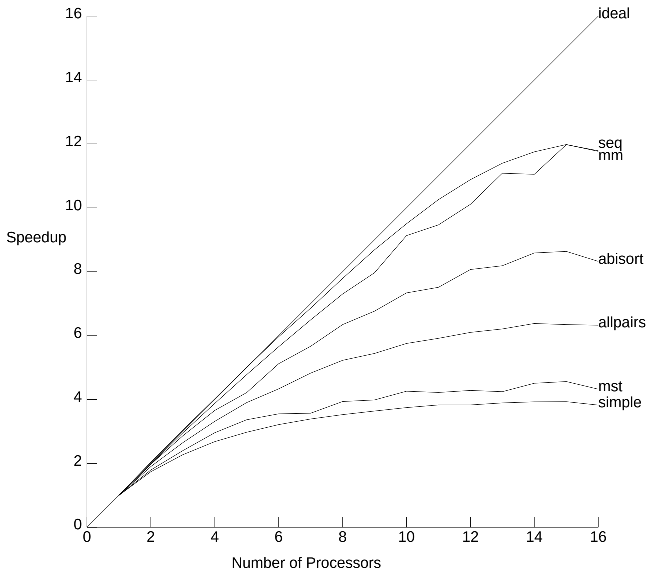

Figure 6: Self-Relative Speedup for ML Threads Benchmarks (Sequent)

to export restartable images, replace a 600 line machine-independent module in the original SML/NJ runtime. An additional 280 lines of the code, which emulate signals for Mach kernel threads, replace a 101 line operating system-independent module. For comparison, the entire SML/NJ runtime system (version 0.75 including **MP** support) for a MIPS machine, consists of approximately 6,750 lines of C code and 650 lines of assembly.

Another indication of portability is the time it takes to complete an implementation of the platform for a new architecture or operating system. After the initial port to the SGI, the Luna implementation was completed within two days by the first author. An initial Sequent implementation was completed within a week by someone familiar with SML/NJ but unfamiliar with **MP**. The current Sequent implementation took longer, primarily because of complications involved in exporting restartable images which could have been avoided by taking the machine-specific approach of the Luna port.

To evaluate the performance of our **MP** implementations, we measured benchmarks running on top of a complete thread package running on top of **MP**. The package was similar to that shown in Figure 3, with the addition of a distributed run queue and a signal-based preemption mechanism. Shared memory locations were protected with user-level mutex locks built on top of `Lock` mutex locks. Tests used a 16-processor Sequent Symmetry S81 with 16MHz Intel 80386 processors, running Dynix V3.1.4. Tests used 100 MB of main memory; all measurements include garbage collection time.

Figure 6 shows the *self-relative* speedup curves for five applications:

- `allpairs`: Floyd's algorithm for computing all shortest paths between two nodes of a 75 node graph [27].
- `mst`: Computes the minimum spanning tree on 200 randomly distributed points using Prim's algorithm [27].
- `abisort`: An adaptive bitonic sorting algorithm [9] run on an input of 2^{12} integers [27].
- `simple`: The Simple hydrodynamics benchmark [14], which solves a set of differential equations across a grid of size 100×100, run for one time step.
- `mm`: Matrix multiply of two 100×100 integer matrices.

One application, `mm`, shows excellent self-relative speedup, which appears to be limited primarily by main-memory bus contention due to SML/NJ's heap allocation. Separate measurements indicate that the bus has a maximum achievable bandwidth of about 25 MB/sec; with 16 processors `mm` generates about 20 MB/sec of bus traffic in allocation alone. Evidence that lock contention and other parallelism issues are not at fault is given by the curve labeled `seq`, which represents the speedup obtained on p processors by running p independent copies of a simple SML/NJ application; `mm` does almost as well.

Speedup for the other benchmarks is limited by other factors, especially our sequential garbage collection strategy; if garbage collection time were omitted, the maximum speedups for `abisort` and `allpairs` would be considerably higher, although the rough shape of their curves would be the same. The applications with the poorest speedups lack enough available parallelism; they leave processors idle much of the time. For example, `simple`, the worst case, has average processor idle rates above 50% for 10 processors or more. `simple` also displays moderate contention for access to the run queues and data locks; none of the other applications showed any significant lock contention. Sequential garbage collection, idle time and lock contention were not significant factors in earlier experiments we conducted on the SGI 4D/380S machine, which has much faster processors but only slightly larger bus bandwidth;[4] on that machine, main-memory contention problems swamped all other effects.

7 Related Work and Conclusions

Although multi-threaded extensions to symbolic computing languages already exist on several multiprocessors, our approach is distinctive because we continue to rely on first-class continuations to represent thread state (a la Wand), and because we stress portability of the multiprocessor platform. The former aspect of our approach is attractive for SML/NJ because continuations play a central role in the existing SML/NJ implementation; the latter aspect is consistent with SML/NJ's commitment to supporting multiple target architectures. Using continuations allows us to describe important aspects of the thread system, such as scheduling policies and synchronization mechanisms, within ML itself rather than burying them in the runtime system. Machine portability allows us to take advantage of a wide variety of hardware, to compare platform performance, and to make the system more readily available for others to use.

Most existing multiprocessor implementations of symbolic languages have been based on Lisp. Multilisp [19] and Qlisp [17] provide concurrency extensions to Lisp and have been implemented for various multiprocessors [18, 22, 25, 32]. These systems have served as the basis for substantial research on problems such as efficient thread scheduling and avoiding excess parallelism. However, they tend to support only one primary synchronization model (e.g., futures

[4] Approximately 30MB/sec. To illustrate the difference in processor speed, locking and unlocking an MP mutex takes only 6μsec on the SGI versus 46μsec on the Sequent.

in Multilisp) and rely on sophisticated runtime system support; they don't attempt to be extensible or portable.

The Lisp-based system most closely resembling ours is STING [20, 21], a concurrent dialect of Scheme specifically intended as a substrate for building higher-level parallel computing environments. STING's basic data types are threads and virtual processors; the system provides flexibility in scheduling, storage allocation and thread migration policies while attempting not to compromise efficiency. The thread data type is simultaneously lower-level and more complex than an SML/NJ continuation; for example, it includes a thread control block with associated stack, heap segment, saved registers, genealogy information, and other state. Threads are therefore expensive to create, and STING includes several optimizations, such as "thread stealing," to avoid creating new threads when possible. In SML/NJ, threads are represented by simple continuations, which cost almost nothing until they begin executing, so there is no need or opportunity for such optimizations. Similarly, **MP** has no processor data type; clients can emulate any of STING's scheduling options by explicitly varying queuing disciplines. By using an explicitly stack-based thread model, STING implementations may be able to offer better performance than our continuation-based approach. However, we believe **MP** presents a simpler view of the multiprocessing environment, which will be easier to port to new hardware and operating system platforms.

Matthews [24] describes several implementations of a concurrent extension to Poly/ML using CSP-style communication operators [8]. He describes several implementations, including one for the DEC Firefly multiprocessor. Support for the thread and communication model is hard-wired into the Poly/ML runtime system, and is not designed for easy porting to different thread models or hardware platforms.

Some existing concurrent systems do stress portability at the runtime-system level. The Portable Common Runtime system (PCR) [36] offers portable and language-independent facilities for threads, storage management, and other runtime-system features. PCR's implementation of threads is similar to ours: user-level threads are multiplexed on top of kernel threads. However, PCR does not allow the thread package or its scheduling policy to be customized.

Although we have stressed simplicity and portability in our **MP** implementation, better performance is also important. One critical task for the future is to improve the present system's poor locality of reference and consequent memory bus contention. One cause of this problem is SML/NJ's heap-based allocation strategy for procedure frames, which makes `callcc` and `throw` cheap relative to ordinary computation but re-uses memory only after garbage collections. When the size of the collector's *from* and *to* spaces exceed the size of a processor's cache, this strategy insures a cache-miss on almost every allocation. Potentially better strategies include using a multi-generational collector with very small young generations that can fit in the cache, or using a stack to allocate "non-escaping" procedure frames. We also need to provide more efficient access to hardware locking facilities, though doing so will unfortunately make the code generator sensitive to the target's operating system as well as its architecture.

Other important areas to address include concurrent garbage collection, compiler-level support for different memory consistency models, and improved I/O interfaces and processor utilization using an operating system based on scheduler activations [2]. Finally, some progress has recently been made in the exploration of a semantics for a multiprocess SML [8, 31]. We hope to to use these results to give a more formal semantics for **MP**, which would reinforce the advantages of using Standard ML as a base language for multiprocessing.

Acknowledgements

Lorenz Huelsbergen made the initial port of **MP** to the Sequent. Suresh Jagannathan and Jim Philbin of the NEC Research Institute, Princeton, NJ, graciously provided access to their SGI machine for benchmarking and profiling. The `allpairs`, `mst`, and `abisort` benchmarks were adapted from Scheme originals written by Eric Mohr; the `simple` benchmark was translated into ML by Lal George.

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

[10] W. Clinger and J. Rees. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-Sep. 1991.

[11] 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.

[12] E. C. Cooper and R. P. Draves. C Threads. Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June 1988.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] D. A. Kranz, R. H. Halstead, Jr., 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.

[23] 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.

[24] 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.

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

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

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

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

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

[30] J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.

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

[32] 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.

[33] A. P. Tolmach. *Debugging Standard ML.* PhD thesis, Princeton University, Oct. 1992. Also Princeton Univ. Dept. of Computer Science Tech. Rep. CS-TR-378-92.

[34] 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.

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

[36] 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.

[37] J. M. Wing, M. Faehndrich, J. G. Morrisett, and S. Nettles. Extensions to Standard ML to support transactions. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications, San Francisco*, pages 104–118, June 1992.