This paper was given at the AIM/ONR Workshop
on Parallel and Distributed Debugging,
Canta Cruz, CA, May 20-21, 1991.

DEBUGGABLE CONCURRENCY EXTENSIONS FOR STANDARD ML

Andrew P. Tolmach
Andrew W. Appel

CS-TR-352-91

October 1991

Debuggable Concurrency Extensions for Standard ML*

Andrew P. Tolmach[†] and Andrew W. Appel[†]
Princeton University

CS-TR-352-91

October 1991

Abstract

We are developing an interactive debugger with reverse execution for the language Standard ML extended to include concurrent threads in the style of Modula-2+. Our debugging approach is based on automatic instrumentation in the source language of the user's source code; this makes the debugger completely independent of the compiler back-end, run-time system, and target hardware. The debugger operates entirely inside the concurrency model and has no special concurrency privileges. In this paper, we consider some of the challenges of debugging a non-deterministic concurrent symbolic language "in itself." Issues considered include logging non-deterministic activity, obtaining more secure semantics for our concurrency primitives, controlling distributed computations, and defining suitable time models. We conclude by suggesting an alternative simulation-based approach to dealing with non-determinism.

1 Debugging Standard ML

Standard ML [22] is a general purpose programming language featuring first-class functions, strong typing with polymorphism, and a powerful exception mechanism. Normal programming style in ML is mostly functional, but the language also supports the creation and manipulation of mutable objects (`ref` cells). Our highly optimizing compiler for the language, Standard ML of New Jersey (SML-NJ) [1], also supports mutable arrays and first-class continuations [7].

ML's compile-time type-checking guarantees that programs have no run-time insecurities ("core is never dumped"). This means that one can always study the run-time behavior of a program—even a buggy one—from *inside* the language, without reference to the underlying machine model (assuming the compiler functions correctly). We have used this fact to build an effective interactive debugger for sequential ML based on automatic *source code instrumentation* [25]. When code is compiled under debugger control, special hooks are inserted at frequent intervals (roughly once per basic block). Later, when the program runs, the debugger can use these hooks to gain control, answer user queries, and support breakpointing. All the added code is itself written in ML. Since the instrumented code is just ordinary ML code from the perspective of the compiler and the run-time system, the resulting debugging system is completely independent of the back-end and target machine, and cannot be damaged by the compiler's aggressive optimization strategies (indeed, it can benefit from them).

An important feature of our debugger is a *reverse execution* facility. Points in the program's execution history are specified by *times*; these are values of a software instruction counter (SIC) that is incremented each time a debugger hook point is encountered (we call each such encounter an *event*). The programmer can use debugger commands to jump back and forth in time, freely interleaving forward execution and replay. Breakpoints may be set either at source locations or at particular time values.

Reverse execution is also used internally by the debugger. In SML-NJ, all data (including stack frames) are allocated on the heap; a datum has no statically known location and is subject to being moved by the garbage collector, so it must be accessed via a pointer. The straightforward way for the debugger to keep track of calling history and the locations of variables would require logging very large numbers of pointers, most of which would never be needed. To avoid this, the debugger answers user queries (e.g., for the value of an identifier) by jumping back in time to obtain pointers only *after* they are known to be relevant.

Reverse execution is supported by a combination of checkpointing and re-execution. Checkpointing is effi-

*Paper presented at the ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, May 20-21, 1991.

[†]Supported in part by NSF Grant CCR-9002786.

cient in SML-NJ because we have an efficient way of capturing current continuations, which describe the immutable part of the store, and the remaining, mutable, part of the store is typically small.

2 ML Threads

Our present work aims to apply these debugging methods to a version of ML extended with concurrency features. Instrumentation and reverse execution are well-established techniques in the concurrent debugging world, (see, e.g., [20, 18, 23]), so it is natural for us to extend our sequential debugger in this direction. In particular, we hope to show that our enthusiasm for instrumenting code at *source* level, with all the portability and independence benefits this offers, remains justified in concurrent systems.

We have chosen to base our research on the ML Threads package [5] being developed at Carnegie Mellon University as part of an effort to make ML useable as a systems programming language. The concurrency primitives provided by this package are closely modelled on those of DEC SRC's Modula-2+ [2, 3], which in turn derive from work on Mesa [19], and ultimately from Hoare's monitors [12]. The principal Threads primitives are listed in Figure 1.

New threads are created using `fork` *function*, which executes *function* in a new thread of control running in parallel with the parent thread. The thread lives until it executes `exit` or returns from *function*. Threads are scheduled preemptively on as many processors as are available; they can also `yield` a processor voluntarily. The model provides no guarantees about the relative progress of concurrent threads. Threads may be used at any granularity of parallelism the application desires; Threads operations are assumed to be efficient enough to make fine-grained applications feasible. Note that there are no user-visible thread identifiers, and no `join` primitive; these can be synthesized out of shared variables if needed.

Threads communicate using a shared memory model. Communication is allowed via any `ref` cell or array that is in common scope. (Message passing primitives can be implemented on top of this model if desired.) It is intended that shared memory be protected with mutexes, but this is not enforced by the model. Mutexes are created dynamically using `mutex`. Mutexes can be acquired in blocking mode (`acquire`) or non-blocking mode (`try_acquire`). If blocked, the acquiring thread is (at least conceptually) put to sleep on a queue associated with the mutex until the owner issues a `release`. Mutex queues are serviced in arbitrary order; there are no fairness guarantees. The `owner` function returns true iff the caller currently owns the mutex.

Threads can arrange their own synchronization and scheduling using conditions, which are created dynamically using `condition`. Each condition is governed by a mutex and is normally used to represent some computable predicate on shared `refs` also governed by the mutex. The semantics of the `wait` operation are as follows: the waiting thread atomically releases the governing mutex and puts itself on the condition queue; when wakened off the queue by a `signal`, it reacquires the governing mutex before returning. The atomicity makes it possible to avoid "lost wakeup" problems. The `signal` operation normally wakes up one thread off the associated queue (if there is any), but it is allowed to wake more. Receipt of a signal should be only be used as a hint that the governing mutex should be acquired and the predicate retested. The `broadcast` operation wakes all threads off the queue.

Per-thread state, i.e., named mutable cells that take on different values in different threads, can be stored in vars, which are created using `var` and accessed using `getvar` and `setvar`.

In addition, the `callcc` (call with current continuation) operator used to capture continuations in sequential ML is also present in ML Threads. It is perfectly possible to capture a continuation in one thread and throw to it in another thread, although this can lead to confusing code.

Since Threads relies heavily on communication via the mutable store, it does not make a particularly idiomatic extension to ML. But its primitives are powerful and general, and there is a large body of experience in writing programs within its model. Moreover, Threads can be used to build a number of "cleaner" higher-level communication mechanisms with reasonable efficiency, and it can itself be implemented efficiently on a range of uniprocessor and multiprocessor architectures. As of July 1991, there are working ML Threads implementations for each of the uniprocessors supported by SML-NJ (VAX, Motorola 68020, SPARC, MIPS), for VAX-based multiprocessors under Mach, and for Silicon Graphics MIPS-based multiprocessors under Irix.

3 Debugging ML Threads

We have aimed to carry over as many of the basic techniques of our sequential debugger as we can to the ML Threads world. The main points of the debugger architecture are as follows:

- Each thread forked in the user program also runs as a distinct thread in the debugged version.

```
val fork : (unit -> unit) -> unit
val exit : unit -> 'a
val yield : unit -> unit

type mutex
val mutex : unit -> mutex
val try_acquire : mutex -> bool
val acquire : mutex -> unit
val release : mutex -> unit
val owner : mutex -> bool
```

```
type condition
val condition : mutex -> condition
val wait : condition -> unit
val signal : condition -> unit
val broadcast : condition -> unit

type 'a var
val var : '1a -> '1a var
val vget : 'a var -> 'a
val vset : 'a var -> 'a -> unit
```

Figure 1: Threads primitive functions

- The debugger itself runs as a separate thread, with no special privileges vis-a-vis the concurrency model.
- All user code is instrumented (by a transformation of abstract syntax) to include hook-points.
- There is a special debugger version of the Threads library, containing instrumented versions of the threads routines. This library is implemented entirely on top of the standard Threads library. Each Threads primitive has a hook-point.
- A separate software instruction counter (which counts *events*) is maintained in each thread. This and other per-thread data of use to the debugger are stored in vars.
- Breakpoints can be specified either by source location or by SIC value (*time*) within a thread. When a breakpoint is triggered, the entire computation is halted, and its global state can be examined.
- The global state consists of a set of continuations, one for each thread, plus a description of the global mutable store.
- After stopping at a breakpoint, execution may be resumed at the current state or at any state in the past. The debugger reverts to a past state by finding a suitable prior checkpoint and re-executing forward as necessary.
- Checkpoints are simply copies of the global state. They are taken at regular intervals during execution, and also at breakpoints.

In the remaining sections of this paper, we will describe some of the interesting implementation problems that arise from applying our approach to debugging to a multi-threaded world. Perhaps the most significant issue is that ML Threads programs are non-deterministic. This makes reverse and repeated execution considerably more complicated. The conventional debugging approach, which we discuss in section 4 and assume throughout most of the rest of the paper, is to log the order of non-deterministic events during initial execution, and use the log to govern subsequent replays. Often non-determinism is unintentional, and results in access anomalies; in section 5 we suggest a way of building more security into the semantics of the ML Threads primitives to reduce unintended non-determinism and make it easier to log what remains.

The mechanics of stopping, restarting, and reversing a distributed computation are complicated, especially when the debugger is not given any special privileges vis-a-vis the concurrency model. Section 6 describes how the debugger versions of the Threads primitives deal with one important task: "stopping the world" in order to execute a breakpoint or a checkpoint. Another major problem is that our simple linear model of program time must be modified to cope with the tree-like (or dag-like) nature of multi-threaded computation; this is discussed in section 7. Finally, in section 8 we suggest taking a more active approach to eliminating non-determinism from the user's program.

Throughout the design of the debugger, we have been particularly concerned with efficiency issues. To conclude this section, we mention a number of particularly important efficiency considerations.

- It is crucial to keep the overhead of instrumentation as low as possible during "normal" execution. In our sequential debugger instrumented code executes only 2-4 times slower than uninstrumented code, and we expect the raw performance of the Threads version to be similar. We think this is an acceptable slowdown from the user's point of view. Clearly, however, our instrumentation is extremely invasive, and leaves no hope of recovering the pattern of shared-memory accesses produced by the non-debugged code; we are dealing not so much with a "probe" effect as with a "ton of bricks" effect. As a result, we are interested in finding new ways to deal with race condition bugs; two of these are discussed in sections 5 and 8.

- Adding debugging instrumentation should not destroy any scalable speedup properties enjoyed by the concurrent program. That is, it should slow the program down by a constant factor independent of the number of processors in use. To achieve this aim, we try hard to avoid adding extra synchronization traffic between threads; some of the consequences are mentioned in sections 6 and 7.
- Space utilization is a serious concern, because of our need to keep checkpoints and logs. Since ML allocates huge numbers of heap cells, typically with very short lifetimes, it is desirable that the debugger not inhibit the system's ability to collect objects that are created and become garbage in the interval between two checkpoints. The sequential debugger already manages this for continuations and `ref` cells. We would like to manage it for threads too, since they also can be very short-lived. Unfortunately, synchronization logs produced by a thread (section 4) cannot be garbage-collected; this is a major impetus for finding a way to do without them (see section 8).

4 Logging non-deterministic effects

ML Threads programs are non-deterministic because their behavior can be influenced by the order in which threads perform *synchronizing operations* (acquires, releases, reads, and writes) on *shared objects* (mutexes, refs, and arrays) in the mutable store. Non-determinism may be benign (as when several threads compete to enter a critical section, and we can't predict which will win) or it may represent a bug (as when the programmer neglects to use a critical section to protect a shared memory access).

The most direct approach to supporting replay of non-deterministic concurrent computations is for the debugger to log the order of synchronizing operations. A log entry is made each time a shared object is accessed by a thread other than the one that last accessed it. To aid in building the log, the debugger keeps information about the the last accesser of each `ref` or array variable; this information must itself be protected by mutual exclusion. Recording the log is straightforward; replaying it efficiently is less trivial, since it involves introducing new synchronization mechanisms tailored to the replay. The basic issues were addressed in [4]; some clever shortcuts are discussed in [11].

The logging process consumes lots of space and time, so it is important to minimize the number of shared mutable objects that require logging. Fortunately, mutable variables must be explicitly declared in ML, and typically constitute only a small fraction of all identifiers, because most ML programs are mostly functional. (This situation compares favorably with that of Multilisp, in which every object is potentially mutable.) On the other hand, in ML Threads *any* mutable variable can be used as a shared variable, without explicit declaration. Because ML supports first-class functions, compile-time analysis of which mutable variables are actually shared will tend to be overly conservative.

There is also another way in which we might hope to improve efficiency. Correct Threads programming practice requires that all shared variables be governed by a mutex: a thread can only read or write the variable if it has successfully acquired the mutex. If the debugger could be sure that the programmer had obeyed this convention, it would only need to log the order of mutex acquisitions, rather than of individual variable references. This would represent a savings each time a single mutex is acquired to make multiple references.

Unfortunately, potentially buggy programs cannot be expected to obey mere conventions; what we would like is for the compiler and/or run-time system to enforce them for us. Original Threads implementations do not do this; indeed, they cannot, since the usual definition of the the Threads primitives does not provide a means for the programmer to specify that a particular variable is governed by a particular mutex. Actually, Threads implementations don't even enforce all the rules that they might: for example, they allow a thread that does not hold a mutex to unlock it (and then perhaps acquire it for itself)! Thus, it is easy for two or more threads to believe that they hold the same mutex, and accordingly access shared variables at the same time. (Amusingly, in this situation there is no point in having the debugger log the order of mutex acquisitions, since they don't actually affect the synchronization semantics of the program!)

5 Safe Threads

This laissez-faire attitude derives from a concern for efficiency: mutexes are typically represented by single bits, and no explicit state information is held as to which threads own which locks. Efficiency for correct programs is important, but so are safety features. Type checking and array bounds checking, for example, are valuable features of ML, even though they are not needed in correct programs.

For debugging we will demand a version of the Threads package with a stronger semantics, which we call Safe Threads. This system is based on a new class of shared objects called shared refs (`srefs`).[1] Every `sref` creation takes a mutex argument, indicating that the thread must hold the given mutex to access the `sref`. (Variables intended to be private to a single thread should be governed by a mutex that the thread keeps permanently locked.) Violating the ownership condition raises an exception. Only one thread can hold a mutex at any one time, and a thread can only release a mutex it already holds, again on pain of raising an exception.

There is an obvious run-time implementation strategy for Safe Threads: each mutex object is expanded to include a field holding the (internal) thread_id of its current owner, and each mutable object is expanded to include a pointer to its governing mutex. On each access to a mutable object, the system fetches the mutex pointer, fetches the mutex's owning thread_id, and checks that it equals the current thread_id. If not, an exception is raised. This sort of ownership checking is strongly analogous to array bounds checking:

- It provides extra safety at extra cost (two or three fetches and a compare) per access.
- Static analysis techniques can be used to elide some checks at compile time; for example, in any straight-line sequence containing accesses having the same governing mutex, only the first access requires a test.
- Checking can always be turned off if the user wants to live dangerously.

Although our Safe Threads proposal is motivated by the needs of the debugger, we believe it is valuable for other reasons. First, for programs in which shared variables are protected by locks, it provides access anomaly detection at a cost similar to other published proposals [6] and in a much simpler fashion. Second, it prevents the common programmer "short-cut" of performing single-word writes to shared memory without the overheads of arranging mutex protection. This works on shared-memory architectures that maintain sequential consistency [17], but will become increasingly dangerous as architectures begin to sacrifice sequential consistency in favor of improved cache performance (see, e.g. [9]). As hardware comes to rely on being handed explicit information about synchronization, it will become essential that compilers guarantee that all object code (even for buggy programs) obeys basic synchronization rules.

[1] Shared arrays can be introduced in a similar fashion.

Finally, we recognize that it is not always natural, convenient, or acceptably efficient to protect every shared variable with an exclusive lock. We may well want to extend the idea of Safe Threads to include more sophisticated mechanisms for controlling access to data, such as n-reader/one-writer locks, write-once variables, etc.

6 Stopping the world

The debugger often needs to freeze the overall program state, either at a user breakpoint or as part of a periodic checkpoint. This requires the following steps:

1. Stop each thread.
2. Arrange the release of all mutexes held by each thread.
3. Have each thread pass its continuation ("functional state") and other local state to the debugger, and then terminate.
4. Save a copy of the (global) mutable store (possibly only those parts that have changed since the last checkpoint).

The information from steps 3 and 4 is combined to form a checkpoint. At this point, the debugger can either continue the computation, by restarting from this checkpoint, or retrieve a previous checkpoint and restart from there. Restarting means forking a new thread to execute each continuation noted in the checkpoint, after resetting the local state and reacquiring any held mutexes. Step 2 is essential when restarting from an earlier point, since the mutexes may have belonged to a different thread at that point. Under Safe Threads, this step is also necessary to allow the (unprivileged) debugger thread to read the values of mutable cells governed by locked mutexes while executing step 4.

Since the debugger thread has no way to directly interfere with a running thread (there are no asynchronous signals or alerts in this version of the Threads model), it must stop threads by arranging for them to stop themselves. The simplest way is to define a global `pleaseStop` flag and augment the instrumentation code at each hook point to check whether the flag has been set. If it has, the instrumentation code executes the steps outlined above. (Since the hook point instrumentation is executed very frequently, and must therefore be kept as simple as possible, it will be better to set time-based breakpoints every few thousand events, and check `pleaseStop` only when these breakpoints go off.)

This method obviously doesn't work for a thread that is sleeping on a mutex or a condition when the debugger decides to stop the world. These threads need to be artificially wakened from their sleep long enough to stop themselves properly. Unfortunately, our adoption of Safe Threads makes this difficult. A thread blocked waiting for a mutex will only wake up when the mutex is freed, and again, under Safe Threads, the debugger cannot free mutexes it does not own. At first sight, this problem has a natural solution: if each running thread releases all its mutexes before stopping, eventually any thread waiting on such a mutex will be wakened long enough to notice the `pleaseStop` flag, release *its* mutexes and stop. But this scheme doesn't cope with the possibility of deadlocked threads, which will never be reached by this unwinding process.

Our solution to this problem is to *emulate* user-level mutexes with private debugger-controlled structures (see Figure 2).[2] This emulation makes use of the primitive Threads condition mechanism; it also uses primitive mutexes, but only as "spin locks" (i.e., locks guaranteed to be held only a short time), which protect the fields of the `Dmutex` record. When the emulation routines are in place, a user thread never actually sleeps on a primitive mutex; rather it sleeps on the `cond` field of the `Dmutex` structure, so the debugger can wake it by signalling the condition. Moreover, since the debugger has access to the `owner` field, it can lock and unlock Dmutexes at will. Ironically, this removes one reason for needing the wake the sleeping thread in the first place, though that is still necessary in order to gather the thread's continuation.

The `guardedWait` routine encapsulates support for stopping the thread when the `pleaseStop` flag is set. From the caller's perspective, `guardedWait` behaves like ordinary `wait` except that it will return after being wakened *either* by a legitimate `signal` *or* by the debugger trying to stop the world. The caller cannot tell which of these cases has occurred, but this doesn't matter since the the `owner` field is always explicitly retested. (This retesting is necessary anyway given the loose semantics of `signal`, which is allowed to wake any number of sleeping threads.)

The efficient implementation of `guardedWait` and the corresponding debugger routine `stopWorld` is rather complicated; a somewhat simplified version is shown in Figure 3. The key idea is that whenever a thread is sleeping on a condition, that condition will be in the globally accessible set `sleepers`; when the debugger wants to stop the world, it wakes up every member of the set. The mutex `masterLock` is needed to avoid race conditions that might occur if the debugger tries to stop the world just as a thread is deciding to go to sleep. Any thread that successfully acquires `masterLock` is responsible for acting immediately on `pleaseStop`; this must be done both before and after the actual `wait`.[3] To stop itself, a thread releases its spin locks and calls routine `stop` with the current continuation as argument; `stop` (not shown here) puts the continuation and other per-thread data in a globally accessible list, decrements a count of running processes, and exits. When the debugger continues from this checkpoint, it will fork a new thread which will start executing in `checkStop` just after the `callcc`.[4]

It is crucial that the debugger versions of Threads primitives preserve execution behavior when switching from the original version to the instrumented version of a user program. Since Threads programs are nondeterministic, equivalence of behavior will take the form of equality between *sets* of *possible* execution histories. In Safe Threads, execution history can be represented by a trace of mutex acquisitions, e.g., by the contents of a notional *acquisition log* recording the new owner after each successful acquisition. We will call two programs having the same sets of possible acquisition logs *acquisition equivalent.* Our task then is to show that substituting instrumented routines for the built-in Threads primitives preserves acquisition equivalence.

As an example, we will sketch an informal proof of this for the primitives of Figure 2.[5] It is convenient to do this in two stages. First, consider a version of `Dacquire` just like the one in Figure 2 except that ordinary `wait` is used in place of `guardedWait`. We claim that this function (call it `Dacquire*`), `Drelease`, and `Dmutex` can be substituted for the built-in primitives while preserving equivalence. One approach to proving this is to prove the stronger claim that the debugger routines conform to the same *specification* of the Threads primitives as the built-in versions do. (In our particular implementation this is not surprising, since the code was copied

[2] The actual implementations of these routines are considerably more complicated than those shown; in particular, support for logging and replay must be added.

[3] In practice, we want to avoid contention for `masterLock`, so we use a distributed hierarchical family of locks instead; this makes the code for `stopWorld` more complicated, but doesn't change the essential algorithm.

[4] As the reader may have realized, the debugger could also emulate conditions and so dispense with "sleeping" altogether. Under such a scheme, conditions would be represented by explicit queues of continuations. A thread would `wait` by putting its own continuation on the queue and calling `exit`; a signalling thread would take the next continuation off the queue and `fork` a new thread to execute it. This might well be a cleaner approach than the one we have detailed in the main text, but we suspect that it would be slower.

[5] Similar considerations apply to the debugger versions of `condition`, `wait`, and `signal`, which also involve emulation, but we lack space to show the details of these routines here.

```
val noOne : threadId                        fun Dacquire (DMUTEX{slock,owner,cond}) =
val myThreadId : threadId var                 (acquire slock;
                                               while (!owner <> noOne) do
type spinlock = mutex                            guardedWait cond;
                                               owner := (vget myThreadId);
datatype Dmutex =                              release slock)
  DMUTEX of
    {slock : spinlock,
     owner : threadId sref,                 fun Drelease (DMUTEX{slock,owner,cond}) =
     cond : condition                         (acquire slock;
     }                                         if (!owner) = (vget myThreadId) then
                                                 (owner := noOne;
fun Dmutex() =                                    signal scond;
  let val slock = mutex()                         release slock)
  in DMUTEX{slock=slock,                       else
            owner=sref(noOne,slock),             (release slock;
            cond=condition slock}                 raise Mutex))
```

Figure 2: Debugger versions of mutex primitives

from the built-in versions!) Formal specifications of this sort do exist for other Threads packages [3], though not yet for ML Threads.

Second, we show directly that substituting **Dacquire** for **Dacquire*** will also preserve equivalence. As noted earlier, **guardedWait** differs from ordinary **wait** only in that the former may return because the debugger stopped (and restarted) the world, whether or not the condition has actually been signalled since the waiting thread went to sleep. (We assume here that the actual call to **stop** and the subsequent restarting of the thread are transparent from the thread's point of view.) Now there are two cases:

- If the condition *was* actually signalled, the thread *might* have been legitimately wakened anyway. (Recall that the semantics of **signal** permit any number of sleepers to be wakened, in any order, after an arbitrary delay.) Anything the **Dacquire**-based thread does after being wakened, including a successful acquisition, might also have been done by the **Dacquire***-based thread, so we have acquisition equivalence.

- If the condition *wasn't* signalled, the **Dmutex** must still be owned by some other thread (since **Drelease** uses the **slock** spin lock to guarantee that clearing the owner field and waking the **cond** queue are performed atomically). Thus, the spuriously wakened thread will fail when it retries the acquisition, and simply go back to sleep; it cannot make an entry in the acquisition log, so it cannot affect acquisition equivalence.

Putting our two claims together gives us the desired result. A more formal proof approach would clearly be welcome, especially since the proof depends on the fine details of the semantics of Threads primitives. We are currently investigating the application of CCS-like proof methods to this problem[21].

7 Temporal Models

Our sequential ML debugger gives the user an abstract model of *program time*, based on the values of a software instruction counter that is incremented each time an event occurs during execution. This definition of program time has three key properties:

A Each event in the program execution has a unique corresponding program time, and each legal program time corresponds to a unique event.

B Program times form a total order corresponding to the causal ordering of events.

C Since only a bounded (though irregular) amount of computation can occur between events, the difference between a pair of program times offers a reasonable measure of the computational effort required to execute from the earlier time to the later one.

In extending our notion of time to the multi-threaded case, we would like to maintain as many of these properties as possible. To begin with, it is natural to define *local time* for each thread by associating a software instruction counter with the thread. These times have

```
(* Assume a data type of sets, supporting insertion, removal, and iteration:
  type 'a set
  val insert : 'a set * 'a -> unit
  val remove: 'a set * 'a -> unit
  val forEachIn : 'a set -> ('a -> unit) -> unit  (* iterate over set *)
*)

val masterLock : spinlock = mutex()
val sleepers : condition set sref = sref(empty,masterLock)
val pleaseStop : bool sref = sref(false,masterLock)

fun guardedWait (cond:condition) =
  let val cond_mutex = mutex_for_cond cond (* extract condition governing mutex *)
      fun checkStop () =
        if (!pleaseStop) then     (* check state of flag *)
          (release cond_mutex;
           release masterLock;
           callcc stop;           (* stop thread arranging to restart here... *)
           acquire masterLock;
           acquire cond_mutex;
           checkStop())           (* loop in case we're supposed to stop again! *)
        else ()
  in acquire masterLock;
       checkStop();               (* is flag set? *)
       insert (sleepers,cond);    (* notify debugger we are about to wait *)
     release masterLock;
     wait cond;                   (* do the actual wait! *)
     acquire masterLock;
       checkStop();               (* is flag set? *)
       remove (sleepers,cond);    (* notify debugger we are no longer waiting *)
     release masterLock
  end

fun stopWorld() =
  (acquire masterLock;
     pleaseStop := true;            (* set flag *)
     forEachIn sleepers broadcast; (* broadcast to each condition in sleepers set *)
   release masterLock)
```

Figure 3: Communicating the pleaseStop request

all the above properties with respect to events within a single thread.

There are several options for extending local per-thread times to a notion that will be meaningful across threads and be useful for debugging. We describe first an approach which emphasizes the distributed nature of multi-threaded computation, and requires no additional synchronization to compute. The *facetted time (f-time)* of an event in thread t is a list of integers defined recursively as:

(local time of event) :: (f-time of fork event that created t)

where the f-time of the forking of the root thread is taken to be the empty list.[6] These facetted times have property A (although the definition of which lists of integers constitute legal times is rather complex).

It is handy to identify a thread with the fork f-time of the event that created it (recall that threads have no intrinsic user-visible id's). With this understanding, facetted times can be interpreted as:

(local time within thread) :: (identifier for thread).

If the user is working with a particular thread, it is an easy matter to assign a temporary variable (e.g., `t`) to the thread identifier; times within the thread can then be entered as a single number consed with the thread identifier variable (e.g., `27::t`). This helps reduce the inconvenience of working with long lists of numbers.

We can define a partial ordering $<$ on f-times as follows. Let t and t' be f-times, and write

$$t = t_j :: t_{j-1} :: \ldots :: t_1,$$
$$t' = t'_k :: t'_{k-1} :: \ldots :: t'_1$$

Then

$$t < t' \iff \begin{cases} j \leq k \\ t_i = t'_i \text{ for } 1 \leq i \leq j-1 \\ t_j < t'_j \end{cases}$$

It is easy to verify that $t < t'$ implies that t occurred before t' in the program execution. Summing the facets of a facetted time produces a single integer *summed time (s-time)*, representing total computation effort on the path from the root to the event, which gives us an approximation to property C. Using these tools the debugger can determine (conservatively) whether it is possible to execute from a given checkpoint to a desired f-time, and about how much computation it will take; this is a fundamental operation when jumping back in time.

The major drawback of f-times is that they do not reflect the synchronizations between threads that result from shared memory operations during any *particular* program execution, so property B cannot hold. To put it another way, the partial order just described is only a proper subset of Lamport's happens-before relation [16]; we cannot deduce $t < t'$ from the fact that t occurs before t' in a particular program execution, or even in *every* program execution, if this causal constraint results from synchronization operations. Similarly, s-times don't reflect computation effort that may be required in other threads to enable synchronizations.

We can fix these problems, if they prove serious, by using Lamport's clock adjustment technique. The debugger can keep a "last accesser's" s-time with each shared object. Whenever a synchronizing operation is performed on a shared object, the acting thread first reads the s-time and makes sure its own s-time is greater, by artificially adjusting its local time upward as necessary; after doing the operation, it stores its (adjusted) s-time into the object. These *adjusted summed times (as-times)* correctly capture the happens-before relation. We can also produce a total ordering of events consistent with happens-before by using as-times with some arbitrary ordering of thread id's to break ties.

The cost of this adjustment is in keeping more synchronization information, and in introducing artificial discontinuities in local time (or keeping more than one local clock). The benefit is that it will be easier for the debugger (and perhaps for the user) to use adjusted times to reason about causal connections between events in different threads. In particular, adjusted times provide a simple framework for defining and detecting access anomalies, since two program sections can potentially execute concurrently iff their adjusted local time intervals overlap. It remains to be seen where the balance of interests lies; we are implementing the system without adjustments initially, because there number of the debugger's algorithms currently rely on time having no gaps.

8 Simulating Determinism

Thus far we have dealt with the intrinsic non-determinism of an ML Threads program in a fundamentally passive way: we let the program run under whatever external conditions (processor configuration, relative clock rates, system load, scheduling algorithm, etc.) currently obtain, and *record* the order of synchronizing operations that result. These synchronization logs can quickly grow very large, and they must be kept for the duration of program execution even for threads that do not otherwise appear in checkpoints.[7] Moreover, be-

[6] :: represents list cons in ML.

[7] They will have to be kept for even longer (and dumped into a more permanent format) if the user wants to preserve a particular

cause of the intrusiveness of our instrumentation, we may well fail to capture the interesting (i.e. buggy) aspects of the program's synchronization behavior in any case.

A more radical approach is to make the program behave deterministically on every run, including the very first. Since it seems very difficult to fix the underlying environment, we take the alternative approach of transforming the program itself, or, more precisely, the underlying Threads library used by the program. A conceptually simple way to do this is to use s-times (or as-times) as a *defining* total order for synchronizing operations. Each thread keeps track of its own s-time, and tags each of its synchronizing operations with the current s-time value. The Threads implementation must guarantee that the synchronizing operations on a single shared object (e.g., mutex) are performed in s-time order (an ordering on thread id's can be used to break ties).

It is easy to implement this guarantee on a uniprocessor version of Threads that executes one thread at a time: each time the system comes to a synchronizing operation, it pauses the current thread and continues by executing the runnable thread with the lowest s-time value. But naturally we would like to support execution on a multiprocessor! The most straightforward approach is to allow a thread to proceed with a synchronizing operation only when it is certain that no other thread will attempt an operation on the same object with a lower s-time tag. But it is easy to see that in the presence of dynamically forked threads and changing communication patterns, this approach can perform little better than the sequential implementation.

A promising alternative approach to this problem was introduced by Jefferson in his TimeWarp system[14], designed primarily for distributed discrete event simulation. We are using s-time as a species of what Jefferson calls *virtual time*. TimeWarp takes an *optimistic* approach to maintaining virtual time ordering of synchronizing operations. In this approach, the system normally allows threads to execute these operations without pause, but it keeps a log of the operations with their virtual time tags. If a thread t attempts to execute a synchronizing operation on an object and discovers that some other operation has already been done on that object at a virtual time greater than its own, the system *undoes* the operation having the later time and forces the thread that performed it to *roll back* to t's virtual time. Rolling back a thread involves restoring its old state (the system takes periodic per-thread checkpoints to make this more efficient) and undoing any other synchronizing operations it performed in the interim. This may in turn force rollbacks of other threads, in a cascading fashion. Obviously, we hope not to need to roll back too often.

execution sequence for further debugging at a later date.

An important concept in this approach is the *global virtual time (gvt)*, which is the minimum of the current virtual times of all threads. Clearly, the system never needs to roll back past the gvt, so log entries and checkpoints need only be kept until their virtual times fall behind the gvt. Maintaining the gvt exactly would require a lot of expensive synchronization traffic, but we can easily make do with a lazily computed conservative approximation. Using this technique to limit the storage costs of supporting rollback is a key to making the system feasible.

This optimistic approach is particularly attractive in our ML debugging setting because we already have rollback and logging mechanisms in place. Moreover, our thread rollback, based on `callcc`, is very time and space efficient; studies of TimeWarp have suggested that making process state saving efficient is essential to delivering adequate performance.

As noted above, most research on TimeWarp has been motivated by simulation applications. In fact, we can view our method of "making the program deterministic" as a simulation process, in which the definition of virtual time is based on parameters representing external conditions. For example, if we define virtual time to be as-time, our scheme represents a simulation of program behavior on a system with an unlimited number of processors, all running at about the same speed. By using a more sophisticated definition of virtual time, we can simulate a more realistic environment, such as a multiprocessor with a fixed number of processors, each making progress on our program at a somewhat different rate. To do this, we introduce the notion of a *virtual multiprocessor*, consisting of n *virtual processors* $p_1 \ldots, p_n$, each characterized by an execution rate r_i (defined relative to a "normal" rate of 1.0). We also define an explicit scheduling algorithm for assigning threads to virtual processors. We now want virtual time to measure elapsed time on our virtual system, so that it again makes sense to handle synchronization requests in virtual-time order. Each thread computes its progress in virtual time by multiplying its local time (maintained in the ordinary way using a SIC) by the execution rate factor for the virtual processor to which is currently assigned. Suppose, for thread t currently executing on processor p_i, we have:

vt_0 = virtual time at beginning of t 's current time slice

lt_0 = local time at beginning of t 's current time slice

Then the virtual time of a synchronization request at local time lt is given by:

$$vt_0 + r_i * (lt - lt_0)$$

Again, threads tag their synchronization requests with virtual times, and the simulator handles requests in virtual time order (using an ordering on processor id's to break ties).

Of course, this simulation is quite rough, but we believe it can prove useful. By varying the values of n and the r_i, we can obtain different program runs that may help expose time-dependent behavior, without worrying about probe effects. In particular, it is easy to experiment with extreme values; for example, we may set n to be (much) larger than the number of physical processors available to us. We can also vary n and the r_i over the course of a program run, e.g., to simulate the effects of processor failures or contention with other applications, so long as we do so in a deterministic, repeatable manner.

Easy repeatability is obviously the main virtue of this approach. We can replay during an execution, or reproduce an entire execution at a later date, simply by remembering the parameter values; there is no need to save full synchronization logs either during or after execution. (Partial logs are necessary to support optimistic simulation, but these can be bounded in size as described above; moreover, the space required can be traded off against the synchronization time needed to update gvt more frequently.) Implementing the simulation doesn't appear to be much more complex than implementing full logging. The key question is whether the optimistic simulation approach is indeed efficient enough to be useable (i.e., to meet our criterion of not destroying speedup); preliminary experience with TimeWarp in the simulation community seems encouraging[8]). In the best case, we could even consider using the simulator in production mode, allowing us to combine the advantages of deterministic execution with the freedom of expression provided by a non-deterministic language.

9 Status, Related Work and Conclusions

A prototype version of the ML Threads debugger described is operational on top of a uniprocessor implementation of ML Threads, and is in testing phase on a multiprocessor platform (as of July 1991). Our next priority is to obtain some performance measurements, and, of course, to see how useful the debugger is in practice.

Relatively little work has been done on debugging concurrent symbolic processing languages, although a debugger was written for Mul-T [15], and another is currently under development by Halstead and Kranz at DEC CRL [11]. Their work is similar to ours in many respects, but so far they have focused more on integrating synchronization logging with minimal probe effect and on user-interface design, a major problem area we have neglected so far.

Our current implementation uses logging techniques to capture non-determinism, but we are also interested in pursuing the simulation-based approach discussed in section 8. Methods of inducing deterministic behavior in concurrent programs have been touched on by other researchers (e.g., Carver and Tai [4]). As far as we know, however, no one has proposed using optimistic simulation mechanisms in the style of TimeWarp to systematically remove non-determinism from programs, either for debugging or production purposes. Obviously performance issues will loom large in any such effort.

Although writing a debugger for a language within the language itself may not be a typical application, it is an illuminating one. Fortunately, since there is as yet no general agreement as to what concurrent ML should look like, we have an opportunity to feed "debuggability" considerations back into language design at an unusually early stage. Our work so far has revealed enough deficiencies in the Threads model that we are ready to consider other approaches to concurrency, including `futures` in the style of Multilisp [10] and various message-passing systems deriving from Hoare's CSP [13] (notably Reppy's CML [24]). We think that comparing the suitability of these different concurrency models as vehicles for instrumentation-based debuggers will help us understand more about their relative overall strengths.

References

[1] A.W. Appel and D.B. MacQueen, "A Standard ML compiler," in *Functional Programming Languages and Computer Architecture,* ed. G. Kahn, LNCS, vol. 274,301-324,Springer-Verlag, 1987.

[2] A.D. Birrell, "An introduction to programming with threads," Research Report 35, DEC Systems Research Center, January 1989.

[3] A.D. Birrell, J.V. Guttag, J.J. Horning, and R. Levin, "Synchronization primitives for a multiprocessor: a formal specification," in *Proc. 11th ACM Symposium on Operating Systems Principles,* published as *Operating Systems Review,* 21(5), 94-101, November 1987.

[4] R.H. Carver and K-C Tai, "Reproducible testing of concurrent programs based on shared variables," *Proc. 6th International Conference on Distributed Computing Systems,* 428-433, May 1986.

[5] E.C. Cooper and J.G. Morrisett, "Adding Threads to Standard ML," Tech. Rep. CMU-CS-90-186, Carnegie Mellon University, December 1990.

[6] A. Dinning and E. Schonberg, "An empirical comparison of monitoring algorithms for access anomaly detection," in *Proc. 2nd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming,* 1-10, March 1990.

[7] B.F. Duba, R. Harper, and D.B. MacQueen, "Typing first-class continuations in ML," in *Proc. 18th Annual ACM Symposium on Principles of Programming Languages,* 163-173, January 1991.

[8] R.M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM,* 33(10):30-53, October 1990.

[9] K. Gharachorloo, *et al.*, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *Proc. 17th Annual Symposium on Computer Architecture,* 15-26, May 1990.

[10] R.H. Halstead, Jr., "Multilisp: A language for concurrent symbolic computation," *ACM Transactions on Programming Languages and Systems,* 7(4):501-538, October 1985.

[11] R.H. Halstead, Jr., and D.A. Kranz, "A replay mechanism for mostly functional parallel programs," Tech. Rep. CRL 90/6, DEC Cambridge Research Lab, November 1990.

[12] C.A.R. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM,* 17(10):549-557, October 1974.

[13] C.A.R. Hoare, "Communicating sequential processes," *Communications of the ACM,* 21(8):666-677, August 1978.

[14] D.R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems,* 7(3):404-425, July 1985.

[15] D.A. Kranz, R.H. Halstead, Jr., and E. Mohr, "Mul-T: A High-Performance Parallel Lisp," *Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation,* published as *SIGPLAN Notices,* 24(7):81-90, July 1989.

[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM,* 21(7):558-565, July 1978.

[17] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers,* c-28(9), 690-691, September 1979.

[18] T.J. LeBlanc and J.M. Mellor-Crummey, "Debugging parallel programs with Instant Replay," *IEEE Transactions on Computers,* 36(4):471-482, April 1987.

[19] B.W. Lampson and D.D. Redell, "Experience with processes and monitors in Mesa," *Communications of the ACM,* 23(2):105-117, February 1980.

[20] B.P. Miller and J.-D. Choi, "A mechanism for efficient debugging of parallel programs," *Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation,* 135-144, June 1988.

[21] R. Milner, *Communication and Concurrency,* Prentice Hall, 1989.

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

[23] D.Z. Pan and M.A. Linton, "Supporting reverse execution of parallel debuggers," *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Computing,* Published as *SIGPLAN Notices,* 24(1):184-197, January 1989.

[24] J.H. Reppy, "CML: a higher-order concurrent language," *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation,* to appear June 1991.

[25] A.P. Tolmach and A.W. Appel, "Debugging Standard ML without reverse engineering," *Proc. 1990 ACM Conference on Lisp and Functional Programming,* 1-12, June 1990.