-

Debugging Standard ML Without Reverse Engineering

*Andrew P. Tolmach**
*Andrew W. Appel**
Department of Computer Science
Princeton University

CS-TR-253-90

March 1990

(Paper presented at 1990 ACM Conference on LISP and Functional Programming, Nice, June 1990.)

Abstract

We have built a novel and efficient replay debugger for our Standard ML compiler. Debugging facilities are provided by instrumenting the user's source code; this approach, made feasible by ML's safety property, is machine-independent and back-end independent. Replay is practical because ML is normally used functionally, and our compiler uses continuation-passing style; thus most of the program's state can be checkpointed quickly and compactly using call-with-current-continuation. Together, instrumentation and replay support a simple and elegant debugger featuring full variable display, polymorphic type resolution, stack trace-back, breakpointing, and reverse execution, even though our compiler is very highly optimizing and has no run-time stack.

1. Introduction

Traditional ''source-level'' debuggers do their real work at machine level. They rely on detailed information about the underlying machine model, compiler back end, and run-time system. Although debuggers typically have access to the original source text and some symbol table data, coordinating source and object at run time requires extensive ''reverse engineering,'' which is difficult. As a result, source-level debuggers are typically characterized by limited functionality, poor portability, and considerable internal complexity [Bruegge85]. These problems are greatly exacerbated by the presence of compiler optimization, which often makes the task of mapping the machine code back to the original source essentially impossible [Hennessy82,Zellwegger84].

Standard ML of New Jersey (SML-NJ) [Appel87a] is a very highly optimizing compiler for the Standard ML language [Milner90]. In fact, it completely transforms the code several times: first into lambda-calculus, then into continuation passing style, and then through several global optimization phases. It would be difficult or impossible to write a traditional debugger that could deal with the resulting machine code. Moreover, the standard approach of turning off optimization would not work here; the compiler's whole methodology is based on code transformation.

Faced with the task of getting a program to run without a debugger, programmers commonly *instrument* their code at key points to print the values of variables or trace the flow of control. In one respect, this works unusually well for ML, because the language is *safe*; that is, compile-time type checking guarantees that there are no run-time insecurities (''core is never dumped''). This means that we can always understand the run-time behavior of ML programs—even buggy ones—without reference to the underlying machine model (assuming the compiler functions correctly). Since the instrumentation is part of the code, our ML compiler's back end guarantees not to modify its semantics when performing optimization.

Of course, instrumenting code by hand is tedious and

* Supported in part by NSF Grant CCR-8806121.

time-consuming. The key idea of our debugger is to *automatically* insert instrumentation into the user's source code to support subsequent debugger queries. Thus, wherever an identifier is bound, we add code to report its value; wherever a function is called, we add code to report the caller and the callee. Whether or not this added instrumentation is executed is decided at run time, so that information is generated only when and if it is wanted; we can also conditionally break out of the program at any instrumentation point.

Since we cannot predict in advance what information will be wanted by the programmer, our debugger supports reverse execution. This means we can arrange that information is collected only *after* it is known to be relevant to a user request. For example, to determine the value of an in-scope variable, we jump back to the time when it was bound, turn on the conditional instrumentation that reports its value, record that value, and return to the original time. Time travel of this kind turns out to be remarkably versatile: we also use it to implement location-based breakpoints and to display ‘‘stack trace-backs’’—though our run-time system actually has no stack, and does tail-call elimination.

Replay debuggers are typically implemented by taking periodic checkpoints of the program's state; it turns out that SML-NJ can do this quite efficiently. ML is a mostly functional language, and the ‘‘functional part’’ of its execution state can be completely captured in a *continuation*. SML-NJ has a call-with-current-continuation (**callcc**) primitive [Friedman84], which gives a way to save and reset continuations at any point in program execution. Because the SML-NJ compiler uses continuation-passing style with no run-time stack [Appel89b], its implementation of **callcc** is extremely simple and time-efficient. Moreover, although a single continuation may be large, continuations taken at adjacent checkpoints will typically point to many of the same memory cells, so that it is feasible to keep multiple checkpoints in main memory. ML does have non-functional features, including a mutable store and I/O support; different and less efficient methods must be used to capture the non-functional part of the program's state, but this part of the state is typically much smaller than the rest.

Instrumentation is done by a one-pass transformation of the abstract syntax tree produced by the compiler's parser; this is conceptually the same as preprocessing the source code, but easier. Although our instrumented code runs a few times slower than ordinary code, it runs about as fast as unoptimized code and much faster than an interpreter. Since the other debugging strategies known to us require either inhibition of optimization or extensive interpretation, we believe our approach is practical and time-competitive. Our method does consume a good deal of memory, but we believe that memory is a relatively inexpensive resource in most computer systems. The debugger implementation is completely machine-independent, and uses only a few well-defined back-end features (for state checkpointing).

2. Events and Time

The debugger modifies source code of "debuggable" functions by adding instrumentation in the form of *events*. Events are located at value declarations, at the top of each function (and each **case** branch), and immediately prior to each function call (excluding built-in functions that cause no side-effects). (Note that we do not place events *after* calls; this insures that the compiler's tail recursion elimination methods will still apply.) There is at least one event within each basic block, and at each binding location. (In practice, adjoining events within the same basic block are coalesced to reduce overhead, but we ignore this complication in what follows.) This enables events to serve both as potential breakpoint locations, and as convenient points at which to collect the values of bound variables. Each distinct event in the program text is given a unique *event number*. The debugger maintains a mapping between event numbers and locations in the original abstract syntax tree, which is retained during debugging.

As the instrumented program runs, the debugger maintains a counter that is incremented each time an event is executed (whether or not the associated breakpoint is taken). This counter is a form of software instruction counter, which we use to uniquely identify points in the program's execution history. We refer to the value of this counter as the *current time*, and talk about the corresponding time of any event execution.

Breakpointing is controlled entirely on the basis of time: a break is taken at an event execution if the time of that event matches the value of a **targetTime** variable maintained by the debugger. In addition, the debugger keeps an array **eventTimes**, indexed by event number, that records the last time each event was executed. This array allows location-based breakpointing to be simulated by time-based breakpointing, as will be described in Section 4. A summary of the instrumentation code described so far is shown in Figure 1. Note that **event** is shown as a subroutine here; in practice, the code will normally be placed in-line for execution efficiency.

```
fun event(eventNum,lastBindTime,boundValues) =
  (currentTime := !currentTime + 1;
   if (!currentTime) = (!targetTime) then
     break (eventNum,lastBindTime,boundValues)
   else ();
   update (eventTimes,eventNum,!currentTime))
```

Figure 1.

```
fun¹ rev (h::t) =²
      let val⁴ r = rev³ t
      in r @⁵ [h]
      end
  |  rev nil =⁶
      nil
```

Event#	Event type
1	fun binding
2	fn entry
3	application
4	val binding
5	application
6	fn entry

User code for the **rev** function
Figure 2.

```
val bindTime1 = !currentTime + 1
fun rev (h::t) =
      (event(2,bindTime1,h,t);
      let val bindtime2 = !currentTime in
        let val r = (event(3,bindTime2);
                     rev t)
            val _ = event(4,bindTime2,r)
            val bindTime4 = !currentTime
        in event(5,bindTime4);
           r @ [h]
        end
      end)
  | rev nil =
      (event (6,bindTime1);
       nil)
val _ = event(1,0,rev)
```

Instrumented Code for the **rev** function.
Figure 3.

The debugger adds further instrumentation to each binding event (**val** and **val rec** declarations, and each rule within a **fn** or **case**) to create a new variable named **bindTime***n*, where *n* is the event number. This variable has the same scope as the variables bound at the associated event; its value is the time at which the binding occurred. The importance of these variables is described in Section 5.

The user program and the debugger are related as coroutines. When a breakpoint is taken, the user program transfers control to the debugger by calling **break** with these arguments: the event number, the time of the previous in-scope binding event execution (which will be an appropriate **bindTime** variable), and a list of the values bound at this event (if any). The values are not tagged but are in a fixed order that can be deduced by the debugger from the abstract syntax corresponding to the event.

Figure 2 shows the user code for a short ML function which reverses a list. The locations of events in this code have been indicated by annotating with event numbers; the adjacent table lists the type of each event. Figure 3 shows the instrumented version of the code, with the original program in bold font and the added instrumentation in italics; the **event** function is as in Figure 1.

3. States and Time Travel

The debugger starts up a user program by setting **targetTime** to a suitable value and invoking the program as a coroutine. The user program then executes normally until the target time or the end of the program is reached, upon which the debugger is re-entered via **break**. Whenever it receives control, the debugger makes a checkpoint of the current *state* of the user program, tagged with the current time.

To continue normally from a breakpoint, the debugger simply resumes the user program coroutine, leaving the state unchanged. But the debugger can also restart the user program from any previous time for which it has a state checkpoint, simply by restoring that state before resuming the user program coroutine. To restart the user program from an *arbitrary* previous time t, the debugger resets the program state to the latest time $\leq t$ for which it has a checkpoint, sets **targetTime** to t, and re-executes forward. To make sure that we won't have to do too much re-execution, automatic **breaks** are caused at regular intervals. In particular, if the user explicitly wishes to break at a particular time t, we assume that the period immediately prior to t is also likely to be of interest, with our level of interest increasing exponentially as we approach t. If we are asked to go to time t from the nearest previous stored time t', we first break at time $t'' = (t + t') / 2$, then halfway between t'' and t, and so on, until we estimate that it would cost less to re-execute directly to t than to store another state. Taking these breakpoints during forward execution costs a small amount of extra time, but saves a great deal of time later if we want to jump back into the period before t.

This time-travel mechanism is encapsulated in a routine called **gotoTime**, which is used as a primitive by many debugger commands. **GotoTime** takes a primary argument **newTime**, which specifies a time in the past to reset to or a time in the future to advance to, and a secondary argument **errorMargin**, which is used when we only need to reset a time approximately and it would be wasteful to bother re-executing to a precise time; if there is a stored checkpoint within **errorMargin** units of **newTime**, it will be used and no further forward execution will take place.

Taking breakpoints is expensive in execution time, and storing checkpoints is expensive in space, so it is important to keep the set of checkpoints we remember reasonably small; therefore, we manage it as a cache. We can choose the size of the cache dynamically by getting memory-demand information from the run-time system. For any given cache size, whenever a new state is added to the cache, some previously stored state may need to be thrown out. Ideally, we would like to keep the states that will be most useful in future restarts, but we cannot predict precisely which these are, so some heuristic is needed. One heuristic is to use a form of least-recently-used: throw away the states that have not been used for restarts in a long time. Another approach is to rank the states based on how expensive it would be to regenerate them, and to throw away the least expensive. It is easy to construct scenarios in which one or the other of these heuristics fails; we would like to find a compromise scheme.

4. Breakpoints

We allow the user to set breakpoints either at particular source program locations (as in traditional debuggers) or at particular times in the program's execution history (past or future). Our debugger is unusual in making time-based breakpointing the fundamental mechanism and using it to implement location-based breakpoints. We think programmers will find efficient time-based breakpoints very useful; conventional debuggers can achieve a similar effect (in the forward direction) only by laboriously repeated single-stepping.*

As with most debuggers, we keep a list of breakpoints; the user can execute either forward or backward to the nearest breakpoint. At present, encountering a breakpoint simply returns control to the user; in principle, it would be easy to associate actions with these breakpoints as well. The remainder of this section assumes that there is only one breakpoint, but the methods described can handle multiple breakpoints and mixtures of time- and location-based breakpoints without difficulty.

---

* In one conventional debugger (GDB) we have measured single-stepping to be more than 25000 times slower than ordinary execution. Our debugger can execute to a time-based breakpoint at a rate only 2 to 4 times slower than ordinary execution (see Section 8).

Implementation of time-based breakpoints is trivial, given the **gotoTime** primitive. Reverse execution to a location-based breakpoint is also easy: we simply look up the last time for the given event in the **eventTimes** array and **goto** that time. Forward execution is much more complicated. Our goal is to find the first time at which the **eventTimes** entry for the event has changed. To do this, we jump into the future looking for an outside bound on the time. Since we have no *a priori* idea of the true distance to the desired time, we first jump forward by an arbitrary delta; if the entry hasn't changed we jump forward again by twice as much as before. We repeat this, doubling our jump each time, until we find the entry changed or the program finishes. Note that in this phase of the algorithm we never go more than twice as far as the true distance, and the total number of breaks we take is proportional to the log of the true distance. Having bounded the desired time from above and below, we next perform a binary search to pinpoint the time when the entry *first* changed. The number of breaks we take in this phase is again proportional to the log of the true distance from our initial starting time. In all our invocations of **gotoTime** we specify a broad error margin in hopes of being able to reuse stored checkpoints without re-execution.

We have chosen this implementation of location-based breakpoints to keep event instrumentation simple. An alternative implementation of forward location-based breakpoints would be to maintain a separate boolean array, **breakWanted**, indexed by event number. User code instrumentation would be extended so that a break at event *i* occurs if **breakWanted[*i*]** is true *or* the target time has been reached. This would give simpler forward breakpointing but would increase the cost of *every* event, whether or not breakpoints were in use. Basing all breakpoint decisions on time keeps the per-event test simple and efficient.

Moreover, our binary search method can be extended to pinpoint any event having a monotonic indicator function, perhaps user-specified.* This could be far more efficient than the traditional implementations of watch-pointing by repeated breakpoints or memory protection tricks.

* E.g., "break when index variable i reaches 100".

5. Displaying values

Recall that there is an event associated with each ML statement that binds a value to an identifier; the bound value is passed to the debugger when the associated **break** is executed. The basis of our technique for displaying the value of an identifier is to jump back to that identifier's binding time, cause the **break** to occur, collect the associated value, and return to our original time. Thus values are typically passed to the debugger only after a specific request by the user. (This works for the large proportion of ML objects that are immutable; we discuss the handling of mutable objects below.)

A user's request for a value is always interpreted in the context of the particular time and event occurrence at which the program is currently halted. The variable requested must be in scope at this point, and any other variables with the same name are hidden by the one in scope. If the variable has just been bound at the current event, the debugger will already have obtained its value when control was received from the program. Otherwise, we jump back to the latest prior event execution that bound some in-scope variable; the time of this execution is passed to the debugger via the **lastBindTime** argument to **break**. We repeat this process until we find the desired value; every in-scope variable must be bound somewhere on this *binding chain* of event executions.

As an example, consider the **rev** program of Figures 2 and 3. Suppose we are stopped at event #5 (just before doing the list append), and that the user asks for the value of **h**. No values were passed at event #5, so we jump back to the **lastBindTime** for event #5, which was given as **bindTime4**, and will in fact be just one time unit earlier. The event for this time is #4, which can only give us the value of **r**, so we must repeat the procedure, jumping back to **bindTime2**, which is associated with event #2. (Notice that this will generally not be the most recent execution of event #2, which will have occurred during the recursive call to **rev** used in calculating **r**, and its time bears no simple relation to our original starting time.) Since the desired value of **h** is available at event #2, the process completes here.

Note that the **bindTime*n*** variables act something like access links in a conventional run-time system for a block-structured language; unfortunately, they occupy storage proportional to the depth of recursion in the program. However, the length of the binding chain, and hence the number of calls to **gotoTime** needed to look up a variable, is proportional only to the static size of the program, not to execution time. Moreover, if we look up the values of multiple variables from the same context we can expect to revisit many of the same times along the chain; since checkpoints are cached, no re-execution will be needed to obtain the second and subsequent values.

Our method for displaying values of objects in the mutable store (references and arrays) is somewhat different. Again we go back to the binding site for the object, but in this case we collect a pointer to the object, which is passed to **break** instead of the object's contents. We then jump back to the current time and fetch the current contents of the object (which may have changed repeatedly in the interim). For this method to work correctly, it is essential that whenever code for creating a store object is re-executed (as it typically will be when finding the bind time for that object) it reuses the same object pointer that was created during the original execution. If this is not done, we may have multiple ‘‘versions’’ of the object referenced from different saved checkpoints and thus get erroneous results. The mechanism for avoiding this problem is described in Section 7.

We also plan to support *modification* of store values. The debugger can get a pointer to the **ref** variable as just described, and can then change its value directly (taking care not to violate type constraints expressed in the abstract syntax). This has the effect of changing the execution history of the program, and so all future stored states must be thrown away. Furthermore, we must record the change and make sure it gets re-executed whenever we pass through the same time again; this is essentially an (internal) action associated with a time-based breakpoint.

To display a variable's value, we need to know its type. ML supports polymorphic variables (e.g., **h** in the example above) whose concrete types may depend on the types of the actual arguments to the enclosing functions, and cannot be deduced at compile time. Moreover, our compiler (like most ML compilers) has no run-time type tagging scheme either! Happily, we can use the debugger's ability to find the variable's binding time to deduce types. This is done by finding the calling function's variables, determining their types (recursively applying this algorithm if necessary), and rerunning the compiler's type-unification algorithm. (A similar scheme for a stack-based tag-free run-time environment was described in [Appel89a].) The only disadvantage of this algorithm is that it may require time proportional to the depth of function-call nesting; we hope to improve this for some common cases by specialized code analysis.

Finally, we can use our knowledge about the binding site for a variable to display not only its value but also the program source that defined it. This is particularly useful for variables representing functions, which have no other printable "value". At present, we simply pretty-print the abstract syntax for the binding, which may be a simple assignment, function call, or **fn** (lambda) expression. In the future, we plan to analyze the definition and recursively find and print the bindings of any functions referenced by the right-hand side of the binding; this should make it much easier to debug programs that use functions as first-class values.

6. Reconstructing Call Histories

SML-NJ doesn't maintain a stack, and it optimizes tail calling into iteration. Nevertheless, our debugger can easily produce a "stack trace-back" showing a complete history of function calls and their arguments. This is possible because: (i) there is an event immediately prior to each application; (ii) there is an event at the top of each function rule body, which is always executed immediately after an application event; and (iii) all top-of-function events appear in the binding chain. Thus we can always determine the caller of the current function by skipping back through the binding chain to find the current top-of-function event and then stepping back by one time unit to the corresponding application event. We can repeat this process to display calling history to any depth desired.

7. Checkpointing Program State

A program's state has three separate parts:

1. The current continuation; i.e., the continuation that will be invoked when we resume from **break**. The continuation encompasses the values of all immutable objects and all control flow.
2. The contents of the mutable store, i.e., the values of all **ref** cells and arrays.
3. The I/O state, i.e., the history of activity on all I/O streams.

Each part of the state is checkpointed using a different method. At present, we keep all parts of a state checkpoint in main memory; some parts could be maintained on backing store instead, as will be noted below.

A fundamental assumption is that the continuation part of the state will normally be by far the most voluminous. Fortunately, it is also the easiest part to capture, using the *call-with-current-continuation (***callcc***)* primitive provided by the SML-NJ run-time system; in fact, the existence of this feature was a key motivation for our approach. Storing the current continuation is very fast (involving only the copying of a few registers), and only costs space when it keeps live pointers to objects that would otherwise be garbage collected. Thus continuations are an inherently incremental checkpointing mechanism. We actually capture the current continuation at a **break** as a byproduct of switching to the debugger coroutine; the operation is straightforward, and we will say no more about continuation state here.

To capture I/O state, we record all I/O operations in a time-indexed log. In normal mode, each I/O operation appends its result to the log before returning. On replay, the I/O operation is not performed and the log entry for the current time is returned instead. We obviously cannot do better than this for interactive I/O, but we could avoid keeping a log of input from files by assuming that the files are stable and checkpointing the file pointers; we may implement this approach in future. It would also be reasonable to keep our log on backing store.

Capturing mutable store state is our biggest challenge. Fortunately, we expect the store to account for a relatively small part of total data; for example, 99.7% of the objects created when SML-NJ compiles itself are immutable. It is therefore acceptable to penalize programs that make heavy use of the store. The most obvious method for checkpointing the store would be to copy the contents of all arrays and **ref** cells wholesale, but this could be very wasteful if there are many store objects of which only a few have been created or updated since the last **break**. Instead, we keep *delta lists*; that is, at each **break** we record only changes that have occurred since the last **break**. We build these lists by instrumenting every array creation and update so that it appends a pointer to the updated element to a global list. (This global list can temporarily occupy a great deal of space; a very similar list is already maintained by our generational garbage collector, and we could make use of it at the cost of increased dependence on the details of the back end.)

When a **break** occurs, the debugger retrieves the list, removes duplicate entries (typically very numerous), and fetches a copy of the contents of each entry. The resulting delta list is stored tagged with the current time. To reset the store to a given time *t*, we must consult, in order, the contents of all delta lists with tags between 0 and *t*, and reset the values of each element in each list. If the same objects are repeatedly updated, they will tend to appear in many lists, with all but the last appearance being overwritten. To improve efficiency in this circumstance, we periodically merge adjacent lists, removing duplicate elements. In principle, we could also keep the lists on backing store.

Our algorithm for looking up store object values (see Section 5) puts a further demand on the store history system: we must insure that if an object creation is re-executed when the original object is still live then the original object is reused. To do this, we maintain a table of store objects, hashed on creation time, and instrument each creation to check this table. If there is a relevant entry it is reused; otherwise the object is created and inserted in the table. This is a significant source of inefficiency, and we are looking for alternative methods to solve the problem.

We use special *weak pointers* to the store objects from within the debugger's data structures. Weak pointers are like ordinary ones except that the garbage collector is allowed to remove objects pointed to only from weak pointers. Suppose, as is often the case, that a large number of the store objects created in the user program are short-lived (e.g., local to the scope of a loop body), and would normally be reclaimed before we take a **break**. Our use of weak pointers allows this to continue happening; using ordinary pointers would prevent the objects from being treated as garbage.

8. Implementation and Performance

We have implemented the debugger (in ML) as an extension to the SML-NJ compiler. The implementation is divided into several modules, with support for checkpointing, time travel, code instrumentation, and query commands clearly separated. Debugger code is wholly independent of the compiler's back end except for continuation checkpointing (which uses **callcc**) and store checkpointing (which uses weak pointers and some basic knowledge about run-time data formats). The code is roughly 3200 lines long (about 10% of the size of the compiler itself).

		Normal		Instrumented for Debugging			Unoptimized	Interpreted
Program	Store Events	Code Size	Execution Time	Code Size	Execution Time	Debug Time	Execution Time	Execution Time
RedBlack	2.5 %	2784 b	10.3+1.0 s	10208 b	22.4+2.6 s	31.7+3.7 s	28.1+2.2 s	758.9+18.6 s
Lexer	5.6	14012	7.9+0.1	37892	16.8+6.4	27.7+7.4	16.1+0.7	219.2+4.1
UnionFind	8.4	8060	5.4+0.6	38204	17.7+9.7	31.6+15.2	10.6+1.5	123.4+3.7
TermRewrite	0	7096	6.9+0.1	34900	20.1+0.3	26.2+0.6	18.9+0.1	297.0+9.7
RefCreate	90.9	512	0.2+0.0	4232	23.2+17.7	30.0+24.5	0.8+0.0	19.6+0.7
RefUpdate	90.9	568	0.1+0.0	6368	2.0+7.5	3.6+10.6	0.2+0.0	13.3+0.8
I/O	0	760	2.5+0.1	2980	8.2+1.6	41.9+1.8	2.6+0.0	5.6+0.1

Benchmark Statistics

(Code Size in bytes of VAX machine code.)

(Times in form *execution-time* + *garbage-collection-time*, both in CPU seconds.)

Table 1

The debugger is fully integrated into the SML-NJ interactive system; code can be compiled and executed in debug mode by using simple commands, which are ordinary ML functions loaded into the user's top-level environment. The user can write new debugger functions based on the existing primitives, using ML itself as a ‘‘customization language.’’

We have benchmarked the debugger (on a 32 MB VAXstation 3500 running Ultrix) with the results shown in Table 1. The top section of the table contains entries for typical programs: **RedBlack** does insertions into a red-black tree, **Lexer** does lexical analysis on ML source text and counts tokens, **UnionFind** reads strings from a file and does union and find operations with path compression, and **TermRewrite** (the only purely functional program benchmarked) does symbolic differentiation by term-rewriting. The entries in the bottom section of the table represent some worst-case program types: **RefCreate** simply creates mutable reference cells, **RefUpdate** just updates such cells, and **I/O** just copies an input file to an output file. The column labeled Store Events gives the percentage of event executions that represent creation or update of mutable store cells. We report execution time and garbage collection time separately because we believe the latter can be made arbitrarily small given sufficient memory [Appel87b] and the use of virtual-memory techniques [Shaw87] that would be expected in a production compiler.

The left-hand side of the table compares normal (uninstrumented) optimized code to instrumented code, for which two timing figures are given. Execution Time is the time needed to execute the instrumented program to completion without taking breaks; it measures the overhead of instrumentation. Debug Time is the time needed to execute to a location-based breakpoint at the end of the program; this involves taking periodic checkpoints and re-executing some part of the tail end of the program (up to two-thirds of the program in the worst case); it is intended to measure the cost of executing the whole program in a more typical debugging context. Nearly all the added time is in re-execution; separate measurements (not detailed here) show that checkpointing alone costs very little. It should be emphasized that the Debug Time figure reported is dependent on several arbitrary parameters (e.g., minimum and maximum checkpointing intervals) that we have made no effort to tune as yet.

These timings confirm the practicality of our approach. The typical programs execute only 2 to 4 times slower under the debugger than normally. Debug Times are at worst twice as long. This seems an entirely tolerable price to pay for debugging functionality. Reference operations slow things down very substantially, as the worst-case benchmarks make clear, but this does not cause the typical programs to suffer too much. I/O is particularly expensive on replay; this accounts for the Debug Times being higher than might be expected.

The two right-hand columns of Table 1 give evidence that our instrumentation approach compares well with other possible debugger methods in time performance. Unoptimized Execution Time is the time to execute the normal program when it is compiled with all optional optimizations turned off. This approximates the time that could be expected if we had tried to build a conventional debugger on top of a compiler that didn't do code-rewriting optimizations. Times in this column are comparable to Instrumented Execution Times. Interpreted Execution Time is the time to execute the normal program under our lambda-calculus interpreter; with such an interpreter, users could reasonably debug by manually instrumenting their own code as needed at run time. Interpretation is typically an order of magnitude slower than instrumented execution.

The table also shows that instrumenting code increases its size significantly, and compile time increases accordingly (somewhat worse than linearly). We can mitigate this effect by not compiling event instrumentation in-line; this cuts compilation time by 30% to 50%, while increasing execution time by 25% to 75%. We would also like to speed up our compiler!

To compare the memory demands of debugging versus normal execution, we separately consider *(i)* the memory used to compile the program, *(ii)* the *static* memory occupied by the program after it has been compiled, and *(iii)* the *dynamic* memory occupied by program data while it is running.*

We have not extensively measured memory use during compilation. A debuggable program will temporarily require more memory because a second, larger, instrumented version of the abstract syntax is produced, and all the other intermediate representations will be correspondingly larger.

Except for code size (see Table 1), the static space requirements of instrumented and non-instrumented code are very similar. The instrumented version requires only a small additional data structure describing events; this structure points into the original abstract syntax, which is large but is kept by the standard system anyway.

Dynamic memory use is much more interesting. Figure 4 shows dynamic memory usage (in MB of live data) measured at regular intervals over the course of each typical benchmark run, with checkpoint caching disabled. The lowest curve in each graph shows the amount of data generated by ordinary non-instrumented code. The middle curve shows the the amount of data generated by instrumented code; the extra data is accounted for by the I/O log, delta lists for the mutable store, and larger code closures. The top curve includes the additional space needed to store uncompressed delta lists for the mutable store; this final increment is not part of the checkpoint but can contribute significantly to temporary memory requirements. Note that all figures were obtained by measuring live data size after full garbage collections. They are valid for comparative purposes, but must be multiplied by a substantial factor (at least 3) to get an estimate of the actual peak memory requirements of a adequately functioning system.

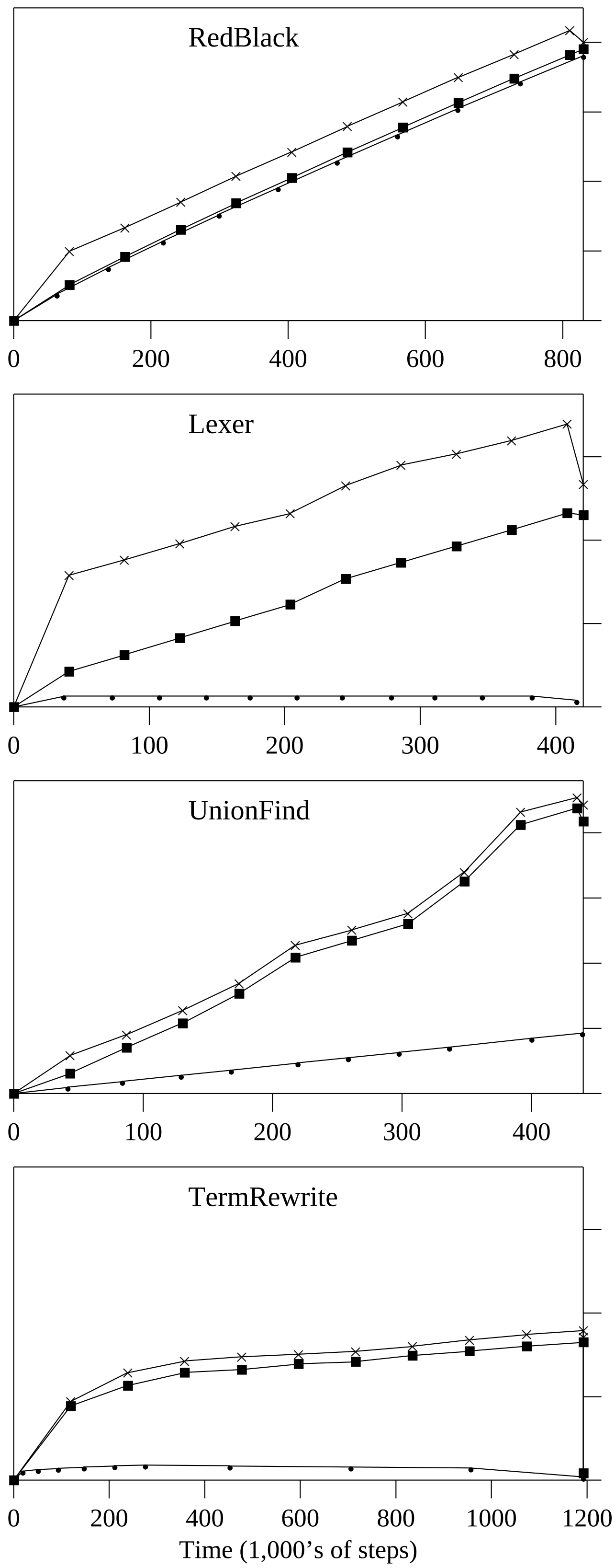
Figure 5 illustrates the phenomenon of storage sharing by continuations for each typical benchmark. In each case, a series of checkpoints was taken at exponentially decreasing intervals approaching a particular time towards the end of the computation (a log scale has been used on the x-axis to display the points more clearly). The lower curve in each graph shows memory use when each checkpoint was thrown away before the next was stored. The upper curve represents the case where all checkpoints were retained in the cache; the resulting exponential distribution of cache entry times is typical for many debugger operations. Substantial sharing occurs in all cases; in programs like **RedBlack** that build up a data structure, sharing is almost 100%. Moreover, even for programs like **TermRewrite**, which turn over their live data rapidly, the marginal space cost of additional checkpoints approaches zero as the interval between them decreases.

9. Related Work

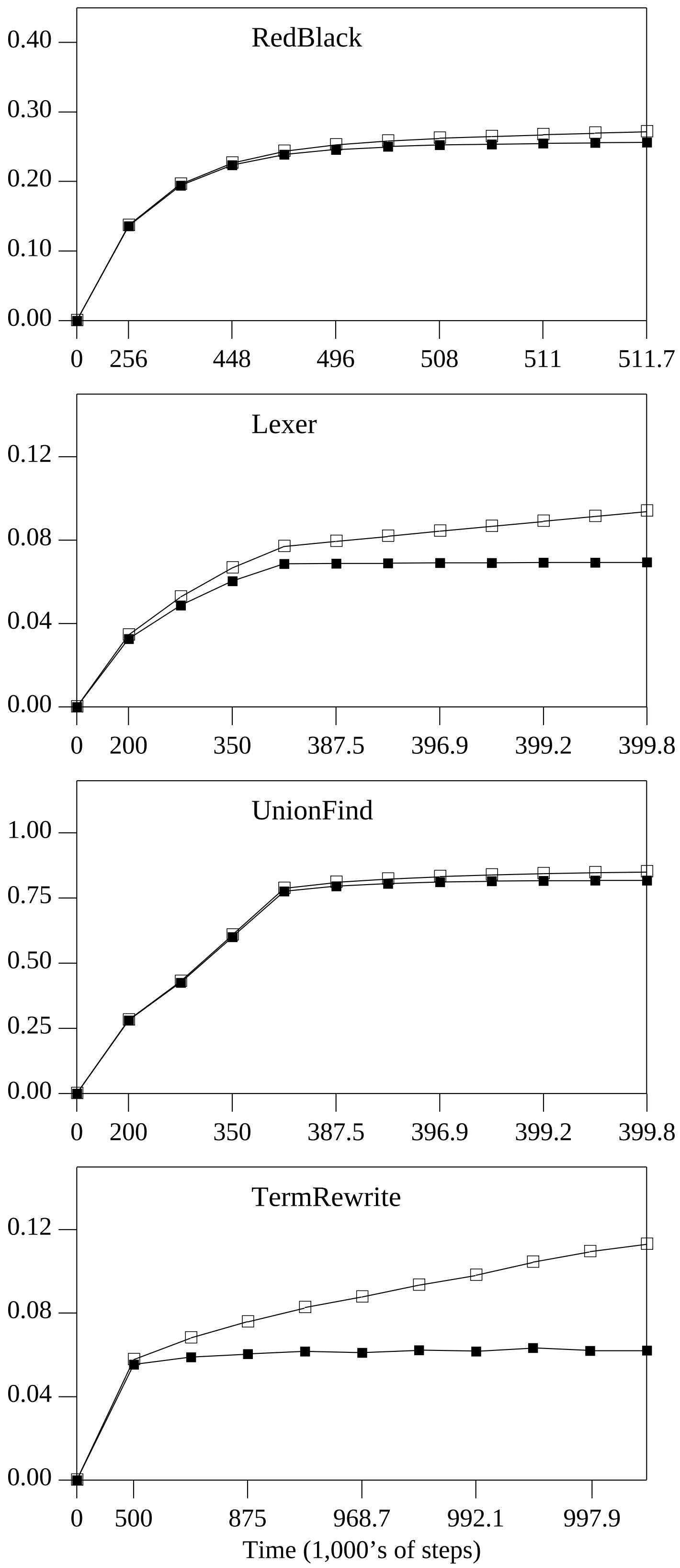
Many of our ideas were anticipated in Balzer's seminal EXDAMS system [Balzer69]. He uses a similar instrumentation method to log important events (first to a buffer and eventually to a file) for subsequent post-mortem analysis. Our ideas on using binding site information resemble his ''flowback analysis,'' which has also been revived in [Miller88]. A major difference in our work is that we collect event data only on request, which keeps execution overheads low enough to support interactive debugging; low-cost, conditionally-triggered events are exploited in a similar way by the Parasight system [Aral89].

Automatic instrumentation has been used for interactive debugging before. [Hanson78] describes how a

* To get an overall picture of memory demand on the machine, one must also add the size of the underlying compiler and run-time system (1.67MB for the standard system vs. 1.93MB for the system supporting debugging), and of the operating system.



Dynamic Memory Usage
Figure 4.



Overlapping of Checkpoint Storage
Figure 5.

generalized event mechanism can be built into a programming language (SNOBOL4) and used to support debugging. [Dybvig88] describes a LISP debugging mechanism based on instrumentation via a powerful macro system. We have previously used instrumentation in ML to produce execution profiles [Appel88].

[Johnson88] explores the use of continuations and stores as first-class objects in the language GL, and suggests their utility for debugging.

Many replay debuggers have been proposed, especially in connection with parallel programming systems [Curtis82,LeBlanc87,McDowell89], and the idea of using a log to govern or assist re-execution is well-established. One important issue in such systems is how to measure time: [Cargill87] proposed a hardware instruction counter; [Mellor-Crummey89] use assembly-code-level instrumentation to implement a counter in software. A higher-level software counter is easier to use with a compiler that performs significant rewriting or optimization. Another key issue is how to support checkpointing efficiently. A very thorough checkpointing system is proposed and analyzed in [Wilson89], which supports the idea that the whole history of very large systems can be preserved at acceptable cost.

10. Conclusions and Future Work

We have built a practical ML debugger based on source code instrumentation and time travel. The debugger provides novel solutions to the challenges posed by our compiler methodology, which prevents ordinary source-level debugger techniques from working. Execution time for instrumented code is only a few times slower than ordinary code. Space requirements for checkpointing functional state are minimized because continuations share storage. The space needed to keep track of changes to the mutable store is kept low by simple coordination with the garbage collector. Except for this and the use of **callcc**, our debugger is completely independent of the compiler's back end and the underlying machine. As a result it was relatively simple to write and would be trivial to port.

One obvious goal at present is to build up a body of users and experience using the debugger in "real life" applications. This will enable us to develop a set of realistic benchmarks representing typical debugging sessions; having these is a prerequisite for any serious performance analysis, in particular for testing alternative checkpoint caching strategies. We would also like to see whether time travel, which the debugger uses internally in a very comprehensive way, will prove equally valuable as a user-level facility. Finding suitable metaphors and methods for navigating through the history of a computation will become increasingly important as replay mechanisms become a common part of programming environments.

The main use of replay debugging to date has been in parallel programming systems, and we expect that our debugging approach will extend naturally to such systems, although some of our time-travel tricks may not carry over to a multi-thread environment. Our notion of event can easily be extended to cover various kinds of communication and synchronization operations; because we instrument code at source level our approach will mesh particularly well with languages that support such operations explicitly.

We believe that instrumentation will be a key technique in future debugging systems. Compilers are becoming increasingly aggressive in transforming code. A computation originally expressed by the programmer in the source language may be transformed into machine code that performs a quite different computation. This is normally acceptable if the new computation's external behavior is consistent with that of the original. But to debug the program effectively, we need the machine code to conform closely to the original program text. The best way to make the compiler produce such code is to extend the program's external behavior to make its internal state observable, e.g., by recording a trace of statement execution or permitting display and update of intermediate variables. When translating this extended program, the compiler will be constrained to produce machine code that mirrors the original computation to whatever degree of precision required. Source-level instrumentation is the natural way to produce the extended version of the source.

We are particularly interested in applying our approach to lazy and parallel languages with indeterminate execution order. Many such languages have transformational compilers of the type just discussed. In addition, they offer a further challenge: for debugging, it is often desirable to fix a particular repeatable execution order for a program. Again, this may be achieved by transforming the program, this time into one that performs the same computation as the original but is determinate, at least during replay. We believe instrumentation can be useful here as well.

Acknowledgements

David Tarditi implemented support for run-time resolution of polymorphic types. The type-checking of **callcc** in Standard ML is due to Bruce Duba and David MacQueen.

References

[Appel87a].
A.W. Appel and D.B. MacQueen, ''A Standard ML compiler,'' in *Functional Programming Languages and Computer Architecture*, ed. G. Kahn, LNCS , vol. 274, pp. 301-324, Springer Verlag, 1987.

[Appel87b].
A.W. Appel, ''Garbage collection can be faster than stack allocation,'' *Information Processing Letters*, vol. 25, no. 4, pp. 275-279, 1987.

[Appel88].
A.W. Appel, B.F. Duba, and D.B. MacQueen, ''Profiling in the presence of optimization and garbage collection,'' Technical Report CS-TR-197-88, Princeton University Dept. of Computer Science, 1988.

[Appel89a].
A.W. Appel, ''Runtime tags aren't necessary,'' *Lisp and Symbolic Computation*, vol. 2, pp. 153-162, 1989.

[Appel89b].
A.W. Appel, ''Continuation-passing, closure-passing style,'' *Sixteenth ACM Symp. on Principles of Programming Languages*, pp. 293-302, 1989.

[Aral89].
Z. Aral, I. Gertner, and G. Schaffer, ''Efficient debugging primitives for multiprocessors,'' *Proc. 3rd International Conf. on Architectural Support for Programming Languages and Operating Systems*, 1989.

[Balzer69].
R.M. Balzer, ''EXDAMS - EXtendable Debugging and Monitoring System,'' *AFIPS Proc. Spring Joint Computer Conference*, vol. 34, pp. 567-580, AFIPS Press, Arlington, VA, 1969.

[Bruegge85].
B. Bruegge, ''Adaptability and portability of symbolic debuggers,'' (Thesis) CMU-CS-85-174, Carnegie-Mellon University Dept. of Computer Science, Sept 1985.

[Cargill87].
T.A. Cargill and B.N. Locanthi, ''Cheap hardware support for software debugging and profiling,'' *Proc. SIGPLAN '87 Symposium on Compiler Construction*, pp. 82-83, June 1987.

[Curtis82].
R. Curtis and L. Wittie, ''Bugnet: A debugging system for parallel programming environments,'' *Proc. 3rd International Conf. on Distributed Computing Systems*, pp. 394-399, October 1982.

[Dybvig88].
R.K. Dybvig, D.P. Friedman, and C.T. Haynes, ''Expansion-Passing style: A general macro mechanism,'' *Lisp and Symbolic Computation*, vol. 1, pp. 53-75, 1988.

[Friedman84].
D.P. Friedman, C.T. Haynes, and E. Kohlbecker, ''Programming with continuations,'' in *Program transformation and programming environments*, ed. P. Pepper, pp. 263-274, Springer, 1984.

[Hanson78].
D.R. Hanson, ''Event associations in SNOBOL4 for program debugging,'' *Software Practice and Experience*, vol. 8, pp. 115-129, 1978.

[Hennessy82].
J. Hennessy, ''Symbolic debugging of optimized code,'' *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 323-344, July 1982.

[Johnson88].
G.F. Johnson and D. Duggan, ''Stores and partial continuations as first-class objects in a language and its environment,'' *Proc. 15th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, San Diego, CA, January 1988.

[LeBlanc87].
T.J. LeBlanc and J.M. Mellor-Crummey, ''Debugging parallel programs with Instant Replay,'' *IEEE Transactions on Computers*, vol. 36, no. 4, pp. 471-482, April 87.

[McDowell89].
C.E. McDowell and D.P. Helmbold, ''Debugging concurrent programs,'' *ACM Computing Surveys*, vol. 21, no. 4, pp. 593-622, December 1989.

[Mellor-Crummey89].
J.M. Mellor-Crummey and T.J. LeBlanc, ''A software instruction counter,'' *Proc. 3rd International Conf. on Architectural Support for Programming Languages and Operating Systems*, 1989.

[Miller88].
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*, pp. 135-144, Atlanta, Georgia, June 22-24, 1988.

[Milner90].
R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*, MIT Press, Cambridge, Mass., 1990.

[Shaw87].
Robert A. Shaw, ''Improving garbage collector performance in virtual memory,'' STAN-TR-87-323, Stanford University Computer Science Department, 1987.

[Wilson89].
P.R. Wilson and T.G. Moher, ''Demonic memory for process histories,'' *Proc. SIGPLAN 89 Conference on Programming Language Design and Implementation*, June 1989.

[Zellweger84].
P.T. Zellweger, ''Interactive source-level debugging of optimized programs,'' CSL-84-5, Xerox Corporation Palo Alto Research Center, May 1984.