J. Functional Programming **1** (1): 1–000, January 1993 © 1993 Cambridge University Press 1

A Debugger for Standard ML [1]

ANDREW TOLMACH
Dept. of Computer Science, Portland State University
P.O. Box 751, Portland, OR, USA 97207-0751
email: apt@cs.pdx.edu

ANDREW W. APPEL
Dept. of Computer Science, Princeton University,
35 Olden Street, Princeton, NJ USA 08544-2087
email: appel@cs.princeton.edu

Abstract

We have built a portable, instrumentation-based, replay debugger for the Standard ML of New Jersey compiler. Traditional "source-level" debuggers for compiled languages actually operate at machine level, which makes them complex, difficult to port, and intolerant of compiler optimization. For secure languages like ML, however, debugging support can be provided without reference to the underlying machine, by adding instrumentation to program source code before compilation. Because instrumented code is (almost) ordinary source, it can be processed by the ordinary compiler. Our debugger is thus independent from the underlying hardware and runtime system, and from the optimization strategies used by the compiler. The debugger also provides reverse execution, both as a user feature and an internal mechanism. Reverse execution is implemented using a checkpoint and replay system; checkpoints are represented primarily by first-class continuations.

1 Introduction

Most "source-level" debuggers for compiled languages actually operate at machine level. They rely on detailed knowledge of object-code formats, runtime layout, and hardware properties of the target machine. They also require accurate mappings between source and object code. These factors make machine-based debuggers complex, dependent on their compilers, runtime systems, and target machines, and intolerant of compiler optimizations that distort the source-to-object mapping.

We have built a debugger that uses a different approach, based on automated *source-code instrumentation.* This approach works by preprocessing source code to insert debugging statements at all potentially interesting points in the program. Because the instrumented code is (almost) ordinary source, it can be processed

[1] This work was completed while the first author was at Princeton University. A preliminary version of this paper appeared in *Procedings of the 1990 ACM Conference on Lisp and Functional Programming,* Nice France, June 1990. The work was supported, in part, by the National Science Foundation under grants CCR-8806121, CCR-8914570, and CCR-9002786.

by the ordinary compiler. The debugger is thus independent from the underlying hardware and runtime system and from the optimization strategies used by the compiler.

Debugger features that might appear hopelessly inefficient in the machine-based model are more plausible when instrumentation is used. One such feature is *reverse execution*—the ability to reset the program to an earlier state in its execution history. Reverse execution can also be used in the debugger implementation to decrease the overhead of more conventional debugging commands. We implement reverse execution using a checkpoint and replay system.

Our work is based on the Standard ML language (Milner et al., 1990), and the Standard ML of New Jersey (SML/NJ) compiler (Appel, 1992; Appel and MacQueen, 1991).[1] Although our instrumentation and time-travel methods are applicable to many systems, SML/NJ has a distinctive combination of characteristics that make it a particularly good target. First, ML is a secure language (Hoare, 1989); programs cannot "dump core" due to runtime typing or bounds errors. Security means that the behavior of all ML programs—even buggy ones—can be understood fully without reference to the underlying compiler implementation, runtime system, or machine. Second, the SML/NJ compiler performs extensive optimizations, which makes a traditional debugger implementation very difficult. SML/NJ uses continuation-passing style (CPS), which eliminates storage for dead variables; it performs tail-recursion elimination, which destroys information about the call chain; and it does not use a runtime stack. Third, SML/NJ supports efficient *first-class continuations*, which provide a cheap and elegant mechanism for checkpointing immutable variables and control state. Since most ML programs have few side-effects, and mutable variables are distinguished statically, a checkpoint in a typical computation can be described by a continuation plus a small amount of additional information describing the values of mutable variables. We assume basic familiarity with Standard ML in the remainder of this paper.

Our source-level debugger is mostly conventional: it includes support for breakpoints, value display, and call traceback. To implement these features, the debugger inserts instrumentation code at each interesting *event site*, e.g., just before function calls, just after function entries, and when local variables are bound. The instrumentation code is inserted by modifying the abstract syntax tree that the compiler produces from the user's source text. At each event, the instrumentation code can build an *event record* describing the event and preserving the values of any variables bound there. Static and dynamic links between event records enable the variable environment and the call chain to be reconstructed at runtime by simple debugger algorithms. The instrumentation code also checks whether a breakpoint has been set at the event, and if so, transfers control to an interactive debug monitor. This monitor is a special version of ML's top-level read-eval-print loop, modified so that

[1] Standard ML of New Jersey is being developed jointly at Princeton University and AT&T Bell Laboratories. Source code for the the compiler and debugger and binaries for several processors are available via anonymous FTP from `princeton.edu` (128.112.129.1), directory `pub/ml`, or from `research.att.com` (192.20.225.2), directory `dist/ml`.

expressions are evaluated in the binding environment corresponding to the current event.

The debugger also supports an unconventional feature: reverse execution. This feature enables users to investigate the origins of an error without having to re-execute their programs from the beginning. Execution events are given sequential *time* values, and users can set breakpoints at particular times in the future or the past. Time travel is also used by the debugger itself to implement conventional features. For example, times are used as indirect pointers to event records; the records themselves can be generated on demand by jumping back to the time of the event. Ordinary location breakpoints and "skipping" (single-stepping over, rather than into, function calls) can be simulated using time travel to perform a binary search over execution history, avoiding expensive per-event checks during forward execution. Internally, the debugger implements all program control mechanisms in terms of time travel.

Reverse execution is itself implemented by taking periodic checkpoints of program state and re-executing as necessary. The control state and the values of immutable identifiers are captured using first-class continuations. Mutable store information is captured separately by instrumenting store update operations to maintain a history log; log entries use special "weak" pointers so that objects referenced only by the log can be reclaimed. Stream input operations are also logged so that they can be repeated on re-execution. Since memory is finite, not all checkpoints taken can be saved; the debugger implements a checkpoint cache managed using simple heuristics.

The debugger is portable and independent of the underlying ML implementation and executes almost entirely within the source-language model. The only exception is a carefully controlled violation of the type system needed to pass runtime values to the debug monitor. Time travel relies on certain SML/NJ language features not found in official Standard ML (e.g., first-class continuations and weak pointers) but not on the details of their implementation. The debugger runs on all of SML/NJ's current hardware platforms, including the MIPS, SPARC, Intel 386, IBM RS6000, and MC68020.

Instrumentation makes code run about three times slower than normal and causes code size (and thus compile time) to increase by about five times. Programs being debugged require three to eight times more memory than normal programs. Although these are large resource demands, we estimate that the alternatives for debugging are also expensive. Any hypothetical machine-based debugger would have to switch off many of SML/NJ's optimizations. We estimate that the resulting code would run close to three times slower than normally-compiled code, and might require much more memory. Another possible approach to debugging is to use an interpreter: SML/NJ's runs 10 to 100 times slower than compiled code. Thus, the performance of our debugging approach appears competitive with the alternatives.

This paper describes the design, implementation, and performance of the instrumentation and replay mechanisms. Additional details may be found in Tolmach's Ph.D. thesis (1992). We have also used our instrumentation and replay approach to build a debugger for an extended version of ML that supports shared-memory concurrent programming (Tolmach, 1992; Tolmach and Appel, 1991).

2 Debugging ML

ML programmers can make good use of a conventional debugger, because the source-level model suitable for understanding the execution of an ML program is fairly conventional. In particular, although ML is often used in a mainly applicative style, users must be aware of evaluation order in any program that updates the mutable store, performs I/O, or uses exceptions. The possibility of side-effects also makes it natural to think of function application as a procedure call rather than as a λ-calculus β-substitution. Thus, for debugging purposes, ML is closer in character to C or PASCAL than it is to pure *lazy* functional languages such as Miranda (Turner, 1985) or Haskell (Hudak and Wadler, 1990), in which precise information about evaluation order is more likely to be a hindrance than a help.

2.1 Debugging Model

Our model for executing ML programs under debugger control is based on the notion of *events*, i.e., interesting junctures in program execution. Events occur at moments in execution where the user might wish to stop the program and examine its current state, or *context*. A context includes a source code *site*, an *environment*, mapping identifiers in scope at the source site to typed values, and a *call chain*, a list of contexts representing the "stack" of suspended function applications.[2] The user sees program execution under debugger control as a sequence of atomic events; the internal behavior of the program between events is invisible. Asynchronous phenomena that occur between two events, such as keyboard interrupts or uncaught arithmetic exceptions, are mapped to a neighboring event.

Every interesting change in context, e.g., the addition of a new variable into the environment or the entry of a new function onto the call chain, requires an event. Thus, events occur immediately before each function application, at the beginning of each function rule body, and immediately after each declaration has been evaluated. Since all branching in ML, conditional or otherwise, is expressed in terms of rule pattern matching, placing an event at the top of each rule body guarantees that there is an event after each branch and hence within each loop. There are no events prior to the application of constructors or to the formation of closures because these operations do not affect the context in an interesting way. There are also no events at function returns; some debuggers report the values returned by functions, but ours does not.[3]

Program events can be identified by logical *time* values, integers assigned consecutively to events as they occur during forward execution. Forward execution can be visualized as progress rightward along a time line marked off at each event (see

[2] Exception handling contexts also appear in the call chain.
[3] This restriction preserves tail-recursion in our instrumented code; see Section 4.5.

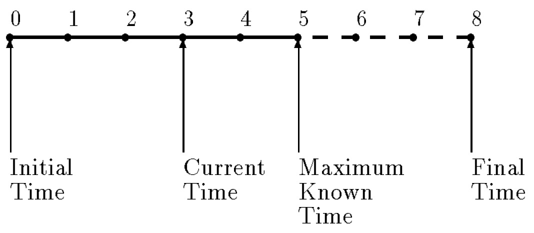

Fig. 1. A sample time line. The program has been executed as far as time 5, but the current time is presently reset to time 3, perhaps after having been reset to other values in the interim. Execution will terminate at time 8, either by reaching the end of program or by raising an uncaught exception, but this fact is not yet known to the debugger or, perhaps, the user.

Figure 1). Whenever the program is halted at an event, there is a well-defined *current time*, i.e., the time label assigned to the event when it was originally executed. To "execute in reverse" means to reestablish the context associated with an event that occurred at an earlier time. In this sense, directing execution to a specific state in the past or future of the computation is *time travel.*

2.2 Debugger Features

There is a small set of fundamental debugger control and query functions.

- Set a *location breakpoint* at a specified site in the source text. When the program reaches any event corresponding to that site, while executing in either direction, control passes to the debugger, with the *current context* set to the context for that event.
- *Advance* or *reset* to a specified time, starting from the current time, subject to currently set breakpoints. The time line expands to the right whenever the user advances the program to a target time that lies beyond the previous maximum known time. If the end of the program is reached or an uncaught exception occurs, the right endpoint of the time line is fixed at the *final time*; it is impossible to advance past this time, but resetting to a known time is always possible.
- Halt execution when a keyboard interrupt or uncaught exception is raised, setting the current context to the event where the interruption occurred.
- Obtain the values and types of identifiers in scope at the current context.
- Alter the current values of mutable variables in scope at the current context. The time line is truncated to the right of the current time, to avoid possible temporal paradoxes.
- Obtain the times and corresponding context information for all events on the call chain, allowing display of a call trace.
- Adjust the current context by moving up (or down) the call chain.

```
(* Some built-in functions on events *)
type time = int
val currentTime: unit -> time
    (* Returns the current time. *)
val caller : time -> time
    (* Given an event time within some function, returns time of the
       application event that invoked function. *)
val advanceTo: time -> unit
    (* Executes forward to the specified time. *)

(* Synthesized functions *)

fun singleStep () : unit = advanceTo(currentTime() + 1)

(* ``Skip'' over a call.
   If executed at an application event, continues single-stepping until
     that application event is no longer on the call chain.
   If executed at any other event, just single-steps. *)

fun skip () : unit =
    let val startTime = currentTime()
        fun startOnChain t =
             if t = startTime then
               true
             else if t < startTime then  (* use time ordering ! *)
               false
             else startOnChain(caller t)
    in singleStep();
       while startOnChain(currentTime()) do
         singleStep()
    end
```

Fig. 2. Synthesizing `singleStep` and `skip` from primitive functions.

Users interact with the debugger via an *interactive debug monitor*, which is a recursively invoked, specialized variant of the compiler's standard top-level loop. The monitor's top-level environment emulates the dynamic environment for the current user-program context, which allows the standard expression evaluator to be used to display, deconstruct, and operate on values in the current debugging context. Other debugging primitives, such as displaying the call history, controlling breakpoints, and restarting execution, are implemented as functions built into a pervasive debugging support environment and executed for side–effect. Like other ML functions, they can be embedded in more complicated user-defined functions, invoked from external files, assigned to new names or abbreviations, etc. As an example, Figure 2 shows how a single step function and a naive "skipping" function can be synthesized from built-in primitives that perform time travel and call-chain lookup.

```
let val pair = fn x => fn y => (x,y)
    val pair1 = pair 1
    val pairt = pair true
    val f = fn (h::t) => pairt h
in f [1,2,3]
end
```

Fig. 3. Example of dynamic types.

2.3 Values and Types

Displaying values in a manner appropriate to their type is non-trivial for a language that includes user-defined data types, first-class functions, functors, and polymorphic identifiers. There are obvious limits to the abilities of any general-purpose pretty printer to deal with user-defined types or with large aggregate data structures. Our approach is to let the user display values via the standard top-level expression evaluator, modified so that identifiers are looked up in the environment of the current context. This approach allows the user to deconstruct complex values by evaluating arbitrary ML expressions. The current context's environment is accessed using a special set of emulation functions. These functions translate an identifier lookup, which the ordinary top-level environment would handle by consulting a hash table, into a call to the debugger's lookup routine, implicitly parameterized by the current context. To the top-level loop and parser, this "synthetic" debugger environment looks just like a "real" symbol table.

Variables bound to functions have no directly printable "value." The expression evaluator can be used to examine a function's behavior by applying it to sample arguments. The debugger can also report the binding context, including source-code site, for an identifier; this feature, used recursively as necessary, enables the user to find and inspect the λ-expression that defines the code bound to a function-valued variable.

Displaying values of variables with polymorphic types is a major debugging challenge. Consider the program in Figure 3. Here `pair` has the polymorphic type scheme $\forall\alpha,\beta.\alpha \rightarrow \beta \rightarrow (\alpha \times \beta)$, and is applied first to an integer, later to a boolean. If the user stops at a breakpoint just inside an invocation of `pair`, after `x` has been bound to the actual argument, how is the debugger to determine the dynamic type of `x`, when the compiled code for `pair` is deliberately oblivious to its argument's type, and runtime values themselves do not carry precise type tags?

To solve this problem, the debugger relies on the fact that the dynamic type of a value doesn't change when it is passed to a function; i.e., the dynamic type of the formal parameter must match that of the actual argument (Appel, 1989a). To find the actual argument's type, the debugger looks up the call chain to find the event where the function was called and consults the corresponding code site. In this example, both applications of `pair` take as argument a constant of manifest type, so the dynamic type of `x` is immediately deducible. In general, more work may be required. Consider what happens if the user breaks execution just after `pairt` has been applied and `y` has been bound, and asks for `y`'s type. As before, the debugger looks up the call chain, and determines that the dynamic type of `y` is the same as that of `h`. But `h`'s dynamic type is not immediately obvious; it must be computed by a recursive application of the same procedure. By following another link in the call chain, the debugger can determine that since the argument to `f` has static (and dynamic) type `int list`, `h` must have dynamic type `int`. In general, the debugger may need to traverse an arbitrary number of links in the call chain.[4] Moreover, it is not always sufficient to consult the current call chain (Goldberg and Gloger, 1992). Suppose that the user, still halted at the same event after the binding of `y`, asks for the type of `x`. At this point, the application of `pair` to `true` is no longer on the current call chain; the debugger must retrieve the call chain that was current when `x` was *bound* before beginning the reconstruction process. Thus, the type reconstruction algorithm requires a mechanism to access the binding-time call chain of every variable in scope. An efficient mechanism for this purpose is described in Section 4.3. We believe that this procedure always terminates with a secure dynamic type, although we have not proved this fact formally.

Most source-level debuggers allow the user to alter the course of execution, usually by changing the values of variables. In ML, this feature makes good sense only for mutable `ref` and `array` variables. It is provided by permitting the user to evaluate assignments for their side-effects. Allowing other identifier values to be changed amounts to changing the program's source code, and would typically require dynamic recompilation.

A debugging session has exactly one history time line representing one consistent execution history. However, the debugger permits the user to reset to a time t before the maximum known time and then change the values of mutable variables. Since these changes might alter the course of execution, the debugger automatically resets the maximum known time to t, truncating the time line in order to avoid possible temporal paradoxes. Other user actions that might change execution history, such as altering an input stream during re-execution, are prohibited. To avoid complicating the user interface, we deliberately do not support an "undo/redo" system for speculative computing or "parallel worlds" of execution.

2.4 Debugger Interface

The debugger uses a screen, keyboard, and mouse-driven interface, implemented as an extension to the GNU Emacs editor, roughly in the style of `gdb` (Stallman and Pesch, 1991). It is independent of the debugger implementation and, to a large degree, of the specific functionality the debugger provides. Whenever execution halts under debugger control at a particular event, the corresponding source code is displayed with a pointer to the current site; the user can select other sites in the

[4] In principle, the debugger could avoid traversing recursive applications, since these can add no useful type reconstruction information. If we implemented this optimization, the complexity of the reconstruction process would be limited by the depth of static nesting rather than by the dynamic call depth.

source by scrolling and pointing at the text. Debugger commands can be issued using single keystrokes which are translated into the equivalent typed commands and passed to the debug monitor.

3 Debugging SML/NJ

The SML/NJ system poses many problems for conventional machine-based or interpretive debuggers. This section describes the compiler and runtime system and the difficulties they provoke.

3.1 Compiler and Runtime System

SML/NJ implements the full Standard ML language with a few trivial omissions and variants. It makes extensive internal use of continuations, and also exports them as first-class user objects, similar to those in Scheme (Clinger and Rees, 1991), but typed (Duba et al., 1991). Continuations are captured with the `callcc` primitive and invoked using the `throw` primitive.

SML/NJ compiles a program by performing a complex sequence of rewritings from one intermediate representation to another. Source programs are parsed into conventional abstract syntax trees, which are then elaborated and type-checked. Elaborated abstract syntax is translated into λ-*language*, an intermediate representation resembling an applied call-by-value λ-calculus. Various simplifications are made at this stage, notably the compilation of pattern matches into explicit test and branch code. The SML/NJ system includes a simple interpreter that operates directly on λ-language.

Next, λ-language is converted into the continuation-passing style (CPS) language. This is another λ-calculus-like language, but with all control flow made explicit. All function operands must be atomic, i.e., they must already have been explicitly evaluated and bound to a variable. Also, functions never return; they terminate by calling a continuation function representing "the rest of the program." Most of SML/NJ's optimizing is done on the CPS representation (Appel, 1992). CPS optimizations include in-line expansion, η-reduction and splitting, flattening of tuple arguments and uncurrying, and code hoisting.

After optimization, the CPS is further rewritten to convert references to free variables into explicit code to create and access heap-allocated closure records. Then, the CPS is translated to code for an abstract machine; a further optimization that chooses function argument registers is performed as part of this translation. Finally, the abstract machine code is translated to binary code for a specific target. Only this last phase is machine dependent. On RISC machines, target code generation includes instruction scheduling to fill branch and load/store delay slots.

In addition to these explicit optimizations, the SML/NJ compiler implicitly performs certain operations that other compilers might treat as elective optimizations. Live variable analysis is performed implicitly as part of CPS conversion: only "live" variables appear free in continuation functions, so "dead" variables never occupy space in continuation closures. Whereas a conventional compiler may view register assignment of local data as an optimization on the "standard" placement in activation records, SML/NJ looks at things the other way around: it assumes all data reside in registers unless explicitly needed in a closure record or spilled. These optimizations cannot be disabled; they are intrinsic to SML/NJ's compilation approach.

The SML/NJ system also contains a runtime component, implemented in C, that provides access to operating-system facilities, including I/O and signals, and a garbage collector for the heap. SML/NJ uses a two-generation copying collector (Appel, 1989b); frequent *minor* collections scavenge the younger generation, and occasional *major* collections scavenge the entire heap. The heap is very heavily used, because SML/NJ has no runtime stack of activation records. Its role is played by heap-allocated closures for continuation functions, which hold on to the values needed for continuing the computation of suspended functions higher up the call chain. Furthermore, closures are not stored into after their creation and initialization. This scheme makes execution of `callcc` almost free: the current continuation can be saved simply by taking a new pointer to a linked set of heap-allocated records, and there is no stack to copy. Of course, there are allocation and garbage collection costs associated with using a heap representation, although these costs might not be higher than those of stack management (Appel, 1987).

3.2 Problems with Machine-Based Debugging

The number and complexity of SML/NJ's optimizations and the nature of its runtime system make the prospect of writing a conventional machine-based debugger quite daunting. To provide a source-level interface, a machine-level debugger must have access to a bidirectional map between source program locations and object code addresses, and a map from source code variables to machine data locations (addresses and/or registers). In addition, the debugger must understand the runtime format of each data type it must print (e.g., strings, arrays, and user-defined types). Finally, it must be able to determine the current call chain (typically by knowing the runtime stack format) and represent it in terms of the source code using the maps described above. All of these tasks are difficult in SML/NJ.

Optimization greatly increases the complexity of the debugger's maps. A typical ML source function is split into a set of CPS functions; an elaborate mapping scheme would be needed to translate correctly between source-code and object-code locations. To avoid reporting incorrect data values, a similarly elaborate scheme would be needed to map source identifiers to values as they move in and out of registers and closures. In-line expansion, η-splitting, argument flattening, uncurrying, etc. all cause the source-to-object code mapping to become one-to-many, and the inverse object-to-source map to become many-to-one. Keeping track of these mapping changes would be fundamentally straightforward and should not carry any runtime cost, but it would be a substantial bookkeeping task even if we demanded merely truthful behavior from the debugger (Coutant et al., 1988). More seriously, previous research on debugging optimized code (Hennessy, 1982; Zellweger, 1984) suggests that it is impossible to provide cost-free expected debugger behavior in the presence of η-reductions, since they remove the reduced functions from the object code; either η-reduction must be prevented or an additional parameter to indicate which original function was applied must be passed at runtime, which might be expensive. And visible but "dead" variables must somehow be preserved despite CPS conversion.

Polymorphic functions prevent any simple static correlation between variable names and types, and runtime values are essentially untagged (although a very weak form of tagging is used to support garbage collection). Finally, there is no conventional stack, and it is not obvious how to extract call chain information from continuation function closures.

3.3 Interpreters

Interpretive environments offer a number of advantages for debugging. Since the interval between making a source edit and running the resulting program is typically shorter than in compiled systems, at least for small programs, interpreters encourage debugging by source-code modification and experimentation. Moreover, the interpreter's evaluator can be modified to support internal debugging features, such as tracing function calls, arguments, and return values, at relatively small cost in added execution time. Many Lisp systems have powerful and elaborate debugging environments based on interpretation (e.g., Interlisp (Teitelman, 1978)), and this approach has also been used for C (e.g., Saber-C (Kaufer et al., 1988)). Since interpreters are typically fairly machine-independent, debugging systems based on them are also portable.

Unfortunately, interpretation is usually one to two orders of magnitude slower than compiled code execution. To make interpretation a feasible basis for debugging, it must be possible to intermix interpreted and compiled procedures at link time, and preferably at runtime (without halting the program) (Goldberg and Robson, 1983). SML/NJ would require significant changes to support dynamic recompilation.

4 Source-code Instrumentation

Our ML debugger avoids the problems of conventional implementations by using an unconventional approach: it automatically instruments programs at compile time so that they can be interactively debugged at runtime. This approach derives from one of the oldest manual debugging techniques: inserting explicit code into the source program to print values at points of interest. In the absence of a source-level debugger, this "insert a PRINT statement" method may be the only one available. Instrumenting by hand can be crudely effective, but it has serious drawbacks, particularly when the location and nature of a bug are unknown. It is hard to predict at compile time what information will be of interest at runtime; indeed, the focus of interest often changes in the course of a debugging session. It is then necessary to add or alter the instrumentation, re-translate, and re-run the program. If the compilation cycle is slow, the method becomes tedious.

Moreover, although the source language is a flexible medium in which to describe what information to display, it is not always a convenient one. For example, few languages provide any direct method for inspecting the call chain. ML doesn't even provide a built-in mechanism for printing variables, because the necessary runtime type information is not normally available. Finally, to instrument a program by hand the user must alter the source text, which can be time-consuming and error-prone. Unless instrumentation is carefully planned as a part of the program from the beginning, adding it (and later removing it) can easily make the source code less readable and maintainable, and can itself introduce bugs.

If the program is large, and instrumentation is planned from the beginning, it may be reasonable to include support for more complex operations than simply printing the values of variables: a trace history, breakpoints and watch points, and even an interactive interface for controlling these features might be added. In fairly short order, the user will have built most of a debugger.

A better idea is to generate the debugging code automatically. Our debugger inserts simple and uniform instrumentation code that supports breakpointing, variable lookup, and control-flow tracing at arbitrary event sites. This instrumentation is itself written in (almost) Standard ML. Instrumentation can be performed as a source-to-source transformation prior to compiling; the resulting instrumented code is legal compiler input, and the instrumented program works as expected in spite of any compiler optimizations and with any back-end. Only one caveat is required: the added code is not quite valid ML because it violates the type system in order to pass the recorded state to the debugger. Runtime values do not carry type information, so they must be treated as having generic type `object`.[5] Fortunately, this violation does not weaken the overall security of the system significantly, because user programs with type errors are still prevented from executing, and other kinds of user-program errors cannot disrupt the instrumentation code or the debugger. In short, we have a simple, reliable, platform-independent debugging strategy.

This approach to debugging may seem awkward or inelegant. In fact, source modification is a natural way to cope with an aggressive optimizer. Optimizing compilers may make any changes to a program so long as the observable behavior of the program remains the same. Unfortunately, debuggers must expose the internal behavior of the original program, which may be altered by optimization. The instrumentation-based debugger solves this problem by transforming the original source into a new program in which the internal state of the original is made potentially observable at frequent intervals. This transformation constrains the compiler to maintain the original form of the computation.

The remainder of this section describes the instrumentation process in more detail. For simplicity, the discussion is given in terms of the core language subset shown in Figure 4.

[5] The compiler already has to bypass ML's type system in order to manipulate code objects and values of uncertain type in the top-level loop. These type violations mean that the compiler itself does not have the security property of well-typed ML programs, but the violations are few and isolated.

exp ::= *nconst*	(nullary constructors, i.e., constants)
uconst	(unary constructors)
var	(variables)
op	(primitive operators)
exp_{opr} † exp_{arg}	(function applications)
`let` *dec* `in` *exp* `end`	(local declarations)
`fn` $rule_1$ | … | $rule_n$	(functions)
(exp_1 , exp_2 , … , exp_n)	(tuples)
(exp_1 ; exp_2 ; … ; exp_n)	(sequences)
nconst ::= `0,1,2,...`	(integer constants)
`nil`	(empty list constructor)
uconst ::= `::`	(list cons constructor)
`ref`	(reference constructor)
op ::= `+,-,*,/`	(integer operators)
`!,:=`	(reference operators)
`array,update,sub`	(array operators)
rule ::= *pattern* `=>` † *exp*	
pattern ::= `_`	(wildcard, i.e., match anything)
var	(variable)
nconst	(constant)
(*uconst* *pattern*)	(constructions)
($pattern_1$, $pattern_2$, … , $pattern_n$)	(tuples)
dec ::= `val` † $pattern_1$ `=` exp_1 `and` … `and` $pattern_n$ `=` exp_n	(ordinary declarations)
`val rec` † var_1 `=` exp_1 `and` … `and` var_n `=` exp_n	(recursive declarations)
dec_1;…;dec_n	(sequences)

Fig. 4. A subset of Core Standard ML. Primitive operators and constructors whose arguments are pairs can be written in infix, e.g., `a+b` for `+(a,b)`, and `[`exp_1 , exp_2 , … , exp_n`]` is syntactic sugar for the pattern or expression exp_1`::(`exp_2`::(`…`::(`exp_n`::nil)))`. Superscript daggers (†) indicate the positions of the source-code sites described in the text. Semicolons separating members of a sequence of declarations may be omitted.

4.1 Sites

Instrumentation code is placed at each event site; the event "happens" at runtime when the instrumentation code is executed. Each site has an associated *site type*. For the subset language, the debugger defines **`FN`** sites, located at the top of each function rule body; **`APP`** sites, located just prior to each function application; and **`VAL`** and **`VALREC`** sites, located just after each **`val`** or **`val rec`** declaration. To debug the full language, a somewhat larger set of site types is defined. There is a well-defined mapping from sites to source-code locations, illustrated in Figure 4 for the subset language. There is also an inverse mapping from each source-code location to the nearest neighboring sites. **`FN`**, **`VAL`**, and **`VAL REC`** sites are called *binding sites*, since

```
let val¹ cons = fn (x,y) =>² x :: y
    val³ rev =
      fn list =>⁴
        let val rec⁵ r =
              fn (h::t,a) =>⁶
                   r⁸(t, cons⁷(h,a))
               | (nil,a) =>⁹ a
        in r¹⁰(list,nil)
        end
in rev¹¹ [1,2,3]
end
```

Fig. 5. The rev function.

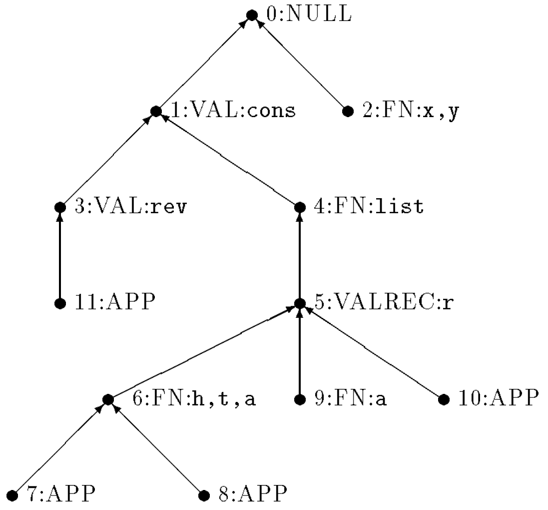

Fig. 6. Event sites for `rev`. Each binding site is annotated with *site number:site type:bound variables*. Arrows connect each site to the site corresponding to its immediately enclosing binding under ML's standard static scoping rules.

identifiers may be bound there; the names of these identifiers can be determined by examining the source code at the site. In our implementation, a site actually points to a node in the program's abstract syntax tree, and its associated bound variables can be extracted from the node. Each site in a program is given a unique identifying *site number*, which can be used to index tables or lists of site information.

Figure 5 shows a fragment of code that reverses a short list; this fragment serves as a running example in the remainder of this section. The program is fairly realistic except for the redefinition of the built-in cons operator (`::`) as the `cons` function, which has been introduced for illustrative purposes only. The superscripts are site numbers; the corresponding sites are described in Figure 6, organized into a tree showing how sites are related under static scoping. Each branch in the definition of `val rec r` has a separate `FN` site; these sites would be present even for branches

```
(* Basic types *)
type siteNumber = int
type time = int
type value = object

(* Event records *)
datatype eventRecord =
     EVENT of {time: time,
               siteNumber: siteNumber,
               binding: eventRecord,
               caller: eventRecord option,
               boundValues: value list}
   | NULLEVENT  (* corresponds to siteNumber 0 *)
```

Fig. 7. A naive `eventRecord` data type.

that failed to bind any variables. Constructors such as `::` are *not* instrumented, because they operate atomically; `cons`, on the other hand, is instrumented.

4.2 Event Code

The code at each site must determine whether the user has requested a halt at the current event; if so, it stops the user program and passes context information about the event to the debugger. In a conventional debugger, it is usual to specify where to halt by giving a source code site; we call this a "location breakpoint." In a time-travel debugger, however, it is also very useful to be able to specify halts at a particular *time* (before or after the current time). Deciding whether to break at an event is thus expressed as a test of the current time against a particular *target time* together with a check of whether a location breakpoint is set for this site.

The context information that must be retrievable at an event includes the event site; the current call chain, i.e., the identity of each pending `APP` event; the current values and types of all identifiers in scope at this site, or at any site on the call chain; and the call chain that was current when each of these identifiers was bound. The context at any given event is only incrementally different from that at a previous event; e.g., a variable is added to the environment or a call is added to the call chain. Thus, the context can be succinctly described by a pointer to a previous context plus the incremental changes.

A naive approach to obtaining context information would be for instrumentation code to build an *event record* at each event, whether or not a breakpoint occurs; the contents of this record are shown in Figure 7. The `boundValues` list contains an entry for each identifier bound at the event, if any. The `caller` field is present only for `FN` (top-of-function) events; it points to the record for the `APP` event that caused the function to be entered. The `binding` field points to the record that describes the most recent lexically-enclosing binding event; it is the dynamic analogue of the static scope nesting relation shown in Figure 6.

From a given event record, the debugger can determine the source location of the corresponding site (using a table built during instrumentation; see Section 4.4) and the values of the identifiers bound there. By traversing the *binding chain* defined by the `binding` pointers, it can find the value of any identifier in scope at the initially given event. As a byproduct of this search, the debugger also determines the event at which the identifier was bound. This information is useful for identifiers representing functions, whose values cannot be printed. It also gives access to the call chain that was current when the value was bound, as required for dynamic type reconstruction (see Section 2.3). Finding an identifier value traverses $O(d)$ `binding` pointers, where d is the static nesting depth of the initial event site, which is at worst proportional to the size of the source text.

Identifier lookup can start from *any* event record, so if the debugger can obtain the event record for an event on the call chain, it can display the value of any variable in scope at that event. To find these call chain event records, the debugger relies on the following properties of the instrumentation.

- Each function rule body has exactly one `FN` site, located at the beginning of the body.
- Each `FN` event is treated as a binding event (even if the rule binds no identifiers) and appears on the binding chain for every event in the function body.

Thus, given an event record, the debugger can find the record of the caller's application event by following the chain of `binding` pointers until a `FN` event is reached and then following the `caller` pointer. Iterating this process produces the event record of the caller's caller, and so forth.

The `caller` and `binding` fields are similar to the control (or dynamic) and access (or static) links, respectively, used to link activation records in languages with nested procedures, such as Pascal (Aho et al., 1986, Chapter 7). A Pascal compiler typically generates code to maintain static links and relies on the hardware stack to maintain dynamic links; SML/NJ generates code to maintain similar information in the form of closure and continuation pointers. We, however, must invent source-language instrumentation to do both jobs.

This naive approach incurs an expensive space penalty, especially in the case of tail-recursive functions. These functions remain tail-recursive, thanks to our policy of not placing events *after* applications, so the compiler can continue to avoid building a continuation closure for the recursive call. But the values that would have been referenced from this closure, which would ordinarily become garbage, are now stored explicitly into event records. The resulting space penalty is proportional to the depth of dynamic function call nesting (*before* tail-recursion elimination) and can be substantial for deeply recursive programs with large data structures. Moreover, we also pay this penalty for each in-scope binding. Measurements of the space overhead incurred by this naive approach are given in Section 8.1.

4.3 Lazy Event Record Construction

The fundamental drawback of the naive approach is that it builds a complete record of the values bound at each event even though most of these values are never needed

```
(* Lazily-generated event records *)
datatype eventRecord =
     EVENT of {siteNumber: siteNumber,
               binding: time,
               boundValues: value list}

val currentTime : time ref
val targetTime : time ref

val breakWanted : bool array

fun event(siteNumber,binding,boundValues) : time =
    let val newTime = !currentTime + 1
    in currentTime := newTime;
       if (newTime = !targetTime) orelse
          (breakWanted sub siteNumber) then
            break (EVENT{siteNumber=siteNumber,
                         binding=binding,
                         boundValues=boundValues})
       else ();
       newTime
     end
```

Fig. 8. Lazy `eventRecord` data structure and event instrumentation code. Other types are as in Figure 7. The `break` function is described in Figure 13.

by the debugger. A more elegant solution is to use time travel *internally* to recreate values lazily, upon demand. During forward execution, debugger instrumentation generates an event record only if a `break` is taken at the event; otherwise, the *time* of the event is preserved but no values are recorded. If more detailed information proves to be needed, the debugger can jump back to the event in question and collect it. This approach makes *controlled execution*, i.e., execution under debugger control but with no debugging queries, cheaper in time and space at the possible expense of longer times for query operations.

The event record has a revised form (see Figure 8). The explicit `binding` pointer is replaced by the *time* of the relevant binding event. To obtain a record on the binding chain, the debugger internally jumps back to the recorded time, generating a full event record, and jumps forward again to the original time. The `caller` pointer is unnecessary because times provide a sequential ordering for events. Since each FN event is immediately preceded by its corresponding `APP` event in the calling function, the record for that `APP` event can be obtained simply by internally jumping back one time unit.

The process of obtaining an event record by jumping back in time is encapsulated into an internal function

```
eventRecordAt: time -> eventRecord
```

Time travel performed during execution of this function is invisible to the user, except perhaps for a delay in evaluating queries.

Figure 8 shows the lazy instrumentation code inserted at each site in the form of an `event` function, which is ordinarily expanded in line to improve execution speed. Location breakpoints are established by setting the appropriate entry in `breakWanted`, an array of booleans indexed by site number. Note that current and target times are simply ML integer `ref`s. Overflow of time values is a potential problem; SML/NJ integers occupy only 31 bits, which allows for 2.1×10^9 events in a single debugged execution. Using pessimistic assumptions, this number should allow debugging a program that takes up to about 15 minutes of CPU time (not including garbage-collection time) on a typical workstation. Unfortunately, any form of multiple-precision counter appears to be much more expensive; we eagerly await 64-bit machine architectures.

4.4 Instrumentation Algorithm

Figure 9 describes the instrumentation process for the subset language of Figure 4 as a set of mutually recursive instrumenting functions that transform the concrete syntax of expressions, rules, and declarations. Function $\mathcal{E}$ takes a pair of arguments: a source-code *expression* to be instrumented and an auxiliary source-code expression *lbexp* that evaluates (at runtime) to the `binding` pointer; it returns the instrumented version of the first source-code expression. Function $\mathcal{R}$ is similar except that its first argument is a source-code *rule*. Function $\mathcal{D}$ takes a source-code *declaration* and *lbexp* expression as arguments and returns a pair: the instrumented declaration and a new value for *lbexp* to be used within the scope of the declaration. A top-level declaration *dec* is instrumented by calling $\mathcal{D}(dec, \mathtt{0})$. The auxiliary metafunction $\mathcal{S}$ takes a site type and source-code position (represented by †) as arguments, creates a new site with that type and position, and returns the new site's number. A table of all sites so created is produced as a by-product of the instrumentation process.

Most of the transformations are straightforward. Application expressions are complicated slightly because the instrumented code must evaluate both the operator and argument subexpressions *before* executing the application event. The most complicated case is for `val rec` declarations: the name of a recursive function is visible inside the function's own body, but the function's value is not available until after the `val rec` binding has been executed, so execution of the binding event needs to be deferred until then. Fortunately, the *time* of this event can be calculated in advance, since the body of a `val rec` is always a non-executable `fn` expression.[6]

Figure 10 shows the instrumented version of the `rev` program of Figure 5, exactly as it would be produced by the execution of the algorithm in Figure 9. Site numbers are as in Figures 5 and 6. The algorithm may produce unnecessary temporaries (e.g., many of the instances of `opr` and `arg`) and other bindings (e.g., `event_2`), but these are removed by the optimizer. `event` is expanded in line, so the code expands by an even larger factor than shown here. Also, Figure 10 is not strictly legal Standard

[6] In practice, this technique is unnecessary because `VALREC` events are coalesced into neighboring `APP` events (see Section 4.6).

$$
\begin{array}{lcl}
\mathcal{E}(nconst, lbexp) & = & nconst \\
\mathcal{E}(uconst, lbexp) & = & uconst \\
\mathcal{E}(var, lbexp) & = & var \\
\mathcal{E}(op, lbexp) & = & op \\
\mathcal{E}(exp_{opr} \;^{\dagger}\; exp_{arg}, lbexp) & = & \texttt{let val opr = } \mathcal{E}(exp_{opr}, lbexp) \\
 & & \quad\;\; \texttt{and arg = } \mathcal{E}(exp_{arg}, lbexp) \\
 & & \texttt{in event(}\mathcal{S}(\texttt{APP}, \dagger)\texttt{,}lbexp\texttt{,nil);} \\
 & & \quad \texttt{opr arg} \\
 & & \texttt{end} \\
\mathcal{E}(\texttt{let } dec \texttt{ in } exp \texttt{ end}, lbexp) & = & \texttt{let } dec' \texttt{ in } \mathcal{E}(exp, lbexp') \texttt{ end} \\
 & & \qquad where\ (dec', lbexp') = \mathcal{D}(dec, lbexp) \\
\mathcal{E}(\texttt{fn } rule_1 \mid \ldots \mid rule_n, lbexp) & = & \texttt{fn } \mathcal{R}(rule_1, lbexp) \mid \ldots \mid \mathcal{R}(rule_n, lbexp) \\
\mathcal{E}(\texttt{(}exp_1\texttt{,}exp_2\texttt{,}\ldots\texttt{,}exp_n\texttt{)}, lbexp) & = & \texttt{(}\mathcal{E}(exp_1, lbexp)\texttt{,}\mathcal{E}(exp_2, lbexp)\texttt{,}\ldots\texttt{,} \\
 & & \qquad \mathcal{E}(exp_n, lbexp)\texttt{)} \\
\mathcal{E}(\texttt{(}exp_1\texttt{;}exp_2\texttt{;}\ldots\texttt{;}exp_n\texttt{)}, lbexp) & = & \texttt{(}\mathcal{E}(exp_1, lbexp)\texttt{;}\mathcal{E}(exp_2, lbexp)\texttt{;}\ldots\texttt{;} \\
 & & \qquad \mathcal{E}(exp_n, lbexp)\texttt{)}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{R}(pattern \texttt{ =>}^{\dagger} exp, lbexp) \quad = \\
\qquad pattern \texttt{ =>} \\
\qquad\quad \texttt{let val event\_}n \texttt{ = event(}n\texttt{,}lbexp\texttt{,}\ vars\texttt{)} \\
\qquad\quad \texttt{in } \mathcal{E}(exp, \texttt{event\_}n) \\
\qquad\quad \texttt{end} \\
\qquad\qquad where\ n = \mathcal{S}(\texttt{FN}, \dagger)\ and\ vars\ = list\ of\ variables\ bound\ in\ pattern
\end{array}
$$

$$
\begin{array}{l}
\mathcal{D}(\texttt{val}^{\dagger} pattern_1 \texttt{ = } exp_1 \texttt{ and } \ldots \texttt{ and } pattern_m \texttt{ = } exp_m, lbexp) \quad = \\
\qquad \texttt{(val } pattern_1 \texttt{ = } \mathcal{E}(exp_1, lbexp) \texttt{ and } \ldots \texttt{ and } pattern_m \texttt{ = } \mathcal{E}(exp_m, lbexp)\texttt{;} \\
\qquad\quad \texttt{val event\_}n \texttt{ = event(}n\texttt{,}lbexp\texttt{,}\ vars\texttt{),} \\
\qquad \texttt{event\_}n\texttt{)} \\
\qquad\qquad where\ n = \mathcal{S}(\texttt{VAL}, \dagger) \\
\qquad\qquad and\ vars\ = list\ of\ variables\ bound\ in\ pattern_1, \ldots, pattern_m \\
\mathcal{D}(\texttt{val rec}^{\dagger} var_1 \texttt{ = } exp_1 \texttt{ and } \ldots \texttt{ and } var_m \texttt{ = } exp_m, lbexp) \quad = \\
\qquad \texttt{(val event\_}n\texttt{\_time = !currentTime + 1;} \\
\qquad\quad \texttt{val rec } var_1 \texttt{ = } \mathcal{E}(exp_1, \texttt{event\_}n\texttt{\_time}) \\
\qquad\qquad \texttt{and } \ldots \\
\qquad\qquad \texttt{and } var_m \texttt{ = } \mathcal{E}(exp_m, \texttt{event\_}n\texttt{\_time})\texttt{;} \\
\qquad\quad \texttt{val event\_}n \texttt{ = event(}n\texttt{,}lbexp\texttt{,[}var_1\texttt{,}\ldots\texttt{,}var_m\texttt{]);} \\
\qquad \texttt{event\_}n\texttt{)} \\
\qquad\qquad where\ n = \mathcal{S}(\texttt{VALREC}, \dagger) \\
\mathcal{D}(dec_1\texttt{;}dec_2\texttt{;}\ldots\texttt{;}dec_n, lbexp) \quad = \quad (dec'_1\texttt{;}\ dec'_2\texttt{;}\ldots\texttt{;}\ dec'_n, lbexp'_n) \\
\qquad\qquad where\ \ (dec'_1, lbexp'_1) = \mathcal{D}(dec_1, lbexp) \\
\qquad\qquad and\ \ \ \ (dec'_2, lbexp'_2) = \mathcal{D}(dec_2, lbexp'_1) \\
\qquad\qquad and\ \ \ \ \ldots \\
\qquad\qquad and\ \ \ \ (dec'_n, lbexp'_n) = \mathcal{D}(dec_n, lbexp'_{n-1})
\end{array}
$$

Fig. 9. Instrumenting functions $\mathcal{E}$,$\mathcal{R}$, and $\mathcal{D}$. Fragments of concrete syntax are shown in `typewriter font` and metavariables representing source text are shown in *slanted font*.

ML code, because the lists of values passed to `event` contain entries having multiple types. In practice, these are cast to a generic `object` type and are treated as vectors (immutable arrays) rather than lists.

The debugger instruments code by transforming the compiler's elaborated abstract syntax representation, after parsing and type checking, but before translation into λ-language. A more direct method might be to pre-process the source code before feeding it to the compiler, but our approach avoids parsing the source twice. ML is a complicated language to parse; neither the effort required to rewrite or extract the parser nor the inefficiency implied by parsing twice was attractive. The only serious disadvantage of our approach is that derived forms, such as `fun`, have already been expanded into core forms before instrumentation is performed, making it difficult to assign source-code locations to event sites accurately.

4.5 Performance Effects

How much will lazy instrumentation slow down code, in the overwhelmingly common case when a `break` does not occur? The direct costs are clear: `currentTime` must be incremented and compared against `targetTime`, and the `breakWanted` array must be checked. The indirect costs of executing this code are more subtle. Each `event_`n binding time variable must be kept live as long as the associated binding is live, which increases demand for registers and the size of closures. Fortunately, the total number of such variables live at any one time is limited by the depth of static nesting in the program, which is typically small. In addition, the addresses of `currentTime`, `targetTime`, `breakWanted`, and the `break` routine must be kept live throughout the code, which puts further pressure on registers and closures.

If `break` is called, the compiler must build a continuation closure to pass to it, which can be costly if there are many live variables.[7] It is desirable to avoid building this closure if the break test fails; unfortunately, the optimizer occasionally hoists the closure creation above the break test.

Another subtle effect is that every named identifier must now remain live from its binding point until the last event that may put its value on a `boundValues` list. Again, more registers or closure space may be needed, although most of these values may be live anyway, since the distance from binding to reporting event is usually short. More interestingly, the liveness requirement also applies to identifiers that the optimizer would ordinarily eliminate by constant folding or η-reduction. In such cases, the original optimization may still take place, with the resuscitated variables used *only* to fill in event records.

Function in-lining and its special case, loop unrolling, are still possible, but are less likely to be invoked by the optimizer's heuristic because of the growth in function size caused by the insertion of instrumentation. When in-lining does occur,

[7] This continuation is constructed and passed in the CPS version of the program generated internally by the compiler. Essentially the same continuation is subsequently "reified" into a user-level continuation by a `callcc` within the body of `break`; see Section 6.

```
let
  val cons =
      fn (x,y) =>
        let val event_2 = event(2,0,[x,y])
        in x::y
        end
  val event_1 = event(1,0,[cons])
  val rev =
      fn list =>
        let val event_4 = event(4,event_1,[list])
        in let
             val event_5_time = !currentTime + 1
             val rec r =
                 fn (h::t,a) =>
                   let val event_6 =
                             event(6,event_5_time,[h,t,a])
                   in let val opr = r
                          and arg = (t, let val opr = cons
                                            and arg = (h,a)
                                            in event(7,event_6,nil);
                                               opr arg
                                            end)
                      in event(8,event_6,nil);
                         opr arg
                      end
                   end
                  | (nil,a) =>
                   let val event_9 =
                             event(9,event_5_time,[a])
                   in a
                   end
             val event_5 = event(5,event_4,[r])
           in let val opr = r and arg = (list,nil)
              in event(10,event_5,nil);
                 opr arg
              end
           end
        end
  val event_3 = event(3,event_1,[rev])
in let val opr = rev and arg = [1,2,3]
   in event(11,event_3,nil);
      opr arg
   end
end
```

Fig. 10. Instrumented version of **`rev`**. Original program code is shown in **`bold typewriter font`** and instrumentation code is shown in `light typewriter font`. Event types are shown in Figure 6.

the instrumentation is copied along with the original code, so the pattern of event records is unaffected.

It is instructive to examine the instrumented code generated for `rev`. The original program is optimized away to almost nothing by a combination of loop unrolling and constant propagation; all that remains to do at runtime is to cons together the result list `[3,2,1]`.The instrumented version, on the other hand, executes the full computation in its original form plus all the debugging support code. No in-line expansion is performed; the compiler doesn't even η-reduce the function `cons`, although it is η-split, so that the call from inside `r` can be optimized to avoid packing its arguments into a pair.

4.6 Improvements and Extensions

In practice, the debugger instruments significantly fewer sites than shown here. Adjacent sites in the binding tree are *coalesced* with neighboring sites whenever they lie in the same basic block. In particular, `VAL` and `VALREC` events are always coalesced with neighboring `APP` and `FN` events. The resulting "augmented" sites are instrumented to build event records containing values for the identifiers bound at all the constituent original sites. Coalescing improves runtime performance by reducing the number of calls to `event` and the number of event records built.

To handle the full ML language, the debugger uses a number of other site types. Events at type and infix declarations, and at module declarations and functor applications, permit the debugger to emulate environments correctly; events at `raise` and `handle` expressions support exception tracing, and so forth. Most of these events are coalesced with others, so they cost little at runtime. In some cases, features in the full language merit new site types even if they could be handled with the core set of types. For example, `case` expressions could be treated as function applications, but doing so would require two event executions per case execution. Since the "function" body is only called from one site, there is no need for a separate `APP` event at runtime; removing it halves the execution overhead of instrumenting. Since `if-then-else` expressions are simply derived forms of `case` expressions, they benefit from this optimization as well. In addition, `APP` events for arithmetic primitives can be omitted to improve efficiency, although this technique reduces the accuracy of the call chain information provided if an uncaught arithmetic exception occurs.

5 Speculative Computation and Time-Line Search

Time travel can be used for *speculative* as well as retrospective computation. If the user wishes to halt in a particular state, there is no need to make sure that the halt occurs during forward execution. If the target state is overshot, the user can back up to it; indeed, it is often easier to identify the target after the fact.

For example, suppose the program has a loop, with a `ref` cell `c` used as the loop counter, and the user wants to halt the program when `c` has just been incremented to a particular value `v`. Assuming that the counter is never decremented, the user can perform a two-step procedure to direct the program to this time. First, the user

```
val timeDelta = ...

fun advanceToRealizationTime (realizedAt:unit -> time) =
  let fun narrow(falseTime,trueTime) =
       if falseTime+1 < trueTime then
         let val targetTime = (falseTime + trueTime) div 2
         in resetTo(targetTime);
            let val trueTime' = realizedAt()
            in if trueTime' < infinity then
                 narrow(falseTime, trueTime')
               else narrow(targetTime,trueTime)
       else resetTo(trueTime)
      fun expand() =
       let val startTime = currentTime()
       in advanceTo(startTime + timeDelta);
          let val trueTime = realizedAt()
          in if trueTime < infinity then     (* predicate now true *)
               narrow(startTime,trueTime)
             else if currentTime() < !finalTime then
               expand()                      (* predicate not yet true *)
             else ()                         (* predicate never true *)
          end
        end
  in expand()
  end
```

Fig. 11. Finding realization time for monotonic predicate by binary search.

repeatedly executes forward at full speed, stopping periodically to check the value of `c`, until `!c >= v`. At this point, the target time lies between the current time, t_{late}, and the time of the last unsuccessful check, t_{early}. The user then performs binary search on the temporal interval $(t_{early}, t_{late}]$ to pinpoint the earliest time at which `!c = v`. This procedure is likely to be faster than checking the value of `c` at every time step during forward execution, even if there is built-in support for doing the check efficiently.

Call t the *realization time* for a predicate P if P first becomes true at t. The binary-search technique can be automated to find the realization time for any monotone predicate on execution states, i.e., any predicate that stays true once it becomes true. Moreover, the algorithm that directs the search can be abstracted into a user-level function, `advanceToRealizationTime`, that takes P as an argument. A simplified version of the code for this function is shown in Figure 11. The `realizedAt` function consults the current state to determine if the predicate defining the desired target state is true. If so, it returns the earliest time for which it knows the predicate to be true; this might be simply the current time, but some predicates can return better bounds. If not, it returns `infinity`.

The debugger uses this function internally to implement certain targeted execution commands as binary searches over the execution time line. One important example is the implementation of location breakpoints. Suppose we modify the code for `event` to maintain an auxiliary array `lastTimes`, indexed by site, and holding the time of the last event executed at each site. Forward execution from an initial time `t0` to a location breakpoint at site `s` can now be implemented by invoking `advanceToRealizationTime` with the function:

```
fn () =>
   let val last = lastTimes sub s
   in if last > t0 then last else infinity
   end
```

Now the debugger can stop checking the `breakWanted` array at each event (see Figure 8), so that the test for whether to `break` is just a comparison of the current and target times. Of course, the debugger must also now update `lastTimes` at each event, so this technique may not save much time during controlled execution. However, `lastTimes` can also be used to support location-based breakpointing during reverse execution, since invoking

```
resetTo(lastTimes sub s)
```

resets the current time directly to the desired breakpoint.

A more sophisticated application is to a "skipping" function. The naive implementation of this function by single stepping, shown in Figure 2, can be very slow; a skip may involve an arbitrary number of steps, and the overhead of single-stepping and checking the call chain must be paid for each of them. To implement skip via speculative execution, we note that the `startOnChain` function is actually a monotone predicate on the contents of the call chain, so binary search can be used in place of sequential search.[8] The revised `skip` function is shown in Figure 12. Interestingly, time-travel primitives will be invoked recursively to determine the contents of the call chain when evaluating the skip predicate.

In principle, users can apply binary search to predicates of their own design. At present, however, we do not provide a way to test the values of local program variables within predicates. Some support for dynamic scoping and typing of variable names is required; such support does not fit easily into ML, but could be provided via special debugger functions that explicitly query, extract, and type-test values from the current environment.

User-supplied predicates can be arbitrarily complicated, and arbitrarily expensive to compute. The advantages of the binary search approach increase with the cost of predicate evaluation. On the other hand, the user will also typically pay a price in "extra" speculative execution beyond the actual realization time, though this execution will not be wasted if the user later decides to proceed forward from the breakpoint. The amount of "extra" computation is bounded by the `timeDelta` parameter in Figure 11.

[8] This method doesn't work if the user program contains `callccs`, since these can be used to switch call chains (e.g., to implement coroutines), which destroys the monotonicity of the predicate.

```
fun skip () =
    let val startTime = currentTime()
        fun pred () =
           let fun startOnChain (t,bound) =
                 if t = startTime then
                   infinity
                 else if t < startTime then
                   bound
                 else startOnChain(caller t,t)
               val ct = currentTime()
           in startOnChain(ct,ct)
           end
    in singleStep();
       advanceToRealizationTime(pred)
    end
```

Fig. 12. Implementing `skip` using speculative execution.

6 Controlling Program Execution

When the user program is halted, the debug monitor's code is active, and vice-versa; transitions between debug monitor and program control are essentially coroutine hand-offs, implemented with `callcc` in a well-known fashion (Wand, 1980). Simplified code for the basic control functions is shown in Figure 13. `setCompilationUnit` sets the debugger ready to execute a user program. Control passes to that program via `recordTo` or `replayTo` when the user issues a forward execution command and back to the monitor when the instrumented code calls `break`. The distinction between recording and replaying governs whether logs describing changes to the mutable store or external inputs are written or read; see Section 7.

During recording, it is possible that the user program will be interrupted before reaching the `targetTime` because the end of the program is reached, an uncaught exception is raised, the program is signaled, or `CTRL/C` is typed. In these cases, control is still passed to the debug monitor via `break` with a suitable event record. For end of program, this record contains a special `END` event. For uncaught exceptions, it contains a "pseudo-event" `UNCAUGHT` lacking a code location; the debugger must jump back one time unit to find the last real event and pinpoint the source of the exception.

The debugger handles CTRL/C from the keyboard via a special ML signal handler.[9] Interrupts are asynchronous with respect to events in our debugging model, so the handler must delay the effect of the interrupt until the next event. It does so by setting a flag indicating that an interrupt has occurred and resetting

```
targetTime := (!currentTime + 1)
```

When the program breaks at the next event, the debugger notices the interrupt

[9] SML/NJ extends the Standard ML definition with a continuation-based signal handling mechanism (Reppy, 1990).

```
local
  val debuggerCont: unit cont ref
  val userCont: unit cont ref
  type mode = RECORD | REPLAY
  val userMode: mode ref
  val currentEvent: eventRecord ref

  fun execTo (time:time) : unit =
    (targetTime := time;
     callcc (fn c => (debuggerCont := c;
                      throw (!userCont) ())))
in
  fun setCompilationUnit (f:unit -> 'a) : unit =
     callcc (fn c => (callcc (fn c1 => (userCont := c1;
                                        throw c ()));
                      f ();
                      ()))

  fun recordTo (time:time) : unit = (userMode := RECORD; execTo time)
  fun replayTo (time:time) : unit = (userMode := REPLAY; execTo time)

  fun break(eventRecord:eventRecord) : unit =
     (currentEvent := eventRecord;
      callcc (fn c => (userCont := c;
                       throw (!debuggerCont) ())))
end
```

Fig. 13. Implementation of basic control functions.

flag and interprets it as a request to "halt execution." The interpretation of this request depends on context. If the debugger was performing explicit, user-initiated, forward time travel, travel is halted at the current time, which allows the user to halt a lengthy or infinite computation in the natural way. If the debugger was resetting to an earlier time, or performing implicit time travel in support of a query command, the debugging operation is aborted and the program is reset to the state it was in before the operation began.

7 Implementing Time Travel

The debugger supports reverse execution by taking periodic checkpoints of program state during initial execution. To reset to a particular target time, the debugger first restores the nearest previous saved checkpoint and then re-executes to the target. We divide the state of an SML/NJ program at a particular execution event into three parts.

- The *functional state* consists of the current program counter, the contents of the call chain, and the values of all identifiers live at any point on the call chain; i.e., the program continuation at this event.

- The *store state* describes the contents of the mutable store (`ref` and `array` variables) when the event is executed.
- The *external state* describes the stream of external information (input via the I/O library, signals received, etc.) that will be seen by the program from this event until the maximum known time.

In a "typical" ML program, the size of the functional state dominates the overall size of a checkpoint, because most heap cells are immutable. For example, more than 99% of the objects created when SML/NJ compiles itself are immutable. Of course, it is easy to write an ML program that generates a large store or I/O state, but most ML programs are mostly functional.

Saving and restoring first-class continuations is fast. Moreover, these continuations are inherently *incremental*: if the debugger saves a sequence of continuations, the total space required is proportional to the total number of *different* heap cells referenced, rather than to the sum of the sizes of each continuation. Finally, SML/NJ first-class continuations are type-safe objects that can be manipulated inside the language model; in particular, they are garbage collected just like other objects without requiring any special back-end features.

Unfortunately, SML/NJ has no such convenient built-in mechanism for saving the state of the mutable store, so the debugger synthesizes its own store checkpoints. It uses an approach that preserves the existing representation of the store: store operations are instrumented to build lists of changed objects and so form incremental checkpoints. This approach can be implemented entirely within the source language, but to implement it efficiently requires unsafe type coercions and some support from the garbage collector.

To ensure correct re-execution, the debugger must log all external influences on the initial execution. These include the contents of input streams read by the program using the standard I/O interface and explicit debugging interpolations made by the user, normally to change the value of a mutable variable. The logging process is straightforward and is implemented by instrumenting the relevant input operations and handlers or providing special versions of standard library routines. The instantaneous state is represented as a set of pointers into these logs. There are other ways in which the external environment can affect SML/NJ programs, e.g., through signals or via system calls that return the time of day or read files directly. The debugger does not support reverse execution for such programs, although extending logging methods to them would not be difficult in principle.

7.1 Checkpoints

A checkpoint is a compact characterization of a state suitable for storage and retrieval. Primitive

```
getState: unit->checkpoint
```

produces a checkpoint from the current state, and

```
resetState: checkpoint->unit
```

resets the current state from a checkpoint. Once taken, a checkpoint remains valid until it is discarded, however many times it is used and regardless of other execution and checkpointing activity. A checkpoint may be discarded to conserve memory (see Section 7.5) or when the maximum known time is reset to a time before that of the checkpoint (see Section 2.3).

We briefly sketch the implementation of `getState` and `resetState`. Capturing the functional part of the current state (the current continuation) is trivial; it is already stored in `userCont` during the control transfer from the user program to the debug monitor. To reset this part of the state, the debugger just changes `userCont`. The bulk of the state-related code in the debugger deals with mutable program state, which requires special treatment both during execution and when saving or restoring checkpoints. This code is organized into subsystems, each devoted to a separate piece of mutable state. Subsystems include the mutable store, the stream I/O library, and the `lastTimes` array. Each subsystem maintains significant internal state, such as a log filled under record mode and used to control replay. Each subsystem implements a function

```
remember:unit -> memory
```

that, when invoked, encapsulates that subsystem's part of the current state into a `memory` object that can be stored as part of a checkpoint. A `memory` consists of two reset functions, `undo` and `redo`. To reset the subsystem to the state it had when the `memory` was created, the debugger invokes the appropriate reset function.

Each subsystem communicates with the executing user code by means of instrumentation code or via special versions of runtime libraries. Typically, interaction with the running program involves executing some fixed piece of code each time a particular user event occurs, e.g., logging each mutable cell creation. Code executed in this context has access to useful debugger globals, such as `currentTime`, `targetTime`, `userMode`, etc.

7.2 Time-travel Primitives

Figure 14 gives code for `advanceTo` and `resetTo` in terms of `getState` and `setState`, `replayTo` and `recordTo`, and the cache functions described in Section 7.5. The code is quite imperative in style. It would be possible to present the basic time-travel functions as side-effect-free transformations from states to states, giving them something of the flavor of "engines" (Haynes and Friedman, 1984). However, since parts of the state—especially the mutable store—are expensive to save and restore, we make the notion of a "current" state explicit and discourage excessive saves and restores.

In practice, the time-travel primitives are complicated somewhat by the need to handle multiple compilation units. Each compilation unit initiated from the top-level loop is considered to extend the existing time line; on re-execution, special steps are taken to pass control from one unit to its successor without re-entering the top-level loop. Moreover, at any breakpoint the user may enter a command (itself a compilation unit) that alters the mutable state; the debugger must arrange

```
local
  fun restoreBestPrev (target:time) =
     let val (bestTime, bestCheckpoint) =
               findPrevCheckpointInCache target
     in if (target > !currentTime andalso bestTime > !currentTime)
        orelse (target < !currentTime) then
          (currentTime := bestTime;
           resetState bestCheckpoint)
        else ()
     end

  fun saveCurrentState () =
     let val checkpoint = getState()
     in putInCache (!currentTime,checkpoint)
     end

  val maxKnownTime : time ref = ref 0
in
  fun resetTo (target:time) =
     if target <> !currentTime then
       (restoreBestPrev target;
        if !currentTime < target then
          (replayTo target;
           saveCurrentState ())
        else ())
     else ()

  fun advanceTo (target:time) =
     if target <> !currentTime then
       (restoreBestPrev target;
        if !currentTime < target andalso
           !currentTime < !maxKnownTime then
          replayTo(min(!maxKnownTime,target))
        else ();
        if !currentTime < target then
          (recordTo target;
           maxKnownTime := !currentTime)
        else();
        saveCurrentState ())
      else ()
end
```

Fig. 14. Time-travel functions. Functions `getState` and `setState` are described in Section 7.1; `recordTo` and `replayTo` are defined in Figure 13; and the cache access functions are described in Section 7.5.

to re-execute this *interpolated* unit at the same program time whenever the program is replayed.

When the debugger recreates an old state by re-executing, the resulting state is not identical to the one produced during the original execution, because values allocated during re-execution will occupy different memory locations. ML has no notion of "pointer equality" for ordinary immutable values, i.e., such values cannot be distinguished by memory location. Thus, the fact that the original and re-created states may point to different copies of a value poses no semantic problems; in fact, the debugger can't tell the copies apart anyway.[10] A `ref` variable's "value," however, is actually a pointer to a memory cell, and re-executing a `ref` creation would create a new cell. This cell would not have been affected by any of the updates that were made to the original cell, so looking up the `ref` variable would return a pointer to a valid cell having the correct type, but typically containing the wrong value. To avoid this problem, the debugger instruments `ref` creations so that actual creation occurs only on initial execution, while on replay the original cell is reused.

7.3 Checkpointing the Mutable Store

The debugger uses an incremental *delta list* technique to checkpoint the mutable store. Each creation and update of a `ref` cell is instrumented to add a pointer to the cell to a global list. Arrays are handled similarly on an element-by-element basis; the list entry for an array element consists of a pointer to the whole array plus an offset. When a `memory` is needed, the mutable store subsystem retrieves the global list, removes duplicate cell pointers (which are typically numerous), fetches a copy of the current contents of each remaining cell, and produces a set of (pointer,value) pairs; the global list is then cleared. The resulting set of (pointer,value) pairs describes the incremental change (or delta) in the store since the previous `memory` was taken. The new `memory` consists of this delta plus a pointer to the previous delta.

By the time a `memory` is requested, the global list may contain many mutable cells that are no longer reachable from the current continuation and whose values are therefore not worth saving. Normally, these cells would have become garbage, but if the global list holds pointers to them, the garbage collector must consider them live. Our solution is to use *weak pointers*, i.e., pointers that the garbage collector ignores when tracing live data and are invalidated when the object pointed to is collected. The global lists consist of such weak pointers; invalidated pointers are omitted from the delta list. Once stored in a checkpoint, the pointers become ordinary and the cells to which they point can never be collected so long as the checkpoint exists.

As program execution progresses, a series of incremental store deltas is generated, each tagged with the checkpoint's time. To reset the store forward to a given time t from an earlier time t_0, the system must consult, in order, the contents of all delta lists with tags between t_0 and t and reset the values of each cell in each list. To

[10] There is, however, a problem with space efficiency: in SML/NJ the two copies of the value will always occupy separate locations, though a cleverer runtime system might merge them.

reset backward to time t from a later time, all delta lists with tags between 0 and t must be consulted. If the same objects are updated repeatedly by the program, they tend to appear on many delta lists, and so will be reset repeatedly. To improve the efficiency of the reset operation, the system merges adjacent deltas tagged t_1,t_2 into a single delta t'_2 whenever the delta at t_1 is no longer referenced directly from a checkpoint. To determine when a delta is no longer needed, the system keeps a weak pointer to it and waits for that pointer to be invalidated.

As explained in Section 7.2, actual `ref` and `array` creations occur only during recording; on replay, the object created originally is reused. To arrange this reuse, the debugger records a special log of mutable object creations and uses it to guide replay; a pointer into this log is part of a mutable store `memory` object. Reuse is necessary only if the original object is still referenced from some continuation, i.e., still live independently of its appearance in the log. Therefore, log entries use weak pointers and a new object is created on replay if the corresponding entry pointer has been invalidated.

The instrumentation of creations and updates described above is either inserted in line (for `ref` creation and assignment) or placed in special versions of library routines (for the `array` operations). This instrumentation violates ML's typing rules. The global list itself contains pointers to cells of different types. Moreover, to remove duplicate update entries the debugger uses a type-unsafe algorithm that involves marking `ref` cells. The algorithm operates in linear time and requires no extra space (Tolmach, 1992). There are type-safe algorithms that take linear time and linear extra space, or take quadratic time.

The SML/NJ garbage collector maintains a "store list" that tracks stores of young generation pointers into old generations. With minor modifications, this list could be used as the debugger's global update list. An earlier version of our debugger did so; we switched to an instrumentation approach to decrease the debugger's dependence on the collector's implementation.

7.4 Stream I/O

Debugger support for repeatable stream I/O is in a special version of the I/O library. Stream input performed via this library is logged during recording phase; on replay, input is taken from the log. At present, the log is kept in main memory; it could be kept on disk instead. The state of an input stream, as recorded in a `memory`, is a pointer into the log. In principle, logging of input from files that do not change during execution could be avoided; a pointer into the file itself could be used to represent the stream's state. We have not implemented this optimization.

Programs that interact with their environment by performing I/O or receiving signals affect that environment during time travel. Advances beyond the maximum previously known time generally act like normal execution; input is requested, signals are accepted, and output to all devices is displayed or written in the usual way.

Speculative computation, e.g., for location breakpointing, introduces a difficulty: if the user has asked to break at a particular location, the debugger must not show output (or, worse yet, ask for input) that actually occurs after the breakpoint has been reached and passed over. A simple solution, at the cost of increased overhead for I/O events when executing speculatively, is to test the predicate before each potentially visible I/O and terminate forward execution if the predicate is true.

When resetting to a known time, the debugger does not reverse side-effects on the file system or other parts of the environment that hold state; it leaves the outside world unaffected. The user terminal, however, is treated as a stateless device; the debugger supports two terminal I/O modes when resetting forward to a known time. In *noisy* mode, the user sees the same terminal I/O as occurred during initial execution: the same output is displayed, and input from the terminal is simulated by the debugger and echoed so that input and output are correctly interleaved. In *quiet* mode, intended for implicit time-travel operations used internally by debugger, no I/O is visible.

7.5 Checkpoint Caching

Average re-execution time, and hence average total reverse execution time, can be minimized by keeping as many checkpoints as possible. But the debugger holds all checkpoint data in the heap; under the current SML/NJ garbage collector, this approach effectively constrains checkpoint storage to fit in main memory, which limits the number of checkpoints that can be kept. Therefore, the debugger maintains a cache containing checkpoints that are expected to be useful.

The cache is simply a set of checkpoints, ordered by time. It is accessed by two routines:

```
findPrevCheckpointInCache:time->(time * checkpoint)
```

returns the cache entry at or most nearly preceding its argument;

```
putInCache:(time * checkpoint)->unit
```

inserts its argument into the cache if there is room.

The targets of explicit user jumps are usually clustered in short intervals of the execution time line, typically tens to thousands of events long, separated by wide gaps that are never visited. The targets of implicit jumps made in support of `eventRecordAt` are to events on the static or dynamic chain. These time-travel patterns suggest that the cache should include entries spaced evenly throughout execution history, with additional entries near recent time-travel targets.

Checkpoints are generated at the end of every `advanceTo` or `resetTo`, i.e., at the target times of explicit and implicit jumps and also at periodic intervals during untargeted execution, as governed by the `timeDelta` parameter (see Figure 11). The value of `timeDelta` therefore has a major influence on the number of checkpoints generated during execution. Obviously, most of these checkpoints are *not* useful, so our cache replacement policy must be able to weed them out easily. Evidently, the policy should contain an LRU component, since we expect recently visited targets to be revisited. However, a pure LRU policy allows arbitrarily large gaps to appear in the cache set, so we have developed an alternative heuristic that favors retention of checkpoints that would be most expensive to recompute. Experiments show little difference between these heuristics, probably because cache replacement is fairly rare.

The debugger controls cache size on the basis of memory availability rather than by maintaining a fixed number of entries. Memory availability is measured as the ratio r between physical memory size and live-data size. The higher the ratio, the less frequently garbage collections will be needed, and the better overall system performance we can expect. Normally, live-data size is a fixed characteristic of the user program, so SML/NJ simply lets r grow as large as the available physical memory size allows.

However, the debugger can control the amount of live data it requires by varying the size of the checkpoint cache, so we must make some *a priori* choice of good target ratio r_0. Appel's experiments (1992, page 193) suggest using a value between 3 and 6; we somewhat arbitrarily take $r_0 = 5$. The debugger's cache sizing policy allows insertions if $r > r_0$; if $r \leq r_0$, the debugger attempts to delete enough entries to bring r back to r_0 and retries the insertion. This simple feedback mechanism keeps r near r_0, but is rather expensive, because accurately calculating r requires a major garbage collection.

Since `eventRecordAt` is typically called repeatedly with the same set of time arguments (e.g., the binding times for global variables), it makes better sense to memoize the function independently from the checkpoint cache, so that a given event record need be reconstructed at most once. Memoization of `siteNumber` and `binding` fields can be enabled separately from memoization of `boundValue` fields, since the former are accessed more frequently and take only a small fixed amount of space per event, whereas the latter can be indefinitely large. Memoization for `eventRecordAt` is implemented using a separate hash table, which is also updated by `putInCache`.

8 Performance

To assess the debugger's practicality, we measured its performance on real programs. The benchmark suite consisted of a program implementing the game of Life (Reade, 1989), running 50 generations of a glider gun; Simple, a spherical fluid-dynamics program originally developed as a FORTRAN benchmark (Crowley et al., 1978); the Knuth–Bendix completion algorithm processing some axioms of geometry; a program to build a dictionary using RedBlack trees from an input of 300,000 integers; a machine-generated Lexer for ML that counts tokens, run on a 5760-line input file; UnionFind, a program that performs 5,000 union and find operations with path compression on a database of strings; and a TermRewriter originally written in Scheme (Kamin, 1990), performing symbolic differentiation on polynomials. The benchmarks ranged in size from 65 to 764 lines of ML. Compile times for the uninstrumented, optimized programs ranged from 5.5 to 67.7 seconds and code size ranged from 5 to 93 KB. Live data size ranged from 3 to 5266 KB and total execution time ranged from 0.8 to 46.3 seconds. Some additional characteristics of the benchmark suite are shown in Table 1.

Table 1. *Benchmark event characteristics.*

Key	Name	Sites/ Line[a]	Millions of Events Executed[b]	Events/ Instruction[c]	% Store Events[d]	% I/O Events[e]
l	Life	3.4	13.2	.031	0	< 0.01
s	Simple	1.7	56.6	.076	0.52	< 0.01
k	Knuth–Bendix	1.7	25.9	.124	0	0.02
r	RedBlack	0.9	37.8	.056	0.79	0
x	Lexer	0.3	1.0	.039	2.70	< 0.01
u	UnionFind	1.4	0.6	.042	7.98	1.72
t	TermRewriter	1.6	2.4	.083	0	0

[a] Average number of instrumentation sites per source line (after coalescing).
[b] Total for a simple execution of the program.
[c] Average number of events executed for each machine instruction executed by the *uninstrumented* program, a measure of dynamic event density.
[d] Dynamic percentage of executed events that alter the mutable store.
[e] Dynamic percentage of executed events that perform stream I/O.

Benchmarks were run on a MIPS Magnum 3000 workstation with 128 MB of memory, 32KB direct-mapped instruction cache, and 32KB direct-mapped write-through data cache, under the RiscOs 4.51 operating system. Benchmarks used a variant of SML/NJ version 0.69. The benchmarked version of the debugger built event records lazily, coalescing events where possible, and used speculative computation to support location breakpoints.

8.1 Measuring Instrumentation Overhead

We consider first the time and space overheads introduced by instrumentation, by measuring execution under debugger control *without* taking checkpoints. We execute each benchmark from beginning to end, without performing any debugging operations. Figure 15 shows how the execution times of each benchmark compare under the different compilation disciplines.

On average, instrumentation slows program execution by a bit less than 3 times (again, in the absence of checkpointing), though individual benchmarks have widely varying behavior. UnionFind runs particularly slowly under under the debugger; this anomaly disappears if the overhead of logging updates and I/O is excluded (debugger nolog), which is not surprising since UnionFind has the highest frequencies of store update and I/O events among the benchmarks.

We have no machine-based debugger for ML for comparison, but we can make a rough estimate of how such a debugger would perform. It would suffer no slow down from instrumentation, but it would need to inhibit some optimizations to support expected behavior. Many debuggers turn off optimization altogether; wholly unoptimized SML/NJ code runs about 3 times slower than normal code, hence about

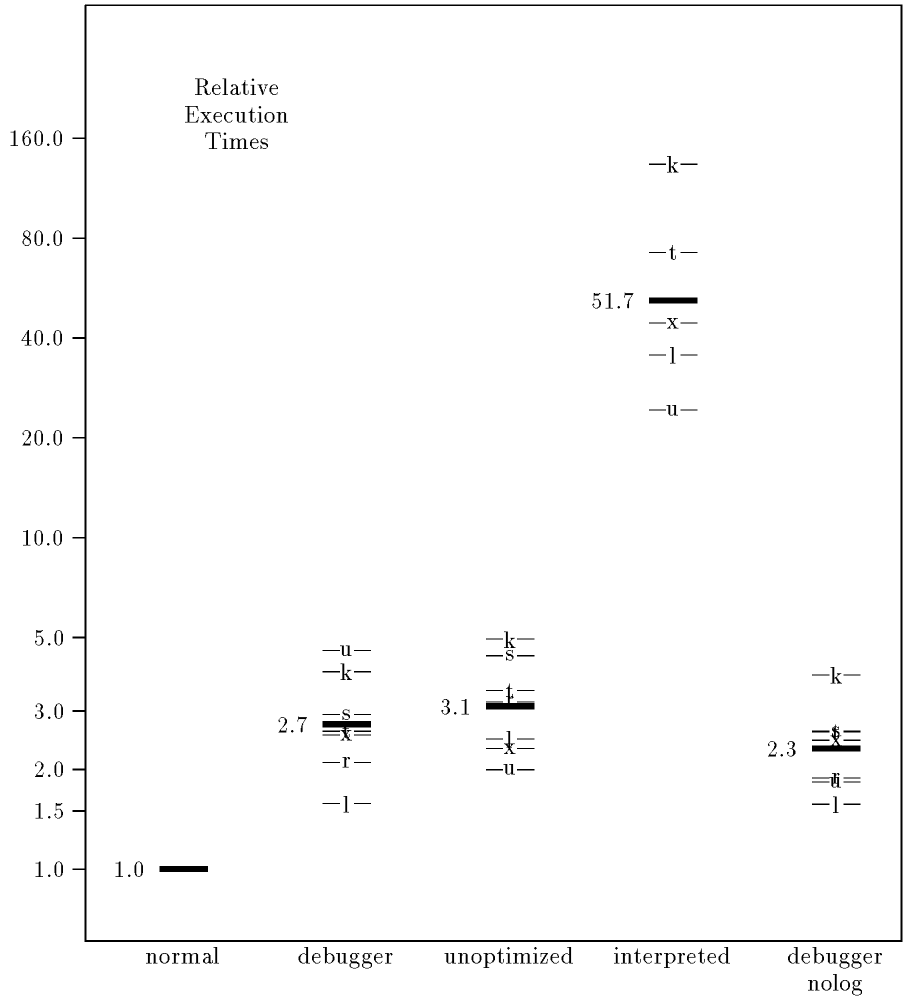

Fig. 15. Relative controlled execution costs (measured as the sum of user, garbage collection, and system times) under various compilation disciplines described in the text. The vertical scale is logarithmic; the labels indicate the ratio of execution time for each discipline to that of the normal discipline. The thick bar is the geometric mean of the benchmark time ratios. The thin bars marked with letters indicate the time ratios for the individual benchmarks; see Table 1 for the key. Two of the interpreted benchmarks were aborted when they didn't complete within one hour, so the geometric mean time ratio shown for the interpreted discipline is artificially low.

comparably with our debugger.[11] We could expect a well-engineered machine-based debugger to do somewhat better than this, since not all optimizations are fatal to debugging, but not as well as unoptimized code. Another approach to debugging is to use an interpreter, but interpreted code runs one to two orders of magnitude slower than any of the compiled approaches.

Code generated under the debugger is, on average, five times larger than normal code. There is relatively little variation in code size expansion from one benchmark to another. By comparison, the unoptimized discipline causes code size to increase by an average factor of three. Increased code size is not in itself a significant problem, but compile time also increases with code size at a somewhat superlinear rate. For Simple, the largest benchmark, a normal compile time of about one minute corresponds to a debugger compile time of more than five minutes, which is too long for use in edit–compile–test cycles.

One way to reduce code size and compilation time at the expense of execution time is to implement the `event` instrumentation code as a true function instead of in-lining it. Experiments on earlier versions of the debugger suggest that this technique could cut compilation time by 30–50% and increase execution time by 25–75%. For large programs with short execution times, another option is to interpret the instrumented code. Since instrumentation is performed before translating abstract syntax to λ-language, the debugger can operate in interpretive mode without modification. Execution will be slow, but the time from edit to completed test might be reduced. Compiling the instrumented code with some optimizations disabled is another option.

To see how much live data the instrumented program generates compared to normal execution, we compare live-data profiles measured over the course of the execution, omitting checkpoint storage. Each profile is constructed by measuring the amount of live data at a selection of frequently executed sites. The benchmarks can be divided into two distinct classes based on their normal execution profiles. RedBlack, Knuth–Bendix, and UnionFind build internal data structures that grow steadily (typically linearly) during execution. Life, Lexer, TermRewriter, and Simple use an essentially fixed amount of live data (although the content of the data may be changing).

Figure 16 shows profiles for a characteristic member of each class. For the first class of benchmarks, the debugger discipline generates a constant factor more live data than normal. Comparison with the debugger nolog profile shows that log space often accounts for much of this increase; this fact holds even for RedBlack, for which less than 1% of events are store updates, and is much more marked for UnionFind, with 8% store update events. The unoptimized discipline also generates a constant factor more live data, though the factor is smaller. The second class of benchmarks shows more varied behavior. While the benchmark shown, Life, generates only a constant amount of additional data under the debugger, other benchmarks that have substantial numbers of store events, such as Lex, generate a linearly increasing

[11] In fact, the "unoptimized" discipline still performs certain optimizations, such as those associated with CPS conversion, that cannot be turned off in SML/NJ.

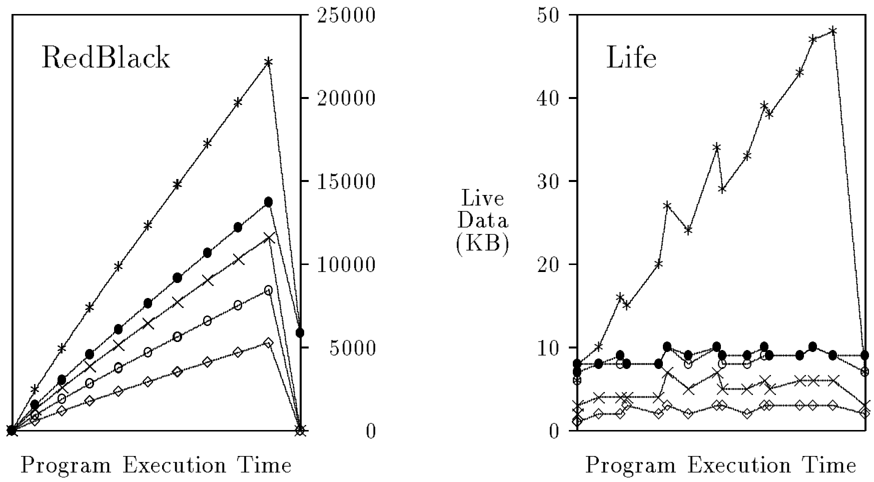

Fig. 16. Live data measured for RedBlack and Life, without checkpointing. Execution disciplines: normal (◇), unoptimized (×), debugger (•), debugger nolog (∘), naive (∗). The points of each profile are connected for graphical clarity; the connecting lines should not be used for interpolation.

amount of extra data, primarily due to logs. The same behavior occurs with the unoptimized discipline, probably because of the loss of tail-recursion elimination.

We also show how much live data is generated by an implementation of the naive event record creation scheme described in Section 4.2—generally much more than by our lazy event record scheme. Of course, checkpoints are needed to make the lazy scheme reasonably efficient, so we can't claim an overall space savings.

8.2 Time vs. Space

The debugger's time and space use cannot be measured independently. On the one hand, like any garbage-collected system, SML/NJ will run faster given more memory. On the other hand, the debugger takes advantage of increased memory to enlarge its checkpoint cache, which should make reverse execution faster but will slow garbage collection. To compare performance under different disciplines, we measure execution time over a range of memory sizes for each discipline, and compare overall time vs. space curves; Figure 17 shows two characteristic sets of curves. Each point on the curve shows the total elapsed time for a single simulated debugging session with total system memory held constant. Each debugging session is driven by a synthetic command script designed to represent typical but simplified patterns of user behavior, including some explicit reverse execution. For the normal and unoptimized disciplines, which don't implement reverse execution, we estimate the execution time by charging the cost of re-executing from the beginning of the program each time the script performs a reverse motion. This estimate is unfair, since a user without a reverse-execution debugger would probably avoid such re-executions, but it provides a rough basis for comparison.

To compare the debugger's performance with other systems, we construct curves

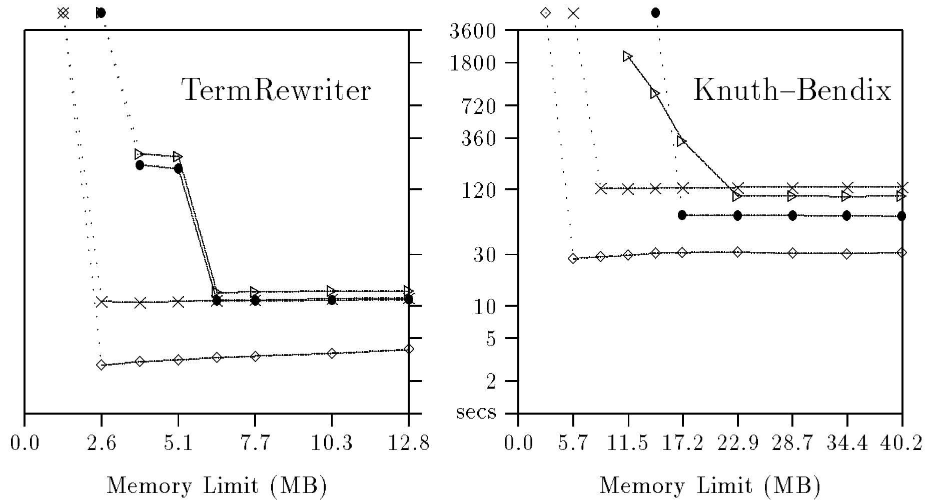

Fig. 17. Time vs. Space for TermRewriter and Knuth–Bendix, with checkpointing. Execution disciplines: normal (◇), unoptimized code (×), debugger (•), debugger without memoization (▷). The checkpointing interval for the debugger disciplines was 16,000 events for TermRewriter and 256,000 events for Knuth–Bendix. Each plotted point represents a single execution of a particular debugging command script with memory held fixed. Elapsed times (user+gc+system time) are shown on a logarithmic scale. The points for each discipline are connected for graphical clarity; the connecting lines should not be used for interpolation. Runs that exceeded the available memory aborted; their symbols are shown above the graph at the appropriate memory coordinate, and they are connected to the curve with a dotted line.

for fully optimized normal execution and unoptimized execution. We also show the effects of disabling memoization. Each curve typically shows a clear "knee" at a particular memory size; at smaller memory sizes, execution time increases sharply or the system halts due to lack of memory. The memory size at the knee is a "natural" minimum for "comfortable" execution, although in many cases the system can execute with less memory at a slower speed. Increasing memory above the knee produces little benefit under any of the measured disciplines. We can compare knees to estimate the "extra" memory needed for the debugging disciplines.

By examining many such sets of curves, we can draw some basic conclusions about performance. The checkpointing interval has a significant effect on script execution time. The best checkpointing interval for each benchmark was chosen by running the benchmark repeatedly under each script with an interval of 4,000, 8,000, 16,000,…, 64,000,000 events. The best interval varies with program and script, but consistently lies in the range 8,000 to 256,000 events, roughly 10-150 msec of user program CPU time on our hardware; within this range, most of the benchmarks are fairly insensitive to the exact value chosen. Thus, while being able to change the checkpointing interval may be useful, a default value seems satisfactory for most programs.

The debugger uses much more memory than normal execution. How much more varies widely among the benchmarks: for example, debugging Lexer requires about three times as much memory as running it normally, while debugging Simple seems to require about five times as much. The ratio is also sensitive to the debugging script used; for Life, it varied from four to eight. If the memory to run the debugger comfortably is available, its speed is competitive with the hypothetical debugging approaches for most benchmarks. We estimate that a machine-based debugger should give performance somewhere in the band between the normal and unoptimized curves. For scripts involving reverse execution, the debugger's execution time lies in or below the machine-based band for all benchmarks except UnionFind and Life (under one script); UnionFind is substantially slowed by its need to log mutable memory updates. For scripts not involving reverse execution, a machine-based debugger will generally do better than ours and a debugger based on the naive record event approach (which doesn't require time travel) usually does about as well; these facts suggest that time travel is not worth implementing solely for internal use by other parts of the debugger.

One way to save space at the expense of more time is to disable memoization. The curves shown for this discipline in Figure 17 are typical: there is a large increase in execution time, but the debugger can continue executing with less memory. A milder version of the same effect can be seen by disabling only memoization of bound values in event records.

8.3 Query Times

Since the debugger is an interactive tool, it must service requests promptly. A reasonable, though arbitrary, standard is that most commands should require less than 1 second of elapsed time. Commands in this class include variable lookup, changing scope along the call stack, showing a call frame, and single-stepping forward or backward. Commands that require an arbitrary amount of execution, such as "proceed to next breakpoint," are excluded from this standard.

We measured elapsed time for the above commands for each benchmark under each of our simulation scripts, using the lowest memory size "comfortable" for all scripts. Nearly all the tested commands executed in less than 1 second; many ran an order of magnitude faster. Execution time exceeded 2 seconds only for showing a frame under Life and Knuth–Bendix (10% of calls required as much as 10 seconds) and for most commands under Simple (about 50% of commands to change scope, show a frame, and single-step backward required up to 10 seconds). The "show frame" problem may reflect the high cost of dynamic type reconstruction, which invokes an expensive unification algorithm and has not been optimized. The results for Life probably include a run in which a major garbage collection occurred during the execution of a "show frame." The poor times under Simple, our largest benchmark, may be due to the cost of saving and restoring the program's large mutable state when time-traveling.

9 Discussion and Related Work

We have used source-code instrumentation to build a simple, portable debugger for a complex language with an optimizing compiler. Our debugger supports time travel, which, in addition to being a valuable user-level feature, is used to implement conventional debugger features elegantly and efficiently.

Despite the existence of several working compilers for Standard ML, and a substantial body of users, there are no other full-featured source-level debuggers for the language. This situation may be due in part to lack of demand: there is considerable anecdotal evidence that compile-time type checking leads to relatively fewer runtime bugs than in conventional languages (Cardelli, 1984), and a debugger is therefore less important. Limited debugging support, in the form of tracing, has been developed for two non-standard ML implementations, ANU ML (Ophel, 1991) and CAML (Projet Formel, INRIA-ENS, 1990). Both these systems support tracing by patching the internal representation of functions. Neither prints polymorphic variable values correctly.

Our debugger is independent of the rewriting, optimization, and code-generation phases of the SML/NJ compiler. During the course of this research, these phases have been modified several times, without requiring any changes to the debugger. The debugger has also become available on several new target architectures as a result of back-end ports, without any changes to the debugger's code. Currently, the debugger does rely on the compiler's front end for parsing and type information; these dependencies could be reduced if the compiler exported the ability to manipulate abstract syntax directly, and, in principle, they could be avoided by preprocessing the source.

The debugger is simple. It is implemented in about 7,900 lines of ML, versus about 44,000 for the compiler as a whole.[12] Supporting debugging is much simpler than compiling because the debugger is back-end independent. Most machine-based debuggers supporting multiple architectures are larger relative to their compilers; for example, `gdb` is roughly the same size as `gcc` (about 100,000 lines of C). Most of the complexity in our debugger stems from the complexity of ML itself, rather than from elaborate debugger algorithms.

9.1 Instrumentation

Automated instrumentation of source code is not a new idea. BUGTRAN (Ferguson and Berner, 1963), one of the first source-level debuggers, transformed FORTRAN source programs to support batch debugging. Balzer's Extendable Debugging and Monitoring System (EXDAMS) (1969) instrumented FORTRAN source-code to generate a "history tape" containing all conceivably interesting information about the course of execution. Later, this tape could be replayed, forward or backward, to study the program's behavior. Our approach is very similar, although we support

[12] This figure for the compiler includes the code generator for just one target architecture. The code generators for additional architectures average about 1500 lines each.

interactive rather than batch-based debugging. LISP systems have long used macro packages (e.g.,(Dybvig et al., 1988)) and dynamic scope to implement debuggers by automatically replacing user-defined functions with instrumented variants that perform tracing, breakpointing, etc.

Debugging instrumentation may also be inserted at lower levels. Reflective systems (Friedman and Wand, 1984; Maes, 1987; Smith, 1982) give the ability to examine and alter the internal state of a language interpreter from within the source language being interpreted. Reflective mechanisms can be used to modify the normal operation of the interpreter to support debugging features such as single-stepping and tracing. Hanson (1978) describes a symbolic debugger for SNOBOL4 programs built using event associations, a mechanism for automatically invoking a source-language function each time an event of interest occurs during program execution. Hanson's approach has much in common with ours; whereas he implements the association mechanism inside the runtime system, we make associations explicit in the modified source code, and rely on general-purpose optimization techniques to render the resulting code efficient.

Kishon, Hudak, and Consel (1991) describe how a standard continuation semantics for a language such as Haskell may be systematically transformed into a non-standard *monitoring semantics* in which the domain of program meanings (answers) is enhanced to include monitoring results as well as the original answer. A monitor is implemented using two levels of partial evaluation. First, a standard interpreter is specialized with respect to a monitoring specification to yield a monitoring interpreter. Then, this interpreter is specialized with respect to a user program to yield an instrumented version of the program, which, like our instrumented code, is fed to an ordinary compiler. By comparison, our hand-crafted instrumentation process appears quite ad-hoc, but it is orders of magnitude faster than the partial evaluation approach, and produces more efficient, direct-style instrumented code.

9.2 Time Travel

Our debugger's user features, which include value querying, breakpointing, and single-stepping, are fairly ordinary (Beander, 1983; Linton, 1990; Stallman and Pesch, 1991); our system is unusual because it integrates these features with a flexible time-travel mechanism. Although reverse execution is not provided by many commercial debuggers, it has a long research and prototyping history. Most previous systems have implemented reverse execution by building a log of execution steps or memory updates (Balzer, 1969; Grishman, 1970; Teitelbaum and Reps, 1981; Teitelman, 1978; Zelkowitz, 1971); such systems typically limit the size of the log, and hence the number of reverse execution steps permitted. Some recent systems (Agrawal et al., 1991; Choi et al., 1991) maintain information about the entire execution history by taking what amount to incremental checkpoints at selected program points; in principle, these systems could support reverse execution to arbitrary points, as we do, by restoring a suitable checkpoint and re-executing as necessary.

Reverse execution is usually supported as a meta-level facility external to the standard semantics of the language being debugged. However, many of the semantic and implementation issues uncovered in generalizing and automating explicit "undo" facilities for programming languages (Archer et al., 1984; Leeman, 1986; Teitelman, 1978; Vitter, 1984) are also relevant to replay debugging. Some Prolog debuggers use the language's built-in backtracking model to support limited reverse execution (Bowen et al., 1984; Byrd, 1980).

Ordinary debuggers typically conflate the notion of an event site with that of an event execution, sometimes relying on a repetition count to indicate which execution is wanted (e.g., "stop on the fifth repetition of the call at line 17"). Identifying events via simple integer "time" values has many advantages for both the user and the debugger's internal bookkeeping needs. We use a machine-independent software clock, but there are other ways to provide the essential features of a clock, namely predictability, monotonicity, and an "alarm" feature. One alternative approach is to count machine instructions, either in hardware (Cargill and Locanthi, 1987) or software (Mellor-Crummey and LeBlanc, 1989), and cause an interrupt after a certain number of instructions have been executed. Another, suitable for heap-based languages like SML/NJ, is to use the allocation pointer as a sort of clock whose alarm is set by altering the heap limit (Wilson and Moher, 1989).

Unfortunately, times have little intrinsic meaning for the user. A debugger should also be able to locate events specified by other predicates, involving site, variable values, and meta-information such as the length of the call chain. Debuggers supporting arbitrarily complicated predicates have mostly used single-stepping to locate matching events (Feldman and Brown, 1988; Grishman, 1970), which is prohibitively slow.[13] Our implementation of speculative computation shows that, for monotone predicates, a binary search based on time travel can be much more efficient. Our approach was inspired by Cargill and Locanthi (1987); a similar idea was used in more limited fashion in IGOR (Feldman and Brown, 1988). Several existing systems use reverse execution primarily as a foundation for more sophisticated debugging aids. These include visualization (Teitelbaum and Reps, 1981); "flowback analysis," the automated display of the assignments that have led a variable to have a particular value (Balzer, 1969; Choi et al., 1991); and dynamic program slicing (Agrawal et al., 1991). We could build similar tools on top of our time-travel primitives.

9.3 Checkpointing

One important source of simplicity in the debugger is the use of `callcc` to capture most of the program's state at a checkpoint. Interestingly, we rely heavily on the ability to throw to a continuation more than once; most other applications, such as exceptions and coroutines, seem to need only one-shot continuations. `callcc` is efficient in SML/NJ, but even if it were slower, it would remain an elegant way to

[13] This characterization ignores the possibility of hardware support mechanisms such as memory protection traps, which can sometimes be used to speed up debugger execution substantially.

gather checkpoint information without requiring the debugger to understand the compiler's back end.

The debugger could be implemented more elegantly and portably if SML-NJ supported first-class stores as well as first-class continuations. First-class stores might be implemented using delta lists (Morrisett, 1993), persistent trees (Johnson and Duggan, 1988), or page-level checkpointing mechanisms (Archer et al., 1984; Feldman and Brown, 1988; Wilson and Moher, 1989). These schemes rely on support from the underlying runtime system, especially the garbage collector.

A more sophisticated, multi-generational garbage collector would also allow the debugger to maintain a larger checkpoint cache: under such a collector, checkpoints should automatically migrate to older generations and eventually to backing store. It would also be possible for the debugger to place some checkpoint data onto backing store explicitly; this would be easy for external input logs, whose contents are already formatted for I/O, but much harder for continuations or store update lists, which contain pointers that need to be forwarded during garbage collections.

9.4 Performance

Debugger performance is adequate for small programs. Execution speed under debugger control is typically about three times slower than normal execution; this makes the debugger performance competitive with our estimates for hypothetical machine-based debuggers and much better than for interpretation. Debugger-controlled execution generates more live data than normal execution, although the worst effects of a naive approach are avoided by generating event records lazily. Still, the debugger requires large amounts of space for logs and checkpoints in order to run "comfortably," although it can continue to operate at slower speed even when few checkpoints are cached.

For large programs, these performance overheads limit the debugger's practicality. A further serious drawback to using the debugger for large programs is that the increase in source-code size caused by instrumentation increases compilation times by roughly a factor of five, and the SML/NJ compiler is already slow. It is possible to trade smaller code size for increased execution time by not in-lining the instrumentation code at events. A better long-term solution would be to speed up the compiler enough so that it compiles even large instrumented programs quickly.

9.5 Modularity

A related problem is that the debugger does not handle modular programs well. At present, all parts of the user program that perform side-effects or manipulate higher-order functions must be instrumented in order for the debugger to function correctly and securely. To simplify debugging a large system, users typically assume that some "trusted" portions of the system, e.g., library routines, are correct. They do not wish to see the internal details of trusted functions or associated data structures. The debugger itself should take advantage of trusted functions by instrumenting them minimally, and, in the case of an imperative function, by logging only the net side-effects of the function rather than its internal, individual side-effects.

It would be straightforward to introduce a user mechanism for declaring specified abstract units such as `abstype`s or `structure`s, or perhaps entire source files, to be trusted. It should be possible to change these declarations at runtime, which would require support for dynamic recompilation of selected units. Such a scheme would resemble dynamic deoptimization (Hölzle et al., 1992; Zurawski and Johnson, 1991). Higher-order functions introduce another complication: a trusted library routine, such as `map`, may invoke a non-trustworthy user function that requires debugging support, and it is not clear how much contextual information from the invoking library routine may be needed to understand the user function's behavior.

Developing a mechanism for users to specify the net side-effects of trustworthy units is a harder problem, but it represents the key to making the debugger genuinely extensible. Our application of the debugger technology to ML Threads (Tolmach, 1992) offers several examples.

10 Acknowledgements

David Tarditi designed and implemented the original version of the dynamic type reconstruction algorithm. Adam Dingle designed and implemented the debugger's original `emacs` user interface.

References

Agrawal, H., DeMillo, R. A., and Spafford, E. H. (1991). An execution-backtracking approach to debugging. *IEEE Software*, 8(3):21–26.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA.

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

Appel, A. W. (1989a). Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2(2):153–162.

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

Appel, A. W. (1992). *Compiling with Continuations.* Cambridge University Press.

Appel, A. W. and MacQueen, D. B. (1991). Standard ML of New Jersey. In Wirsing, M., editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–13, New York. Springer-Verlag.

Archer, Jr., J. E., Conway, R., and Schneider, F. B. (1984). User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages and Systems*, 6(1):1–19.

Balzer, R. M. (1969). EXDAMS–EXtendable Debugging and Monitoring System. In *Proceedings AFIPS 1969 Spring Joint Computer Conference*, volume 34, pages 567–580, Montvale, NJ. AFIPS Press.

Beander, B. (1983). VAX DEBUG: An interactive, symbolic, multilingual debugger. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging*, pages 173–179. Published as *SIGPLAN Notices*, 18(8), Aug. 1983.

Bowen, D., Byrd, L., Pereira, F., Pereira, L., and Warren, D. (1984). *Prolog-20 User's Manual.*

Byrd, L. (1980). Understanding the control flow of prolog programs. In *Proc. Logic Programming Workshop, Debrecen, Hungary*, pages 127–138. Also Univ. of Edinburgh Dept. of Artificial Intelligence Research Paper 151.

Cardelli, L. (1984). Compiling a functional language. In *Proc. 1984 ACM Conference on Lisp and Functional Programming*, pages 208–217.

Cargill, T. A. and Locanthi, B. N. (1987). Cheap hardware support for software debugging and profiling. In *Proc. Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 82–83.

Choi, J.-D., Miller, B. P., and Netzer, R. H. B. (1991). Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530.

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

Coutant, D. S., Meloy, S., and Ruscetta, M. (1988). DOC: A practical approach to source-level debugging of globally optimized code. In *Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 125–134. Published as *SIGPLAN Notices*, 23(7), July 1988.

Crowley, W. P., Hendrickson, C. P., and Rudy, T. E. (1978). The SIMPLE code. Technical Report UCID 17715, Lawrence Livermore Laboratory, Livermore, CA.

Duba, B., Harper, R., and MacQueen, D. (1991). Typing first-class continuations in ML. In *Eighteenth Annual ACM Symp. on Principles of Programming Languages*, pages 163–173.

Dybvig, R. K., Friedman, D. P., and Haynes, C. T. (1988). Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–75.

Feldman, S. I. and Brown, C. B. (1988). IGOR: A system for program debugging via reversible execution. In *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112–123. Published as *SIGPLAN Notices,* 24(1), Jan. 1989.

Ferguson, H. E. and Berner, E. (1963). Debugging systems at the source language level. *Communications of the ACM*, 6(8):430–432.

Friedman, D. P. and Wand, M. (1984). Reification: Reflection without metaphysics. In *Proc. 1984 ACM Conference on Lisp and Functional Programming*, pages 348–355.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation.* Addison–Wesley, Reading, MA.

Goldberg, B. and Gloger, M. (1992). Polymorphic type reconstruction for garbage collection without tags. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, pages 53–65. Published as *LISP Pointers* V(1), Jan.–Mar. 1992.

Grishman, R. (1970). The debugging system AIDS. In *Proc. AFIPS 1970 Spring Joint Computer Conference*, volume 36, pages 59–64, Montvale, NJ. AFIPS Press.

Hanson, D. R. (1978). Event associations in SNOBOL4 for program debugging. *Software—Practice and Experience*, 8(2):115–129.

Haynes, C. T. and Friedman, D. P. (1984). Engines build process abstractions. In *Proc. 1984 ACM Conference on Lisp and Functional Programming*, pages 18–24.

Hennessy, J. L. (1982). Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344.

Hoare, C. A. R. (1989). Hints on programming-language design. In *Essays in Computing Science*, pages 193–216. Prentice Hall. Keynote address to the ACM SIGPLAN conference, Oct. 1973.

Hölzle, U., Chambers, C., and Ungar, D. (1992). Debugging optimized code with dynamic deoptimization. In *Proc. SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 32–43. Published as *SIGPLAN Notices*, 27(7), July 1992.

Hudak, P. and Wadler, P. (1990). Report on the programming language Haskell: A non-strict, purely functional language, Version 1.0. Technical Report YALEU/DCS/RR-777, Yale University, Dept. of Computer Science.

Johnson, G. F. and Duggan, D. (1988). Stores and partial continuations as first-class objects in a language and environment. In *Proc. 15th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 158–168.

Kamin, S. N. (1990). *Programming Languages: An Interpreter-Based Approach.* Addison-Wesley, Reading, MA.

Kaufer, S., Lopez, R., and Pratap, S. (1988). Saber-C: An interpreter-based programming environment for the C language. In *Proc. Summer '88 USENIX Conference*, pages 161–172.

Kishon, A., Hudak, P., and Consel, C. (1991). Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 338–352. Published as *SIGPLAN Notices*, 26(6), June 1991.

Leeman, Jr., G. B. (1986). A formal approach to undo operations in programming languages. *ACM Transactions on Programming Languages and Systems*, 8(1):50–87.

Linton, M. A. (1990). The evolution of dbx. In *Proc. Summer USENIX Conference*, pages 211–220.

Maes, P. (1987). Concepts and experiments in computational reflection. In *Proc. 1987 Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 147–155.

Mellor-Crummey, J. and LeBlanc, T. (1989). A software instruction counter. In *Proc. Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78–86. Published as *Computer Architecture News* 17(2), Apr. 1989.

Milner, R., Tofte, M., and Harper, R. (1990). *The Definition of Standard ML.* MIT Press, Cambridge, MA.

Morrisett, J. G. (1993). Generalizing first-class stores. In *Proc. ACM SIGPLAN Workshop on State in Programming Langauges (SIPL '93), Copenhagen, Denmark*, pages 73–87. Published as Yale University Dept. of Computer Science Tech. Rep. YALEU/DCS/RR-968.

Ophel, J. L. (1991). *AIMLESS: A Programming Environment for ML.* PhD thesis, The Australian National University.

Projet Formel, INRIA-ENS (1990). Caml reference manual (version 2.6.1). Technical Report 121, INRIA.

Reade, C. (1989). *Elements of Functional Programming.* Addison-Wesley, Reading, MA.

Reppy, J. H. (1990). Asynchronous signals in Standard ML. Technical Report TR 90-1144, Cornell University, Dept. of Computer Science.

Smith, B. (1982). Reflection and semantics in a procedural language. Technical Report MIT-LCS-TR-272, Massachusetts Institute of Technology, Cambridge, MA.

Stallman, R. M. and Pesch, R. H. (1991). *Using GDB: A Guide to the GNU Source-Level Debugger (GDB version 4.0).* Free Software Foundation, Inc.

Teitelbaum, T. and Reps, T. (1981). The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573.

Teitelman, W. (1978). *Interlisp Reference Manual.* Xerox Palo Alto Research Center.

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

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

Turner, D. A. (1985). Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag.

Vitter, J. S. (1984). US&R: A new framework for redoing. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 168–176. Published as *SIGPLAN Notices*, 19(5), May 1984.

Wand, M. (1980). Continuation-based multiprocessing. In *Proc. 1980 LISP Conference*, pages 19–28.

Wilson, P. R. and Moher, T. G. (1989). Demonic memory for process histories. In *Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 330–343. Published as *SIGPLAN Notices*, 24(7), July 1989.

Zelkowitz, M. (1971). *Reversible Execution as a Diagnostic Tool.* PhD thesis, Cornell University.

Zellweger, P. T. (1984). *Interactive Source-level Debugging of Optimized Programs.* PhD thesis, University of California, Berkeley. Also Xerox Corporation Palo Alto Research Center Tech. Report CSL-84-5.

Zurawski, L. W. and Johnson, R. E. (1991). Debugging optimized code with expected behavior. Unpublished manuscript.