DEBUGGING STANDARD ML

Andrew Peter Tolmach

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

October 1992

© Copyright by Andrew Peter Tolmach 1992
All Rights Reserved

Abstract

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 type-safe languages, however, debugging support can be provided without reference to the underlying machine, by adding instrumentation to program source code before compilation. Because the instrumented code is (almost) ordinary source, it can be processed 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.

This dissertation describes the design and implementation of instrumentation-based debuggers for the Standard ML of New Jersey compiler. Three systems are described: a simple debugger with ordinary variable display and breakpoint facilities, a debugger supporting—and taking internal advantage of—reverse execution, and a debugger for ML Threads, a concurrent extension of ML running on multiprocessors. A formal operational semantics for a subset of ML is used to define debugging operations precisely, and to provide a basis for determining the concrete types of polymorphic variables at runtime. Debugger performance measurements are presented for a range of real programs.

For my parents,
in joy and in sadness.

Acknowledgments

My advisor, Andrew Appel, has given greatly of his time, enthusiasm, and technical ingenuity while guiding me through this research. In particular, I wish to thank him for his perpetually open door, and for his understanding attitude about deadlines.

David MacQueen has supported this project with his advice and enthusiasm from its inception. Much of the debugger was implemented while I was working as a consultant for AT&T Bell Laboratories, by his arrangement. The text of this dissertation has been much enhanced by his comments, as well as by the eagle eye and vigorous red pen of David Hanson.

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.

Greg Morrisett designed and implemented the multiprocessor version of SML/NJ, and keeps me cheerful by claiming that he will use the concurrent debugger one day. Suresh Jagannathan and Jim Philbin graciously provided access to the NEC Research Institute's SGI multiprocessor, which made it possible to benchmark the concurrent debugger.

Lal George implemented the ML version of the Simple benchmark in both sequential and parallel forms, and offered much good counsel on parallel processing. Several of the parallel benchmarks were adapted from Scheme originals written by Rick Mohr.

The office staff of the Computer Science Department have helped keep life at Princeton livable. Thanks especially to Sharon Rogers for her frequent and indispensable aid over the years.

Finally, I could not have completed this degree without the love and support of my wife, Alycia, who has kept me saner and happier than any graduate student has a right to be.

This research was funded, in part, by the National Science Foundation under grants CCR-8806121, CCR-8914570, and CCR-9002786.

Contents

List of Tables

List of Figures

Chapter 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.

This dissertation describes debuggers that use a different approach, based on *source-code instrumentation*. This approach works by preprocessing source code and inserting statements at potentially interesting points in the program to support runtime debugging operations. Because the instrumented code is (almost) ordinary source, it can be processed 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 event in its execution history. Reverse execution can also be used in the debugger implementation to decrease the overhead of more conventional debugging commands. Reverse execution can be implemented using a checkpoint and replay system.

We apply instrumentation and reverse execution techniques to build several debuggers for the language Standard ML. We treat both the standard sequential language and a multiprocessor version with concurrency extensions.

Our work is based on the Standard ML language [88] and the Standard ML of New Jersey

(SML/NJ) compiler [11, 13]. Standard ML is type-safe (programs cannot "dump core" due to runtime typing errors). Type-safety 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. ML encourages the use of first-class functions and recursion, so naive compilation strategies do not produce efficient code. The SML/NJ compiler therefore performs extensive optimizations.

Our definition of source-level debugging is conventional; it includes support for breakpoints, value display, and call traceback. These features would be difficult or impossible to implement using a machine-based debugger in conjunction with SML/NJ's optimizer. Although some problems could be resolved by careful coordination between the compiler back-end and the debugger, others are more fundamental, since they arise because the optimized program doesn't compute or keep information that the debugger needs to provide "expected behavior" to the user. The most serious problems arise because SML/NJ uses continuation-passing style (CPS), which eliminates storage for dead variables, performs tail-recursion elimination, which destroys information about the call chain, and does not use a runtime stack.

Our **smld** debugger for SML/NJ avoids these problems by using source-language instrumentation instead of machine-based support. **Smld** inserts instrumentation code into the abstract syntax tree obtained from the user's source text at each interesting *event site*, e.g., just before function calls, just after function entries, and when local variables are bound. At each event, the instrumentation code builds an *event record* describing the event and preserving the values of any variables bound there. The record is connected to other event records via static and dynamic links to enable the variable environment and the call chain to be reconstructed by simple runtime debugger algorithms. The instrumentation code also checks whether a breakpoint has been set at the site, 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 expressions are evaluated in the binding environment corresponding to the event where the program halted.

Smld 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, similar to a violation the standard runtime system requires in order to implement the top-level loop. The debugger runs on all of SML/NJ's current hardware platforms, including the MIPS, SPARC, VAX, and MC68020.

Smld-t is an enhanced debugger that supports reverse execution. With reverse execution, users can 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 **smld-t** itself to implement standard 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) are simulated using time travel to perform a binary search over execution history, avoiding expensive checks at each event during forward execution. **Smld-t** implements all program control mechanisms in terms of time travel.

Reverse execution is implemented by taking periodic checkpoints of program state and re-executing as necessary. SML/NJ's use of CPS and heap-allocated closures allow it to implement first-class continuations efficiently. These continuations make checkpointing the "functional" part of program state efficient, and the resulting checkpoints share a single copy of their common data. The state of the mutable store is captured separately by instrumenting store update operations to maintain a history log; log entries use a special "weak" form of pointer that is not followed by the garbage collector, so 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; **smld-t** implements a checkpoint *cache* managed using simple heuristics. **Smld-t** relies on certain SML/NJ language features not found in ordinary ML (e.g., `callcc` and weak pointers) but not on the details of their implementation.

Standard ML has a precise formal definition [88], which describes the typing rules and operational semantics of the language and the relationship between them. However, these semantics say nothing about intermediate points in execution or the values of variables visible from such points. Defining these concepts is non-trivial because of ML's parametric polymorphism; functions can act on values of arbitrary type provided they do so in a uniform manner. Parametric polymorphic functions are not allowed to inspect the types of their actual arguments; this restriction makes it possible to avoid tagging values with runtime type information. But the debugger needs to discover the runtime type of a variable's value in order to print it or use it in expressions; this task is *dynamic type reconstruction.* This reconstruction may also be useful for garbage collectors and for adding polymorphic printing or `eval`-like capabilities to the language.

It is not obvious that dynamic types are even well-defined. To address this problem, we present a modified operational semantics for a subset of ML, in which the notions of an execution event and its environment are made explicit, and give a precise definition of dynamic typing. We also describe an algorithm for reconstructing dynamic types on demand, based on walking the dynamic call chain and the static binding chain.

To assess the performance of **smld** and **smld-t**, we compare their speed and space requirements with those of ordinary code and hypothetical debuggers that use other approaches. Performance is studied using a range of real programs and synthetic "scripts" representing hypothetical debugging sessions. Instrumentation makes code run about three times slower than normal and causes code size (and thus compile time) to increase by about five times. **Smld** generates large amounts of additional live data to store event records and the values to which they refer, which are kept for every pending function application on the call chain. **Smld-t** requires requires less space for event records because it generates them lazily, but more space for logs, and it will fill as much space as it is allowed for checkpoints.

Although these debuggers make large resource demands, we estimate that the alternatives for debugging are also expensive. Any hypothetical machine-based debugger would have to inhibit at least some of SML/NJ's optimizations. We estimate the performance of such debuggers by measuring the effect of inhibiting various sets of optimizations. These measurements suggest that to keep the debugger reasonably simple would require slowing down compiled code by a factor close to three, and would increase live data requirements by failing to eliminate tail-recursion. Another possible approach to debugging is to use an interpreter: SML/NJ's runs 10 to 100 times slower than compiled code. The performance of our debugging approach, though expensive, seems competitive with the alternatives.

Our **smld-c** debugger applies the instrumentation and time-travel technology to ML Threads [32], an extended version of ML that supports concurrent threads of control in the style of Modula-3 [90]. ML Threads has thread, mutex, and condition data types and operators on them. Threads can communicate using shared memory; any `ref` cell or array in the common scope of two threads can be used to pass information. Since the relative execution rates of different threads are unpredictable, Threads programs can be non-deterministic. Our version of ML Threads provides more secure semantics for control of shared memory than most other threads packages.

Maintaining the spirit of the instrumentation approach, **smld-c** operates entirely inside the Threads model. It has no special concurrency privileges, which in principle makes it portable to any implementation of ML Threads. This respect for abstraction makes control of distributed computations (e.g., halting all threads for a global checkpoint) a challenge. Our **smld-c** implementation also defines a temporal model for multi-threaded computations and includes mechanisms for logging non-deterministic events so that programs can be replayed correctly. Having secure programming-language semantics for memory accesses helps reduce the size of these logs.

Prototype versions of ML Threads and **smld-c** run on Silicon Graphics MIPS-based multiprocessors under Irix and Omron multiprocessors under Mach, and on uniprocessors with signal-based preemptive scheduling. We present performance measurements taken on the Silicon Graphics machine.

Portions of the work described here have been previously published as References [111] and [112].

Chapter 2

Debugging Standard ML of New Jersey

Most of the debugging technology discussed in this thesis can be applied to a wide range of programming languages. However, our work focuses on a specific target language, Standard ML, and a particular compiler and runtime system for that language, Standard ML of New Jersey (SML/NJ).[1] SML/NJ has a distinctive combination of characteristics that make it a particularly good test bed for our instrumentation and time-travel methods.

ML supports first-class functions, so compilers must perform extensive optimizations to achieve reasonable execution performance. These optimizations make traditional debugging based on machine code difficult to implement. The language also includes features, such as polymorphic types and exceptions, that are challenging to support properly from any debugger.

ML is *type-safe*; it is possible to understand and debug program behavior using an abstract execution model that ignores the underlying machine. Moreover, most ML programs have few side-effects, and mutable variables are distinguished statically; thus, the state of a typical computation can be described by a continuation plus a small amount of additional information describing the values of mutable variables. Since SML/NJ supports efficient capture of first-class continuations, a checkpoint-based reverse execution mechanism is easy and efficient to implement.

[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 the **smld-t** 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`.

2.1 Standard ML of New Jersey

2.1.1 The Language

This section reviews features of Standard ML that are relevant to sequential debugging. Concurrency extensions for the language are described in Chapter 9. For a general introduction to ML, see the books by Paulson [95] or Reade [97].

Standard ML resembles a typed lambda calculus and supports first-class functions and lambda expressions. Normal programming style in ML is "mostly functional," i.e., side-effect free, but the language has facilities for creating and manipulating mutable objects, for exception handling, and for non-applicative I/O. Because of these imperative features, evaluation order is significant: function arguments are evaluated strictly and from left to right. Functions can be defined as a sequence of alternative *rules*, each consisting of a pattern and a body; when the function is applied, the arguments are compared against each pattern in turn to determine which body to execute. All conditional branching in ML is expressed in terms of this pattern matching mechanism.

ML has an extensible type system, which includes tuples, records with named fields, disjoint unions labeled by *constructors*, and function types. Functions and types can be *parametrically polymorphic*, i.e., a function can be applied to arguments of different types, provided it operates on them in a uniform manner. This restriction ensures that a polymorphic function need only be compiled once, regardless of how it is applied. Thus ML's polymorphic functions differ from, for example, Ada's generic functions, which must be recompiled for each new argument type.

Programs can be checked for type correctness at compile time. The compiler can *infer* the types of identifiers during type checking, so that the programmer need not declare them explicitly. Data objects cannot be deallocated explicitly, so "dangling pointers" cannot occur; all ML implementations provide automatic reclamation of unreachable data. Runtime data objects do not need precise type tags in order to ensure type-safety, in contrast with, e.g., Scheme [29] or Smalltalk [54]. In this spirit, the language does not contain primitives that require precise runtime determination of types, such as polymorphic printing; SML/NJ supports an *overloaded* `print` function, but it is disambiguated at compile time. ML does have a polymorphic structural equality predicate. To support this feature, runtime objects carry enough tag information to distinguish pointers from non-pointers and to determine the size of records; this information also supports garbage collection.

Standard ML has a formal definition [87, 88], which gives explicit static semantics (typing rules) and dynamic semantics (evaluation rules) for both a Core language and an extended Modules language. The definition provides a reference for implementors and also serves as a basis for proving theoretical results about the language. In particular, it is possible to formulate precise soundness theorems stating that statically well-typed programs never perform type-unsafe operations at runtime. These theorems have been proved for substantial fragments of the language and are believed to hold for the full language [87, 110]. The static safety property is crucial: in addition to providing a basis for efficient implementations without runtime type checking, it implies that the execution model for ML programs can be expressed in entirely abstract terms using the structural operational semantics, or some more informal semantics. Details of the underlying implementation and machine can be ignored.

The ML definition specifies a standard interactive interface to the compiler and runtime system. This interface is a read-eval-print loop (the *top-level loop*), familiar from LISP systems, that allows value and function declarations to be entered, compiled, and evaluated. Each successfully executed *compilation unit* adds definitions to a *top-level environment* from where they can be referenced by subsequent units. Initially, the top-level environment is equated with the *initial basis*, which defines standard library functions for arithmetic, list manipulation, I/O, etc.

Standard ML supports dynamically scoped exception handling. Built-in exceptions are raised as a result of internal errors, e.g., arithmetic faults reported by the hardware; new exceptions can be defined by `exception` declarations and raised by explicit `raise` expressions. If no user handler catches a raised exception, control is returned to the top-level interactive loop, which prints the exception's name and terminates execution of the compilation unit. There is no built-in mechanism for reporting the program location from which an uncaught exception was raised, which is an important *desideratum* when debugging.

Standard ML's *module* system manages the name spaces of large programs and supports libraries. A set of declarations can be collected into a named *structure*. Structures may be nested to any depth. Identifiers in a structure are referenced by qualifying them with the structure name, e.g., `Real.sin`; alternatively, an `open` *structurename* declaration loads all the identifiers of the named structure into the environment. The visible components of a structure can be constrained by applying a *signature*, which specifies the name and type of each identifier to be exported. The language also supports a form of module, called a *functor*, that is parameterized by a signature, which is useful for composing abstractions. The types of identifiers declared in a functor may depend on types in its argument structure. Structures may be nested within other modules, but functors and signatures may not. The modules system can be viewed as a language mechanism for controlling compile-time linking; as such, it complicates issues such as symbol-table management and type checking, but has little impact on execution semantics.

Standard ML of New Jersey implements the full Standard ML language with a few trivial omissions and variants; it also provides several extensions to the language. In addition to mutable scalar `ref` variables, SML/NJ supports mutable arrays, implemented so that access or update of any element takes constant time.[2] A first-class continuation is an encapsulation of the state of a program at a particular point in execution. SML/NJ continuations are captured using the primitive call-with-current-continuation (`callcc`).[3] Semantically, a continuation acts like a function that, when invoked (`thrown` to), carries out the remainder of the program's calculation from the point of capture, with the important caveat that the contents of the mutable store are *not* affected. A continuation can be invoked any number of times, at any point in the subsequent execution of the program. In a conventional runtime environment, a continuation could be thought of as a data object containing a copy of the contents of the stack and program counter at the point of capture, but, again, no copy of the mutable store. The SML/NJ programmer can install customized signal handlers for most Unix signals; this feature is chiefly used for catching keyboard and timer interrupts. A signal handler can change the flow of control by `throw`ing to a new continuation, or returning to the interrupted program point.

2.1.2 The Compiler

The SML/NJ compiler [11, 13] processes a source program by successively rewriting it into a series of intermediate representations. Each rewriting step is largely independent of the others. The principal phases of compilation are illustrated in Figure 1. Source programs are parsed into conventional abstract syntax trees. Each syntactic element is tagged with the line number and column position of the original source text from which it

[2] Most other ML implementations also provide arrays.

[3] This operator is similar to `call-with-current-continuation` or `call/cc` in Scheme [29]. SML/NJ continuations are *typed*, however [42].

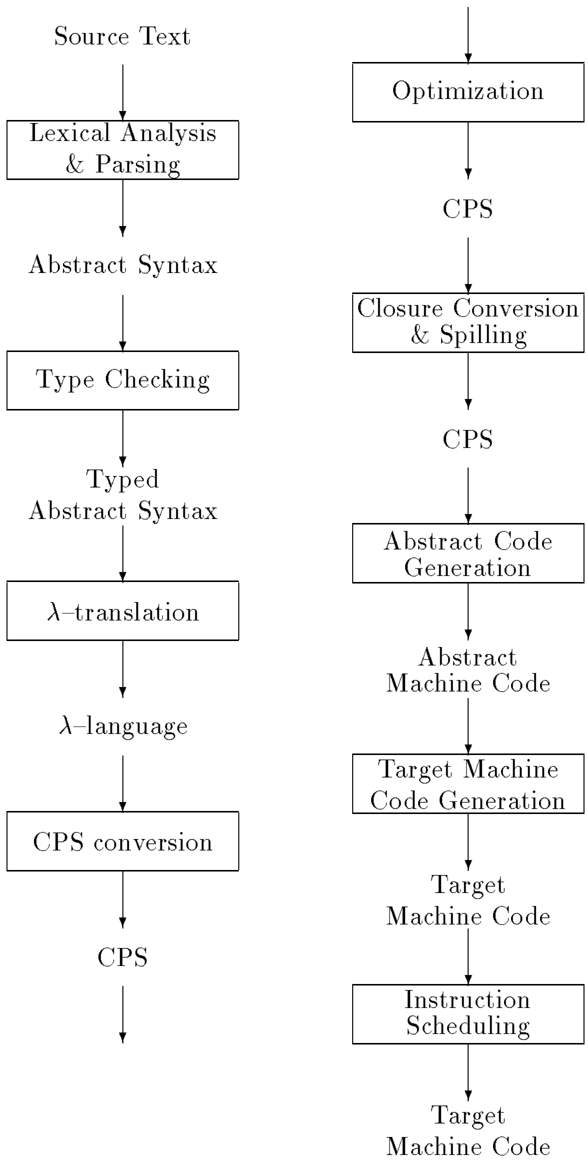

Figure 1: SML/NJ compiler phases. Parsing and Type Checking are actually interleaved.

was derived. In the typed abstract syntax, the program represented is guaranteed to be well-typed, and typings are attached to all identifier nodes. Abstract syntax is translated into a λ-language intermediate representation resembling an applied call-by-value λ-calculus. Various simplifications are made at this stage, including the compilation of pattern matches into explicit test and branch code, and the conversion of inter-module references into record field extractions. The SML/NJ system includes a simple interpreter that operates directly on the 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, and functions never return. Instead, they terminate by calling a continuation function representing "the rest of the program." CPS is a useful representation for performing optimizations, and most of SML/NJ's optimizing is done here (see Section 2.1.4).

After optimization, the CPS is transformed to make it look more like assembly code for a Von Neumann machine. Non-local variables of user functions (which appear as free variables in CPS functions) must be stored in memory, so *closure conversion* rewrites the CPS, replacing references to free variables by explicit fetches from heap-allocated records. The standard representation of a function is a pointer to a *closure record*, which contains a pointer to the function's executable code plus the values of its free variables. Code is added to generate a closure record whenever a function is defined, unless all call sites of the function can be determined statically, in which case a more efficient *ad-hoc* function representation may be possible. 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. 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 [8].

Local variables in user functions can be mapped to machine registers, but Von Neumann machines have a limited number of these, so a spilling phase may be needed to guarantee that no CPS subexpression has more free variables than there are registers. At this point, it is straightforward to generate assembly code directly from the CPS, first to an abstract machine code, then to assembly code for a specific target. This phase is the first machine-dependent phase encountered during compilation, although previous phases depend on a few machine parameters, such as the number of general-purpose registers. Supported targets include the MIPS, SPARC, VAX, and MC68020. Instruction scheduling is performed as a further pass for the RISC targets. Machine code is emitted as a position-independent binary byte stream.

The compiler is about 39,000 lines of (almost) Standard ML. In order to implement the top-level loop, which involves generating a machine-code object (treated as a stream of integers) and then executing it (treating it as a function), the compiler needs to bypass ML's type system by permitting code objects to be cast from one type to another. In addition, since ML data objects do not carry runtime type tags, the values produced by executing a compilation unit must be treated as having a generic type `object` when they are passed to the top-level pretty printer and stored in the top-level environment. These type violations mean that the compiler itself does not have the safety property of well-typed ML programs, but the few violations are isolated.

2.1.3 Garbage Collection

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.

Efficient garbage collection is particularly important in this system because all function closures are kept on the heap. SML/NJ uses a two-generation copying collector [10]. Available memory is divided into two half-spaces, and allocation occurs at the low end of the upper space, designated the *new space*. When the new space becomes full, all live data in the new space is appended into the other, *old space*, but it typically doesn't fill up the old space. The allocator is then reset to begin filling the new space again. After some number of these *minor collections*, the old space occupies half the total available memory, though much of its data will no longer be live. This condition signals the need for a *major collection* in which the entire live data graph is traversed, all live data are copied from the old space to the new space, and finally block-moved back to the old space so the cycle can begin again.

Minor collections are much quicker than major ones, since they traverse and copy only those parts of the live data graph that lie in new space. Every object in new space is newer than any object in old space, so immutable pointers in old space objects cannot point into new space. Since most ML objects and pointers are immutable, minor collections normally don't have to traverse pointers into old space; the subgraph of live data accessible from these pointers lies entirely in old space. However, `ref` and `array` variables space *are* mutable, and may therefore point from old space into new space. These pointers are handled by generating extra code after each pointer update that appends the updated object to a global *store list*; minor collections use this list as an auxiliary set of roots. This solution is not entirely satisfactory: in addition to the overhead added to the update itself, the time required to traverse the store list, which typically contains many duplicates, can be a bottleneck in programs that make many updates.

The SML/NJ garbage collector also supports a special language extension that allows user programs to detect when a particular heap object has become inaccessible. In the standard language, there is no way to detect such objects, since holding any reference to the object keeps it live. The solution is to use a *weak pointer*, which is like an ordinary pointer except that the garbage collector does not follow it. Objects pointed to only by weak pointers are treated as inaccessible during a collection, and the weak pointers are marked invalid by the garbage collector. On the other hand, if an object has both ordinary and weak pointers, and is copied during a garbage collection, the weak pointers are forwarded correctly just like the ordinary ones. Weak pointers have long been part of the folklore of Lisp implementations. Weak pointers were introduced into SML/NJ specifically for our debuggers, but they have application in any situation where the user program would otherwise need to do reference counting to manage a resource.

2.1.4 Optimizations

This section describes some of the optimizations that have a significant impact on the speed of compiled code; they are discussed in more detail in Appel [11]. Except as noted, these optimizations are performed as CPS-to-CPS transformations.

In-line expansion. Functional programs often contain many small functions. Expanding these in line, provided they are not too large and are not called too frequently, can increase performance by removing the overhead of argument passing and—more importantly—opening the possibility for further optimizations on each specialized copy of the function. Since loops are represented as recursive function calls in ML, repeated in-line expansion corresponds to loop unrolling. Functions that are called only once can always be in-lined for both a time and space savings.

η-reduction. If the system is compiling a call to a function `f`, defined as `fun f x = g x`, then it may as well arrange to invoke `g` directly instead and not bother generating code for `f`. This situation is rare in user code but arises frequently as a result of CPS conversion; in particular, eliminating tail-recursion in the source program requires η-reduction.

Argument flattening and uncurrying. All ML functions take a single argument, which is often a tuple. For *known* functions, i.e., functions whose call sites can all be determined at compile time, it may be more efficient to pass the individual elements of the tuple as separate arguments. This optimization can be applied more often if the system first tries to convert curried functions into uncurried versions taking a single tuple argument: this conversion can be done when all call sites are known and all provide the full complement of arguments.

η-splitting. This optimization is the converse of η-reduction: it creates two entry points for a single function, which can be desirable when some, but not all, of the function's call sites are statically known. One entry point is optimized for calls from known sites, as above; the other is used for calls from unknown sites.

Hoisting. This optimization changes the position of a definition in the nesting hierarchy of the CPS expression, thus effectively changing execution order. Definitions can be hoisted up, e.g., out of a recursive function, or down, e.g., into a single branch of a conditional. Hoisting does not ordinarily have a large impact on program performance, but it can make a significant difference in the speed of instrumented code (see Section 4.4).

Argument Representation. If all call sites of a function are known, the representation of its arguments can be specialized so that at least one of the calls requires no register-register moves. This optimization is done during abstract machine code generation.

Instruction scheduling. On RISC machines it is often beneficial to rearrange instructions in order to fill delay slots due to load and store latencies or branch latencies. These optimizations are done during machine-dependent code generation.

These optimizations are similar to those expected of a good optimizing compiler for more conventional languages. The main difference is the emphasis on improving function call code or eliminating calls by in-lining.

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 lives 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.

2.2 Interactive Source-level Debugging

ML programmers can make good use of a conventional debugger, because the source-level model suitable for understanding the execution of an ML program does not differ significantly from that for more conventional languages. In particular, although ML is often used in a mainly applicative style, programmers must consider 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 substitution process. Thus, there is also a natural notion of a current call chain, which it can be useful to display or traverse.

These points distinguish ML from pure *lazy* functional languages (e.g., Miranda [113] or Haskell [66]), in which precise information about evaluation order is likely to hinder rather than help the user's understanding of program execution. The desired behavior of a pure function is specified solely in terms of the output values it computes from given inputs; there are no underlying state changes. Moreover, since functions cannot interact in hidden ways, they can be tested independently. These characteristics encourage debugging approaches organized around the static structure of the program rather than the dynamic sequence of control flow [92]. Shapiro has formalized such approaches into debugging algorithms and shown how they can be partially automated [102]. Structured debugging techniques are also applicable to partially imperative languages such as ML, but the possibility of side-effects can make it more difficult to specify and test functions.

2.2.1 Debugging Model

The standard facilities of our interactive source-level debuggers for ML are drawn from well-known existing tools such as VAX/VMS `debug` [17], `dbx` [30], and `gdb` [105]; Zellweger [120] specifies a similar set of features. Unusual facilities, such as reverse execution and concurrent programming support, are discussed in Chapters 5 and 9.

The model for executing ML programs under debugger control is based on the notion of *events*, i.e., interesting junctures in program execution. Events are located wherever it would be useful to be able to stop the program and examine its current state, or *context*. From the debugger's perspective, each transition from an event to the next is atomic; that is, the internal behavior of the transition is invisible and cannot be explored. Program execution is simply a sequence of events.

To make this model more precise requires an explicit operational semantics for ML, which is discussed in Chapter 7. Informally, events occur at the following junctures.

- Immediately before each function application, i.e., after operator and argument expressions have been evaluated, but before the "call" has been made.
- At the top of each function rule body, i.e., after pattern matching has occurred, a rule has been chosen, and the environment has been updated to reflect the binding of formal parameters to actual argument values, but before evaluation of the body.
- Immediately after each declaration has been evaluated, i.e., just after the defined identifiers have been added to the environment.

Recall that all branching in ML, conditional or otherwise, is expressed in terms of rule pattern matching, so placing an event at the top of each rule body guarantees that there is an event after each branch. There are no events prior to the application of constructors or to the formation of closures because these operations do not affect the state in an interesting way. There are no events at function returns; some debuggers report the values returned by functions, but ours does not. The reasoning behind this somewhat arbitrary choice is as follows: if there is some runtime action associated with an event—as is the case in our implementations—then placing events at the ends of functions would destroy the possibility of tail-recursion. Transitions between events are atomic, so asynchronous phenomena that might arise between two events, including keyboard interrupts and uncaught arithmetic exceptions, must be mapped to a neighboring event.

In a source-level debugger, we expect the program's state to be displayed and queried using the concepts, terminology, and the syntax of ML itself. However, our model also includes aspects of program execution, such as the notion of a call chain, that are not explicit in the language. The context information associated with each event consists of the following data.

- A unique source code *site*, i.e., a position in the source text in or near the source expression being evaluated at the event; the exact positioning of sites is a matter of convention.
- An *environment*, mapping identifiers in scope at the source site to typed values.
- A *call chain*, a list of contexts representing the "stack" of suspended function applications.

A context is similar to a continuation, but contains more information: for example, continuations do not include the values of identifiers that are dead, even if they are still in scope. Also, of course, a continuation is a functional object, which cannot be decomposed into the components of interest here.

2.2.2 Debugging Commands

There is a small set of essential debugger control and query commands.

- Set a *breakpoint* at a specified site in the source text. When the executing program reaches any event corresponding to that site, control passes to the debugger, with the *current context* set to the context for that event.
- Halt execution when a keyboard interrupt is raised, setting the current context to the event at the time of the interrupt.
- Halt execution when an uncaught exception is raised, setting the current context to the event where the raise occurred.
- Restart execution from the current event.
- Display the values of identifiers in the environment of the current context, formatted appropriately according to their type.

- Adjust the current context by moving up (or down) the call chain. As a variant of this operation, display the *call trace*, i.e., the sequence of contexts on the call chain above the current context.

We assume that these functions can be invoked via an *interactive debug monitor* that reads and processes commands from the programmer, controls the program's execution accordingly, and displays the results before prompting for additional commands. This definition of "interactive debugging" excludes batch-oriented debuggers, where the entire command stream must be specified at the beginning of the debugging session. It also excludes debuggers that record a complete execution history and then emulate execution in response to user commands. These debuggers can give a partial illusion of being interactive, but they fall short in two ways: it is necessary to execute the entire program (which may take a long time) before beginning to debug, and it is not possible to alter the course of program execution by changing the state.

A useful debugger has additional features that can be synthesized from those listed above, including single-stepping from one event to the next; "skipping," or single-stepping from one event to the next event in the same function without stopping in any lower-level function; conditional breakpoints, where control is transferred to the debugger only if a boolean condition evaluates to true at the specified location; programmable breakpoints, for which the debugger is instructed to execute a user-defined procedure when the break occurs; and tracing operations, which can display control flow or the update history of a variable. There are an indefinite variety of compound operations like these, so the basic operations should be designed so that they can be incorporated into more complex, user-defined commands. It may also be possible to produce more efficient versions of these operations by implementing them directly as primitives.

2.2.3 Accessing the Dynamic Environment

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. The best approach is to provide a source-language expression evaluator, parameterized by the environment of the current context. This approach allows the programmer to deconstruct complex values by evaluating arbitrary ML expressions. It should also be possible to call user-defined printing functions, defined within or external to the program.

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. It is also useful for the debugger to be able to report the binding context, including source-code site, for an identifier; this feature, used recursively as necessary, enables the programmer to inspect the λ-expression that defines the code bound to a function-valued variable.

The major challenge faced by an ML debugger in displaying values is coping with polymorphic types. Consider the following simple program, in which `pair` is a polymorphic function with type scheme $\forall\alpha.\alpha \rightarrow (\alpha \times \alpha)$, and is applied first to an integer, then to a boolean:

```
let fun pair x = (x,x)
  in pair 1;
     pair true
  end
```

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 type of `x`, when the code for `pair` provides no information about its argument's type, and runtime values themselves do not carry type tags?

Intuitively, there are several possible solutions to this problem. Each applied occurrence of `pair` is statically typed with a different instance of the type $\alpha \rightarrow (\alpha \times \alpha)$ that depends on the types of the surrounding expressions. If we can keep track of which instance of `pair` is being executed, we can use this information to determine the correct type to substitute for α. Alternatively, we can assign `x` the type of the actual argument to which `pair` is applied [9]. In this example, both applications of `pair` take as argument a base constant of manifest type; in general, the argument may be a variable or expression whose type must be inferred, and the calling function may itself be polymorphic, requiring us to apply this dynamic type reconstruction process recursively. We describe these ideas more formally and give general algorithms for reconstructing dynamic types in Chapter 7.

In its basic form, the call chain is a list of contexts representing the event from which the current function was called, the event from which the function containing that first event was called, and so forth. In practice, it is also important to be able to find the event at which each function was entered, so it is convenient to think of the call chain as a list of (caller, callee) pairs of contexts. This list behaves just like a conventional call stack.

When an exception is raised in some function invocation, it is handled (if at all) by code executing in a function invocation somewhere "up the stack" from the first one. It is conventional to think of the stack as being "unwound" by the exception, and the stack contents below the handling invocation as being lost. But these contents include the context in which the exception was raised, which may be of interest for debugging purposes, so they should be kept. The debugger could store them on a separate "exception stack", but in general an indefinite number of stacks might be needed, because the handler can begin an entirely new computation that raises new exceptions in turn, and so forth. Therefore, the debugger does not "unwind" the call chain when an exception occurs; instead, it treats handled exceptions uniformly with function calls by appending specially marked (raiser, handler) context pairs to the call chain. Thus, the chain is no longer equivalent to a conventional stack, but instead represents a history of all function and handler invocations that have been entered but not properly returned from. The "unwinding" effect occurs only if the function invocation containing the handler returns normally.

Programmers who use `callcc` and `throw` must interpret the call chain with additional care, since these primitives effectively save and restore it; the `throw` itself does not appear on the chain.

2.2.4 Altering Program Execution

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 can be provided by permitting the programmer 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 recompilation; although some debugging systems (notably for Interlisp [109] and Smalltalk-80 [54]) permit dynamic recompilation, we do not.

2.2.5 User Interface

Our debugger uses a screen, keyboard, and mouse-driven interface, implemented as an extension to the GNU Emacs editor, roughly in the style of `gdb` [105]. 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 programmer can select other sites in the source by scrolling and pointing at the text. Debugger commands can be issued from the keyboard by name or using single keystrokes. Specialized debugger commands can be written as ordinary ML functions, typed in or read from a file, and invoked as macros from the keyboard.

2.3 Machine-Level Debugging and Optimization

Most computer systems have a debugger that operates at machine level. Debugger operations, such as setting breakpoints and displaying data values, are performed by directly inspecting and changing machine addresses and registers. As source-level debuggers for compiled higher-level languages have become almost universally available, they have normally been implemented by layering a source-level interface on top of a machine-level debugging kernel. There are two major reasons for this approach.

- Machine-based debugging is efficient. Most debuggers are trap-oriented: they allow the object program to run at full speed most of the time, and arrange to cause a trap to the debugger when an event of interest occurs. The most important kind of trap is a location breakpoint, which is usually arranged by patching the program's instruction stream. Many processors also have special hardware intended to support machine-level debugging, e.g., a mode in which a trap occurs after each instruction, intended to aid single-stepping.
- Many languages are unsafe: they cannot prevent a faulty program from damaging its runtime environment, exposing the underlying machine.[4] To handle such programs, the debugger must provide the programmer with tools to view and manipulate their compiled (machine-level) representation.

2.3.1 Compiler/Debugger Interface

For a machine-level debugger to provide a source-level interface, it must have access to a set of maps between the source program and its machine code. In particular, the following maps must be provided.

[4] Of course, any program may damage its environment if there is a bug in the compiler or a runtime component. We ignore this important problem.

- A map from source-language program sites to the corresponding object-code addresses, needed in order to set breakpoints.
- An inverse map from object-code addresses to sites, needed to determine the site corresponding to an unsolicited exception or interrupt.
- A map from source-language variables to the memory locations or registers where the corresponding values are held, needed to display or update values.

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 model using the maps described above.

Machine-level debuggers depend on their target hardware, their runtime system, and their compiler's code generator, as well as on the source languages being debugged [21]. Although a modular design can be used to isolate the dependencies on each of these components, in practice, most debuggers tend to be wedded to their target machine, or at least to a fixed runtime system interface. There has been somewhat more progress toward language independence, particularly where a common runtime environment or code generator is used [17]. Unfortunately, the desire to present a uniform debugger user interface across multiple source languages typically leads to the introduction of a new debugging command language, which overlaps each of the source languages in functionality but differs in style and syntax from all of them. The need to learn this new language can pose a substantial barrier to user adoption of the debugger [107].

A number of high-quality programming environments have been developed by taking the opposite approach, and eschewing independence in favor of strong integration of debugger, compiler, editor, and other tools [54, 108, 109]. It helps to develop the compiler and debugger together, so that the format and interpretation of mapping information can be fixed by mutual agreement.

2.3.2 Optimization

Providing the mapping information is not too hard for a compiler that translates source code to object code in a fairly direct manner, so that the maps are simple and one-to-one. But an optimizing compiler can introduce many difficulties, primarily because the maps can become complicated. Zellweger [120] identifies two major categories of problems.

- Data problems involve reporting incorrect values. These may arise because the value doesn't exist in the object code (e.g., due to constant propagation), because the debugger looks for it in the wrong place (e.g., the source-to-runtime memory mapping is wrong or unavailable), or because the value has been discarded (e.g., as a dead variable) or has yet to be computed (e.g., due to code motion).

- Location problems concern difficulties in setting breakpoints, displaying the current location at a breakpoint or after an error or interrupt, or displaying the call chain. These problems reflect difficulties with the mapping between source-code and object-code locations: some source code may have no corresponding object code, the mapping may be many-to-one or one-to-many because source code has been duplicated or collapsed, or the mapping may be distorted by code rearrangement.

Traditionally, machine-based debuggers have not handled these problems well. Zellweger distinguishes two reasonable levels of support that a debugger might give: *expected behavior*, where the debugger acts as if the optimizer were not present, and *truthful behavior*, where the debugger allows the user to see that program transformations have occurred, and either presents the restructuring in source terms or admits defeat when it cannot answer a query. We desire expected behavior, but a debugger offering truthful behavior is better than one that simply ignores the effects of optimization. Most systems either disable optimization when debugging, thus offering truthful behavior the easy way, or leave it to the programmer to figure out the effects of the optimization, thus offering less than truthful behavior.

2.3.3 SML/NJ Optimizations

The SML/NJ optimizations described in Section 2.1.4 offer numerous challenges for the would-be designer of a machine-based debugger.

A typical source function is split into a set of CPS functions; an elaborate mapping scheme would be needed to avoid the resulting location problems. To solve the corresponding data problems, 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 is fundamentally straightforward and should not carry any runtime cost, but it appears to be a substantial bookkeeping task. On the other hand, η-reduction is fundamentally hard to deal with, since it removes the reduced function from the object code; to provide expected behavior, either η-reduction must be prevented or an additional parameter to indicate which original function was applied must be passed at runtime, which may be expensive.

There is no runtime stack, so where is the debugger to get the call chain? Presumably the continuation function closures contain the necessary information, but it is not obvious how to extract it. CPS conversion automatically avoids generating storage space for dead variables, so in-scope variables must be kept live artificially, which removes any space benefit of tail-recursion elimination.

The sheer number of phases in the compiler's back-end would make the bookkeeping problems tedious. For example, the argument representation optimization is not particularly difficult to cope with on its own, but it is implemented at a different compilation phase from the bulk of the (CPS-based) optimizations, thus requiring separate bookkeeping. Instruction scheduling, which would introduce further location and data problems, occurs in yet a different phase.

In sum, the prospect of designing a machine-based debugger for SML/NJ is unpleasant. The bulk of this thesis is devoted to exploring an alternative approach.

2.4 Interpreters

Interpretive environments offer a number of advantages for debugging systems. 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. In particular, it is feasible to insert explicit statements into the source code to display the internal state of the computation at a problematic point; this manual instrumentation approach is discussed more fully in Chapter 3. 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., Reference [109]), and this approach has also been used for C (e.g., Reference [71]). Since interpreters are typically fairly machine-independent, debugging systems based on them are also portable.

The main drawback to interpretation is slow execution speed. To deal with large programs, some systems allow interpreted and compiled procedures to be intermixed at link time. The internals of compiled procedures cannot be debugged, although calls to these procedures can still be traced, e.g., by forcing all calls to go through a tracing procedure. If a bug is suspected inside a compiled procedure, the user reloads an interpreted version of that procedure and examines its execution in detail. Since it is difficult to predict which procedures will be faulty, this approach is most useful if the system allows a compiled procedure to be replaced by its interpreted variant dynamically, without halting the program [54].

Debuggers that combine interpreting and compiling must solve many of the same problems as debuggers based on pure compilation. Many of the results of this thesis can therefore be applied to the compilation portion of a combined system.

2.5 Related Work

This section describes general work on debugging ML and debuggers for optimized code. Related work on other relevant aspects of debugging is described in the appropriate chapters. General discussions of source-level debugging may be found in the dissertations of Gaines [51], Satterthwaite [100], Zellweger [120], and Bruegge [21]. The published literature on debugging is sparse; much of it has appeared in the proceedings of three ACM workshops on debugging [69, 85, 86].

2.5.1 Debugging ML

Despite the existence of several working compilers for Standard ML, and a substantial body of users, there are no other source-level debuggers for the language. This situation is possibly 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 [23], and a debugger is therefore less important.

Ophel [93] describes a programming environment for ANU ML, a non-standard variant of ML developed at the Australian National University. His AIMLESS system supports program development based on an extended version of the standard read-eval-print loop. The system maintains a history of entered phrases; phrases can be undone and redone, edited, or grouped together to form larger program units. AIMLESS supports testing by saving the values computed when an expression is entered and evaluated; if the expression is subsequently re-evaluated after a program change, the old and new values are automatically compared. To support debugging, AIMLESS includes a facility for tracing the arguments and return values of specified functions. Tracing is implemented by patching the code pointer for traced functions to point to a variant version of the function that includes appropriate `print` statements. AIMLESS cannot print the values of polymorphic variables properly.

The CAML system [96], another non-standard ML implementation, includes a facility for tracing the arguments and return values of functions. Tracing is implemented by modifying the function's internal closure representation. Polymorphic variable values are not printed properly.

2.5.2 Debugging Optimized Code at Machine Level

Hennessey [64] addresses data problems: the detection and possible recovery of "noncurrent variables," i.e., variables whose values would be incorrect in the optimized program if they were fetched from the memory locations they occupied before optimization. He ignores the possibility of register assignment. He makes a detailed study of optimizations within basic blocks, including common-subexpression elimination, redundant-store elimination, and reordering by the code generator, all considered in the framework of a dag representation. He suggests augmenting the dag so that it reflects both the original and the optimized computation, and making the dag available to the debugger. This approach always provides enough information to determine whether or not a variable value is current. Sometimes it provides enough information to reconstruct the correct value of a non-current variable, typically by recomputing the value of a dag node at debug time; this occurs 58% of the time in an empirical study of pointer-free Pascal programs. Corrected and improved versions of these algorithms are suggested by Wall, Srivastava, and Templin [115].

Hennessey also addresses data problems arising from global optimizations, including code motion out of loops, induction variable elimination, and dead-store elimination. The first two of these have fairly straightforward solutions: unrolling a loop once removes any non-currency problems (this insight is due to Wall, *et al.* [115]), and the values of eliminated induction variables can always be reconstructed given adequate information from the compiler. Dead-store elimination problems, on the other hand, are essentially impossible to detect accurately, much less to correct, without the addition of runtime control-flow tracing that would cost more execution time than the optimization saves.

Zellweger's thesis remains the most comprehensive and readable treatment of the issues involved in debugging optimized code [120]. Much of it is devoted to a survey and classification of optimizations, the problems they pose for a machine-based debugger, and potential solution techniques. The remainder of the thesis focuses on debugging problems caused by two fundamental control-flow optimizations: in-line procedure expansion and "cross-jumping" (automatic detection and merging of identical code sequences by the compiler). To study these problems, Zellweger implemented a debugger called Navigator for the Cedar language. The debugger was closely integrated with the existing Cedar compiler and associated programming environment; in particular, Zellweger modified the compiler to produce additional information for Navigator's use. The main conclusion of this work is that coping with in-line expansion is relatively straightforward—it involves increased bookkeeping at compile time, but no additional runtime cost—whereas coping with cross-jumping is difficult. The fundamental problem is that cross-jumping produces a many-to-one mapping from source to object code, so that the debugger cannot know which of several source locations corresponds to a current object-code location without gathering additional information at runtime. Zellweger uses internally activated breakpoints ("path determiners") to inform the debugger which path has been taken into a merged region. This method slows down execution so much that it appears to be acceptable only when the need for accurate determination is predictable. Therefore, Navigator provides expected behavior at deliberately placed breakpoints (since determiners can be placed around such points), but merely truthful behavior when reporting the current location at exceptions or keyboard interrupts. Moreover, proving the correctness and performance properties of the path-determination algorithms appears to be lengthy and complicated, and interactions between the two optimizations cause additional problems. In Zellweger's words, the ideas behind Navigator may be simple but the implementation is "less than straightforward." She concludes that, in general, truthful behavior may be a more realistic goal than expected behavior.

Coutant, Meloy, and Ruscetta [33] address the data problems resulting from register assignment of data, induction variable elimination, and instruction scheduling. They discuss a prototype debugger (DOC) built to match an existing global optimizer and code generator for the HP9000 Series 800. Guidelines for this effort included no inhibition of optimization in the compiler, minimal performance impact, and tolerating merely truthful behavior. The primary technique is generation of better bookkeeping information by the compiler, based on the notion of a variable range, which describes where a variable may be found in memory or the register set according to the current program counter. Ranges are built during the compiler's initial dataflow analysis and are maintained during copy elimination, register assignment, and spilling. The range record is augmented during instruction scheduling if the scheduler causes a variable-updating instruction to move outside of the range associated with the source program statement containing the update; in this case, the debugger can truthfully inform the user that the value reported is non-current, but does not attempt to reconstruct the correct value. The prototype implementation displays low overheads (3.5% increase in object-file size and and 3% increase in compile time) and quick runtime response. In sum, the optimizations considered by DOC can be truthfully handled in a straightforward way, provided that the necessary bookkeeping work can be assigned to the compiler.

Zurawski and Johnson [121] describe how to produce expected debugging behavior by selectively inhibiting the compiler from performing certain optimizations. The key idea is to define a set of inspection points, i.e., potential breakpoints, at which the debugger may be invoked, and have the compiler compute recovery functions that allow the debugger to map the machine-level state of the optimized program at any inspection point to the corresponding state, or context, in the unoptimized source-level program. The authors formally define contexts and operators on them, and show how standard debugger commands can be implemented in terms of these operations, thus achieving independence from the optimizer and underlying machine. Unlike the other work discussed in this section, Zurawski and Johnson include support for changing values, and even for recompiling parts of a running program and continuing execution using the new code; in fact, the latter technique is used to support breakpoints and apparently single-stepping. This approach improves compiled code performance by minimizing the number of inspection points required.

The bulk of the paper describes how recovery functions interact with, and inhibit the effects of, optimizations including in-line procedure expansion, global dead-store elimination, register allocation, and tail-recursion elimination. One issue considered in depth is how recovery functions can extend the life of variables, preventing their space from being reused.[5] The authors also present a formalism for proving the correctness of the various transformations in the context framework. These ideas have been implemented using a compiler for Typed Smalltalk. Thus far, only toy programs (e.g., bubble sort) have been compiled successfully. These show worst-case execution time slowdowns of 17% and size

[5]This is very similar to the "tracking effect" we measure in Section 8.5.

increases of 35%. Most of the size increase is due to additional register spills in optimized code.

Chapter 3

Source-Code Instrumentation

One of the oldest debugging techniques is to insert explicit code into the source program to print values at points of interest. In the absence of an effective source-level debugger, this "insert a PRINT statement" method may be the only one available, and it can be a useful approach in any case. On the other hand, instrumenting by hand is tedious and time consuming, particularly when the location of a bug is unknown.

Perhaps the most attractive aspect of this instrumentation technique is that it operates by transforming the source code, and so is independent of the system used to implement the language, including any compiler optimizations, and also of the machine on which the system runs.

This chapter shows how a Standard ML program may be *automatically* instrumented at compile time to support interactive source-level debugging at runtime. Chapter 4 gives a detailed description of **smld**, a debugger based on this approach.

3.1 Instrumenting by Hand

Manual instrumentation can be a simple and effective way of diagnosing bugs, particularly if the programmer already has an idea of where to insert diagnostic code and what variables to print. The method is flexible and may be the approach of choice when the diagnostic output desired, e.g., the contents of a complex data structure, is specialized enough to require programming in its own right.

Instrumentation works well for Standard ML because it is a type-safe language. In this respect, ML is similar to recent members of the Modula family (which also use compile-time type checking) and to Lisp and Scheme (which use runtime type checking), but quite different from FORTRAN and C, in which it is easy to generate a program that writes anywhere in memory. ML's safety guarantee has two important consequences.

- Instrumentation can be added to a faulty source program without fear that the instrumentation itself will be corrupted by the bugs.
- The runtime behavior of ML programs—even faulty ones—can be understood without reference to an underlying machine model. That is, it is *never* necessary, or even useful, to examine machine registers or memory locations in order to diagnose a bug. The other side of the coin is that ML doesn't *allow* us to inspect the underlying machine, again in contrast to FORTRAN and C.

Manual instrumentation does have drawbacks. 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, retranslate, and re-run the program. If an interpreter or a fast, perhaps incremental, compiler is available, this approach may be viable, but 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, most languages provide mechanisms for printing the values of in-scope variables, but few provide direct methods for inspecting the call chain.

Finally, to instrument a program by hand the programmer must alter the source text. 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 programmer will have built most of a debugger.

3.2 Automatic Instrumentation

A better idea is to generate the debugging code automatically. The key features of an interactive source-level debugger can be implemented by automatically inserting instrumentation statements at *every* location in the source program where the user might want to make a debugging query, i.e., at every site as defined in Section 2.2.1. For example, wherever an identifier is bound, the debugger can automatically insert code to report its value; wherever a function is called, the debugger can insert code to report the caller, the callee, and the parameter values passed; and so forth. The instrumentation can support both identifier look-up and control-flow tracing queries.

This instrumentation code will look similar to manual instrumentation, but somewhat more sophisticated. The programmer typically doesn't want all available state information to be reported unconditionally at every site, so the debugger must store the information rather than printing it, and arrange a method for selectively retrieving it. Moreover, the debugger must be able to control program execution in addition to monitoring it. To match the model described in Section 2.2.2, the debugger will need to transfer control temporarily to an interactive debug monitor at specified breakpoints; the monitor allows the programmer to query for specific state information. So the general form of an inserted instrumentation block will be

```
(recordState();
 if (shouldBreakHere()) then
   invokeDebugMonitor()
 else ())
```

Function `recordState` preserves the program's state at this point, e.g, the values of variables, the contents of the call chain, etc. Function `shouldBreakHere` returns true if execution should be suspended at this point; evaluating the function will typically involve consulting a runtime data structure controlled by the programmer, such as a list of breakpoints. The debug monitor accepts and acts on debugging requests; when the programmer is ready to continue execution, control returns to the instrumentation point.

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`, just as in the compiler's top-level loop (see Section 2.1.2). Fortunately, this violation does not weaken the overall safety 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 can 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 more and prevents it from significantly altering the internal course of the computation.

An obvious danger is that our approach to debugging will be hopelessly inefficient, either in time, space, or both. A typical program has many sites—an average of 1.5 per source line for the programs we've measured (see Table 3). Since `recordState` and `shouldBreakHere` must be evaluated each time execution passes through any site, execution under the debugger will be several times slower than normal, even if no debugging activity is performed. Moreover, although optimizations are still enabled, many of them will probably be inhibited by the presence of instrumentation, causing further slowdown. The instrumented program may generate or hold on to more live data than the original in order to hold state descriptions. Finally, the instrumented version of a source program will be several times larger than the original, the compiled code will be larger, and compilation time will be longer. Compilation time is a particular concern in the SML/NJ system, because the compiler is slow. Performance is discussed in more detail in Chapter 8.

3.3 Related Work

Automated instrumentation techniques have a long history in debugging. This section surveys some of the relevant work. Instrumentation-based debuggers for parallel programs are discussed in Chapter 9.

3.3.1 Early Systems

BUGTRAN [47], a batch-based debugger for FORTRAN programs, was one of the first to operate at the source level. It supported selective tracing of variables, control flow, and procedure calling as well as snapshot dumps and an assertion mechanism. The system worked by transforming the source program; the transformation was controlled by preceding an ordinary FORTRAN source deck with special BUGTRAN control cards. BUGTRAN's use of instrumentation lies somewhere between manual and automatic: the regions of program code to be instrumented must be specified explicitly, but the instrumentation is generated automatically and the original source deck remains intact.

Balzer's Extendable Debugging and Monitoring System (EXDAMS) [16] is a very ambitious debugger; many of his ideas are still as fresh (and as underutilized) as they were twenty years ago. The author's own words describe EXDAMS best.

> EXDAMS was designed to satisfy three needs: first as a vehicle to test some proposed, but unimplemented on-line debugging and monitoring facilities; second, as an extendable facility to which new debugging and monitoring aids could be added easily, then tested; and, third, as a system providing some measure of independence of not only the particular machine on which it is being run and the particular implementation of the language being debugged and/or monitored, but also of several source languages in which users' programs could be written and debugged and/or monitored [16, pages 567–568].

The newly proposed debugging facilities include dynamic display of the program counter as a position in the source code and display of values as they change, multiple simultaneous displays, and "flowback analysis" of variable histories, which is discussed more fully in Chapter 5. A prototype was developed for FORTRAN programs; its implementation philosophy is unusual. Although EXDAMS gives the illusion of being an interactive system, the user cannot interact with the program while it is actually executing. Instead, when a program is run under the system's control, a history tape is generated containing all conceivable information of interest about the course of execution. The tape is generated by source-language instrumentation statements inserted in the original program as a preprocessing step. All the user-oriented debugging tools operate on this tape, which can be played backward as well as forward. This approach gives the desired flexibility at the cost of efficiency; it also gives independence from the hardware and compiler. If a suitably universal tape format can be designed, the playback tools can also be independent of the source language.

Our instrumentation approach is fundamentally similar to Balzer's; like ours, his system adds instrumentation code at every interesting point in the program (in fact, at somewhat more points than ours), taking care that the instrumented code is still a valid program in the source language. However, we use instrumentation to support interactive debugging rather than to build a history tape.

3.3.2 Language Mechanisms

Hanson [62] describes debugging of SNOBOL4 programs using event associations, a mechanism for automatically invoking a source-language function each time an event of interest occurs during program execution. Events may include the fetching or storing of a variable value, the execution of a specified statement, a function call, an external interrupt, or a runtime error. The associated user functions are ordinary SNOBOL4 code; certain attributes of the event, e.g, the value being stored in a variable or the statement number being executed, are made available as parameters to the invoked function. Associations between a class of events and a user function are established or destroyed via calls to special built-in functions; the actual invocation of the function when the event occurs is arranged by the runtime system.

This mechanism can be used to support either special-purpose debugging, with the user defining associations and associated functions explicitly, or as the basis for a general-purpose debugging tool, with generic functions. As an example of the latter, the mechanism was used to build a symbolic debugger called SNODDT, modeled on the DEC-10 DDT debugger, with an interactive interface supporting expression evaluation in context, data-type checking, breakpoints, and watchpoints. Event associations were implemented as a modification to the SITBOL interpreter running on the DEC-10. Considerable effort was made to avoid performance degradation in the presence of associations, and even more importantly, in their absence. Hanson's approach has much in common with ours; the main difference is that he implements the association mechanism inside the runtime system, whereas we make the association explicit in the modified source code, and rely on general-purpose optimization techniques to render it efficient.

Another implementation of SNOBOL4, based on an abstract machine, included a facility called "The Window to Hell" (TWTH) to read and manipulate the internal state of the abstract machine from within a SNOBOL program [59]. TWTH was used primarily to debug the implementations of the abstract machine and the SNOBOL interpreter itself, but might also have been suitable for implementing source-level debugging support. The ability to examine and alter the internal state of a language interpreter from within the source language being interpreted is sometimes called *reflection* [49, 80, 104]. Reflective mechanisms can be used to modify the normal operation of the interpreter to support debugging features such as single-stepping and tracing.

3.3.3 Macros

Code transformation techniques come naturally to LISP systems, where code has a simple standard representation that is easily manipulated as data by meta-level programs. LISP systems also typically have powerful macro processors, which can provide an elegant way to specify and implement front-end transformations. Dybvig, Friedman, and Haynes [43] describe a particularly powerful macro approach called expansion-passing style. In this scheme, each macro expansion function (called an expander) takes two arguments: the macro expression to be expanded and an expander to be used for any recursive expansions. This enables macros to control the way in which the substituted text is further expanded. The authors show how this mechanism, together with LISP dynamic binding, can be used to implement powerful debugging aids simply. The idea is to use a macro to replace each form in the source code by a call to a debugging wrapper function that takes the original form as argument. The wrapper itself is not a macro, but an ordinary function that takes control before and after the evaluation of the original form. Depending on the level of debugging support desired, the wrapper can trace the arguments and result, or pass control to an interactive debugging loop that supports single-stepping and display or update of variables using an emulated environment. The final program (after full expansion) is similar to our instrumented programs; the main difference is that we wrap specialized debugging code around each expression of interest, rather than using a generic wrapper function. As with our approach, the expansion-passing debugger produces a large increase in code size. Also, although no performance figures are given, the wrapper function must incur a substantial runtime cost; this problem can be mitigated somewhat by limiting the debugging macro's (static) scope of application when the programmer has an idea of where the bug lies.

3.3.4 Monitoring Semantics

Kishon, Hudak, and Consel [72] aim to give a sound semantic foundation to the notion of program monitoring, including debugging, profiling, tracing, etc. They 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. This transformation is performed by augmenting the standard valuation functions with hooks that invoke monitoring functions before and after each evaluation step. These monitoring functions are triggered by the presence of special monitoring labels inserted into the (abstract) source program; it is envisioned that labeling will be performed automatically by a tool. The scheme is parameterized by the desired monitoring activity. Choice of monitoring labels and functions is tailored to the desired activity; for example, a simple profiler might label all named function bodies, and include a monitoring function that maintains a call count for each named function. A more elaborate interactive debugger may be implemented using monitoring functions that communicate with the user via lazy stream I/O. Implementation is by means of partial evaluation; one level of specialization gives an interpreter instrumented with monitoring actions; a second level gives an instrumented user program.

Such an instrumented program resembles the code resulting from our instrumentation process. The main difference is that while Kishon, *et al.* embed monitoring actions in a non-standard semantics, we express them in the source language and rely on the standard semantics. Simply put, they change the language; we change the program. A practical advantage of our approach is that we can rely on a standard compiler, which gives considerably better performance than an interpreter specialized to a source program via partial evaluation. Specializing a Haskell interpreter with respect to a source program roughly doubles its speed, whereas our compiler generates object code orders of magnitude faster than our admittedly primitive interpreter.

Chapter 4

Implementing Instrumentation

Smld is an instrumentation-based debugger for SML/NJ. It inserts simple and uniform instrumentation code that supports execution control, variable lookup, and control-flow tracing. This chapter describes the details of the instrumentation code and the resulting object code. For simplicity, our initial description of **smld** uses a subset of the core Standard ML language, the syntax for which is specified in Figure 2; Section 4.5 extends these methods to the full language.

4.1 Implementing Sites and Events

Smld's implementation is based on the notions of debugging sites and events, introduced in Section 2.2.1. Recall that a program site is a specific location in the program source text, corresponding (informally) to a program point of potential interest during debugging, and that a conceptual event occurs each time the program executes to or through a site. **Smld** places instrumentation code at each site, and the execution of that instrumentation code is the concrete realization of the associated event. Note that sites refer to the *static* program representation, whereas events refer to *dynamic* program execution.

Each site has an associated *site type*. For the subset language, **smld** 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. There is a well-defined mapping from sites to source-code locations, illustrated in Figure 2 for the subset language. A partial inverse mapping from source-code locations to sites is also implied; this mapping can be extended to a total mapping from source-code locations

$exp ::=$	$nconst$	(nullary constructors, i.e., constants)
	$uconst$	(unary constructors)
	var	(variables)
	op	(primitive operators)
	$exp_{opr}\ {}^{\dagger}\ exp_{arg}$	(function applications)
	$\texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end}$	(local declarations)
	$\texttt{fn}\ rule_1 \mid \ldots \mid rule_n$	(functions)
	$\texttt{(}exp_1\texttt{,}exp_2\texttt{,}\ldots\texttt{,}exp_n\texttt{)}$	(tuples)
	$\texttt{(}exp_1\texttt{;}exp_2\texttt{;}\ldots\texttt{;}exp_n\texttt{)}$	(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)
$rule ::=$	$pattern\ \texttt{=>}\ {}^{\dagger}\ exp$	
$pattern ::=$	`_`	(wildcard, i.e., match anything)
	var	(variable)
	$nconst$	(constant)
	$\texttt{(}uconst\ pattern\texttt{)}$	(constructions)
	$\texttt{(}pattern_1\texttt{,}pattern_2\texttt{,}\ldots\texttt{,}pattern_n\texttt{)}$	(tuples)
$dec ::=$	$\texttt{val}\ {}^{\dagger}\ pattern_1\ \texttt{=}\ exp_1\ \texttt{and}\ \ldots\ \texttt{and}\ pattern_n\ \texttt{=}\ exp_n$	(ordinary declarations)
	$\texttt{val rec}\ {}^{\dagger}\ var_1\ \texttt{=}\ exp_1\ \texttt{and}\ \ldots\ \texttt{and}\ var_n\ \texttt{=}\ exp_n$	(recursive declarations)
	$dec_1;\ldots;dec_n$	(sequences)

Figure 2: 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 $\texttt{[}exp_1\texttt{,}exp_2\texttt{,}\ldots\texttt{,}exp_n\texttt{]}$ is syntactic sugar for the pattern or expression $exp_1\texttt{::(}exp_2\texttt{::(}\ldots\texttt{::(}exp_n\texttt{::nil)))}$. 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.

```
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
```

Figure 3: The `rev` function.

to the nearest preceding (or succeeding) site. `FN`, `VAL`, and `VAL REC` sites are called *binding sites*, since 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 (see Section 4.6). To debug the full language, a larger set of sites is defined, as described in Section 4.5. Each site in a program is given a unique identifying *site number*, which can be used to index tables or lists of site information.

Since ML is statically scoped, there is a well-defined function, `enclosingBinding`, from sites to binding sites, which gives, for each site s, the innermost nested site that binds some variable visible at s. This function can be visualized as a directed graph in which the nodes represent sites and there is an edge from s to t if `enclosingBinding`$(s) = t$. For the subset language, this digraph is actually a tree[1] with pointers oriented from child to parent; it is convenient to add a `nullSite` node (having site number 0) at the root. Repeatedly following `enclosingBinding` edges from a site s to the root encounters exactly those sites that bind variables visible at s.

Figure 3 shows a fragment of code that reverses a short list; this fragment serves as a running example in the remainder of this chapter. This program is fairly realistic except for the definition of `cons` in terms of the built-in cons operator (`::`), which has been introduced for pedagogical reasons only. The superscripts are site numbers; these are referenced in Figure 4, which depicts the program's binding tree. Each branch in the definition of `val rec r` has a separate `FN` site; these sites would be present even for branches that failed

[1]Section 4.5 describes features in the full language that prevent it from being a tree.

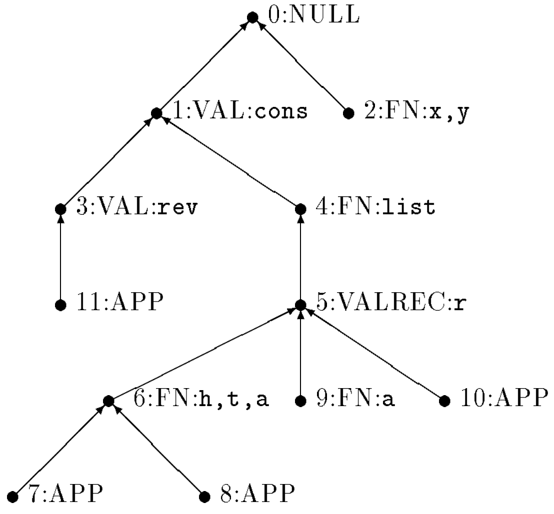

Figure 4: Binding tree for `rev`. Each binding site is annotated with *site number*:*site type*:*bound variables*.

to bind any variables. Constructors such as `::` are *not* instrumented, because they operate atomically; `cons`, on the other hand, is instrumented. Note that because `r` is recursive, it is visible within its own body, so its binding site (5) appears in the scope tree as an ancestor of sites 6–9, as well as of site 10; on the other hand, `rev` is not recursive, so its binding site (3) is only an ancestor of site 11 in the scope tree.

4.2 Instrumentation Code

To support interactive query and control commands, **smld** must be able to halt selectively at any event, and obtain the identity of the current site, the current call chain, i.e., the identity of each pending `APP` event, and the current values of all identifiers in scope at this site or at any site on the call chain. Most of this information changes little from event to event and site to site, so it makes sense to store it incrementally. **Smld** inserts instrumentation to generate an *event record* each time an event is executed. This record contains only the state information unique to the event, together with pointers to other event records that make it possible to reconstruct the entire state. An event record is generated whether or not a break is executed at the event; if a break does occur, the newly generated record is passed to the debug monitor.

Figure 5 gives an ML signature describing event records and basic functions on sites. The

```
(* Basic types *)
datatype 'a option = SOME of 'a | NONE
type siteNumber = int
datatype siteType = FN | APP | VAL | VALREC
type id = string
type value = object

(* Functions on sites *)
val siteTypeOf: siteNumber -> siteType
val idsBoundAt: siteNumber -> id list

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

Figure 5: The `eventRecord` data type.

`boundValues` list contains an entry for each identifier bound at the event. To allow the list to contain values of arbitrary types, all values are cast to the universal type `object`, which is defined in SML/NJ though not in ordinary ML. The identifiers themselves can be found by inspecting the source code at the pointer associated with the site; `idsBoundAt` returns the identifiers for a specified site number, listed in an order consistent with `boundValues`. The `caller` field is present only for `FN` (top-of-function) events; it points to the event record for the most recent preceding event that has been executed, which must be the `APP` event that caused the function to be entered. The `binding` field points to the event record for the lexically innermost-nested binding event that bound a identifier in scope at this event. It is the dynamic analogue of the `enclosingBinding` function on sites, i.e., for an event at site s, the binding pointer always points to an event whose site is `enclosingBinding`(s).

Given an event record, **smld** can directly determine the source location of the corresponding site, and the values of identifiers bound there. To determine the values of other identifiers, it follows the `binding` pointers to obtain a series of event records that corresponds to a path from the initial event's site to the root of the static binding tree. This series of records is the *binding chain*; any identifier that is in scope at the initial event must be bound at some event along the chain. To look up an identifier at a particular event,

```
fun lookup (x:id,event:eventRecord) : value =
  case event of
    NULLEVENT => raise notFound
  | EVENT{siteNumber:siteNumber,
          binding:eventRecord,
          caller:eventRecord option,
          boundValues:value list}) =>
      let val idList = idsBoundAt(siteNumber)
          val pos = index(x,idList)
                  (* return SOME(position of x in idList) or
                           NONE if not present *)
      in case pos of
           SOME n =>
             nth(boundValues,n) (* extract element n from boundValues *)
         | NONE => lookup(x,binding)
      end
```

Figure 6: The identifier `lookup` algorithm.

smld traverses the binding chain starting from that event until it finds a matching binding; at this point, the value can be obtained from the event record. If the null event, which corresponds to the root of the binding tree, is reached, the identifier is not bound at the initial event site. The algorithm for identifier lookup is summarized in Figure 6.

As a byproduct of this algorithm, **smld** also determines the source-code location at which the identifier was bound; this information is useful for identifiers representing functions, whose values cannot be printed. 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, though usually much smaller. As detailed in Chapter 7, determining the type of a polymorphic identifier may be less efficient.

`lookup` can start from *any* event record, so if **smld** 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 get these call chain event records, **smld** relies on the following properties of the instrumentation.

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

```
fun findCaller(event:eventRecord) : eventRecord =
  case event of
    NULLEVENT => raise noCaller
  | EVENT{siteNumber:siteNumber,
          binding:eventRecord,
          caller:eventRecord option,...} =>
      case caller of
        SOME event => event  (* implies siteTypeOf(siteNumber) = FN
                                and type of event = APP *)
      | NONE => findCaller(binding)
```

Figure 7: The `findCaller` algorithm.

- The event executed immediately before a `FN` event is always an `APP` event corresponding to the application that caused the function rule body to be entered.

Thus, given an event record, **smld** 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 `findCaller` algorithm is summarized in Figure 7.

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 [7, 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.

To generate the `caller` fields, the instrumentation introduces a global mutable `ref` variable `lastCaller` containing a pointer to the event record for the last `APP` event executed; it is updated by instrumentation at each `APP` event. When a `FN` event record is created, the value of `lastCaller` is placed in the `caller` field. To generate the `binding` fields, the instrumentation arranges that each binding event record (say for site s) is itself bound to a new identifier called `event_`s. This variable is used to fill in the `binding` field for all event records whose sites are children of s, which is always possible because `event_`s has the same scope as the variables bound at s.

Finally, breakpoint control is supported using `breakWanted`, an array of booleans indexed by site number. If the programmer requests a breakpoint at a particular site, the

```
val breakWanted : bool array
val lastCaller : eventRecord ref

fun event(siteNumber,binding,caller,boundValues) : eventRecord =
   let val eventRecord = EVENT{siteNumber=siteNumber,
                               binding=binding,
                               caller=caller,
                               boundValues=boundValues}
   in if breakWanted sub siteNumber then
          break(eventRecord)
      else ();
      eventRecord
   end
```

Figure 8: **Smld**'s event instrumentation code.

relevant array entry is set to `true`; each instrumented site tests its array entry and transfers control to the debug monitor if it is `true`. If **smld** wants to get control at the next event regardless of its site (e.g., when the user is single-stepping or has typed CTRL/C at the keyboard to halt execution), it sets all entries in `breakWanted` to `true`.

Figure 8 shows the instrumentation code inserted at each site in the form of an `event` function, which is ordinarily expanded in line to improve execution speed (see Section 8.6 for an alternative). Figure 9 describes the instrumentation process for the subset language of Figure 2 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 $\mathcal{D}(dec, \texttt{NULLEVENT})$. The auxiliary metafunction $\mathcal{S}$ takes a site type and source-code position (represented by $\dagger$) as arguments, creates a new site with that type and position, and returns the new site's number. The resulting set of sites is a by-product of the instrumentation process.

$$
\begin{array}{lcl}
\mathcal{E}(\mathit{nconst}, \mathit{lbexp}) & = & \mathit{nconst} \\
\mathcal{E}(\mathit{uconst}, \mathit{lbexp}) & = & \mathit{uconst} \\
\mathcal{E}(\mathit{var}, \mathit{lbexp}) & = & \mathit{var} \\
\mathcal{E}(\mathit{op}, \mathit{lbexp}) & = & \mathit{op} \\
\mathcal{E}(\mathit{exp}_{\mathit{opr}}\ {}^{\dagger}\ \mathit{exp}_{\mathit{arg}}, \mathit{lbexp}) & = & \texttt{let val opr = } \mathcal{E}(\mathit{exp}_{\mathit{opr}}, \mathit{lbexp}) \\
 & & \quad\texttt{and arg = } \mathcal{E}(\mathit{exp}_{\mathit{arg}}, \mathit{lbexp}) \\
 & & \texttt{in lastCaller := event(}\mathcal{S}(\texttt{APP}, \dagger)\texttt{,}\mathit{lbexp}\texttt{,NONE,nil);} \\
 & & \quad\texttt{opr arg} \\
 & & \texttt{end} \\
\mathcal{E}(\texttt{let } \mathit{dec} \texttt{ in } \mathit{exp} \texttt{ end}, \mathit{lbexp}) & = & \texttt{let } \mathit{dec}' \texttt{ in } \mathcal{E}(\mathit{exp}, \mathit{lbexp}') \texttt{ end} \\
 & & \qquad \mathit{where}\ (\mathit{dec}', \mathit{lbexp}') = \mathcal{D}(\mathit{dec}, \mathit{lbexp}) \\
\mathcal{E}(\texttt{fn } \mathit{rule}_1 \mid \ldots \mid \mathit{rule}_n, \mathit{lbexp}) & = & \texttt{fn } \mathcal{R}(\mathit{rule}_1, \mathit{lbexp}) \mid \ldots \mid \mathcal{R}(\mathit{rule}_n, \mathit{lbexp}) \\
\mathcal{E}(\texttt{(}\mathit{exp}_1\texttt{,}\mathit{exp}_2\texttt{,}\ldots\texttt{,}\mathit{exp}_n\texttt{)}, \mathit{lbexp}) & = & \texttt{(}\mathcal{E}(\mathit{exp}_1, \mathit{lbexp})\texttt{,}\mathcal{E}(\mathit{exp}_2, \mathit{lbexp})\texttt{,}\ldots\texttt{,}\mathcal{E}(\mathit{exp}_n, \mathit{lbexp})\texttt{)} \\
\mathcal{E}(\texttt{(}\mathit{exp}_1\texttt{;}\mathit{exp}_2\texttt{;}\ldots\texttt{;}\mathit{exp}_n\texttt{)}, \mathit{lbexp}) & = & \texttt{(}\mathcal{E}(\mathit{exp}_1, \mathit{lbexp})\texttt{;}\mathcal{E}(\mathit{exp}_2, \mathit{lbexp})\texttt{;}\ldots\texttt{;}\mathcal{E}(\mathit{exp}_n, \mathit{lbexp})\texttt{)}
\end{array}
$$

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

$$
\begin{array}{l}
\mathcal{D}(\texttt{val}^{\dagger}\mathit{pattern}_1\texttt{ = }\mathit{exp}_1\texttt{ and }\ldots\texttt{ and }\mathit{pattern}_m\texttt{ = }\mathit{exp}_m, \mathit{lbexp}) \quad = \\
\qquad \texttt{(val } \mathit{pattern}_1 \texttt{ = } \mathcal{E}(\mathit{exp}_1, \mathit{lbexp}) \texttt{ and } \ldots \texttt{ and } \mathit{pattern}_m \texttt{ = } \mathcal{E}(\mathit{exp}_m, \mathit{lbexp})\texttt{;} \\
\qquad\quad \texttt{val event\_}n\texttt{ = event(}n\texttt{,}\mathit{lbexp}\texttt{,NONE,}\mathit{vars}\texttt{)}, \\
\qquad \texttt{event\_}n\texttt{)} \\
\qquad\qquad \mathit{where}\ n = \mathcal{S}(\texttt{VAL}, \dagger)\ \mathit{and\ vars = list\ of\ variables\ bound\ in}\ \mathit{pattern}_1,\ldots,\mathit{pattern}_m \\
\mathcal{D}(\texttt{val rec}^{\dagger}\mathit{var}_1\texttt{ = }\mathit{exp}_1\texttt{ and }\ldots\texttt{ and }\mathit{var}_m\texttt{ = }\mathit{exp}_m, \mathit{lbexp}) \quad = \\
\qquad \texttt{(val event\_}n\texttt{\_ref = ref NULLEVENT;} \\
\qquad\quad \texttt{val } \mathit{var}_1 \texttt{ = } \mathcal{E}(\mathit{exp}_1, \texttt{!event\_}n\texttt{\_ref}) \texttt{ and } \ldots \texttt{ and } \mathit{var}_m \texttt{ = } \mathcal{E}(\mathit{exp}_m, \texttt{!event\_}n\texttt{\_ref})\texttt{;} \\
\qquad\quad \texttt{val event\_}n\texttt{ = event(}n\texttt{,}\mathit{lbexp}\texttt{,NONE,[}\mathit{var}_1\texttt{,}\ldots\texttt{,}\mathit{var}_m\texttt{]);} \\
\qquad\quad \texttt{val \_ = event\_}n\texttt{\_ref := event\_}n, \\
\qquad \texttt{event\_}n\texttt{)} \\
\qquad\qquad \mathit{where}\ n = \mathcal{S}(\texttt{VALREC}, \dagger) \\
\mathcal{D}(\mathit{dec}_1;\mathit{dec}_2;\ldots;\mathit{dec}_n, \mathit{lbexp}) \quad = \quad (\mathit{dec}'_1; \mathit{dec}'_2; \ldots; \mathit{dec}'_n, \mathit{lbexp}'_n) \\
\qquad\qquad \mathit{where}\ \ (\mathit{dec}'_1, \mathit{lbexp}'_1) = \mathcal{D}(\mathit{dec}_1, \mathit{lbexp}) \\
\qquad\qquad \mathit{and}\quad\ (\mathit{dec}'_2, \mathit{lbexp}'_2) = \mathcal{D}(\mathit{dec}_2, \mathit{lbexp}'_1) \\
\qquad\qquad \mathit{and}\quad\ \ldots \\
\qquad\qquad \mathit{and}\quad\ (\mathit{dec}'_n, \mathit{lbexp}'_n) = \mathcal{D}(\mathit{dec}_n, \mathit{lbexp}'_{n-1})
\end{array}
$$

Figure 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*.

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 it is not available until after the `val rec` binding has been executed, so execution of the binding event needs to be deferred until then. A `ref` cell is set up as a placeholder for the event record that can be referenced from inside the function body, and filled in later.[2] Figure 10 shows the instrumented version of the `rev` program of Figure 3, exactly as it would be produced by the execution of the algorithm in Figure 9. Site numbers are as in Figures 3 and 4. 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 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.

4.3 Reducing Instrumentation Overhead

Smld must minimize the cost of *controlled execution*, i.e., execution under debugger control, typically directed toward a breakpoint. While it is also important to keep query times as low as possible, it is better to expend some extra time answering a query that the user has explicitly requested than to slow down controlled execution to support queries that might never be made.

4.3.1 Coalescing Sites

Adjacent sites in the binding tree can often be coalesced into neighboring *augmented* sites without affecting the ability to look up identifiers and trace control flow. Roughly speaking, a site s can be removed and its binding information assigned to another site a whenever the execution of an event at s necessarily implies that the next event executed will be at a. In traditional compiler terms, we coalesce all sites that appear in the same basic block.

[2] Since the body of a `val rec` is always a non-executable `fn` expression, there is no danger that the placeholder will be dereferenced before it has been set. In practice, this technique is unnecessary because `VALREC` events are coalesced into neighboring `APP` events (see Section 4.3).

```
let
  val cons =
      fn (x,y) =>
        let val event_2 = event(2,NULLEVENT,SOME(!lastCaller),[x,y])
        in x::y
        end
  val event_1 = event(1,NULLEVENT,NONE,[cons])
  val rev =
      fn list =>
        let val event_4 = event(4,event_1,SOME(!lastCaller),[list])
        in let
             val event_5_ref = ref NULLEVENT
             val rec r =
                 fn (h::t,a) =>
                   let val event_6 =
                             event(6,!event_5_ref,SOME(!lastCaller),[h,t,a])
                   in let val opr = r
                          and arg = (t, let val opr = cons and arg = (h,a)
                                        in lastCaller :=
                                             event(7,event_6,NONE,nil);
                                           opr arg
                                        end)
                      in lastCaller := event(8,event_6,NONE,nil);
                         opr arg
                      end
                   end
                  | (nil,a) =>
                   let val event_9 =
                             event(9,!event_5_ref,SOME(!lastCaller),[a])
                   in a
                   end
             val event_5 = event(5,event_4,NONE,[r])
             val _ = event_5_ref := event_5
           in let val opr = r and arg = (list,nil)
              in lastCaller := event(10,event_5,NONE,nil);
                 opr arg
              end
           end
        end
  val event_3 = event(3,event_1,NONE,[rev])
in let val opr = rev and arg = [1,2,3]
   in lastCaller := event(11,event_3,NONE,nil);
      opr arg
   end
end
```

Figure 10: Instrumented version of **`rev`**. Original program code is shown in **`bold typewriter font`** and instrumentation code is shown in `light typewriter font`. The types of events can be distinguished as follows: `APP` event records are assigned to `lastCaller` and `FN` event records reference `lastCaller`; the remaining events are of type `VAL` or `VALREC`.

In particular, `VAL` and `VALREC` events can always be coalesced into neighboring `APP` and `FN` events. The coalescing process can be visualized as a transformation of the binding tree. We fix the `APP` and `FN` nodes in the tree, and push each `VAL` and `VALREC` node down the tree toward the leaves until it reaches a fixed node, which is then augmented with information about the node that has been coalesced into it. If a node is pushed through a fork in the tree, it must be duplicated and propagated down each branch. At the end of this process, each augmented node contains a list of *associated* sites, and the event record built at the augmented site contains value entries for all identifiers bound at any of the associated sites. Traversing the transformed tree from any fixed site to the root, pausing at each node to traverse in order all the sites coalesced there, encounters exactly the same sequence of sites as in the original tree. Thus `lookup` operates essentially unchanged.

Removing `VAL` and `VALREC` sites won't affect **smld**'s ability to trace control flow. However, since these sites do define possible breakpoint locations and affect how control flow is reported, we must consider how to present coalesced sites to the user. There is a range of choices. At one extreme, **smld** could identify just the augmented sites, and leave it to the user to figure out which augmented sites contains a given value binding, which is usually clear from the source code. At the other extreme, **smld** could give the illusion that the original sites are present by interpreting each coalesced-event execution at debug-time as a sequence of event executions at the associated sites. Our present implementation is a compromise: it reports control flow and permits breakpoints only at augmented sites, but reports binding locations in terms of the original sites.

Figure 11 shows a coalesced version of the binding tree for the `rev` example, and Figure 12 shows the corresponding instrumented code.

4.3.2 Linearizing

We can carry coalescing further by trying to move `FN` sites as well as `VAL` and `VALREC` sites. This process must be done carefully, however, since `FN` sites play an important role in tracing flow control as well as in identifier lookup. To maintain the invariant that each `FN` event is executed immediately after the matching `APP` event, there can be no more than one `FN` site per fixed site, and no `FN` event may be executed more than once. Thus, if a `FN` site is coalesced into more than one fixed site, at most one of these fixed-site events can occur during a single invocation of the function.

In practice, these restrictions nearly always prevent `FN` sites from being pushed down

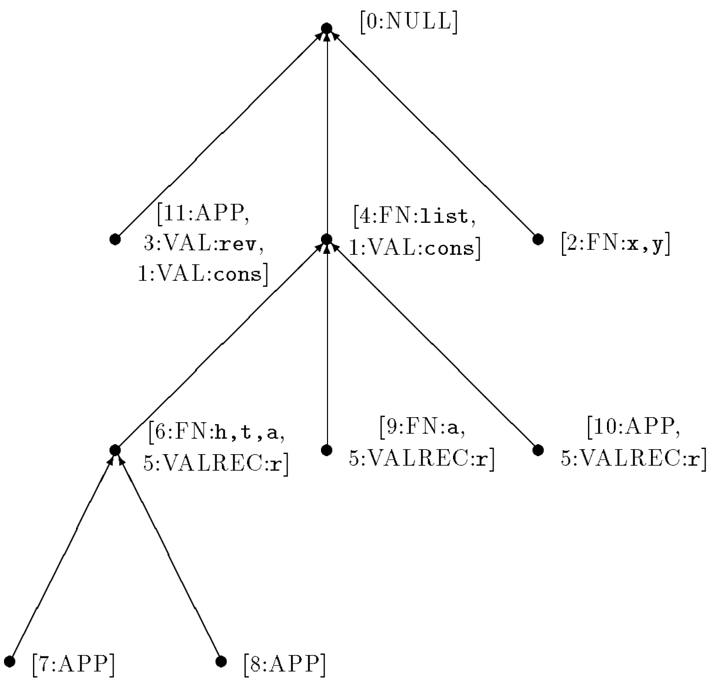

Figure 11: Coalesced binding tree for `rev`. Nodes are described by lists of the associated original events. Since the number of the first event in each coalesced event list is unique, it can be used to identify the coalesced event.

```
let
  val cons =
      fn (x,y) =>
        let val event_2 = event(2,NULLEVENT,SOME(!lastCaller),[x,y])
        in x::y
        end
  val rev =
      fn list =>
        let val event_4 = event(4,NULLEVENT,SOME(!lastCaller),[list,cons])
        in let
             val rec r =
                 fn (h::t,a) =>
                   let val event_6 =
                            event(6,event_4,SOME(!lastCaller),[h,t,a,r])
                   in let val arg =
                             (lastCaller := event(7,event_6,NONE,nil);
                              (t,cons(h,a)))
                      in lastCaller := event(8,event_6,NONE,nil);
                         r arg
                      end
                   end
                 | fn (nil,a) =>
                   let val event_9 = event(9,event_4,SOME(!lastCaller),[a,r])
                   in a
                   end
           in lastCaller := event(10,event_4,NONE,[r]);
              r (list,nil)
           end
        end
in lastCaller := event(11,NULLEVENT,NONE,[rev,cons]);
   rev [1,2,3]
end
```

Figure 12: Instrumented version of **`rev`** with coalescing. Original program code is shown in **`bold typewriter font`** and instrumentation code is shown in `light typewriter font`. Instrumentation code is shown in simplified form, with most unnecessary temporary bindings removed. Event types can be distinguished as follows: `APP` event records are assigned to `lastCaller` and `FN` event records reference `lastCaller`.

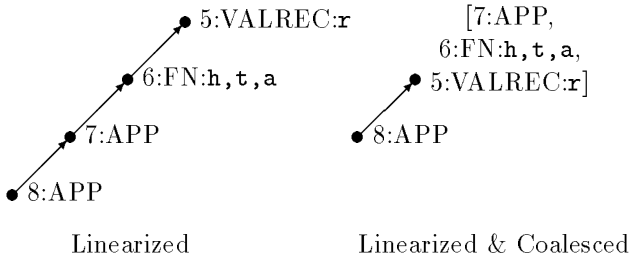

Figure 13: Improved binding trees for `fn (h::t,a)` rule.

through a branch in the binding tree, and so severely limit the opportunities for coalescing. In `rev`, for example, the only potential opportunity for coalescing a `FN` site is in the rule

$$\texttt{fn (h::t,a) =>}^{6}\ \texttt{r}^{8}\ \texttt{(t, cons}^{7}\texttt{(h,a))}$$

corresponding to the [6:...,5:...] subtree of Figure 11. But the two `APP` events will be executed in sequence, so the `FN` event cannot coalesced into both of them.

It is sometimes possible to rearrange the binding tree to remove forks of this kind. Let us expand the definition of legal binding trees to allow edges to point to non-binding (`APP`) nodes, with the understanding that these nodes are simply ignored by `lookup`. Now we can modify a binding tree without changing the results of lookups by replacing a tree edge $s \longrightarrow t$ by a new edge $s \longrightarrow s'$, provided that s, s', and t lie in the same static environment and that any event at s is immediately preceded by an event at s'. This *linearizing* rewrite of the binding tree can open up more opportunities for coalescing. The effect of linearizing on the fragment shown above and the corresponding coalesced subtree are shown in Figure 13. The resulting instrumented code, with unnecessary temporaries removed and slightly simplified scoping for clarity, is

```
fn (h::t,a) =>
  let val event_7 = event(7,event_4,SOME(!lastCaller),[h,t,a,r])
      val arg = (lastCaller := event_7;
                 (t,cons(h,a)))
  in lastCaller := event(8,event_7,NONE,nil);
     r arg
  end
```

The instrumenter can perform linearizing wherever it encounters a sequence of subexpressions guaranteed to be executed in order (e.g., nested applications, sequence expressions, and tuple expressions), and each containing an event site, such that all the event sites share the same environment. In our present implementation, linearizing and coalescing are folded into a one-pass instrumentation algorithm similar to the description in Figure 9. The combination of linearizing and coalescing gives faster execution times than a naive coalescing scheme where all `FN` and `APP` sites are fixed. We could make some further improvements at the cost of making our algorithm more complicated; however, the abstract syntax tree is a poor basis for performing the dataflow analysis needed for more elaborate schemes.

4.3.3 Handling `case` Expressions

We omitted `case` or `if-then-else` expressions from the subset language because they are derived forms (syntactic sugar) for applied `fn` expressions; indeed, the Standard ML definition [88] treats them this way. Specifically

`case` exp `of` $rule_1$ `|` $rule_2$ `|` ... `|` $rule_n$

is a derived form for

`(fn` $rule_1$ `|` $rule_2$ `|` ... `|` $rule_n$`)` exp

and

`if` exp_b `then` exp_t `else` exp_f

is a derived form for

`case` exp_b `of true =>` exp_t `| false =>` exp_f

If expanded `case` expressions are instrumented in the usual way, the resulting code has one site on each `fn` branch plus one for the application of the `fn` expression to *exp*. When executed, two events occur: one `FN` event preceded by its corresponding `APP` event. But in this case, the `APP` site is known statically, so it is wasteful to execute an event prior to the application. Instead, the instrumenter inserts special `CASE` sites in place of the `FN` sites. `CASE` sites carry an extra source-code pointer to the `case` keyword and *exp*, as well as to the top of the case branch chosen. This optimization also applies automatically for `if-then-else` expressions. Although trivial, it is worthwhile because many programs contain conditionals in their inner loops.

Another optimization we have considered for `case` expressions, but have not yet implemented, concerns linearizing. Each branch of a `case` expression induces a corresponding branch in the binding tree. Handled naively, as at present, each branch introduces a nested scope containing the variables bound by the branch pattern, which prevents any enclosing `FN` event from being pushed into `case` branches and defeats attempts to linearize any construct enclosing the `case`. But when the `case` is exhaustive and no variables are bound by any of the `case` branch rules, as for `case` expressions derived from `if-then-else` expressions, we could use the fact that only one `case` branch is executed to allow `FN` events to be pushed down into the branches. The instrumented `case` expression would return an event record pointer in addition to its original value.

4.3.4 Application of Primitives

Since many programs have arithmetic operators in their inner loops, not instrumenting them would save execution time. In most respects not instrumenting them is safe; like data constructors, these functions are ordinarily executed atomically. But some primitive operators may raise an exception; for example, division may raise the divide-by-zero exception `Div`. If there is no `APP` event for the primitive application, the value of `lastCaller` does not indicate the exception-causing event site. Instead, **smld** reports the last instrumented event (typically a non-primitive application) prior to the exception, which may be far away from the actual exception-raising application. It may even be in a different function, e.g., one called earlier in the course of executing the function that calls the exception-raising primitive.

Unfortunately, unexpected exceptions are one of the problems for which the programmer most needs and expects help from the debugger. But, at present, **smld** implements this defective reporting, although it always executes a `FN` event at the top of any function body that calls an exception-raising primitive. If this solution proves unacceptable in practice, we could apply more generous heuristics, such as instrumenting the first of any consecutive chain of arithmetic operators.

4.4 Code Generation and Optimization

This section examines the code generated by the SML/NJ compiler from instrumented source. We consider how instrumentation affects controlled execution performance, and how it inhibits optimizations that would destroy expected behavior during debugging. We are concerned with execution cost when `break` is *not* called, since this is the usual case; costs associated with actually calling `break` are dominated by the cost of the debug monitor.

The code generated for the instrumentation itself is largely straightforward. Executing an event has a direct cost: the event record must be built, which takes time and space, the `lastCaller` reference variable must be updated for `APP` events, and the `breakWanted` entry looked up and tested. The indirect costs of executing this code are more subtle. The addresses of `breakWanted`, `lastCaller`, and the `break` routine must be kept live throughout the code; this increases demand on registers and the size of closures. Each binding pointer must be kept live as long as the associated binding is live, which puts more pressure on registers and closures. However, 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.

If `break` is called, a continuation closure must be built to pass to it, which can be costly if there are many live variables. It is desirable to avoid building the continuation closure if the break test fails; unfortunately, the optimizer sometimes hoists it above the break test, though this hoisting does not normally happen.

Each update to `lastCaller` adds another entry to the garbage collector's store list (see Section 2.1.3); overall, this adds substantially to garbage collection times.

The most serious effect of instrumentation on the performance of the original code is on tail-recursive functions. These remain tail-recursive, thanks to our policy of not placing events *after* applications, so the code generator 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 could be substantial for deeply recursive programs with large data structures. One possible solution is to limit the call-trace depth for which local variable information is kept. A more elegant solution, which involves recreating the values lazily upon demand, is developed in Chapter 5.

Another effect is that every named identifier must now remain live from its binding point until the last event record containing its value is constructed. Again, more registers or closure space may be needed, although most of these values may be live anyway, since even with coalescing the distance from binding to reporting event is 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. We have directly measured the extended-liveness effect, in combination with the cost of maintaining live binding pointers; see Section 8.5.

A potentially worrisome consequence of including all function identifiers in event records is that these functions now *escape*, since they might be called from the debug monitor. The SML/NJ optimizer can generate better calling sequences for functions whose call sites are known (non-escaping functions), so making all functions escape could have performance consequences. Fortunately, the optimizer's η-splitting mechanism (see Section 2.1.4) makes a separate non-escaping version of any escaping function with at least one known call site.

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, 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 of `rev`, on the other hand, does the full computation in its original form plus all the instrumentation support. 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` doesn't need to pack its arguments into a pair. Increasing the permitted body size for in-line expansion (a compiler parameter) by 150% over its default value causes `cons` to be expanded in line at each place it occurs, with one copy of the original function in its "unknown" parameter-passing version still made for inclusion in the binding event record.

The optimizer can improve performance of the instrumented code too, sometimes in unforeseen ways. For example, there is no reason to build an event record for non-`APP` events for which no binding pointers are created (e.g., events at sites 2 and 9 in `rev`) unless a `break` is executed at the event. Thus the code to build the event can be moved to the true branch of the break test at such events; the optimizer does this automatically.

4.5 Coping with the Full Language

This section describes how **smld** instruments some of the more complicated features of Standard ML, including reference variables, declarations, exceptions, and modules. It assumes more in-depth knowledge of ML on the part of the reader than the rest of this thesis. Discussion of one major feature, polymorphic typing, is deferred to Chapter 7.

4.5.1 Mutable Variables

Although most Standard ML variables don't actually vary once they are declared, the language supports mutable `ref` variables, and SML/NJ (and most other implementations) also support mutable arrays. These are distinguishable at compile time as having different types from ordinary variables, and their contents must be explicitly set and retrieved. In essence, the value of a `ref` variable is a pointer to a heap cell holding the `ref`'s contents; the instrumentation of Section 4.2 places the pointer, not the contents, in the variable's binding event record. With this understanding, **smld** can handle a `ref` variable as it would any other identifier, relying on type restrictions to make sure that `ref`s are used legitimately in a query or update expression issued from the debug monitor loop.

4.5.2 Type and Fixity Declarations

ML allows the user to define new discriminated union types using `datatype` or `abstype` declarations, and to define type abbreviations using `type` declarations. These declarations, which contribute type names and constructor names to the environment, follow the same scoping rules as value declarations. **Smld** must be able to look up types and constructors in an event context just like other identifiers. To do so, `TYPE` event sites are located just after each of these declarations and appear in runtime binding chains, where a suitably modified version of `lookup` can find them. These sites are invariably coalesced, so they have no associated runtime cost.

A similar mechanism applies to `infix`, `infixr`, and `nonfix` declarations, which control the fixity of operators: special `FIX` events are located after these declarations.

Each `datatype` declaration generates a distinct type, even if it is syntactically identical to another declaration. The parser assigns each distinct type a unique internal stamp; to maintain type safety in its own event-context environments, **smld** needs a similar type label. To integrate it's environments with the remainder of the SML/NJ system (see Section 4.6), **smld** uses the parser's own stamps, which are available in the abstract syntax and can be extracted via the TYPE site source-code pointer.

4.5.3 `local` Declarations

ML supports locally scoped declarations with the syntax

`local` dec_{local} `in` $dec_{exported}$ `end` indicating that dec_{local} is visible only between `in` and `end`, while $dec_{exported}$ is visible in subsequent declarations. This mechanism violates our notion that the `enclosingBinding` relation is transitive and forms a tree; **smld** must "splice out" dec_{local} from the binding chain when following the chain from a point after `end`. It does so by adding new sites `LOCAL_TOP`, `LOCAL_IN` and `LOCAL_END` at the obvious points; these are ordinarily coalesced, so there is no runtime cost. **Smld** determines whether or not a given binding event is visible from a given starting event by performing simple bookkeeping on the number of `LOCAL` sites of each type that it encounters while traversing the chain.

4.5.4 Exceptions

Smld must be able to find the source of an unexpected and uncaught exception that causes program termination; the inability to do so in the ordinary SML/NJ system is a common user complaint. It is also important to be able to trace the flow of control when exceptions are explicitly handled by user code.

Raising and handling of exceptions is treated similarly to function application and entry, by introducing `RAISE` and `HANDLE` event types to parallel `APP` and `FN` events. **Smld** arranges for `lastCaller` to be set at explicit `RAISE` events and read by `HANDLE` events. As described in Section 2.2.3, `RAISE/HANDLE` pairs are included on the call chain just like `APP/FN` pairs.

Unhandled exceptions that would normally terminate program execution at the top level are caught by **smld** just below that level, so that debugger data structures remain intact. The exception is reported to the user, and control is returned to **smld**'s interactive loop, with `lastCaller` containing the last event that occurred before the exception. If the exception is caused by an explicit `raise`, `lastCaller` is a `RAISE` event at a known location, giving the user direct access to the circumstances leading to the exception. If the exception is raised implicitly by the runtime system in response to an arithmetic error, `lastCaller` points to the last `APP` event prior to the error-causing operator.

Another important class of built-in exceptions consists of `Match` and `Bind`, which are raised if no specified pattern matches the right-hand side of a `val` declaration or the actual argument to a function. Although the compiler can statically detect when a set of patterns is not exhaustive, many programmers ignore the resulting compile-time warnings, so it is important that **smld** report the source of these exceptions. `Match` and `Bind` are actually raised by explicit `raise` expressions inserted by the compiler whenever non-exhaustive matches are detected; this insertion can be arranged to occur before instrumentation, so that the `raise` expressions are instrumented in the normal way.

Information about a raised exception is encapsulated in an *exception packet*. All exception packets have built-in type `exn`, which is like a data type but has an extensible set of constructors defined by `exception` declarations. Exception constructors can carry values. Exception declarations are generative; that is, as each one is executed, a new distinct `exn` constructor is produced, even if the constructor name matches an existing one. Standard ML does not provide a convenient way for printing or deconstructing an arbitrary exception value. However, SML/NJ arranges that the constructor name can be extracted as an ASCII string from the exception value via an internal function; though this facility doesn't provide a unique identifier, it is useful for displaying exceptions raised to the top level. **Smld** uses this function for the same purpose; it also displays the value (if any) carried by known (built-in) exceptions. **Smld** doesn't display the value carried by a user-defined exception constructor, since it has no reliable means to determine its type; however, users can deconstruct the value by hand if the relevant `exception` declaration is in scope.[3]

4.5.5 Module System

Modules complicate **smld**'s identifier lookup mechanism. References to an identifier may be qualified, that is, prefixed by a structure name. To look up a qualified identifier, **smld** must first look up the structure and then look up the identifier inside the structure body. Searching for structure bindings is straightforward: **smld** adds `STRUCTURE` binding events analogous to `VAL` binding events, located so that they execute immediately after the structure declaration is executed. Since structures may be nested, the structure name may itself by qualified, so that looking up the structure may be recursive.

The body of a structure is a list of declarations, instrumented in the ordinary way, and an identifier search starting at an event inside the body can use `lookup`. However, the events within a structure body do *not* lie on the binding chain of events below the structure. To search within a named structure, **smld** needs a pointer from the `STRUCTURE` binding event to the last declaration event inside the structure body (always a special `STRUCTURE_BOTTOM` event). This pointer is written into a special `ref` location that is dereferenced and inserted into the `STRUCTURE` event record when the structure declaration is executed. This situation

[3] If **smld** could access the exception constructor's event record when an exception is raised, it could display the values carried by exceptions, at least for constructors declared to carry ground types. The compiler could be altered to enable this access by allowing an arbitrary pointer to be associated with exception constructors in place of the ASCII string.

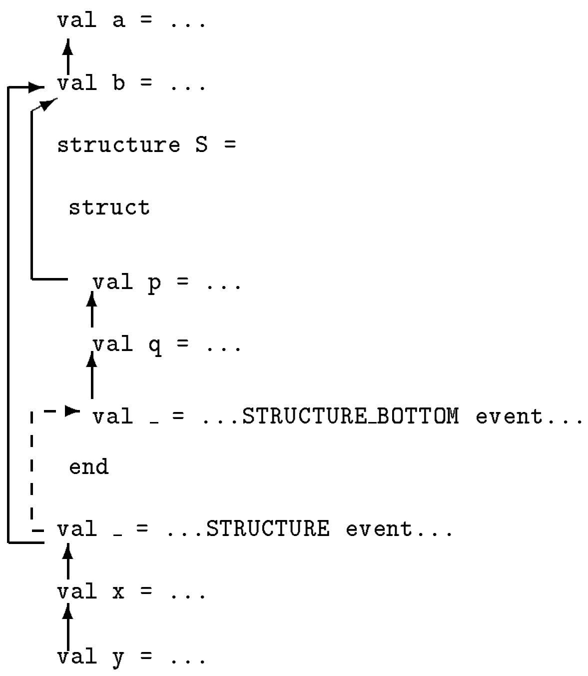

Figure 14: Binding chain for `structure` declarations. Solid arrows represent the `enclosingBinding` relation. The dotted arrow represents the special pointer allowing access to the structure's internal binding chain from the main binding chain.

is illustrated in Figure 14.

A structure may also be referenced by an `open` *structurename* declaration, which has the effect of inserting all identifiers defined by the structure into the environment at the point of the declaration. To deal with these, **smld** places an `OPEN` event at the declaration; if this event is encountered on the binding chain while searching for a particular identifier, the opened structure must be searched for that identifier before checking the remaining declarations in the chain.

This process is further complicated by the existence of signatures, which define the set of identifiers exported from a structure. To avoid finding identifiers that should be hidden by a signature constraint, **smld** can begin a search within a structure body only after the structure's signature has been checked. In principle, **smld** could add instrumentation to signatures to determine the exports of each structure. In practice, it is more convenient to consult the structure description generated as a byproduct of parsing, which lists all the identifiers exported by the structure.

A functor body is much like a structure body that is executed once each time the functor is applied, except that the functor's parameter (itself a structure) is now part of the environment in which the body of the result structure is elaborated. In particular, types of values declared in the functor body can depend on types in the parameter structure. Thus, both value lookup and dynamic type reconstruction initiated from within a result structure need access to the actual parameter. To associate the actual parameter with the result structure, **smld** must instrument functor bodies and applications in a manner similar to ordinary function bodies and applications; it uses the `lastCaller` reference cell to jump from the top of a functor body to its application site.

Declarations of `datatype`s within functors are generative; i.e., a (statically) distinct set of `datatype`s is created each time the functor is applied. However, the parser does not bother to generate stamps for these types unless they are exported from the result structure. As described above, **smld** uses the parser's internal stamp for a generated type whenever possible. For non-exported types, **smld** must create a new stamped copy at debug time when the type is first encountered; subsequent uses of the type must use the same stamp. Similar considerations apply to substructures, which are also stamped.

4.6 Integration with SML/NJ

4.6.1 Instrumentation Mechanics

Smld performs instrumentation by transforming the compiler's abstract syntax representation, after parsing and type checking, but before translation into λ-language. The instrumenting pass produces a new copy of the abstract syntax, with hand-crafted abstract syntax for instrumentation code inserted. A more direct approach might be to pre-process the source code prior to parsing, entirely outside the compiler. The main advantage of the abstract syntax transformation approach is that it 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.

One of the outputs of the instrumenting process is a mapping between instrumented sites and source-code locations in the original concrete syntax. This mapping is made possible by *mark* nodes in the abstract syntax tree that map sub-trees to the concrete syntax that generated them.[4] The main problem with delaying instrumentation until abstract syntax has been generated is that derived forms, such as `if-then-else`, `while`, and `fun`, have already been expanded into core forms. Thus, sites for these constructs will be defined in terms of the expanded syntax rather than the original; for example, `if-then-else` clauses appear as `case` rules, and therefore have associated CASE site types. It can also be difficult to assign source-code location marks to expanded code in a meaningful way. Also, SML/NJ builds the static representations of structures, functors, and signatures at parse time as side-effects, which limits the changes that can be made in modules during instrumentation.

That type checking has already been performed when **smld** instruments is a mixed blessing. On the one hand, the instrumenter needs to understand only type-safe programs, and the instrumentation code itself does not have to be strictly type-safe. On the other hand, the instrumenting transform is not itself type-checked.

For some purposes, it would be convenient to defer instrumentation to a somewhat later stage. For example, the debugger could avoid the need to make a large expanded instrumented copy of the abstract syntax by folding the instrumenting process in with the translation from abstract syntax to λ-language. This approach was initially used in SML/NJ's profiling facility [12], but it complicates the translation code; implementing the instrumentation as a separate pass preserves modularity. If instrumentation were delayed until after the abstract syntax was discarded, i.e., until the λ-language or CPS stage, the debugger would face serious problems in mapping event sites back to the concrete source text—some of the very problems an instrumentation-based approach seeks to avoid.

4.6.2 Debug Monitor

The debug monitor 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. This implementation approach allows the standard expression evaluator to be used to display, deconstruct, and operate on values in the current debugging context; no extra code is needed. Other debugging features, 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.

[4] These marks were originally added expressly to support debugging instrumentation. However, they are general purpose and have also proved useful for improved error reporting in the standard system. They are presently discarded when the abstract syntax is translated to λ-language.

```
(* Built-in functions on events *)
eqtype event
val isNilEvent : event -> bool
val currentEvent : unit -> event
val caller : event -> event
    (* returns APP event from which current function was called *)
val singleStep: unit -> unit

(* Synthesized function for stepping over a call.
   If executed at an APP event, continues single-stepping until
     that APP event is no longer on the stack of callers.
   If executed at any other event, just single-steps. *)

fun skip () : unit =
    let val startEvent = currentEvent()
        fun startOnStack event =
             if isNilEvent event then
               false
             else if event = startEvent then
               true
             else startOnStack(caller event)
    in singleStep();
       while startOnStack(currentEvent()) do
         singleStep()
    end
```

Figure 15: Synthesizing "skip" from primitive functions.

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. Moreover, these functions are designed with carefully defined argument types and clean separation of computation and I/O, so that the user can combine them into macro–like functions to perform specialized debugging tasks. For example, a naive "skipping" function can be synthesized from a combination of built-in single-stepping and call-traceback functions, as shown in Figure 15.

The monitor communicates with the user program via a coroutine mechanism built on the `callcc` primitive in a standard fashion [116]; see Figure 16. `setCompilationUnit` sets the debugger ready to execute a user program. Control passes to that program via `exec` when the user issues a forward execution command and back to the monitor when the instrumented code calls `break`. When the user program finally completes, the result of the

```
val debuggerCont: unit cont ref
val userCont: unit cont ref
val currentEvent: eventRecord ref

fun setCompilationUnit (f:unit -> 'a) : unit =
   callcc (fn c => (callcc (fn c1 => (userCont := c1;
                                       throw c ()));
                    f ();
                    ()))

fun exec() : unit =
   callcc (fn c => (debuggerCont := c;
                    throw (!userCont) ()))

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

Figure 16: Implementation of basic control functions for **smld**.

compilation unit is made available to the top-level loop in the normal way.

In the current SML/NJ implementation, the top-level loop is "hard–coded" and is not accessible from within the interactive system itself. Thus, making any changes or additions to the loop's operation that involve manipulating abstract syntax, type information, or environments requires recompiling the relevant part of the compiler's own source code and building a new executable image. Since the internal organization of SML/NJ is quite modular, this process has not proved too difficult, but it has required coupling the compiler and debugger, and their development cycles, more closely than we might like.

4.6.3 Debug Monitor Environment

The SML/NJ top-level loop, and its debug monitor variant, use a completely functional interface for accessing identifier environments. **Smld** provides the monitor loop and parser with access to the current context environment via a set of special *emulation* functions. These functions translate an identifier lookup, which the ordinary environment would handle by consulting a hash table, into a call to **smld**'s `lookup`, implicitly parameterized by the current context. The top-level loop and parser deal with a "synthetic" debugger environment as if it were a "real" symbol table.

In practice, it is useful to provide a rather elaborate environment within the debug monitor, containing more than just the current context's environment. The user should be able to define new values or functions, e.g., as scratch temporaries, or ad-hoc debugging aids. Such ad-hoc definitions generally remain useful after execution has moved on to a different event, and even after the debugging session has terminated. In addition, it is useful to be able to access identifiers, e.g., debugging functions, that were bound at top level before the debug monitor was invoked but after the target program was compiled. Thus **smld** provides the layered environment: identifiers are first looked up in the set of ad-hoc definitions entered since debugging began, then in the environment at the current event, and finally in the top-level environment that was in place before debugging began. There is a danger that an ad-hoc definition will hide a program identifier; the programmer must simply be careful. At the conclusion of the debugging session, the ad-hoc definitions are added to the ordinary top-level environment.

4.6.4 Keyboard Interrupt

Users must be able to interrupt a program and determine where it is stopped. In our event-based debugger, the interrupt must appear as if it occurs at an event site; that is, event transitions must be atomic with respect to interrupts.

The Standard ML definition calls for a keyboard interrupt facility tied to the exception handling mechanism. Hitting CTRL/C should raise a (built-in) exception `Interrupt`, which normally results in an uncaught exception at top level, thus terminating program execution. SML/NJ uses a more robust and flexible mechanism [98] that lets the programmer install customized handlers for most Unix signals, including keyboard interrupt (SIGINT). **Smld** uses this feature to convert asynchronous interrupts to synchronous ones, by installing a SIGINT handler that sets an `interruptPending` flag, sets all elements in `breakWanted` to `true` to force a break at the next event, and resumes execution at the point where the signal occurred. An additional test is inserted in **smld**'s runtime code after a break occurs to see if `interruptPending` is set.

Chapter 5

Time Travel

Damage caused by a program bug often isn't detectable until well after the execution point where the bug occurred. Therefore, programmers often use a debugger to investigate the program's behavior *prior* to the bug's manifestation. A common strategy is to re-execute the program from the beginning under debugger control, trying to stop it at a point just before the faulty behavior begins. Often, this point is hard to find, so the program must be re-executed repeatedly, stopping a little bit sooner each time. Re-execution can be tedious, particularly if the program had already run for a long time before the bug occurred, or if it reads input that must be provided again. If many re-executions are needed, this method is intolerable.

Debugging such programs would be easier if the debugger supported *reverse* execution, i.e., if it could give the appearance of executing the program backward, either step by step or by returning to the last occurrence of a breakpoint. Generalizing slightly, the programmer may wish to interleave forward and reverse execution arbitrarily. Moreover, the debugger itself may be able to use reverse execution *internally* to implement conventional facilities, such as identifier lookup and location-based breakpointing.

In general, programs cannot actually be executed backward. Instead, the debugger can support the illusion of reverse execution by restoring previous program states. The debugger can reset the program to a desired target state by re-executing it from the beginning, as described above, provided that all external factors that affect the program's execution history can be recreated, e.g., from a log. Of course, this approach will be nearly as slow as manual re-execution, though more convenient. The debugger can make reverse execution faster, at the expense of consuming more storage, by taking checkpoints during

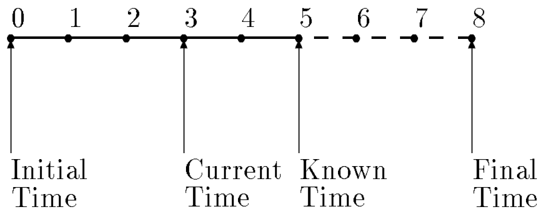

Figure 17: A sample time line. The program has been executed as far as time 5, but the current time is presently reset to time 3, perhaps after having been reset to other values in the interim. The program will terminate at time 8, either by reaching the end of the compilation unit or by raising an uncaught exception, but this fact is not yet known to the debugger or, perhaps, the user.

the original forward execution; to reverse to a particular target state, it re-executes from the previous checkpoint nearest to the target point. SML/NJ's implementation of first-class continuations makes checkpointing most of a program's state fast and simple.

Program states can be identified by logical *time* values, ordered by the sequence in which they appear during forward execution. In this sense, directing execution to a specific state in the past or future of the computation is *time travel*. This chapter describes the design of **smld-t**, a version of **smld** that provides time-travel debugging for SML/NJ. Chapter 6 describes **smld-t**'s implementation in detail.

5.1 Temporal Model

The ML execution model presented in Section 2.2.1 views execution as a sequence of atomic transitions between events; each event has an associated site in the source program and an associated context. Executing in reverse reestablishes the context associated with an event that was previously encountered during forward execution.

A reverse execution system must label and identify events. The simplest approach is to number events with consecutive integer times and to visualize ordinary sequential program execution as progress rightward along a time line marked off at each event. Operations like single-stepping (forward or backward) have an obvious interpretation in terms of times and the time line, and the notion of setting a breakpoint at a specific time is well-defined.

At any given point during execution, the time line represents *known times*, i.e., the history of execution that has already occurred; it extends from the initial time 0 rightward to the maximum known time, i.e., the maximum value that the event counter, or *clock*, has yet achieved at any point during execution. 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. 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 before the desired target time is reached, the right endpoint of the time line is fixed at the *final time*; it is impossible to advance past this time, but resetting to an earlier time is still possible. To advance the program to the final time, the user may request an advance to target time infinity. The programmer may also reset the current time forward or backward to any known time; there is no limit on how many steps backward the programmer can jump. This terminology is illustrated in Figure 17. **Smld-t** is structured around the two time-travel procedures `advanceTo(`*time*`)` and `resetTo(`*time*`)`.

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 act like normal execution; input is requested, signals are accepted, and output to all devices is displayed or written in the usual way.[1] When resetting to a known time, **smld-t** does not reverse side-effects on the file system or other parts of the environment that hold state; it leaves the outside world unaffected.

An exception is made for I/O to the user's terminal, which is treated as a stateless stream device.[2] When resetting from a known time to a later known time, **smld-t** supports two modes. 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 **smld-t** and echoed so that input and output are correctly interleaved. In *quiet* mode, intended for implicit time-travel operations used internally by **smld-t**, no I/O is visible. When resetting to an earlier known time, no input or output is performed.

A debugging session has exactly one history time line representing one consistent execution history. However, **smld-t** permits the user to reset to a time t before the maximum

[1] There is an exception for speculative execution; see Section 5.3.

[2] In some undo/redo systems [14], the user's terminal dialogue can be viewed as a script with a cursor; in such systems, it would make sense to "undo" I/O by resetting the cursor. In fact, **smld-t** maintains such a script internally, but for simplicity the user is deliberately prevented from viewing or editing it.

known time and then change the values of mutable variables. Since these changes might alter the course of execution, **smld-t** 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. **Smld-t** is not an "undo/redo" system for speculative computing, nor does it support "parallel worlds" of execution. While such mechanisms might be useful, they would complicate the user interface.

Executing multiple compilation units in a single interactive session introduces a further wrinkle in the time line metaphor. **Smld-t** treats a sequence of compilation unit executions in a single interactive session as a continuum: the first event of each unit is assigned a time one greater than the last event of the previous successfully executed unit. The top-level definitions exported by some compilation units will be hidden by later redefinitions, but **smld-t** makes no effort to deduce this; the execution histories of all successfully executed units are kept until the end of the session. Failed execution histories are discarded.

Just as in **smld**, all user commands and queries in **smld-t**, e.g., variable displays, call chain displays, etc., are implicitly qualified by the state in which program execution has halted, i.e., by the current time. This policy keeps the user interface conceptually simple, avoiding the confusion that might result if queries or commands could be explicitly qualified by user-specified times. As in **smld**, it is also possible to adjust the current context up or down the call chain, e.g., in order to examine the values of identifiers in other frames, but adjusting the context does not change the current time.

5.2 Laziness

The ability to "jump back in time" helps the programmer gather information about program execution. To find out about a particular execution state, it is unnecessary to record information about it during initial execution. Instead, the programmer can execute forward at full speed until a need for the information arises, and then back up to the desired state to collect it. In short, the programmer can be lazy about collecting execution information, deferring the cost of the collection until the information is known to be needed.

Smld-t uses these same lazy time-travel techniques internally to implement fundamental functions, including identifier lookup and control-flow tracing. During forward execution, **smld-t** records only minimal information at each event—enough to allow the event to be found again if it is needed. If more detailed information is needed, **smld-t** can jump back to the event in question and collect it. This approach makes initial controlled execution cheaper in time and space at the possible expense of longer times for query operations.

To make recording of state information lazy, the instrumentation scheme of Section 3.2 is modified so that `recordState` is invoked only if a `break` is taken at the event:

```
if (shouldBreakHere()) then
  (recordState();
   invokeDebugMonitor())
else registerTime()
```

`registerTime` preserves the time of the event so that other events can reference it. To collect full state information later on, **smld-t** internally jumps back to the registered time, arranges for `shouldBreakHere` to return true, collects the recorded state information, and jumps forward again to the original time. This time travel is invisible to the user, except perhaps for a delay in evaluating queries. To gain any performance benefit from this approach, `registerTime` must cost less than executing `recordState`; this requirement is examined more closely in Section 6.1, where we give an implementation.

Smld-t takes advantage of the sequential ordering of events provided by times. Since the event just prior to a `FN` event must be the corresponding `APP` event in the calling function, the time of that `APP` event is one less than that of the `FN` event. Thus, **smld-t** does not need to record times of `APP` events in order to build the call chain.

5.3 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 use a two-step process to direct the program to this time. First, the user 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 the earliest time at which `!c = v`. This process 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 generalized 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.

Smld-t uses the same function internally to implement many targeted execution commands as binary searches over the execution time line. One important example is the implementation of location-based breakpoints, i.e., breaking at a specific site. Suppose `registerTime` maintains 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 t_0 to a breakpoint at site s is a search for the first time when `(lastTimes sub` s`) >` t_0.

An advantage of this technique is that the `shouldBreakHere` predicate can be expressed solely in terms of times; there is no need to consult a list of location breakpoints. Updating `lastTimes` is not free, 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 `(lastTimes sub` *site*`)` directly gives the time corresponding to the desired breakpoint. The user will also typically pay a price in "extra" speculative execution beyond the breakpoint time, but this execution will not be wasted if the user later decides to proceed forward from the breakpoint.

A more sophisticated application is to a "skipping" function. The naive implementation of this function single-steps repeatedly, starting from an APP site, until the site is no longer on the call chain, as shown in Figure 15. This process can take a long time, since an arbitrary number of steps may occur between skips, and the overhead of single-stepping must be paid for each of them. A better method is to set a location breakpoint on the next site (or sites) that can be executed in the current function (or its caller). When the breakpoint is tripped, the debugger must still check to see if the halt occurred in the original call frame, because of the possibility of recursion; if the recursion is deep, this method can also be very slow. Both methods depend on sequential execution to locate the correct target of the skip.

Smld-t characterizes the target time for a skip in terms of the contents of the call chain.

If the current event is of any type other than `APP`, a skip is the same as a step; if the type is `APP`, a skip should jump to the first time at which the call chain no longer includes this `APP` event, which is just the realization time of a monotone predicate on the contents of the call chain. Thus **smld-t** can use the binary-search method to execute the skip.[3] Interestingly, time-travel primitives will be invoked recursively to determine the contents of the stack when evaluating the skip predicate.

An important consideration in implementing `advanceToRealizationTime` is choosing how often to check the predicate during the initial "blind" search phase, when the debugger has no *a priori* idea of how far it needs to execute. Two obvious possibilities are to keep the checking interval constant, e.g., at some fixed proportion of the estimated cost of checking the predicate, or to let it expand exponentially, e.g., by doubling at each unsuccessful check. The first approach guarantees that the debugger will never do more than a constant amount of "extra" speculative execution or re-execution, but requires a number of checks linear in the actual distance to the realization time. The second approach requires only logarithmically many checks, but may execute speculatively twice as far as necessary in the worst case, and hence require up to three times as much total execution when re-execution is included. At present, **smld-t** uses the fixed interval approach.

Speculative computation introduces difficulties for programs that perform I/O: if the programmer has asked to break at a particular location, **smld-t** 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. This *conditional* I/O mode complements the quiet and noisy modes described in Section 5.1.

5.4 Checkpoints and Re-execution in SML/NJ

Smld-t supports replay by taking periodic checkpoints of program state during initial execution. To reset to a particular target time, **smld-t** first restores the nearest previous saved checkpoint and then re-executes to the target. Average re-execution time, and hence average total reverse execution time, can be minimized by keeping as many checkpoints as possible. On the other hand, checkpoints occupy space, so **smld-t** cannot keep an

[3] 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.

indefinite number of them. The solution is to maintain a cache containing checkpoints that are expected to be useful; the questions of when to take checkpoints and which ones to keep are addressed in Section 6.7. An important way to reduce the space occupied by a series of checkpoints is to make the checkpoints incremental. Since program state usually changes gradually, a typical checkpoint contains substantially the same information as its predecessor with minor variations. Ideally, the checkpointing mechanism should keep just one copy of the shared information plus the incremental changes to it.

The state of an SML/NJ program at a particular execution event consists of 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.

Because SML/NJ uses continuation-passing style and allocates closures on the heap, a first-class continuation is essentially a pointer to an already-existing heap structure, so saving and restoring first-class continuations is fast. Moreover, these continuations are inherently incremental: if **smld-t** 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 **smld-t** synthesizes its own store checkpoints. It uses an approach that does not involve changing the store's representation: operations on the store, which are distinguishable at compile time using type information, 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.

A cleaner approach would be to simulate the mutable store in pure functional style (using balanced-tree representations, for example), so that its state became part of the continuation. Alternatively, an explicitly persistent (but not necessarily pure functional) data structure could be used for the simulation. These approaches would make both reads and writes to the store more expensive by a factor logarithmic in total store size.

If more reliance on back-end facilities is allowed, other approaches become possible. For example, the garbage collector also needs to keep track of changed pointer objects in old space, and this mechanism could be adapted for use by the debugger. If memory layout were controlled so that mutable objects were grouped together on distinct hardware pages, the checkpointing mechanism could take copies of entire pages, although this approach might interact poorly with copying garbage collection.

To ensure correct re-execution, **smld-t** 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 programmer, 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. 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. **Smld-t** does not support reverse execution for such programs, although extending logging methods to them would not be difficult in principle.

Smld-t holds all checkpoint data in the heap. With the current SML/NJ garbage collector, this approach effectively constrains checkpoint storage to fit in main memory, which has a significant impact on **smld-t**'s performance. Under a multi-generational garbage collector, checkpoints should automatically migrate to older generations and eventually to backing store. It would also be possible for **smld-t** 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.

5.5 Related Work

Although reverse execution is not provided in most commercial debuggers, it has been the subject of research and prototyping since the earliest days of source-level debugging. There has also been long-standing interest in generalizing and automating the implementation of "undo" facilities in programming languages [14, 79, 109]. Typically this work is focused on interactive command languages, in which the action to be "undone" is a command explicitly issued by the user. However, it was recognized early on that both the semantic and implementation issues raised by such systems are relevant to general-purpose programming languages, and that "undo" can be used as a debugging aid in such languages [14]. Reverse execution technology is also relevant in the areas of fault-tolerant systems and database systems. Finally, in recent years, debugging researchers have focused on parallelism, and a number of replay debuggers have been built for parallel systems; these are surveyed in Chapter 9.

5.5.1 Execution Histories

One approach to supporting reverse execution is to maintain a complete history of the program steps and memory updates that occur during forward execution. Balzer's EXDAMS system [16], discussed in Chapter 3, produces a "history tape" during program execution that can be read later to simulate re-execution either backward or forward.

Since complete histories can grow very large, most systems settle for a limited reverse execution capability, based on a history list of bounded size. One of the earliest such systems was Grishman's AIDS debugger for FORTRAN on the Control Data 6600, intended for both batch and on-line use [58]. AIDS provides a special-purpose command language, including elaborate conditional trace and breakpoint facilities, and a `RETREAT` command to step backward through the program by a specified number of machine language instructions. `RETREAT` is implemented by keeping a circular log of the last few thousand memory changes and taking periodic snapshots of the register set, of which the most recent two are kept; retreat is possible to any point after the older of the two snapshots. To return to a specified target, AIDS restores a snapshot, undoes memory changes back to the point corresponding to that snapshot, and then re-executes forward to the target. AIDS uses a machine-level debugging model; Grishman argues that this is the only possible approach in his environment, where assembly language routines and faulty compilers were the norm.

Zelkowitz [118, 119] describes modification of PL/C, a PL/I batch compiler, to support an enhanced language including a `RETRACE` statement that backs up execution to a specified label (within the same procedure as the `RETRACE`), by a specified number of statements, or until a specified condition is true (checked only within the procedure). After execution has been backed up, the system executes a specified PL/I statement, which can be used to alter state and try an alternative path through the program. Since PL/C is a batch system, the user must in effect hard-code a debugging strategy—typically a backtracking search of the execution space—into the program. Retracing is implemented by modifying the compiler to insert object code instrumentation that maintains a trace table (a circular buffer of about 2000 entries) reflecting all changes to memory. Performance on a sample of 7 small programs showed an average increase in program size of 40% and in execution time of 70%. Anticipating one of our favorite arguments, Zelkowitz [119] comments that "since core is relatively inexpensive and the usual interpretive debugging systems run at a factor of 40 to 50 slower, the [increased size and execution time] are tolerable costs for these features."

Teitelman's Programmer's Assistant [109] provides an interactive "undo" facility for LISP programmers based on a history list. Each operation that changes system state is supposed to store enough information on the history list to enable its effects to be undone. Since storing this information is expensive, every destructive primitive operation comes in both reversible and irreversible versions. The programmer can specify which version is to be used; for typed-in expressions or when a special test mode is enabled, the reversible versions are automatically substituted for the irreversible ones.

The Cornell Program Synthesizer [108] also includes a mechanism for reverse execution, which is implemented by processing a history file of control-flow decisions and memory updates produced during interpreted forward execution. The history file is of bounded length.

Borland's Turbo Debugger 2.0 for the IBM/PC has a reverse execution facility based on a history of 400 to 3000 machine language steps [45]. The history is built up only when the user is single-stepping forward at source level or machine level, and is discarded if the user continues normal full-speed execution.

These histories are all essentially lists of changes to memory locations or program branch decisions, held in a separate region of memory. An alternative approach is to integrate checkpoint support into the implementation of mutable memory; i.e., treat mutable memory as an abstract data type that supports checkpointing and resetting operations as well as fetches and stores. Johnson and Duggan [68] proposed implementing first-class stores for the language GL (a lambda-calculus derivative extended with data types) by applying the persistent data structure techniques of Driscoll, *et al.* [41] to the language's runtime data structures.

Miller and Choi's PPD debugger [26, 27] uses history logging and re-execution techniques to produce dynamic dependence information for flowback analysis (see Section 5.5.7); in principle it can also support user-level reverse execution. Rather than logging every memory update individually, PPD statically divides the program text into small units called *e-blocks*, and logs the effect of each e-block as a whole. An e-block can be any consecutive set of statements with a single entry point, but is typically a single procedure or loop body. PPD instruments the object code for each e-block to generate a prelog, containing the values of variables read by the block, and a postlog, containing the values of variables written by the block; these sets of variables are determined using data-flow analysis.

To reset the program to a previous execution point, PPD finds the log entry for the e-block execution containing that point and resets the values of the variables mentioned in the prelog, re-establishing the correct conditions for re-executing the code within the block. Postlogs are used to avoid re-executing calls to nested e-blocks. In effect, log entries are a variant of incremental checkpoints. Efficient and balanced performance of the system requires a good choice of e-block size [28]: smaller e-blocks allow faster re-execution at the expense of slower initial execution and larger space requirements for the log, while larger e-blocks slow down re-execution.

Agrawal, DeMillo, and Spafford [4, 5] have implemented a similar form of replay called "structured backtracking." During forward execution, their Spyder debugger maintains statement change sets, i.e., names of variables that are changed by each statement together with their old values. To save space, change sets for statements embedded within a larger block (compound statement or function body) are discarded when the block completes, in favor of a single change set for the entire block. This implementation prevents backtracking directly to an arbitrary point in the middle of a previous block. Instead, the user must backtrack to the beginning of the previous block and then re-execute forward; in principle, the system could do this automatically. If this scheme were applied to ML, the space requirements would be similar to those of **smld**, since all variables in scope at any point on the call chain must be kept.

5.5.2 Page-level Checkpointing

A number of incremental checkpointing schemes have been implemented at machine level; they rely on support from the underlying operating system or runtime system. These systems use the hardware page as their checkpointing increment on the assumption that most applications display locality of update activity. Using pages also allows a compact description of checkpoint contents.

The COPE system [14] uses a centralized, incremental checkpointing scheme to implement an "undo-redo" mechanism for a program development environment. All data describing the state of the environment are stored in COPE's block-structured file system, so the entire effect of a command can be represented as a set of changes to blocks. Each command execution takes a copy of any block it changes; these copies constitute a log of changes that can be reversed to "undo" the command.

IGOR [46], an MC68000-based debugger for C programs running under the Unix-like distributed operating system DUNE, provides reverse execution in a conventional compiled environment. The operating system is modified to provide a list of changed pages on demand; these are periodically checkpointed to disk. Measurements of the IGOR prototype suggest that performance is highly dependent on the amount of update locality in the application program and the frequency of checkpointing; in the experiments described, checkpointing overhead ranged from 40–70% at 1 second checkpointing intervals to 100–400% at 0.1 second intervals, and the percentage of pages touched per checkpoint ranged from 5–75%.

Wilson and Moher [117] suggest viewing process history abstractly as a two-dimensional array indexed by both memory address and time. They present a design for a general-purpose mechanism that implements lookup in such an array lazily, using a "demon" to reconstruct missing states with the aid of periodic incremental checkpoints and replay. This "demonic memory" system would use memory management hardware to detect touched pages when collecting a checkpoint. Demonic memory is intended to support full replay for an execution ranging up to months or years in duration. Over time, checkpoints would be thinned out and moved to cheaper, slower levels of the memory hierarchy, so that retrieval time for a state is correlated with its age. The authors point out the importance of coordinating garbage collection with checkpointing.

5.5.3 Semantic Frameworks

There has been considerable work on giving precise models and definitions for what it means to "undo" or "redo" operations in an interactive programming environment [14, 79, 114]. We avoid most of these issues because our model for execution is so simple: for example, **smld-t** does not provide automated support for debugging alternative executions based on different input parameters.

Some Prolog debuggers use the language's built-in backtracking model to support limited reverse execution [20, 22]. Prolog execution can be viewed as a sequence of clause selection and unification operations; a Prolog system must keep a full history of these operations, in order to undo them if backtracking is required. The debugger can reset the program to an earlier point in the execution by forcing backtracking artificially; this technique requires no extra storage and minimal extra machinery. It does have limitations: the use of cut can remove parts of the backtracking space and make the corresponding part of the execution history unavailable; also, side-effects such as `assert` are not undone by backtracking.

5.5.4 First-class Stores

Smld-t could be implemented more elegantly if SML-NJ supported first-class stores as well as first-class continuations. Wilson and Moher [117] suggest a `call-with-captured-state` (or `call/cs`) primitive for their demonic memory system that packages both the current continuation and the current store into a first-class object. Johnson and Duggan [68] provide a full denotational semantics for GL extended with first-class stores and continuations. There are operators to return the current store (and current continuation) and a `fork` operator that applies a specified continuation to a specified store; these operators are intended to be used for "experimenting" with the effect of a continuation on a variety of stores, which the authors think will be useful for debugging. Several authors have noted that the contents of a saved store can be useful in contexts other than reverse execution. In IGOR [46], for example, the checkpoints for a given page can be used to generate a coarse-grained update history for a given variable, without the need for re-execution.

5.5.5 Clocks

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"). Reverse execution systems must provide a simple, compact way to specify event executions, for use both by the programmer, e.g., specifying time-travel targets, and for internal record-keeping, e.g., logging asynchronous events.

Smld-t identifies states using a software clock whose time is incremented at each instrumented program event, as described in Section 5.1. This approach has the advantage of being entirely independent of the back-end and machine, but there are other ways to provide the essential features of a clock: times that increase predicatably and monotonically during execution and an alarm facility.

Cargill and Locanthi [24] suggest using hardware instruction counting to support repeatable execution for debugging tools including reverse execution and watchpointing, i.e., stopping a process when a particular condition on the data becomes true. They sketch implementation of an hardware instruction counter for an MC68020 workstation: the counter is decremented on each instruction and an interrupt is raised if it goes to zero.

Mellor-Crummey and Leblanc [83] suggest an alternative software instruction counting technique based on instrumenting assembly code. They point out that only (potentially) backward jumps and calls need to be counted; the value of the program counter can be used to get finer resolution among events with the same count. Instrumentation takes the form of explicit decrement and conditional branch instructions on a dedicated (ordinary) register. They measured overhead of 0% to 12.5% for this technique for a range of C programs on an MC68020, and predict overheads of up to 20% on RISC architectures. They use this technique in their Instant Replay system for debugging parallel shared-memory programs, which is examined in more detail in Chapter 9. SML/NJ's back-end has a similar mechanism used to count instructions for performance studies (see Section 8.4).

Wilson and Moher [117] note that in heap-based languages, the allocation pointer (plus the program counter) can be used as a sort of clock whose alarm is set by altering the allocation limit. Since object code generated by SML/NJ allocates frequently and could be forced to allocate in every extended basic block without sacrificing much performance, this clock could be made quite precise.

Unfortunately, counts have no intrinsic meaning for the user. It seems likely that the user will gain a feeling for the meaning of different step distances only after experimenting with each program; e.g., after tracing a few iterations of a loop, the user will have come to know that it takes a certain number of steps and will then be able to issue temporal commands more boldly. Counts based on source-language constructs may be more useful than instruction counts. In the Cornell Program Synthesizer [108], for example, step distances are defined in terms of the same syntactic units used by the editing tools in the Synthesizer environment, i.e., individual statements or block constructs, depending on context.

5.5.6 Predicates

Another approach to identifying states is to use defining predicates, or boolean conditions, resembling the qualifiers on conditional breakpoints in conventional debuggers. A predicate should be definable in terms of any variables in scope in the source program and also in terms of useful meta-information such as the depth of the call stack. Ideally, predicates should be monotone, so that they have a single well-defined realization point. Monotone predicates also have the advantage that they can be reliably tested against checkpointed states to narrow the required interval for re-execution [46].

Cargill and Locanthi [24] note that the realization time for a monotone predicate can be found without single-stepping; they suggest an interval search process essentially similar to our speculative computation.

Executing to a state defined by a general predicate has typically been implemented by single-stepping, with the predicate conditions being checked after each step. An example of this approach is AIDS [58], which allows any valid FORTRAN arithmetic or logical expression to be used as the predicate. The system is implemented by simulating object-code execution; since simulation is at least 60 times slower than ordinary execution, it can be selectively disabled in regions the programmer thought to be bug-free. IGOR [46] also uses an object-code interpreter, which checks the target condition at each step, estimated to be 140 times slower than direct execution.

5.5.7 History-based Debugging

Smld-t's features, which include value querying, breakpointing, and single-stepping, are fairly ordinary: the system is unusual because it integrates these features with a flexible time-travel mechanism. Several earlier systems, including EXDAMS and the Cornell Program Synthesizer, also emphasized making standard features work in a uniform manner forward and backward. These systems also stress the importance of good visual feedback during execution, including displays showing the program counter as a position in the source text and showing the current values of "watched" variables. Indeed, the Synthesizer's authors consider that "the novelty in [the Synthesizer's use of reverse execution] is the presence of enough immediate visual feedback to make the feature meaningful as a real-time control mechanism. [108, page 569]"

EXDAMS [16] also contains a novel debugging aid, called *flowback analysis*, which uses the availability of history information to show how a variable comes to have a particular value at some moment in execution. Requesting a flowback analysis for a variable produces an inverted tree-shaped display of the executed assignments that affected the variable's value, i.e., all assignments from which there is a data dependency. Flowback analysis has been revived more recently in the parallel debugging work of Miller and Choi [26, 27].

Agrawal, DeMillo and Spafford's debugger includes a tool with a similar focus, based on dynamic program slicing [4, 6]. Their debugger highlights the program slice (set of statements) whose execution has affected the value of a specified variable. Dynamic slices can be recorded at varying degrees of precision by building dynamic variants of the program dependence graph at runtime.

Chapter 6

Implementing Time Travel

The implementation of **smld-t** is similar to that of **smld**. In particular, the instrumentation schemes of the two debuggers are identical, once certain uniform changes are made to the instrumentation code and to the interpretation of certain variables. **Smld-t** has additional code to control reverse execution, to save and restore checkpoints, and to maintain the checkpoint cache. **Smld-t**'s performance is described in Chapter 8.

6.1 Instrumentation for Time Travel

The time-travel concepts developed in Chapter 5 are implemented using a slightly modified version of the instrumentation scheme of Section 4.1. The key change is to replace the explicit event record pointers used in **smld**'s instrumentation code with event times. The overall structure of **smld**'s instrumentation code, instrumenting algorithms, and query algorithms is unchanged, but executing the algorithms under **smld-t** may involve time travel. Events are located at the same sites and event records have the same contents, except that the `binding` field becomes an event time and the `caller` field is omitted. The instrumentation algorithm of Figure 9 and the coalescing and linearizing improvements described in Section 4.3 are unchanged with the understanding that every identifier of the form `event_`n is interpreted as a time rather than as a record pointer.

The instrumentation code executed by **smld-t** at each event is shown in Figure 18; it differs significantly from **smld**'s instrumentation. Each event increments the `currentTime` clock and breaks into the debug monitor if `currentTime` matches `targetTime`. There is no `breakWanted` array; location breakpoints are implemented as described in Section 5.3

```
type time = int
val currentTime : time ref
val targetTime : time ref

val lastTimes: time array

datatype eventRecord =
     EVENT of {siteNumber: siteNumber,
               binding: time,
               boundValues: value list}
   | NULLEVENT

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

Figure 18: **Smld-t**'s `eventRecord` data structure and event instrumentation code.

using the `lastTimes` array, which is updated to record when a site has been visited. The event record is built only if a break is taken, in which case it is passed directly to the debug monitor. There is no need for a `lastCaller` reference since the `APP` event corresponding to a `FN` event at time t always has time $t - 1$.

In terms of our schematic description in Section 5.2, the work done in `recordState` for each event consists of updating `lastTimes` and (possibly) keeping the event's time in an `event_`n variable. Executing this code should be faster than building an entire event record (and possibly storing a pointer to it). It should also require significantly less space for live variables because variables in suspended functions on the call chain are *not* kept artificially live in an event record, unless that record has been built explicitly in response to a user query. Also, **smld-t** need not maintain `lastCaller`, which saves time and also avoids the associated garbage-collector store-list overhead. The break test itself should have roughly the same cost as in **smld**: one array load and test is replaced by two scalar loads and a compare. On balance, we expect controlled execution under **smld-t** to be somewhat faster than under **smld** if the costs of logging and checkpointing are ignored; this is confirmed by the performance figures in Section 8.5.

Smld-t uses a simple software instruction counter to track the current time and to signal when the target time has been reached. The costs of this approach are straightforward: pointers to the current and target times must be carried as global variables in the instrumented program, and at each event the current time must be fetched, incremented, and stored, and the target time must be fetched and compared.

Overflow is one potential difficulty: SML/NJ integers occupy only 31 bits, which allows for a total of 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. Thus, substantial programs can be debugged, but many programs will cause overflow. Unfortunately, any scheme representing times in two words is much more expensive, because SML/NJ represents a single integer directly but must use a record and a pointer for a pair. A compromise would be to maintain `currentTime` as a relative clock, counting events since the debug monitor last transferred control to the user program; the length of controlled executions could be artificially limited to guarantee that the relative counter doesn't overflow. These relative times could be used directly for halting execution at a specified target time and could even be written into the `lastTimes` array, if the debugger used a post-execution pass to convert relative times to absolute times. The debugger would still need to generate absolute times for `event_`n variables. This scheme would certainly be slower than our present one; we have not implemented it. In any case, the advent of 64-bit machine architectures should alleviate the overflow problem.

6.2 Controlling Execution

Under **smld-t**, the user program may be *recording*, *replaying*, or *halted* at some event with an associated state. The distinction between recording and replaying governs whether logs describing changes to the mutable store or external inputs are written or read. When the user program is halted, the debug monitor's code is active, and vice-versa; transitions between debug monitor and user control are essentially coroutine hand-offs, implemented with `callcc`, as in **smld**.

Simplified code for the basic control functions is shown in Figure 19. The actual code for these functions is more complicated because **smld-t** may execute one compilation unit "in the middle of" another one; this feature is required to support time-travel between top-level units (see Section 5.1) and to handle programmer interpolations (e.g., explicit updates to mutable variables). The actual `execTo` routine keeps track of nested compilation units and transfers control among them at the correct times. The code shown has also been simplified by omitting the I/O mode specification (i.e., noisy, quiet, or conditional) required by `recordTo` and `replayTo` (see Sections 5.1 and 5.3).

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; **smld-t** must jump back one time unit to find the last real event and pinpoint the source of the exception.

The user program's state consists of the information that distinguishes a particular event and makes it possible to replay execution from that event. The state includes the event record, a user program continuation, a description of contents of the mutable store (including `currentTime` and the contents of `lastTimes`), and pointers into the various logs (such as the stream I/O log). A checkpoint is a compact characterization of a state suitable

```
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;
     userMode := mode;
     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
```

Figure 19: Implementation of basic control functions for **smld-t**.

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. Their implementation is discussed in Section 6.4. Once taken, a checkpoint remains valid until it is discarded however many times it is used and regardless of other execution and checkpointing activity. Checkpoints are taken at the target of each `resetTo` or `advanceTo` operation. Thus, responsibility for choosing the frequency of checkpoints falls upon the callers of the these functions: either explicit time-travel commands or the debugger-generated internal time-travel commands discussed in Section 6.3.

Smld-t's checkpoint cache is accessed using two routines:

```
findPrevCheckpointInCache:time->checkpoint
```

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

```
putInCache:checkpoint->unit
```

attempts to insert its argument into the cache, subject to the cache's membership policies. Cache membership and replacement policies and implementation are discussed in Section 6.7.

Figure 20 gives code for the `advanceTo` and `resetTo` functions in terms of the primitives described in this section. 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" [63]. 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.

6.3 Debugger Functions

Smld-t implements location and time-based breakpoints, variable lookup, and call traceback using time-travel operations as primitives. Time-based breakpoints are implemented in terms of `advanceTo` and `resetTo`. Location-based breakpoints are implemented using the binary-search algorithm sketched in Section 5.3; a simplified version of the code for

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

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

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

Figure 20: Time-travel functions. The `checkpoint` data type, `getState`, and `resetState` are described in Figure 24.

```
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
```

Figure 21: Finding realization time for monotonic predicate by binary search.

this `advanceToRealizationTime` function is shown in Figure 21. 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`. It is faster to reset to target times that have entries in the cache than to arbitrary times; the actual code for `narrow` consults the cache to choose such targets when possible, rather than performing a binary search. To execute forward from time `t0` to a location breakpoint at site `s`, **smld-t** invokes `advanceToRealizationTime` with the function:

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

```
fun lookup (x:id,t:time) : value =
  case eventRecordAt t of
    NULLEVENT => raise notFound
    EVENT{siteNumber:siteNumber,
          binding:time,
          boundValues:value list} =>
      let val idList = idsBoundAt(siteNumber)
          val pos = index(x,idList)
                  (* return SOME(position of x in idList) or
                            NONE if not present *)
      in case pos of
           SOME n =>
             nth(boundValues,n) (* extract element n from boundValues *)
         | NONE => lookup(x,binding)
      end
```

Figure 22: The identifier `lookup` algorithm for **smld-t**.

To execute backward to this location breakpoint, **smld-t** simply calls

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

Smld-t recasts **smld**'s query algorithms to operate on times instead of explicit event records, as described in Section 5.2. A new function

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

maps known event times to the corresponding event records. A simple lazy implementation of `eventRecordAt(`*time*`)` just calls `resetTo(`*time*`)`, extracts the event record from the newly current state, and resets to the original time. Better implementations are discussed at the end of this section.

Smld-t's identifier lookup algorithm is shown in Figure 22; it has the same form as **smld**'s algorithm (see Figure 6), but takes a `time` t as argument and calls `eventRecordAt(`t`)` to get and unpack the corresponding event record. **Smld-t**'s `lookup` may have to perform time-travel, including possible re-execution, to obtain the event record containing the value of an identifier. Unless the desired value is already stored in an earlier checkpoint's continuation, it will be recalculated during the re-execution. Thus, it will not occupy the same memory location as the value calculated during initial execution, and possibly pointed to by continuations in checkpoints at later times. For ordinary immutable values, this behavior

```
fun findCaller(t:time) : time =
  case eventRecordAt t of
    NULLEVENT => raise noCaller
  | EVENT{siteNumber:siteNumber,
          binding:time,...} =>
      if siteTypeOf(siteNumber) = FN then
        t-1    (* corresponding APP event occurred one time unit earlier *)
      else findCaller(binding)
```

Figure 23: The `findCaller` algorithm for **smld-t**.

poses no semantic problems since such values cannot be distinguished by memory location.[1] 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 `lookup` on the `ref` variable would return a pointer to a valid cell of the correct type, but typically containing the wrong value. To avoid this problem, **smld-t** instruments `ref` creations so that actual creation occurs only on initial execution, while on replay the original cell is reused (see Section 6.5).

The time-travel version of `findCaller`, which both takes and returns a time, is shown in Figure 23. It uses the sequential ordering of events provided by times.

During the course a typical debugger session, `eventRecordAt` is called repeatedly with the same set of time arguments (e.g., the binding times for global variables). Therefore, implementing `eventRecordAt` by invoking `resetTo` as described above is naive, since the overhead of performing a reset is large, even if the cache contains a checkpoint for the target time. Instead, **smld-t** memoizes the `eventRecordAt` function: when an event record is computed it is remembered for subsequent reuse. 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.

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

6.4 Checkpoints and State

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. The bulk of the state-related code in **smld-t** 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, the `lastTimes` array, and multiple compilation units. Most subsystems maintain significant internal state, such as a log filled under record mode and used to control replay. The subsystems communicate with the `getState` and `resetState` functions using a standard interface, summarized in Figure 24.

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, **smld-t** 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. Unfortunately, instrumentation code is not well modularized by subsystem since it is all inserted in a single pass through the abstract syntax for efficiency reasons. 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.

6.5 Mutable Store

Smld-t 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 pointer to 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

```
type memory = {undo:unit->unit,
               redo:unit->unit}

val subsystems : (unit -> memory) list (* remember functions *)

datatype checkpoint  =
       CP of {time:time,
              event:eventRecord,
              functionalState:unit cont,
              subsystemMemories:memory list
              }

fun getState() : checkpoint =
       CP {time=!currentTime,
           event=!currentEvent,
           functionalState=!userCont,
           subsystemMemories=app (fn remember => remember ()) subsystems}

fun resetState (CP{time,event,functionalState,subsystemMemories}) =
    let val originalTime = !currentTime
    in currentTime := time;
       currentEvent := event;
       userCont := functionalState;
       app
         (fn {undo,redo} => if originalTime < time then redo() else undo())
         subsystemMemories
    end
```

Figure 24: Saving and restoring checkpoints.

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. The solution is to use weak pointers, described in Section 2.1.3, in the global lists, and to omit invalidated pointers from the delta list. To maximize the number of invalidated weak pointers, **smld-t** forces a minor collection, which typically takes only a few dozen milliseconds, before taking the checkpoint. Once stored in a checkpoint, the pointers become ordinary and the cells they point to 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 subsystem 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 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 subsystem 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 subsystem keeps a weak pointer to it and waits for that pointer to be invalidated.

As explained in Section 6.3, actual `ref` and `array` creations occur only during recording; on replay, the originally created object is reused. To arrange this reuse, **smld-t** 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 **smld-t** uses an algorithm that stores a unique pointer value called the *mark* into the `ref` cells themselves, regardless of type, using an unsafe `cast` operation. The algorithm operates in two passes to produce an output list of unique (pointer, value) pairs from the original global list.

1. The pointers in the global list are processed in reverse order. If the value of the cell pointed to is not the mark, the cell has not been visited during this pass; the cell pointer and value are paired and added to the output list, and the mark is written into the cell. If the cell value contains the mark, it has already been visited during this pass.

2. For each element in the output list, the contents of the cell are reset to the recorded value, overwriting the mark, and leaving the cell in the correct state.

With small extensions, this algorithm works for arrays and handles creation as well as update operations. The algorithm operates in linear time and requires no extra space. There are type-safe algorithms that take linear time and linear extra space, or take quadratic time.

The algorithm described above depends on some basic knowledge about the runtime format of data and makes essential use of weak pointers. We could increase **smld-t**'s dependence on the back-end by using the SML/NJ garbage collector's store list (see Section 2.1.3). This list is the same as the subsystem's global update list except that non-pointer cells (such as integers) are not included, *all* pointer cells *are* included (including internal debugger code `ref`s), and the list is reset at each minor collection. An early version of **smld-t** modified these details and used the store list directly to build delta lists. We switched to an instrumentation approach to decrease **smld-t**'s dependence on the collector's implementation. However, any generational collection algorithm must track updated mutable cells in older spaces, so it is not unreasonable to require that the collector be able to generate a list of changed cells as a byproduct of collection.

6.6 External Input

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 extension.

During noisy forward travel, the debugger's I/O library attempts to reproduce the original sequence of output on the terminal (including the echoing of keyboard input). This effect can be achieved by re-executing the program. However, to avoid re-execution when jumping forward from one known time to another where there is an intermediate checkpoint, **smld-t** logs output to the terminal as well as input. Output is redisplayed directly from the log, at the cost of increased log size.

Smld-t handles `CTRL/C` signals slightly differently from **smld** (see Section 4.6). In place of the `breakArray` technique used there, **smld-t**'s `CTRL/C` handler sets a flag indicating that an interrupt has occurred, sets

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

and returns. When the program breaks at the next event, the debugger notices the interrupt flag and interprets it as a request to "halt execution." The meaning of this request depends on context. If **smld-t** 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 **smld-t** 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 its state before the operation began.

A similar mechanism could be used to synchronize the handling of signals intended for the user program, which could then be logged and replayed. On replay, the log would need to be consulted to determine when to (artificially) interrupt the program and invoke the user's signal handler. We have not implemented such a mechanism.

Both **smld** and **smld-t** support a general mechanism for evaluating an arbitrary fragment of code while stopped at any breakpoint. The code is parsed in the environment of the current event. The primary use for this feature is to update mutable variables by evaluating assignment expressions for their side-effects (see Section 2.2.4). **Smld** uses the ordinary expression evaluator. **Smld-t**, however, must preserve the fragment of evaluated code, and re-evaluate it at the same time on any replay. In fact, **smld-t** should treat the interpolated code as a normal part of the program for the duration of the debugging session, and must allow the programmer to debug it in the usual way by stepping through it, setting breakpoints in it, and so forth. Thus the interpolation is compiled with instrumentation, and executing it updates the time. Since the interpolation typically changes the execution path of the program, **smld-t** must also truncate the time line at the current time as described in Section 5.1, by discarding all future checkpoints and resetting `maxKnownTime`.

It should also be possible to evaluate an expression for debugging purposes *without* changing the current state or truncating history, even if the evaluation invokes debuggable functions. Therefore, **smld-t** allows the user to distinguish interpolations from ordinary expressions to be evaluated.

6.7 Checkpoint Caching

The cache is simply a set of checkpoints, ordered by time, accessed by the operations

```
findPrevCheckpointInCache:time->checkpoint
```

and

```
putInCache:checkpoint->unit
```

described in Section 6.2. Memoization for `eventRecordAt` is implemented using a separate hash table, which is also updated by `putInCache`.

To manage the cache sensibly requires some idea of the pattern of time travel expected. 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 user's choice of targets is unpredictable, but it is reasonable to suppose that targets visited recently have a good chance of being visited again soon. The implicit jumps made by **smld-t** during variable lookup or call-trace generation are to events on the static or dynamic chain. The most recent events in these chains are clustered near the starting event; earlier events in the chains may be scattered widely over the time line. If memoization is not enabled, recently visited events have a strong chance of being revisited, but if it is enabled, events are never revisited. These time-travel patterns suggest that the cache should include entries spaced throughout execution history, with increased density near recent time-travel targets.

As described in Section 6.2, 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 of `advanceToRealizationTime`. The value of `timeDelta` therefore has a major influence on the number of checkpoints generated during execution. Experiments suggest that overall performance is optimized by using a delta size of 8,000-256,000 events, roughly 10-150 msec of user program CPU time on our hardware (see Section 8.8). Obviously, most of these checkpoints are *not* useful, so our cache membership policy must be able to weed them out easily.

Given our analysis of time-travel patterns, it seems natural to implement pre-caching. Suppose the user jumps forward to a time t for which there is no nearby cached checkpoint. It is likely that times in a small interval preceding t will be targets of subsequent jumps. Therefore, it may be worthwhile to pause in the course of executing forward to t, and take a preliminary checkpoint at some time $t - \epsilon$, thereby enabling quick replay to events in the interval from $t - \epsilon$ to t. The effects of pre-caching on performance are explored in Section 8.8.

Smld-t controls cache size on the basis of memory availability rather than by maintaining a fixed number of entries. This policy allows **smld-t** to take advantage of all the available physical memory without being limited to a fixed maximum count. 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, **smld-t** 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. Given that we use a copying collector, we obviously must have $r_0 > 2$. Appel's experiments [11, page 193] indicate that setting $r_0 < 3$ causes too much garbage collection, while setting $r_0 > 6$ ceases to improve overall collection time; $r_0 = 5$ offers a reasonable compromise between execution and collection time and acceptable overall performance. **Smld-t**'s cache sizing policy allows insertions if $r > r_0$; if $r \leq r_0$, **smld-t** attempts to delete enough entries to bring r back to r_0 and retries the insertion.

This simple feedback mechanism should keep r near r_0. However, **smld-t** can only make a guess at how many entries should be deleted from the cache when a ratio adjustment is needed; since checkpoints vary wildly in their space requirements and multiple checkpoints can share spare, this guess may be inaccurate. Therefore, **smld-t** checks the new value of r after a deletion round to see whether an insertion is now permissible. Unfortunately, r

can only be calculated when the live-data size is known accurately, i.e., immediately after a major garbage collection (see Section 2.1.3); thus, **smld-t** must force a major collection after each deletion round. Thereafter, insertions will be permitted until a spontaneous major collection occurs and reports $r < r_0$; the next insertion attempt triggers another deletion round. In the steady state the cache tends to be full, so this mechanism may double the number of major collections. This problem can be ameliorated by overestimating the number of entries to be removed in each deletion round, which tends to decrease the frequency with which the cache fills up.

The cache membership policy defines which candidate checkpoints are actually inserted in the cache and which existing entries are deleted if there isn't enough space for the newcomers. As in any caching scheme, there is an optimal sequence of decisions that can be determined offline given the entire target sequence; it is harder to compute than for ordinary caches because the cost of a "miss" is proportional to the distance from the probe to the nearest earlier entry. **Smld-t** faces an online problem; it uses heuristics to control cache membership. Given the time-travel patterns described above, these heuristics are easy to state. At most one checkpoint should be kept per cluster of target times, as close to the beginning of the cluster as possible, and if a cluster representative must be deleted the cluster chosen should not have been targeted recently; otherwise, checkpoints should be fairly evenly distributed over the total known time line.

To give a precise algorithm based on these heuristics, we use the average time gap g between adjacent cache entries:

$$g = \frac{\texttt{!maxKnownTime}}{\text{number of entries in cache}}$$

and introduce a constant cluster factor $f : 0 < f < 1$; entries that are within $f \times g$ time units of each other are considered to belong in the same cluster while entries further apart are in different clusters. We typically set $f = 0.25$. The algorithm for handling a potential checkpoint at time t is then as follows, where $pred(t)$ and $succ(t)$ are t's temporal predecessor and successor in the cache.

1. If there is room in the cache, insert t.
2. Otherwise, if $t - pred(t) < f \times g$, do nothing.
3. Otherwise, if $succ(t) - t < f \times g$, then replace $succ(t)$ with t.
4. Otherwise, perform a deletion pass on the cache, and insert t if there is now room.

In the deletion pass, each entry to be deleted is chosen by taking the least recently accessed entry t_0 such that $t_0 - pred(t_0) \leq g$. The condition on t_0 is important to help keep the checkpoint distribution fairly even. A pure LRU scheme would allow inter-entry times to grow arbitrarily large in regions of the time line that have not been visited recently; Section 8.8 measures pure LRU against the algorithm described here. Finding t_0 efficiently is difficult because both LRU ranking and g can change after each insertion, so **smld-t** maintains an approximation of g that is updated each time the true value doubles and uses it to compute an approximation of t_0.

Chapter 7

Dynamic Type Reconstruction

Formal semantics can be used to give explicit meaning to the notion of "typed environment at an event." This notion is essential for a debugger to display or to compute safely with the values of variables (see Section 2.2.3). It may also be useful in other contexts, such as tag-free garbage collection [9, 56] or the addition of a LISP-style `eval` operator to the language.

Consider the expression

```
let id = fn x => x in (id (fn y => 3))(id true)
```

When this expression is type-checked, each variable and subexpression is assigned a type. In the function `fn x => x`, variable `x` is assigned the arbitrary type α, since the function body does not constrain `x` to have any specific type; the overall function is `let`-bound to the variable `id`, which is assigned type $\forall\alpha.\alpha \rightarrow \alpha$, indicating that it can be applied to arguments of any type. `id` is applied to arguments of two different types in this example. Suppose the user stops at a breakpoint just after `x` has been bound by one of these applications and asks the debugger to print the value of `x`. To do so, the debugger must know the *dynamic* type of `x`.

ML is deliberately designed so that dynamic type information is not necessary during normal runtime operation. The code associated with `id`'s closure is oblivious to the type of its argument, and ML does not have runtime type tags. For this reason, the SML/NJ `print` function is overloaded rather than polymorphic and demands that its argument have a well-defined static type; thus, it is impossible to insert a `print` statement to display the value of `x`. However, there are several ways to calculate an appropriate dynamic type for `x`. When the overall expression is type-checked, each *use* of `id` is assigned a distinct static type that depends on the context of the use. This type is always an instance of $\alpha \rightarrow \alpha$; in this example, the first use is assigned type (`bool` $\rightarrow$ `int`) $\rightarrow$ (`bool` $\rightarrow$ `int`); the second is assigned `bool` $\rightarrow$ `bool`. The debugger can insert additional instrumentation to track which use of `id` is being evaluated in order to determine the correct substitution for α. Alternatively, the debugger can use the fact that the dynamic type of the formal parameter `x` must match the dynamic type of the actual argument to which it is bound. The actual argument can be located using the binding and caller pointers already in place for value and call-trace lookup; its dynamic type can be calculated recursively.

We explore these approaches to defining dynamic types in the context of a formal semantics for a subset of ML. Our definition differs from superficially similar notions that have appeared in the literature; in particular, dynamic types should not be confused with type `Dynamic`, a type whose runtime values consist of a data value paired with a type tag [1]. Throughout this chapter, we assume familiarity with the basic methods of structural operational semantics. Our reasoning is informal, but our presentation provides a basis for future formal soundness proofs.

7.1 Semantic Framework

We use the structural operational semantics formalism employed by the ML definition [87, 88]. The discussion is based on a small expression language, EXP, that illustrates the main aspects of ML-style polymorphism and avoids some complicating features of the full language, such as recursive functions and imperative `ref` variables. Our treatment closely follows the presentation of Tofte [110, Chapter 2].

The set of expressions, Exp, has syntax

$$e ::= c \mid x \mid \lambda x.e_1 \mid e_1 e_2 \mid \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2$$

where

$$\begin{array}{rcll} c & \in & \text{Const} = \{\texttt{true}, \texttt{false}, \texttt{1}, \texttt{2}, \texttt{3}, ...\} & \text{(e.g.)} \\ x & \in & \text{Var} = \{\texttt{a}, \texttt{b}, \texttt{c}, \ldots\} & \end{array}$$

Variables can be λ-*bound*, `let`-*bound*, or *free*. An expression containing no free variables is *closed*.

We first define a static semantics, or set of typing rules, for EXP. The typing rules embody ML-style polymorphism: `let`-bound variables may be used polymorphically, but λ-bound variables must be monomorphic.

Metavariable b ranges over a pre-defined set of *base types*, BaseTy; function TypeOf maps each constant to its corresponding base type. Metavariable α ranges over an infinite set of *type variables*, TyVar. The set Ty of *types* is

$$\tau ::= b \mid \alpha \mid \tau_1 \rightarrow \tau_2$$

(where $\rightarrow$ is right associative) and the set TyScheme of *type schemes* is

$$\sigma ::= \tau \mid \forall \alpha . \sigma_1$$

A type containing no type variables is a *monotype*. A *type environment* is a finite map from variables to type schemes. To emphasize that λ-bound variables are always mapped to simple types, we write

$$TE \quad \in \quad \text{TyEnv} = \text{Var} \rightarrow (\text{Ty} \cup \text{TyScheme})$$

The domain of TE is Dom TE, and $TE \pm TE'$ denotes the environment formed from TE by adding each definition in TE', possibly superseding existing definitions.

Free and bound type variables in a type scheme are defined in the usual way; a type variable is free in type environment TE if it is free in some type scheme in the range of TE. The function tyvars(τ) returns the set of type variables that occur in type τ, tyvars(σ) returns the free type variables in type scheme σ, and tyvars(TE) returns $\bigcup(\text{tyvars}(\sigma))$ for all σ in the range of TE.

A *substitution* is a mapping S from type variables to types, which can be extended in a natural way to a mapping from types to types. Substitutions can be composed in the usual way: $(S_2 \circ S_1)\tau = S_2(S_1\tau)$. A substitution is *total* if its domain is all of TyVar; it is *ground* if every type in its range is a monotype. Substitution can be extended to type schemes and type environments in a straightforward way, taking care to rename bound type variables as necessary to avoid type variable capture.

The transformation of σ to $\forall\alpha.\sigma$ is *generalization* on α. Conversely, τ' is an *instance* of $\sigma = \forall\alpha_1. \ldots \forall\alpha_n.\tau$, written $\sigma > \tau'$, if there exists a substitution S with domain $\{\alpha_1, \ldots, \alpha_n\}$ such that $S(\tau) = \tau'$. Finally, $\text{Clos}_{TE}(\tau)$ means $\forall\alpha_1. \ldots \forall\alpha_n.\tau$, where $\{\alpha_1, \ldots, \alpha_n\} = \{\alpha \in \text{tyvars}(\tau) | \alpha \notin \text{tyvars}(TE)\}$.

A static semantics for EXP is shown in Figure 25. Inference rules have conclusions of the form $TE \triangleright e : \tau$, which may be read "under type environment TE, expression e

$$\frac{\text{TypeOf}(c) = \tau}{TE \overset{L}{\triangleright} c : \tau} \qquad \text{(CONST)}$$

$$\frac{TE(x) > \tau}{TE \overset{L}{\triangleright} x : \tau} \qquad \text{(VAR)}$$

$$\frac{TE \pm \{x \mapsto \tau'\} \overset{L1}{\triangleright} e : \tau}{TE \overset{L}{\triangleright} \lambda x.e : \tau' \to \tau} \qquad \text{(ABS)}$$

$$\frac{TE \overset{L1}{\triangleright} e_1 : \tau' \to \tau \quad TE \overset{L2}{\triangleright} e_2 : \tau'}{TE \overset{L}{\triangleright} e_1 e_2 : \tau} \qquad \text{(APP)}$$

$$\frac{TE \overset{L1}{\triangleright} e_1 : \tau' \quad TE \pm \{x \mapsto \text{Clos}_{TE}(\tau')\} \overset{L2}{\triangleright} e_2 : \tau}{TE \overset{L}{\triangleright} \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau} \qquad \text{(LET)}$$

Figure 25: Static semantics.

has type τ." Each sequent also has a *label*, L, the format of which is defined below, that identifies the sequent's position within an overall deduction; we sometimes omit the label when considering a sequent in isolation.

In general, an expression e can be given many different typings under the static rules and a given type assignment TE. We are particularly concerned with the *principal type* of e under TE, namely the type τ such that $TE \triangleright e : \tau$ and $TE \triangleright e : \tau' \Rightarrow \text{Clos}_{TE}(\tau) > \tau'$. Any expression that has a type in a given environment has a principal type in that environment, which is unique except for choice of bound type-variable names and can be inferred using the well-known type assignment algorithm ("Algorithm W") [36, 110].

The derivation of the principal type τ for a closed expression e, generated via the type assignment algorithm or any other means, can be represented as a proof tree called the *static tree* for e. The nodes of the tree are sequents, each labeled with a string over $\{1, 2\}$ that represents the path from the root of the tree to the node. The root is $\emptyset \overset{\epsilon}{\triangleright} e : \tau$, where ϵ denotes the empty string; the internal nodes are LET, APP, and ABS inferences, and the leaves are CONST and VAR inferences. Given a node with label L, the node's left or sole child has label $L1$, i.e., the string concatenation of L with 1, and the right child has label $L2$, as shown by the annotations on the typing rules in Figure 25. If $TE \overset{L}{\rhd} e : \tau$ then we refer to TE by TE_L and to τ by Type_L. As an example, Figure 26 shows a static tree for a slightly extended version of our earlier example expression, recast in EXP syntax: `let f = (λz.λx.x) 5 in (f (λy.3))(f true)`.

The shape of the static tree is isomorphic to the term structure of the expression [110]. Thus we can relate certain nodes in the static tree to expression sites, in the sense of Chapter 3; in particular, the children of ABS nodes and the right children of LET nodes correspond to function entry and value binding sites, respectively. This correspondence is explored further in Section 7.2.3.

Our dynamic semantics for EXP is conventional, except that we label each inference with a corresponding node in the static tree, as described below. The semantic objects are:

$$
\begin{array}{rcl}
c & \in & \mathrm{Const} \\
v & \in & \mathrm{Val} = \mathrm{Const} \cup \mathrm{Clos} \\
[x, e, E, L] & \in & \mathrm{Clos} = \mathrm{Var} \times \mathrm{Exp} \times \mathrm{ValEnv} \times \mathrm{Label} \\
E & \in & \mathrm{ValEnv} = \mathrm{Var} \rightharpoonup \mathrm{Val}
\end{array}
$$

The dynamic semantic rules are given in Figure 27. Each sequent has the form $E, L \vdash e \longrightarrow v$, which may be read "under value environment E (and static type environment TE_L), expression e evaluates to value v." As with the static semantics, the dynamic evaluation of a closed expression e to a value v can be viewed as a proof tree called the *dynamic tree* for e. Each node of the dynamic tree for e is labeled with a corresponding node L of the static tree for e. The root of the dynamic tree is $\emptyset, \epsilon \vdash e \longrightarrow v$; the internal nodes are LET and APP inferences and the leaves are CONST, VAR, and ABS inferences. Notice that ABS nodes are leaves of the dynamic tree but not of the static tree, since evaluation does not proceed "under a λ" but type inference does; conversely, APP nodes in the dynamic tree have three children rather than two, with the right child representing the evaluation of the function body. The correct label for the right child of an APP is fetched from the closure value generated by the left child. We include expressions in dynamic sequents for clarity even though the labeling makes them redundant, since if $E, L \vdash e \longrightarrow v$ and $TE \overset{L}{\rhd} e' : \tau$ then e and e' are the same expression. Similarly, Dom E = Dom TE. The principal type environment of a dynamic tree node $E, L \vdash e \longrightarrow v$ is TE_L. Figure 28 shows the dynamic tree corresponding to the static tree of Figure 26.

To explore the consistency of the static and dynamic semantics, we introduce an explicit relation $\models v : \tau$, read "v has type τ," between dynamic values and types and extend

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{TE_{111} \pm \{\mathtt{x} \mapsto \alpha\} \overset{1111}{\triangleright} \mathtt{x} : \alpha}
            {\underbrace{\{\mathtt{z} \mapsto \mathtt{int}\}}_{TE_{111}} \overset{111}{\triangleright} \lambda\mathtt{x}.\mathtt{x} : \alpha \to \alpha}
    }{\emptyset \overset{11}{\triangleright} \lambda\mathtt{z}.\lambda\mathtt{x}.\mathtt{x} : \mathtt{int} \to \alpha \to \alpha}
    \qquad
    \emptyset \overset{12}{\triangleright} \mathtt{5} : \mathtt{int}
  }{\emptyset \overset{1}{\triangleright} (\lambda\mathtt{z}.\lambda\mathtt{x}.\mathtt{x})\ \mathtt{5} : \alpha \to \alpha}
  \qquad
  \dfrac{
    \dfrac{
      TE_2 \overset{211}{\triangleright} \mathtt{f} : (\mathtt{bool} \to \mathtt{int}) \to (\mathtt{bool} \to \mathtt{int})
      \qquad
      \dfrac{TE_2 \pm \{\mathtt{y} \mapsto \mathtt{bool}\} \overset{2121}{\triangleright} \mathtt{3} : \mathtt{int}}
            {TE_2 \overset{212}{\triangleright} \lambda\mathtt{y}.\mathtt{3} : \mathtt{bool} \to \mathtt{int}}
    }{TE_2 \overset{21}{\triangleright} \mathtt{f}\ (\lambda\mathtt{y}.\mathtt{3}) : \mathtt{bool} \to \mathtt{int}}
    \qquad
    \dfrac{
      TE_2 \overset{221}{\triangleright} \mathtt{f} : \mathtt{bool} \to \mathtt{bool}
      \qquad
      TE_2 \overset{222}{\triangleright} \mathtt{true} : \mathtt{bool}
    }{TE_2 \overset{22}{\triangleright} \mathtt{f}\ \mathtt{true} : \mathtt{bool}}
  }{\underbrace{\{\mathtt{f} \mapsto \forall\alpha.\alpha \to \alpha\}}_{TE_2} \overset{2}{\triangleright} (\mathtt{f}\ (\lambda\mathtt{y}.\mathtt{3}))\ (\mathtt{f}\ \mathtt{true}) : \mathtt{int}}
}{\emptyset \overset{\epsilon}{\triangleright} \mathtt{let}\ \mathtt{f}\ \mathtt{=}\ (\lambda\mathtt{z}.\lambda\mathtt{x}.\mathtt{x})\ \mathtt{5}\ \mathtt{in}\ (\mathtt{f}\ (\lambda\mathtt{y}.\mathtt{3}))\ (\mathtt{f}\ \mathtt{true}) : \mathtt{int}}
$$

Figure 26: Static tree for `let f = (λz.λx.x) 5 in (f (λy.3))(f true)`.

$$\frac{}{E, L \vdash c \longrightarrow c} \tag{CONST}$$

$$\frac{E(x) = v}{E, L \vdash x \longrightarrow v} \tag{VAR}$$

$$\frac{}{E, L \vdash \lambda x.e \longrightarrow [x, e, E, L]} \tag{ABS}$$

$$\frac{E, L1 \vdash e_1 \longrightarrow [x, e', E', L'] \quad E, L2 \vdash e_2 \longrightarrow v' \quad E' \pm \{x \mapsto v'\}, L'1 \vdash e' \longrightarrow v}{E, L \vdash e_1 e_2 \longrightarrow v} \tag{APP}$$

$$\frac{E, L1 \vdash e_1 \longrightarrow v' \quad E \pm \{x \mapsto v'\}, L2 \vdash e_2 \longrightarrow v}{E, L \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \longrightarrow v} \tag{LET}$$

Figure 27: Labeled dynamic semantics.

it in stages to a relation between values and type schemes and between value and type environments.

Definition $v \models \tau$ for monotypes τ if

- $v = c$ for some constant c, $\tau = b$ for some base type b, and $\text{TypeOf}(c) = b$; or
- $v = [x, e, E, L]$, $\tau = \mu_1 \to \mu_2$ for some monotypes μ_1,μ_2, and for all v_1,v_2, if $\models v_1 : \mu_1$ and $E \pm \{x \mapsto v_1\}, L1 \vdash e \longrightarrow v_2$ then $\models v_2 : \mu_2$.

Moreover:

- $\models v : \tau$ for arbitrary types τ if $\models v : S\tau$ for every total, ground substitution S;
- $\models v : \sigma$ for type schemes $\sigma = \forall\alpha_1 \ldots \forall\alpha_n.\tau$ if $\models v : \tau$;
- $\models E : TE$ for type environments TE such that $\text{tyvars}(TE) = \emptyset$ if Dom E = Dom TE and $\models E(x) : TE(x)$ for all $x \in$ Dom E;
- $\models E : TE$ for arbitrary type environments TE if $\models E : S(TE)$ for every total, ground substitution S.

Tofte [110] proves that if $\models E : TE$ and $TE \overset{L}{\triangleright} e : \tau$ and $E, L \vdash e \longrightarrow v$ then $\models v : \tau$. In particular, if e is a closed expression and has a type, then the evaluation of e is well-behaved, in the sense that it never leads to an attempt to reference an undefined variable or apply a non-closure value. This property is the essence of ML's "type-safety."

$$E_b \pm \{\mathtt{y} \mapsto \mathtt{true}\}, 2121 \vdash \mathtt{3} \longrightarrow \mathtt{3}$$

$$E_a \pm \{\mathtt{x} \mapsto \mathtt{true}\}, 1111 \vdash \mathtt{x} \longrightarrow \mathtt{true}$$

$$E_b, 222 \vdash \mathtt{true} \longrightarrow \mathtt{true}$$

$$E_b, 221 \vdash \mathtt{f} \longrightarrow [\mathtt{x}, \mathtt{x}, E_a, 111]$$

$$E_b, 22 \vdash \mathtt{f\ true} \longrightarrow \mathtt{true}$$

$$E_a \pm \{\mathtt{x} \mapsto [\mathtt{y}, \mathtt{3}, E_b, 212]\}, 1111 \vdash \mathtt{x} \longrightarrow [\mathtt{y}, \mathtt{3}, E_b, 212]$$

$$E_b, 212 \vdash \lambda\mathtt{y.3} \longrightarrow [\mathtt{y}, \mathtt{3}, E_b, 212]$$

$$E_b, 211 \vdash \mathtt{f} \longrightarrow [\mathtt{x}, \mathtt{x}, E_a, 111]$$

$$E_b, 21 \vdash \mathtt{f\ (}\lambda\mathtt{y.3)} \longrightarrow [\mathtt{y}, \mathtt{3}, E_b, 212]$$

$$\underbrace{\{\mathtt{f} \mapsto [\mathtt{x}, \mathtt{x}, E_a, 111]\}}_{E_b}, 2 \vdash \mathtt{(f\ (}\lambda\mathtt{y.3))(f\ true)} \longrightarrow \mathtt{3}$$

$$\underbrace{\{\mathtt{z} \mapsto \mathtt{5}\}}_{E_a}, 111 \vdash \lambda\mathtt{x.x} \longrightarrow [\mathtt{x}, \mathtt{x}, E_a, 111]$$

$$\emptyset, 11 \vdash \lambda\mathtt{z.}\lambda\mathtt{x.x} \longrightarrow [\mathtt{z}, \lambda\mathtt{x.x}, \emptyset, 11] \qquad \emptyset, 12 \vdash \mathtt{5} \longrightarrow \mathtt{5}$$

$$\emptyset, 1 \vdash \mathtt{(}\lambda\mathtt{z.}\lambda\mathtt{x.x)\ 5} \longrightarrow [\mathtt{x}, \mathtt{x}, E_a, 111]$$

$$\emptyset, \epsilon \vdash \mathtt{let\ f\ =\ (}\lambda\mathtt{z.}\lambda\mathtt{x.x)\ 5\ in\ (f\ (}\lambda\mathtt{y.3))(f\ true)} \longrightarrow \mathtt{3}$$

Figure 28: Dynamic tree for `let f = (λz.λx.x) 5 in (f (λy.3))(f true)`.

7.2 Defining Dynamic Types

Suppose we are given static and dynamic trees for a closed expression, and that $E, L \vdash e \longrightarrow v$ is a node in the dynamic tree. In order to print or compute safely with values from E at debug time, we must provide a dynamic type environment TE such that $\models E : TE$. The obvious candidate for TE is the principal type environment TE_L, but $\models E : TE_L$ does not hold in general. The problem is that principal type environments do not reflect the way in which bound type variables in the type of a `let`-bound program variable are instantiated when the program variable is mentioned. Obviously, these instantiated types cannot appear in the static tree, since a single static node may correspond to multiple dynamic nodes carrying values of different types. For example, the two nodes evaluating `x` in Figure 28 refer to the single node labeled 1111 in Figure 26, but neither $\models [\mathtt{y}, \mathtt{3}, E_b, 212] : \alpha$ nor $\models \mathtt{true} : \alpha$ holds.

7.2.1 Extended Static Trees

One way to "make a place" for these instantiated types is to substitute the defining expression for each `let`-bound variable at every mention of the variable, forming an *extended expression*. For an expression e, the corresponding extended expression is e^*. The extended expression corresponding to our running example is

```
let f = (λz.λx.x) 5 in (((λz.λx.x) 5)(λy.3))(((λz.λx.x) 5) true)
```

Note that e^* includes the original `let` expression, which is typed in the normal way, even though the `let`-bound variable is not used.

The *extended static tree* for e is the proof tree generated for e^* using the ordinary static semantics of Figure 25. The resulting tree for our example is shown in Figure 29.

Expressions e and e^* evaluate to the same value. In e^*, however, each replication of the definition of a `let`-bound variable can be typed differently. These distinct typings filter down to the subexpressions of the definition, including abstractions. For example, in Figure 29, the λ-bound variable `x`, which is given principal type α at node 1111 inside the definition of `f`, is given a different concrete type at each replicated usage: $\mathtt{bool} \rightarrow \mathtt{int}$ at node 211111, and `bool` at node 221111.

Because the replicated instances are typed in the context in which they are used, their types reflect all "top-down" unification constraints as well as the "bottom-up" constraints

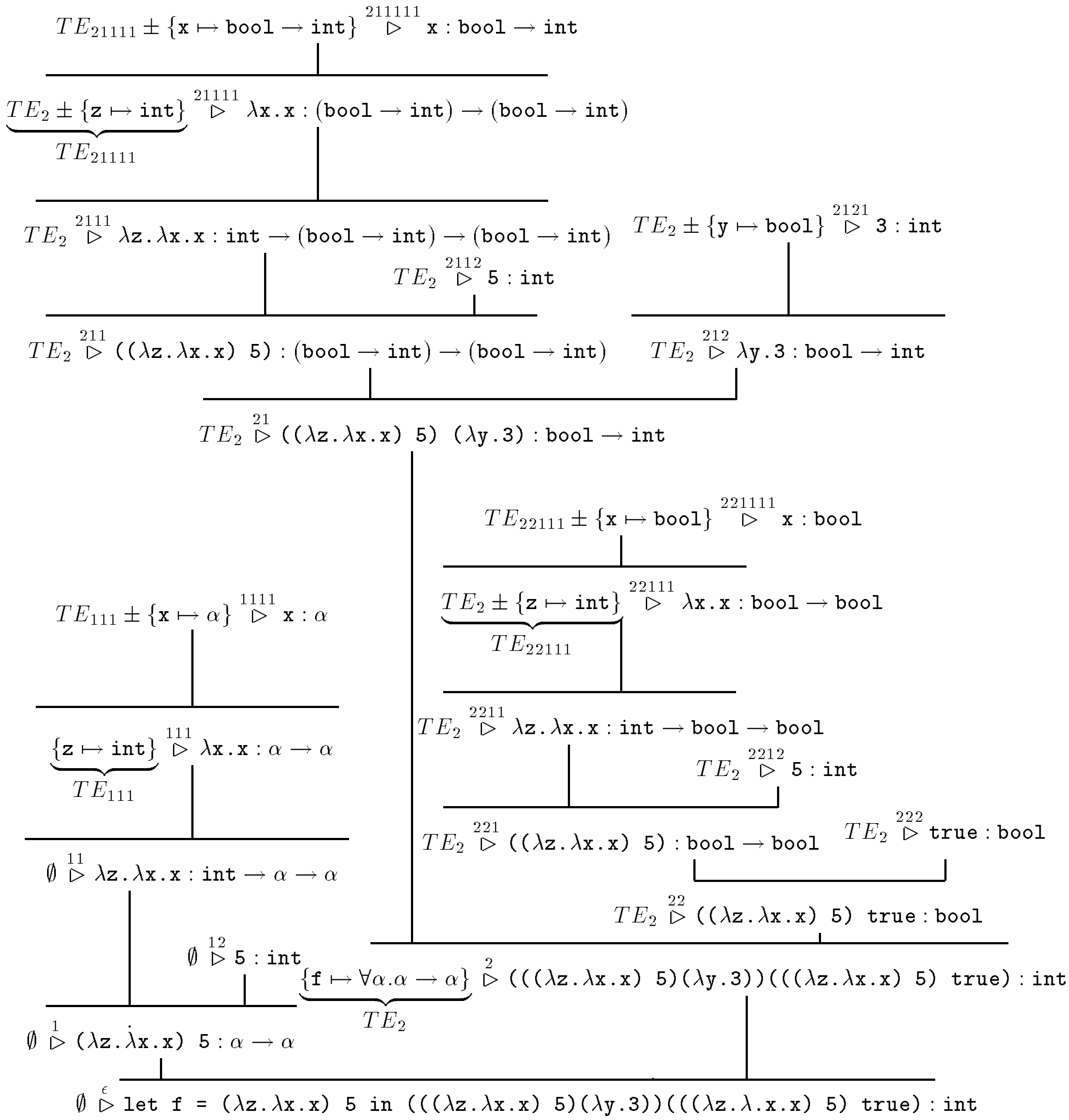

Figure 29: Extended static tree for `let f = (λz.λx.x) 5 in (f (λy.3))(f true)`.

reflected in the principal type. Since the replicated instances differ from the original `let` definition only by being typed in a richer context, the types of the replications will be instances of the principal type. Thus, the type of each replicated instance can be obtained by applying an *instantiating substitution* to the principal type. This substitution is exactly the one required in the definition of the instantiation relation, $>$, as applied by the (VAR) rule in the static semantics of Figure 25.

Thus, building the extended static tree explicitly is wasted effort; we can use the ordinary tree to simulate the lookup of labels in the extended tree using a function `elookup` that maps extended-tree labels to pairs containing an ordinary tree label and the appropriate substitution. We claim that for any extended-tree label L^*, $TE_{L^*} = S(TE_L)$ where $(L, S) =$ `elookup`(L^*).

It is easier to describe `elookup` if we make a minor change to the static semantics of Figure 25 by extending type environment entries for `let`-bound variables to include the root label of the defining LET subtree. This label gives a direct "cross-pointer" from each use of a `let`-bound identifier to its definition. The revised type environment definition and typing rules are

$$TE \quad \in \quad \mathrm{TyEnv} = \mathrm{Var} \to (\mathrm{Type} \cup (\mathrm{TyScheme} \times \mathrm{Label}))$$

$$\frac{TE(x) = (\sigma, L') \quad \sigma > \tau}{TE \overset{L}{\triangleright} x : \tau} \qquad (\mathrm{VAR}_1)$$

$$\frac{TE(x) = \tau}{TE \overset{L}{\triangleright} x : \tau} \qquad (\mathrm{VAR}_2)$$

$$\frac{TE \overset{L1}{\triangleright} e_1 : \tau' \quad TE \pm \{x \mapsto (\mathrm{Clos}_{TE}(\tau'), L1)\} \overset{L2}{\triangleright} e_2 : \tau}{TE \overset{L}{\triangleright} \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau} \qquad (\mathrm{LET})$$

With these revised static semantics, an algorithm for `elookup` is given in Figure 30. The operator $\mathrm{Instsub}(\tau, \tau')$ returns the substitution S' such that $S'(\tau') = \tau$; note that Dom $S' = \{\alpha_1 \ldots, \alpha_n\}$.

```
fun elookup(L*: Label): Label × Subst =
  let fun look(L: Label, L*: Label, S: Subst) =
     if L* = ε then
       (L,S)
     else if L is a VAR node with TE ▷^L x : τ then
       let (∀α1 ...∀αn.τ', L') = TE(x)
           S' = Instsub(τ, τ')
       in look(ε, L' L*, S ∘ S')
       end
     else
       let nL0* = L*
       in look(Ln, L0*, S)
       end
  in look(ε, L*, ∅)
  end
```

Figure 30: Pseudo-code for `elookup`.

7.2.2 Dynamic Types via Extended Labels

Since extended expressions never mention their `let`-bound variables, the proper definition of dynamic type environments for these expressions is not affected by type scheme instantiation. Indeed, we claim that given static and dynamic trees for a closed extended expression e^*, then for any node $E, L \vdash e \longrightarrow v$ in the dynamic tree, $\models E : TE_L$ holds. Of course, we do not want to evaluate extended expressions directly using the dynamic semantics, since doing so involves re-evaluating the definitions of `let`-bound variables repeatedly. For example, executing the expression of Figure 29 evaluates `((λz.λx.x) 5)` three times instead of once.

Instead, we evaluate the original expression e but *simulate* the correct labeling for the evaluation of e^*. For every node $E, L \vdash e_0 \longrightarrow v$ in the dynamic tree for e there is a corresponding node $E^*, L^* \vdash e_0^* \longrightarrow v^*$ in the dynamic tree for e^* with $E = E^*$. Since $\models E^* : TE_{L^*}$ for each node in e^*'s dynamic tree, sound dynamic type environments for the nodes of e's dynamic tree can be computed by tracking the corresponding extended label L^* at each evaluation step.

To do so, we introduce *label schemes* M, analogous with type schemes, which are represented as strings over $\{0, 1, 2\}$. We adopt new dynamic semantic rules, obtained from those of Figure 27 by replacing L by L^* throughout and replacing the VAR and LET rules with

$$\frac{E(x) = [x, e, E', M]}{E, L^* \vdash x \longrightarrow [x, e, E', \mathrm{Inst}(L^*, M)]} \qquad (\mathrm{VAR}_1)$$

$$\frac{E(x) = c}{E, L^* \vdash x \longrightarrow c} \qquad (\mathrm{VAR}_2)$$

$$\frac{\begin{array}{c} E, L^*1 \vdash e_1 \longrightarrow [x, e', E', M] \\ E \pm \{x \mapsto [x, e', E', \mathrm{Gen}(L^*, M)]\}, L^*2 \vdash e_2 \longrightarrow v \end{array}}{E, L^* \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \longrightarrow v} \qquad (\mathrm{LET}_1)$$

$$\frac{E, L^*1 \vdash e_1 \longrightarrow c \quad E \pm \{x \mapsto c\}, L^*2 \vdash e_2 \longrightarrow v}{E, L^* \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \longrightarrow v} \qquad (\mathrm{LET}_2)$$

Figure 31: Extended-label dynamic semantics.

two variants, as shown in Figure 31. The revised definition for closure objects is

$$[x, e, E, M] \quad \in \quad \mathrm{Clos} = \mathrm{Var} \times \mathrm{Exp} \times \mathrm{ValEnv} \times \mathrm{LabelScheme}.$$

The new rules perform "generalization" and "instantiation" for label schemes at dynamic nodes corresponding to the static nodes where type scheme generalization and instantiation takes place. The extended label of a closure built inside a `let` definition is generalized to a label scheme when the closure value is added to the value environment of the LET node's right child. A label is generalized by stripping off the common prefix of the label and the LET node's own label, leaving the relative path from LET node to label node, and then prefixing with a 0. If a VAR node evaluates to a closure with a label scheme, the scheme is instantiated to a label by stripping of the leading 0 and prefixing with the VAR node's label, thus forming a complete path from root to instance. More precisely

$$\begin{aligned} \mathrm{Gen}(L^*, M) &= \begin{cases} 0M' & \text{if } M = L^*1M' \text{ for some } M' \\ M & \text{otherwise} \end{cases} \\ \mathrm{Inst}(L^*, M) &= \begin{cases} L^*M' & \text{if } M = 0M' \\ M & \text{otherwise} \end{cases} \end{aligned}$$

Figure 32 shows the dynamic tree for the running example with extended labels.

The dynamic type environment at a dynamic node with extended label L^* is TE_{L^*}. Given extended static and dynamic trees for an expression, for each node $E, L^* \vdash e \longrightarrow v$

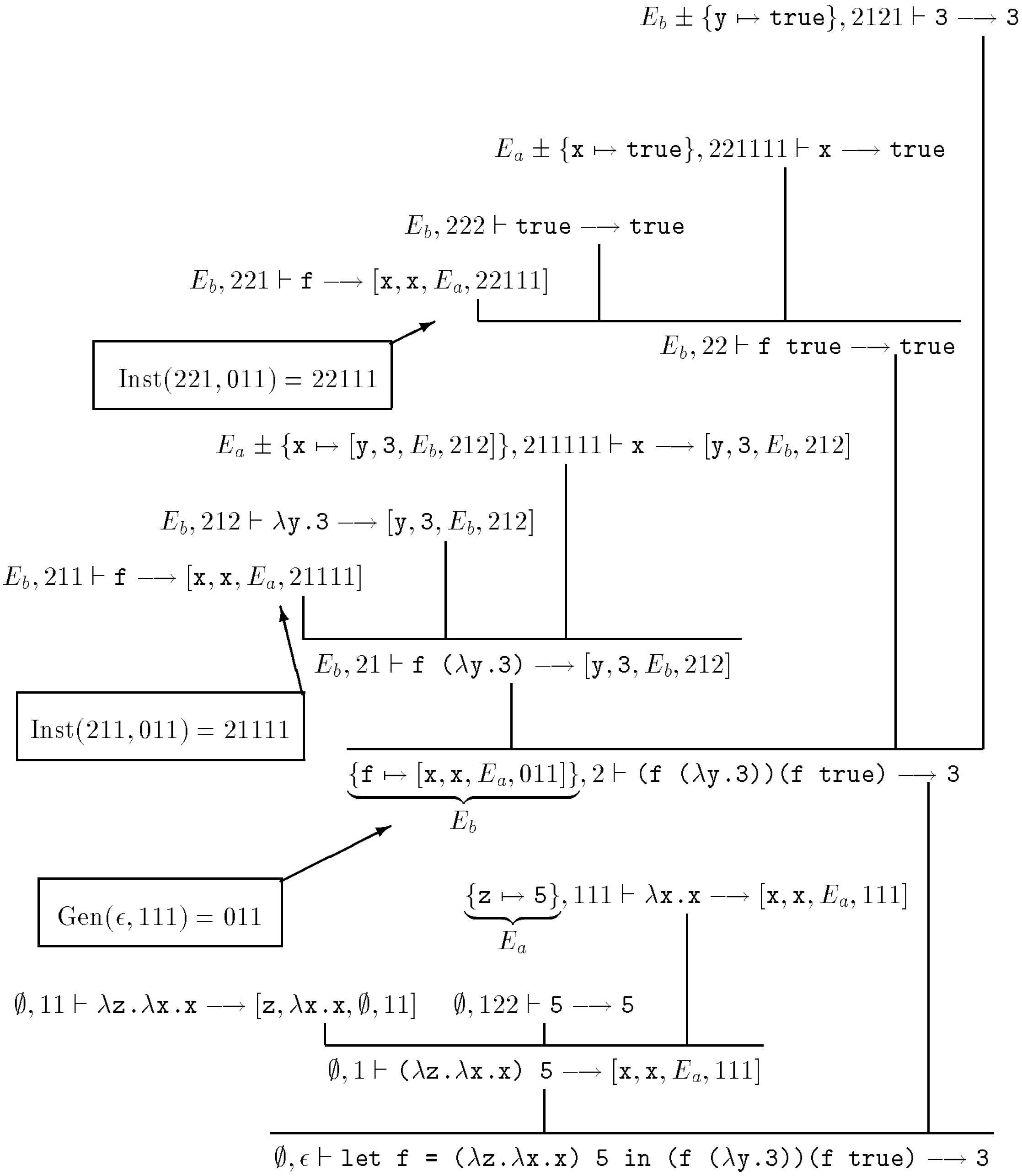

Figure 32: Dynamic tree for `let f = (λz.λx.x) 5 in (f (λy.3))(f true)` with extended labels.

in the static tree, $\models E : TE_{L^*}$. This claim implies that we can safely use values from E in printing and expression evaluation if we give them types from TE_{L^*}.

Dynamic types for closure values may include free type variables. These can arise during the execution of `let` definitions, e.g., for the value of `g` in `let f = (λg.g)(λx.x) in ...`, or when an argument goes unused, e.g., `x` in `(λx.3)(λy.y)`. When using such values, we can safely generalize on the free variables in their types, since the definition of $\models$ says that $\models v : \tau$ implies $\models v : \forall\alpha_1 \ldots \forall\alpha_n.\tau$, where $\{\alpha_1, \ldots, \alpha_n\} = \text{tyvars}(\tau)$, independent of any particular type environment.

Dynamic types are not actually generated dynamically; they are all present in the extended static tree, which can always be generated in finite time from the original tree. This fact would remain true even if we added recursive functions to EXP, since a given mention of a `let`-bound variable always receives a unique, statically determined type.

7.2.3 Implementation Considerations

Applying the machinery of sites and events developed in Chapter 3 to EXP produces straightforward correspondences between static tree nodes and sites and between dynamic tree nodes and events; (non-extended) labels point to the site corresponding to an event, acting just like the site numbers in event records. In particular, right children of dynamic APP nodes correspond to function entry events, right children of LET nodes correspond to value binding events, and APP nodes correspond to application events. The value environment at an event is simply E for the corresponding dynamic node; the type environment is TE_{L^*} for that node. To interpret the dynamic tree operationally, it is helpful to think of events as occurring just *before* the calculation rooted at the node is begun, but this interpretation doesn't affect the definition of the environments.

Assume that we keep the static tree available, e.g., in the form of a type-annotated abstract syntax tree, and that site descriptions point directly to the relevant node in this tree. We could add instrumentation code to manipulate extended versions of these pointers when defining or mentioning `let`-bound variables, just as the Gen and Inst operators in Figure 31 manipulate label schemes at LET and VAR nodes. Unfortunately, this approach might add substantial overhead to debugger execution, since the debugger would not otherwise associate events with references to `let`-bound variables.

$$\frac{}{E, L, S \vdash c \longrightarrow c} \qquad \text{(CONST)}$$

$$\frac{E(x) = v}{E, L, S \vdash x \longrightarrow v} \qquad \text{(VAR)}$$

$$\frac{}{E, L, S \vdash \lambda x.e \longrightarrow [x, e, E, L, S]} \qquad \text{(ABS)}$$

$$\frac{\begin{array}{c} E, L1, S \vdash e_1 \longrightarrow [x, e', E', L', S'] \\ E, L2, S \vdash e_2 \longrightarrow v' \\ E' \pm \{x \mapsto v'\}, L'1, \mathrm{Usubst(L, S, L', S')} \vdash e' \longrightarrow v \end{array}}{E, L, S \vdash e_1 e_2 \longrightarrow v} \qquad \text{(APP)}$$

$$\frac{E, L1, S \vdash e_1 \longrightarrow v' \quad E \pm \{x \mapsto v'\}, L2, S \vdash e_e \longrightarrow v}{E, L, S \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \longrightarrow v} \qquad \text{(LET)}$$

Figure 33: Implemented dynamic semantics. The Usubst operator is defined in the text.

7.3 Dynamic Types via Argument Typing

Our debuggers implement an alternative approach to dynamic type reconstruction that does not require operational steps for generalizing and instantiating the types of `let`-bound variables. This approach is based on the observation that at any function application, the dynamic types of the actual argument and the formal parameter must be the same [9]. By comparing the dynamic type of the argument with the static type of the parameter, we can recover the substitution that instantiates the bound type variables in the function's type. The substitution should be the same as that calculated by `elookup`.

7.3.1 Dynamic Semantics with Substitutions

To describe this idea, we introduce a new version of the dynamic semantics that includes substitutions as explicit elements in dynamic nodes and in closures, which now have contents

$$[x, e, E, L, S] \;\; \in \;\; \mathrm{Clos} = \mathrm{Var} \times \mathrm{Exp} \times \mathrm{Env} \times \mathrm{Label} \times \mathrm{Subst}$$

The resulting dynamic semantics is given in Figure 33. For this semantics, the dynamic type environment for a dynamic node with label L and substitution S is $S(TE_L)$. The operator $\mathrm{Usubst}(L, S, L', S')$ used in (APP) is defined as follows. Let $\tau \to \mu = \mathrm{Type}_{L'}$, so that τ is the static type of x in the closure being applied. Further, let $\tau' = \mathrm{Type}_{L2}$, the static type of the actual argument. Then $S(\tau')$ is the dynamic type of the actual argument, and hence of x within the closure body. Finally, let $S_i = \mathrm{Instsub}(S(\tau'), \tau)$, the substitution such that $S_i(\tau) = S(\tau')$. Then Usubst returns $S' \circ S_i$.

Dynamic trees produced by this semantics are isomorphic to those produced by the extended-label semantics of the previous section. For the running example, the dynamic tree is the same as in Figure 28, with empty substitutions at almost every node and closure; the only non-empty substitutions are for the node evaluating `x` to $[\mathtt{y}, \mathtt{3}, E_b, 212]$, which has the substitution $\{(\mathtt{bool} \rightarrow \mathtt{int})/\alpha\}$, and the node evaluating `x` to `true`, which has $\{\mathtt{bool}/\alpha\}$.

We conjecture that for any node $E, L, S \vdash e \longrightarrow v$ generated by this semantics, whose corresponding node under the extended-label semantics is $E, L^* \vdash e \longrightarrow v$, we have $S(TE_L) = TE_{L^*}$ and therefore $\models E : S(TE_L)$. The challenge of proving this conjecture lies in showing that the definition of dynamic type environments is not affected by delaying application of the instantiation substitution for a closure-valued `let`-bound variable from the point where the variable is mentioned to the point where the closure is applied.

7.3.2 Lazy Reconstruction of Substitutions

Since all type instantiations in this scheme occur at APP nodes, it is possible to maintain runtime type information during debugger-controlled execution by augmenting the code executed at function application sites. In fact, the necessary information can be reconstructed lazily, by performing the instantiations required for a particular variable only if the user references it. To perform lazy reconstruction, the debugger must be able to trace the chain of application events that bind variables in scope at the current event; this information is already provided by the binding pointers used to reconstruct value environments (see Section 4.2).

To describe lazy reconstruction, we use the original dynamic semantics of Figure 27, marking each node with a *dynamic label*, D, analogous to the static-tree labels already encountered: a label is a string over {1,2,3} representing the path from the dynamic-tree root to the labeled node. If $E, L \overset{D}{\vdash} e \longrightarrow v$, we denote L by L_D, define B_D to be the dynamic label of the right child of the statically enclosing APP node, and let P_D be the dynamic label of D's parent. Figure 34 shows the lazy algorithm `getSubst` for computing S at node D; the function returns only that part of S relevant to an initial type τ. The

```
fun getSubst(τ: Type,D: DynamicLabel) : Subst =
   let fun loop(τ: Type,D: DynamicLabel,S: Subst) =
           if tyvars(τ) = ∅ or B_D = ε then
             S
           else let D' = B_D
                    L1 = L_D'
                    τ' → μ = Type_L
                in if tyvars(τ) ∩ tyvars(S τ') = ∅ then
                     loop(τ,D',S)
                   else let D'' = P_D'
                            L' = L_D''
                            τ'' = Type_L'2
                            S' = getSubst(τ'',D'')
                            S'' = Instsub(S' τ'',S τ')
                        in loop(S'' τ,D',S'' ∘ S)
                        end
                end
   in loop(τ,D,∅)
   end
```

Figure 34: Pseudo-code for getSubst.

correct dynamic type for an identifier x at node D is given by

```
let τ = TE_{L_D}(x)
    S = getSubst(τ,D)
in S τ
end
```

getSubst works backwards from D to the root of the tree, stopping at the each dynamic site that λ-binds a variable whose scope includes D; this process is exactly analogous to the event binding chain traversal of Section 4.2. At each such site, getSubst checks whether the static type of the variable being bound contains any type variables in common with τ. If so, it calculates the substitution obtained by instantiating the static type of the bound variable with the dynamic type of its actual argument, which is computed using a recursive call to getSubst. The algorithm terminates when all type variables in τ have been instantiated or when no enclosing λ-bindings remain.

In the worst case, lazily reconstructing the dynamic type environment at node D requires $O(d)$ time, where d is the depth of D in the dynamic tree. If we add recursion to our language, this bound might be unsatisfactorily large; it is possible that additional pointers could be added to short-circuit the lazy computation.

7.3.3 Using Runtime Unification

SML/NJ currently discards the contents of the static type tree after type checking is completed, but it keeps the types associated with both `let`- and λ-bound variables. Therefore, the debugger actually uses lazy reconstruction based on a slightly weaker version of the semantics shown in Figure 33. In this version, labels are not maintained and the Usubst operator is defined differently: τ, the static type for x, is looked up in the compiler's symbol table, and τ', the static type of the actual argument e_2, is recomputed on the fly using the type-assignment algorithm. This recomputation does not take the surrounding context of e_2 into account, so it sometimes yields a less precise type than the true τ'. As an illustration, in our running example (Figure 28), when `f` is applied to λ`y.3` the type of x will be reported as $\alpha \rightarrow$ `int` rather than `bool` $\rightarrow$ `int`.[1]

We conjecture that this problem affects only polymorphic functions, and that $\models E : S(TE_L)$ remains true with this definition of Usubst. However, this formulation of the semantics remains unpleasantly ad-hoc; future versions of the debugger may cooperate with the compiler's type checker to keep the full static tree available, or to regenerate it on the fly.

[1]A technical consequence is that Instsub(τ_1, τ_2) must cope with the possibility that τ_1 is not an instance of τ_2 (though they must be unifiable). In such cases, the substitution returned by Instsub simply fails to include a mapping for some of the type variables in τ_2.

Chapter 8

Sequential Debugger Performance

To assess the practicality of **smld** and **smld-t**, we measured their performance on real programs. We focus on the costs of controlled execution, i.e., forward execution under debugger control, but without actual queries. We compare controlled execution speed and live-data space requirements for several versions of **smld** and **smld-t**, and also for other compilation disciplines intended to simulate hypothetical debuggers that use different technology. We also compare the speed and overall memory requirements of the different debugger versions on various interaction scripts intended to represent typical debugging sessions that do include queries. Finally, we measure **smld-t**'s query response times. These measurements enable us to judge that instrumentation and time-travel can be implemented efficiently enough to be practical.

8.1 Benchmark Programs

Table 1 summarizes the benchmark programs. Each program solves a real, if perhaps only pedagogical, problem. RedBlack, Lexer, and UnionFind have been embedded in synthetic drivers for benchmarking purposes. Slightly different versions of Life, Simple, and Knuth–Bendix have been benchmarked extensively for other purposes [11].

Basic statistics on normal, uninstrumented execution of each benchmark are listed in Table 2. Statistics on the static and dynamic distribution of events in the instrumented versions of the benchmarks are shown in Table 3.

Key	Name	Description
l	Life	A game of Life [97], running 50 generations of a glider gun.
s	Simple	A spherical fluid-dynamics program, developed as a "realistic" FORTRAN benchmark [34], translated into ID [44], and then translated into Standard ML by Lal George.
k	Knuth–Bendix	An implementation of the Knuth–Bendix completion algorithm, implemented by Gerard Huet, processing some axioms of geometry.
r	RedBlack	Builds a dictionary using red-black trees from an input of 300,000 integers.
x	Lexer	A machine-generated lexer for Standard ML that counts tokens, run on a 5760-line input file.
u	UnionFind	Performs 5000 union and find operations with path compression on a database of strings.
t	TermRewriter	A term-rewriter performing symbolic differentiation on polynomials, written in Scheme by Samuel Kamin [70] and translated into Standard ML by Andrew Appel.

Table 1: The benchmark programs.

Key	Name	Source Lines	Compile Time	Code Size	Data Size	User Time	GC Time	System Time
l	Life	121	31.6s	18k	3k	22.5s	0.05s	0.18s
s	Simple	764	67.7	93	2438	43.0	3.28	0.07
k	Knuth–Bendix	497	25.7	40	1479	12.7	1.12	0.06
r	RedBlack	65	6.1	5	5266	31.1	5.27	1.45
x	Lexer	477	25.2	21	7	1.6	0.00	0.04
u	UnionFind	145	5.5	12	220	0.8	0.00	0.04
t	TermRewriter	119	5.9	13	11	1.7	0.01	0.01

Table 2: Benchmark normal execution characteristics. Source Lines is the number of non-blank, non-comment lines in the original program. Compile Time is the elapsed time required to compile the program. Code Size is the number of kilobytes in the compiled program. Data Size is the maximum number of kilobytes of live data observed at any size measurement point (see Section 8.7), counting only data generated by the program while it executes. User Time is the CPU time spent executing the program, excluding garbage-collection and operating-system time. GC Time is CPU time spent garbage collecting, and System Time is CPU time spent in the operating system.

Key	Name	Sites/ Line	Events Executed	Events/ Instruction	Store Events	I/O Events
l	Life	3.4	13.2×10^6	.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

Table 3: Benchmark event characteristics. Sites/Line is the average number of instrumentation sites per source line (after coalescing and linearizing). Events Executed is the total for a simple execution of the program. Events/Instruction is the average number of events executed for each machine instruction executed by the *uninstrumented* program; it gives a comparative measure of dynamic event density. Store Events and I/O Events give the dynamic percentages of executed events that alter the mutable store and perform stream I/O, respectively.

8.2 Hypothetical Debuggers

There are no other ML debuggers available for direct comparison with **smld** and **smld-t**, but we can estimate the performance of other, hypothetical, debugging approaches, including machine-based debugging and interpretation.

Any realistic machine-based debugger must inhibit certain compiler optimizations to support expected behavior (see Section 2.3). Without designing the machine-based debugger and associated compiler in detail, we cannot be certain which optimizations must be prevented; in general, there is a range of possible debugger designs, involving different tradeoffs between the strength of optimization permitted and the complexity of debugger implementation. We consider two hypothetical machine-based debuggers, one at either end of this range. The *simple* debugger guarantees expected behavior only when its associated compiler disables all optimizations; most commercial debuggers are simple. The *clever* debugger takes advantage of close integration with the compiler to provide expected behavior even for heavily optimized code, provided it can compensate for the optimizations without adding runtime instrumentation; these debuggers exist only on paper [121].

To estimate the performance of these hypothetical debuggers using SML/NJ, we choose sets of compilation options, or disciplines, that generate code with appropriate levels of optimization. These disciplines can only approximate the behavior of the compilers that could support machine-based debuggers. As noted in Section 2.1.4, certain optimizations cannot be inhibited without altering the basic structure of the SML/NJ compiler; in particular, CPS conversion implicitly performs dead-variable elimination, which even a clever debugger would need to disable.

To obtain a first estimate of the simple debugger's performance, we turn off all controllable optimizations to generate an unoptimized version. However, turning off all optimization is somewhat misleading, as the CPS conversion process introduces certain inefficiencies that would normally be removed by subsequent optimization passes; even a naive compiler, if it didn't use CPS, could probably produce better code than the unoptimized discipline suggests. Therefore, we also measure performance with a limited set of *cleanup* optimizations intended to restore the code to roughly its pre-conversion efficiency. The true efficiency of code generated to run under a simple machine-based debugger should be somewhere between these two disciplines.

To approximate the performance of the clever debugger, we disable only optimizations that can't be handled even with complicated bookkeeping: η-reduction (including tail-recursion elimination) and load scheduling. This discipline is called *most* optimizations.

Another approach to debugging is to use an interpreter (see Section 2.4). To measure the performance of this alternative, we ran the benchmarks under SML/NJ's simple λ-language interpreter.

8.3 Scripts

We use three debugging scripts to measure the overall performance of our debuggers, including the cost of queries as well as the overhead of controlled execution. Each script is a sequence of commands representing a single debugging session, which can be fed to the debug monitor. Our scripts are synthetic; they are designed to represent typical but simplified patterns of user behavior. Every script works with every benchmark.

Each script focuses on a single location breakpoint, chosen to occur 5–10 times during program execution, but each script uses the breakpoint in a different way. Script `proceed` sets the breakpoint and executes forward. Each time the breakpoint is encountered, the script changes scope up one level, looks up an undefined variable (twice), shows the last five call frames, and continues executing forward. Script `recede` first executes forward to the end of the program; it then sets the breakpoint and repeatedly executes backward, performing the same queries as `proceed` each time the breakpoint is encountered. Script `tofro` sets the breakpoint and executes forward. Each time the breakpoint is encountered (at time t), the script changes scope up one level, looks up an undefined variable, single-steps forward 5 times and backward 10 times, jumps back to time $t/5$, shows the last two call frames, looks up an undefined variable again, and re-executes forward to t, stopping but not performing queries each time the breakpoint is encountered between $t/5$ and t. Finally, it continues executing forward.

The scripts look up an undefined variable in order to generate maximum work for the lookup algorithm, which must consult every binding in the static scope. However, this search does not exercise the dynamic type reconstruction mechanism. The reconstruction mechanism *is* exercised in a realistic way by showing a call frame, which reports the entry point of the current function, the point from which the function was called, and the names, types and values of the arguments.

All scripts can be executed by **smld-t**, and `proceed` can also be executed by **smld**. To estimate the performance under **smld** of scripts that perform explicit reverse execution, we charge the cost of re-executing from the beginning of the program each time a reverse motion occurs.[1] 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 apply the scripts to the hypothetical machine-based debugging systems of Section 8.2, we assume that query execution is free and measure only the cost of forward and reverse execution; as with **smld**, reverse execution is charged as re-execution from the beginning of the program.

8.4 Timing Model

SML/NJ's use of CPU time can be divided into *user* time spent executing instructions generated by the SML/NJ compiler in Unix user mode, *gc* time spent performing garbage collection, also in Unix user mode, and *system* time spent executing in Unix system mode, typically performing I/O.

For programs that use a fixed amount of live data, gc time is highly elastic, because it depends on the amount of available memory and the particular garbage-collection strategy. Therefore, for comparative purposes it is often better to consider just user times. We

[1]For `tofro`, the ten backward single-steps are charged as only a single re-execution.

measure user times in terms of instruction counts rather than elapsed time, because elapsed times vary by 5 to 20%, apparently due to cache interference effects [11]. This variation makes it difficult to use elapsed times to compare runs with different parameters; instruction counts are more consistent. The counts are obtained by instrumenting the MIPS machine code generated by the compiler. The count is held in a register and an instruction to update it is inserted in the branch delay slot that ends every extended basic block[2] in the code; this instruction adds the number of instructions in the basic block just executed (including the add instruction itself) to the count register. Unfortunately, this technique makes it necessary to disable the compiler's branch-scheduling optimization. We assume that the resulting effects do not alter our comparisons, except for reducing the contrast between unoptimized and "normal" code, which already lacks branch scheduling.

However, garbage collection times must also be considered. If one version of a benchmark uses more live data than another, it is bound to require more gc time, regardless of the garbage collection technology and parameters chosen, and this gc time can be a significant component of the elapsed time. Moreover, for the adaptive checkpointing algorithms used by **smld-t**, there is no fixed memory demand: the debugger uses whatever memory it is given. We would like to measure **smld-t**'s performance over a range of memory sizes; if we measured only user time we would presumably find that more memory invariably allows faster execution, but this correlation might not hold when gc time is included. For these reasons, we often compare total user+gc+system times for different versions of a benchmark, especially when studying the adaptive algorithms, with the caveat that the figures may reflect idiosyncrasies of the current garbage collector.

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 with a modified runtime system supporting instruction counting and other measurement functions. Ordinarily, the benchmark was the only user process running on the system. However, some files were accessed via NFS from other computers on which activity could not be controlled, which introduces non-determinism into the behavior of SML/NJ because of varying NFS buffer sizes, and perturbs our measured times. All reported times represent the minimum of at least three benchmark runs in which the sum of measured user, gc, and system time was at least 97% of total elapsed time.

[2] An extended basic block is an acyclic set of blocks with one entry point and one or more exits.

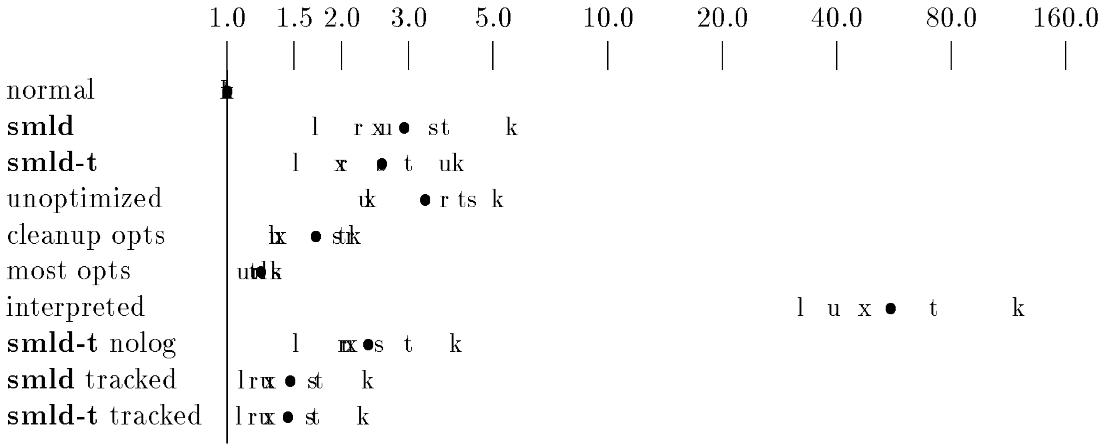

Figure 35: Execution cost (instructions). Relative controlled execution cost under various compilation disciplines described in the text. Cost is the number of user instructions. The scale is logarithmic; the labels indicate the ratio of the cost of each discipline to that of the normal discipline. The dot • is the geometric mean of the benchmarks. The letters indicate the times for the individual benchmarks; see Table 2 for the key. Two of the benchmarks didn't complete within one hour when interpreted, so the geometric mean shown for the interpreted discipline is artificially low.

8.5 Controlled Execution Overhead

Figures 35 and 36 show how the relative execution time of each benchmark varies with compilation discipline. Each benchmark is executed from beginning to end, without query operations in the case of the debugging disciplines, and without checkpointing in **smld-t**. In Figure 35, execution time is the number of instructions executed, excluding garbage collection and system activity. Figure 36 is based on elapsed time including garbage collection and system time; for these experiments, the garbage collector kept the ratio of total available memory to live data fixed at about 5, and there was ample physical memory to maintain this ratio. In addition to the disciplines already described, both figures also show the cost of running **smld-t** with logging disabled (**smld-t** nolog) and two variable-tracking disciplines, described below.

Figure 35 shows that mean user execution time under **smld** is about 3 times slower than fully optimized normal execution, although there is considerable variation among the benchmarks. **Smld-t**'s instrumentation is slightly less expensive; its mean controlled execution

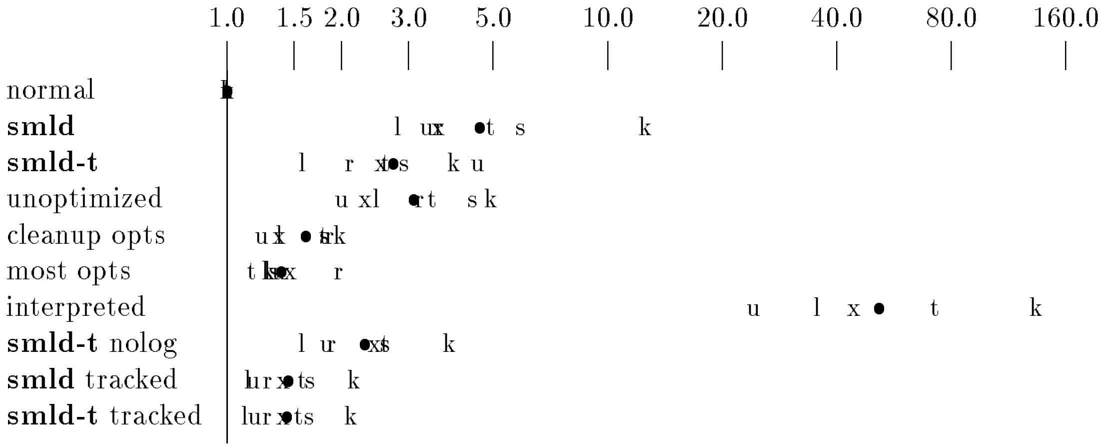

Figure 36: Execution cost (user + gc + system times). Relative controlled execution costs measured as the sum of user, garbage collection, and system times. The key is the same as for Figure 35.

time is about 2.5 times slower than normal execution.

If we include garbage collection time in the comparison (Figure 36), the relative performance of **smld** is roughly 5 times slower than the normal discipline, which is probably due to the overhead of collecting event records (see Section 8.7) and processing the many entries for `lastCaller` on the collector's store list. **Smld-t** without checkpointing, however, remains less than 3 times slower than normal execution when gc time is considered, which shows the effect of the lazy pointer technique of Section 5.2. The behavior of **smld-t** with checkpointing is considered in Section 8.8.

The disciplines representing hypothetical debugging approaches also show significant slowdowns, with or without garbage collection time. Unoptimized code runs about 3 times slower than normal code and somewhat slower than **smld-t**; code with cleanup optimizations is about 60% slower than normal. These figures suggest that the hypothetical simple debugger would have controlled execution speed roughly comparable to **smld-t**. With most optimizations, code slows down by only about 20%, so the hypothetical clever debugger might perform better than instrumentation approaches. The uncompetitive interpreted code runs one to two orders of magnitude slower than any of the compiled approaches.

Individual benchmarks have widely varying behavior. Knuth–Bendix, for example, seems to run much slower under *every* non-normal discipline, perhaps because it has large numbers of small functions, so that increasing per-call overhead and failing to in-line calls have a more significant effect than in the other benchmarks. UnionFind runs particularly slowly under **smld-t**; this anomaly disappears if the overhead of logging updates and I/O is excluded (**smld-t** nolog), which is not surprising since UnionFind has the highest frequencies of store update and I/O events among the benchmarks.

The **smld** tracked and **smld-t** tracked disciplines attempt to measure how much of the slowdown caused by instrumentation is due to disabling dead-variable elimination in the original code, rather than to the cost of executing the instrumentation code itself at each site. These disciplines use a special version of the **smld** and **smld-t** instrumentation that keeps user variables live ("tracks" them) to the site where they would be placed in an event record and generates the `event_`n variables, but does not actually execute code at events. Tracking adds 25–50% to execution times. This measurement provides an upper bound on how much we can improve the performance of an instrumentation-based debugger by streamlining the instrumentation code. It also provides an estimate of the costs faced by the clever debugger in keeping variables live long enough to present expected behavior.

8.6 Code Size and Compilation Time

Figure 37 shows the increase in code size suffered under each of the disciplines defined in Section 8.5. **Smld**'s code is, on average, five times larger than normal code; **smld-t** code size increases by a slightly smaller factor. The various partially optimized disciplines cause code size to increase by 20–200%.

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 an **smld-t** 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 of Chapters 3 and 5 as a true function instead of in-lining it. Experiments on earlier versions of **smld-t** 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

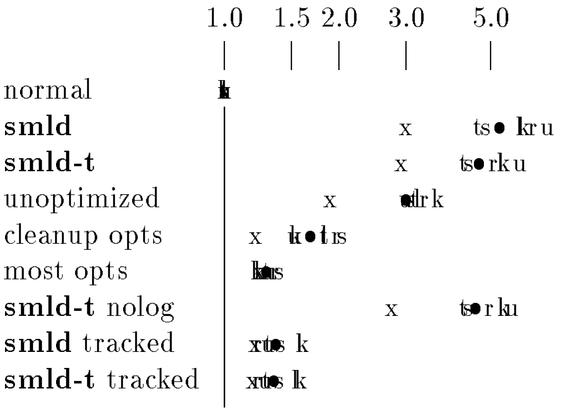

Figure 37: Compiled code size (bytes). Relative compiled code size for different compilation disciplines. A logarithmic scale is used; the labels give the ratio to normal compilation. The geometric mean increase is shown by the dot (•).

λ-language, **smld** and **smld-t** 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 optimizations disabled is another option.

8.7 Live-Data Requirements

No single value can accurately represent how much live data is used by a program, since this amount can vary widely during execution. Although the maximum value may be of particular interest, it cannot be measured without knowing the program point that witnesses it, which cannot usually be determined by inspecting the program text. Therefore, we record entire *profiles* of live-data requirements during execution. We record a profile by measuring the amount of live data at specific points in the program text, chosen because they are executed regularly and frequently. This approach enables us to compare profiles from different runs precisely, which would be impossible if measurements were taken at timer interrupts or other periodic external events.

Measuring the amount of live data requires traversing the entire live-data graph, so it is convenient to collect measurements as a by-product of performing major collections (see Section 2.1.3). Major collections can be forced by a call to the runtime system, which changes the characteristics of subsequent minor collections, but does not affect the accuracy of the live-data measurement. Since the large number of garbage collections increases elapsed time substantially and alters memory behavior, we perform space measurements and time measurements in separate runs.

Figures 38 and 39 show profiles of live data use for complete program executions of each benchmark, under the normal, unoptimized, **smld**, **smld-t**, and **smld-t** nolog disciplines described in Section 8.5. Profiles for the **smld-t** executions exclude checkpoint data.

The benchmarks can be divided into two distinct classes based on their normal execution profiles. Knuth–Bendix, RedBlack, 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).

For the former class of benchmarks, the unoptimized discipline requires a constant factor more live data than the normal version: total live data still increases linearly with execution time, but at a faster rate. The latter class of benchmarks shows more varied behavior under the unoptimized discipline. The space demands of Lexer and TermRewriter increase linearly with execution time, which can be attributed primarily to the lack of tail-recursion elimination under this discipline. For Life, space demand remains constant, though at a higher level than in the normal discipline; for Simple, demand is about the same as in the normal case. Profiles for the other partially optimized disciplines (cleanup optimizations and most optimizations) are not shown; they are broadly similar to the unoptimized discipline, with most optimizations usually requiring the most space of the three.

Smld keeps large amounts of live data in every benchmark; its space demands increase linearly for most benchmarks, including those with fixed-size normal profiles. The extra data consists of event records on or referenced from the current call chain and the dead (but still visible) values pointed to from these records. Thus, even though the **smld** compilation discipline performs tail-recursion elimination, the expected space savings are lost because of explicit storage into event records.

Profiles for **smld-t** and **smld-t** nolog are shown as a further comparative measure of instrumentation overhead; since the profiles do not include checkpoint space, they are not correlated with the amount of memory actually needed to run **smld-t**. Neither of these disciplines build event records. **Smld-t** nolog uses substantially less space than **smld** in most benchmarks; indeed, it seldom uses more than twice as much live data as normal and usually requires less live data than the unoptimized discipline.

When logging is included (**smld-t**) memory use increases dramatically for benchmarks that perform many update events; in UnionFind, **smld-t** actually uses more space than **smld**. However, the **smld-t** measurements are somewhat inflated because they include the size of the global list of changed cells maintained by the mutable store subsystem. This list typically contains large numbers of duplicates and garbage entries, which are removed each time a checkpoint is taken; since the **smld-t** trials measured here took no checkpoints, the list grew much larger than it normally would have.

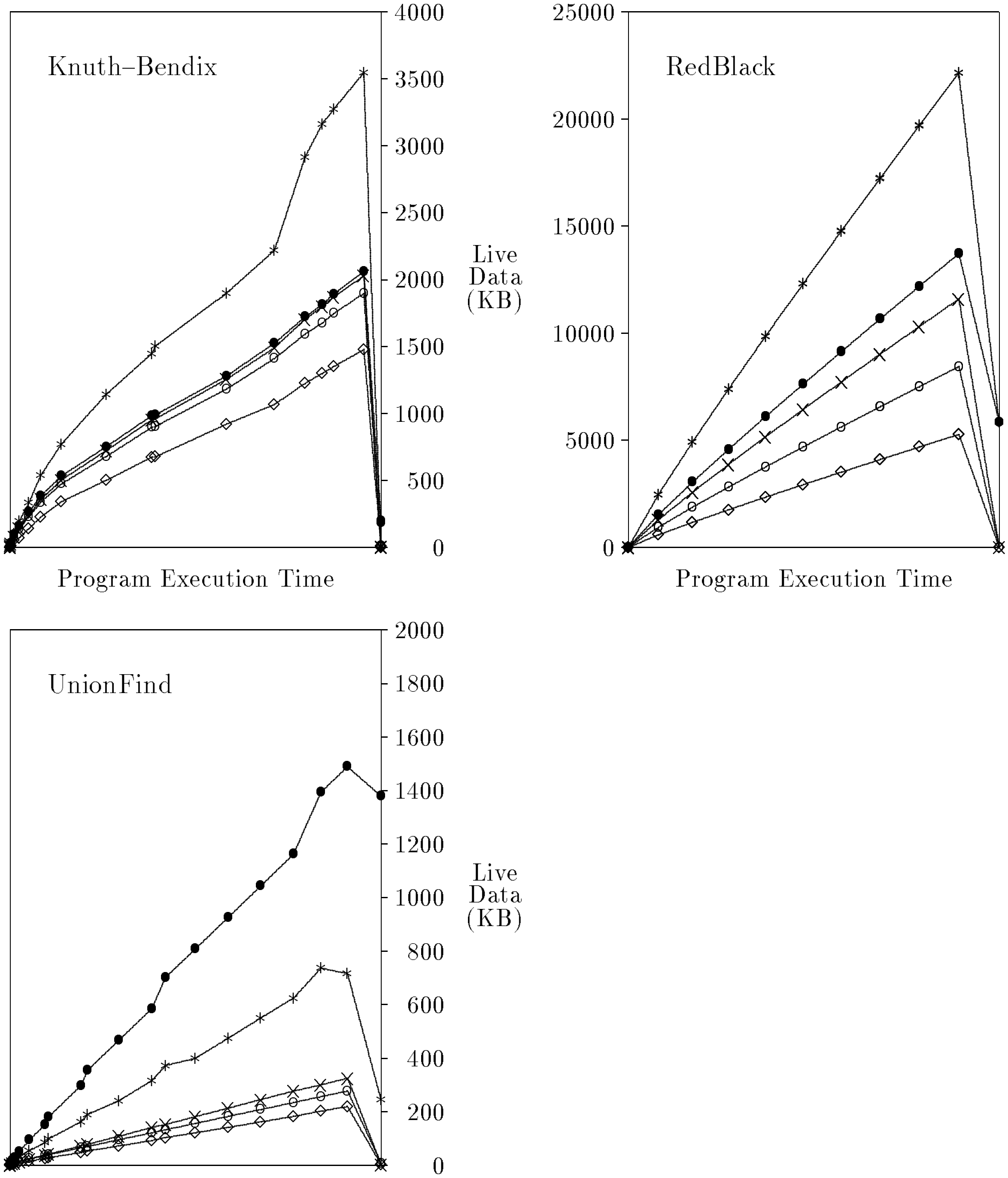

Figure 38: Live data measured for Knuth–Bendix, RedBlack, and UnionFind. Execution disciplines: normal (⋄), unoptimized (×), **smld** (∗), **smld-t** (•), **smld-t** nolog (∘). The points of each profile are connected for graphical clarity; the connecting lines should not be used for interpolation.

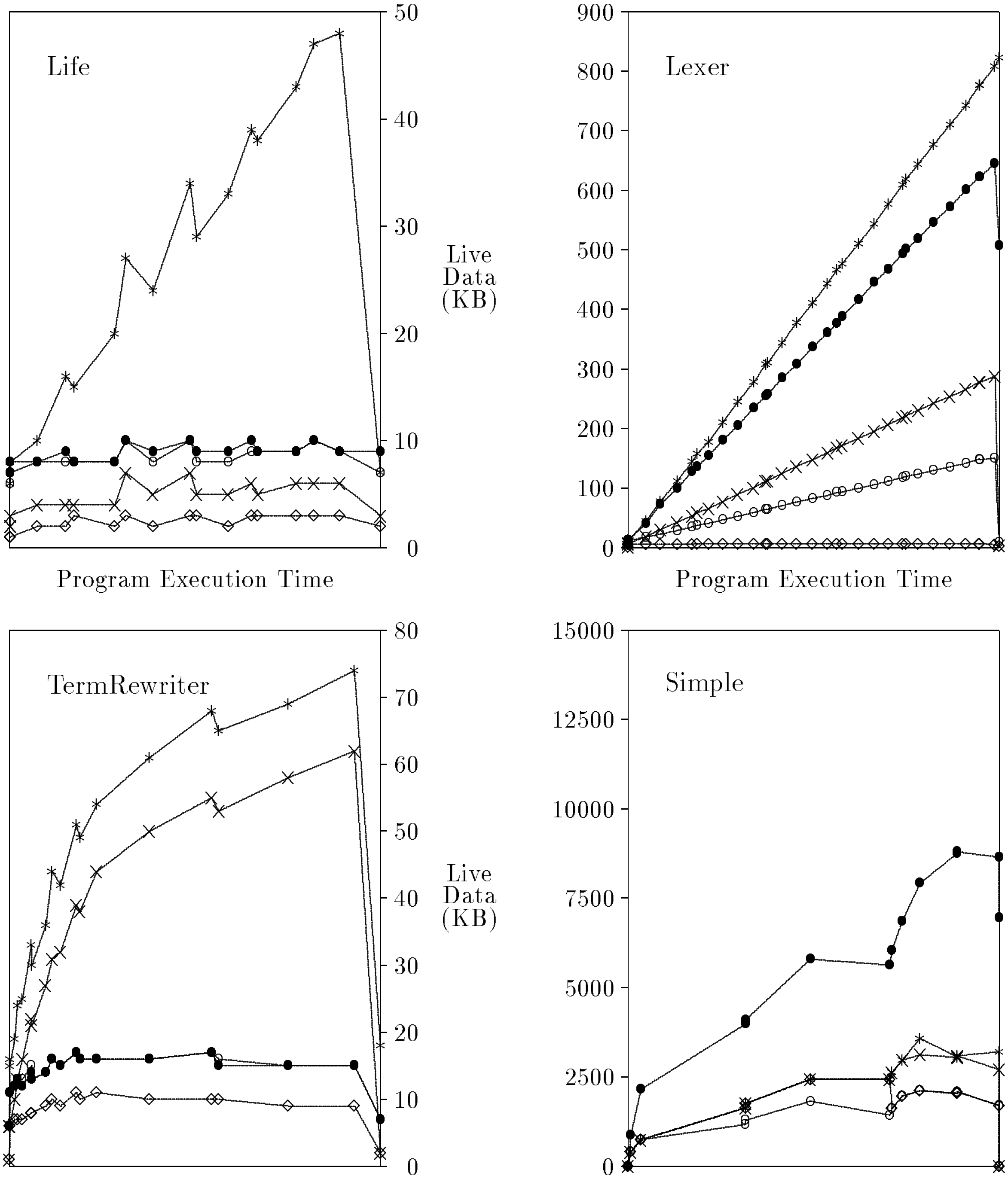

Figure 39: Live data measured for Life, Lexer, TermRewriter, and Simple. Execution disciplines: normal ($\diamond$), unoptimized ($\times$), **smld** ($*$), **smld-t** ($\bullet$), **smld-t** nolog ($\circ$). The points of each profile are connected for graphical clarity; the connecting lines should not be used for interpolation.

8.8 Space vs. Time

This section compares execution time and memory use for compilation disciplines including **smld-t**, **smld**, normal compilation, and various hypothetical debuggers, under the scripts described in Section 8.3. The measurements presented here, unlike those described in previous chapters, include the cost of queries and checkpoints.

Time and space cannot be measured independently. Like any garbage-collected system, SML/NJ offers a tradeoff between execution time and memory use: increasing memory decreases the frequency of collections and hence overall execution time. Moreover, since **smld-t** adapts to the amount of memory available by varying the size of its checkpoint cache, changing the memory limit may have other, complex effects on execution time. Therefore, to compare performance under different disciplines we measure execution time over a range of memory sizes for each discipline, and compare overall space vs. time curves.

These curves typically show 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.

Figures 40 through 46 show space vs. time curves for each benchmark and script under a variety of disciplines. Each curve is constructed from a series of runs; in each run, total system memory (including compiler and runtime data) was held constant and total elapsed user+gc+system time for the script was recorded. The resulting (size,time) pair is plotted as a single symbol. The lines connecting these symbols into curves are intended to improve graphical clarity; they should not be used to interpolate intermediate values. Note that the range of memory sizes shown (along the x-axis) varies from benchmark to benchmark; labels indicate a subset of the values at which measurements were taken. Elapsed times (along the y-axis) are shown on a logarithmic scale. Benchmark runs that didn't complete within one hour were aborted; we generously assume that they would have completed in just over one hour and continue the curves accordingly, using a dashed line. Runs that exceeded the available memory aborted; their symbols are shown above the graph at the appropriate x-coordinate, and they are connected to the curve with a dotted line.

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 is different for each benchmark and is not always the same for all three scripts, but all the optimal values lie in the range 8,000 to 256,000 events; 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.

Each figure includes one curve showing **smld-t**'s behavior for the overall *best interval*, and another for its behavior at a *standard interval* of 64,000 events, a reasonable choice for this set of benchmarks. Both curves were measured with pre-caching disabled, full memoization enabled, and using the cache replacement strategy described in Section 6.7, with $r_0 = 5$ and $f = 0.25$. The preliminary runs for finding the optimal interval (not shown here) used the same parameters, with system memory size fixed at 100MB.

To compare **smld-t**'s performance with other systems, we show curves for fully optimized normal execution, unoptimized execution, and **smld**. For the `recede` and `tofro` benchmarks, these three curves are synthesized as suggested in Section 8.3, by assuming that any reverse execution is accomplished by re-executing again from the beginning to the target point.

Smld-t 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 also varies among scripts: for Life, it varies from 4 to 8.

Smld runs comfortably using only twice as much memory as normal execution in most benchmarks, but in a few cases (RedBlack, Knuth–Bendix) takes as much or more memory than **smld-t**.

If the memory to run **smld-t** comfortably is available, its speed is competitive with the hypothetical debugging approaches described in Section 8.2 for many benchmarks and superior to **smld** for most. A machine-based debugger should give performance somewhere in the band between the normal and unoptimized curves. Even when only forward execution (script `proceed`) is considered, **smld-t**'s execution time is in that band for four of the seven benchmarks, and it runs faster than **smld** in five of the seven. When the cost of reverse execution is considered, as in scripts `tofro` and `recede`, **smld-t**'s execution time lies in or below the machine-based band, and below **smld**, for all benchmarks except UnionFind and Life under `tofro`. **Smld-t** does particularly well for RedBlack and for Life under `recede`; evidently, replay performance depends heavily on the particular pattern of time-travel exhibited by the script.

We also measured **smld-t**'s performance under various non-standard parameter settings, using the overall best checkpointing interval for each benchmark. Enabling pre-caching (see Section 6.7) with $\epsilon = f \times g$ has little effect on most benchmark/script combinations. However, pre-caching significantly improves Life under `tofro`, giving a knee at about half the memory size required by ordinary **smld-t** and running about twice as fast, which suggests that an unusually large percentage of the backward time-travel operations for this case have magnitude less than ϵ events. We have not investigated other values of ϵ.

Separate measurements (not shown here) indicate that most of the benchmark runs do not invoke cache replacement. For the few that do, such as the runs for UnionFind between 9.2MB and 12.2MB, there is no consistent difference between simple LRU cache replacement and the elaborate replacement strategy described in Section 6.7.

Disabling all memoization (see Section 6.3) causes a large increase in execution time. Disabling only memoization of bound values in event records also increases times, but by much less; these results are not shown here. However, memoization takes a good deal of space; moreover, in the current implementation, this space can never be recovered even when memory grows tight. Not surprisingly, therefore, disabling memoization in all or part allows **smld-t** to execute with a smaller memory limit. For Life `proceed` and UnionFind `tofro`, the non-memoized discipline sometimes runs faster when given less memory; this anomaly may be due to excessive expansion of the checkpoint cache when extra memory is available.

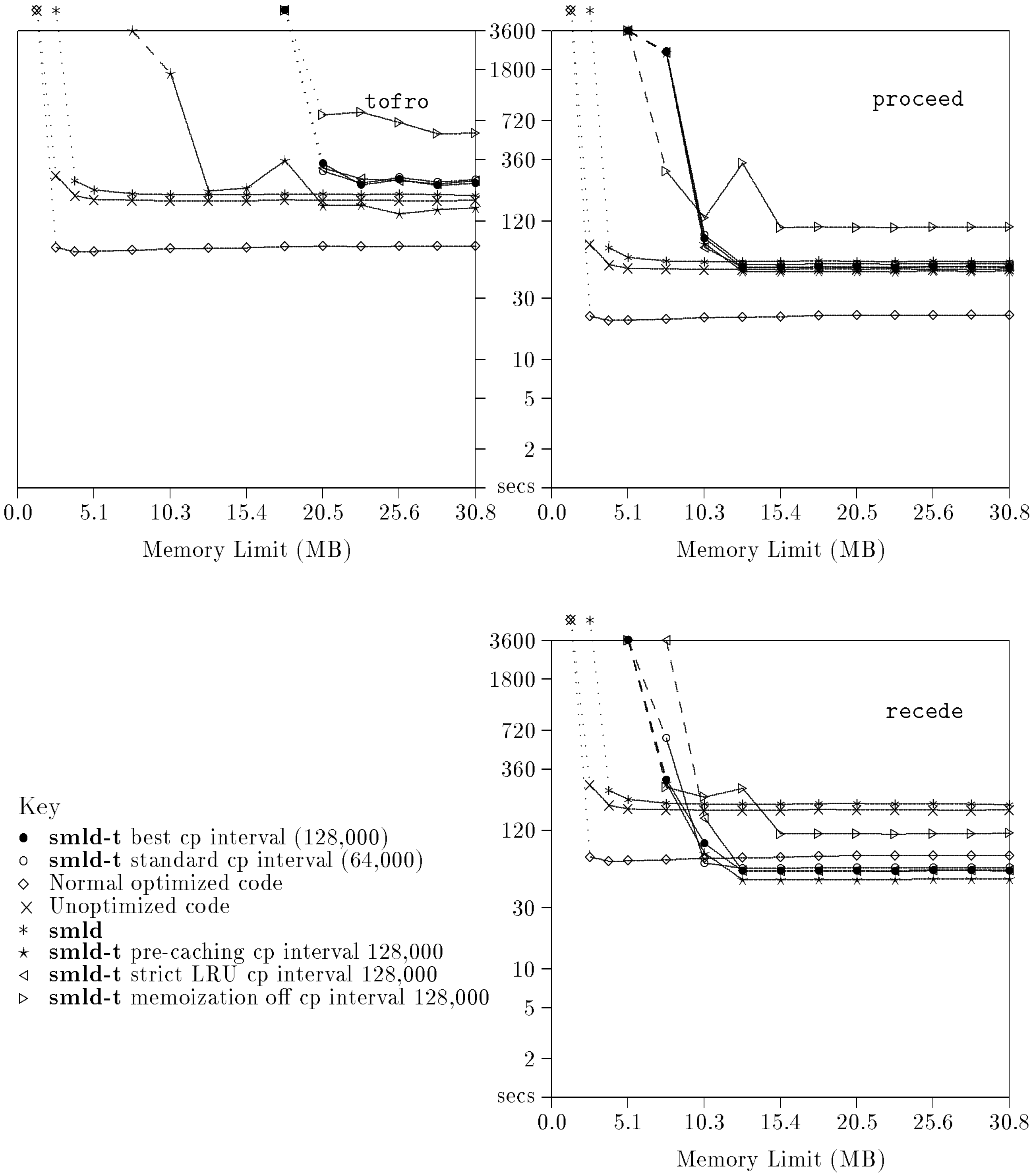

Figure 40: Time vs. space for Life.

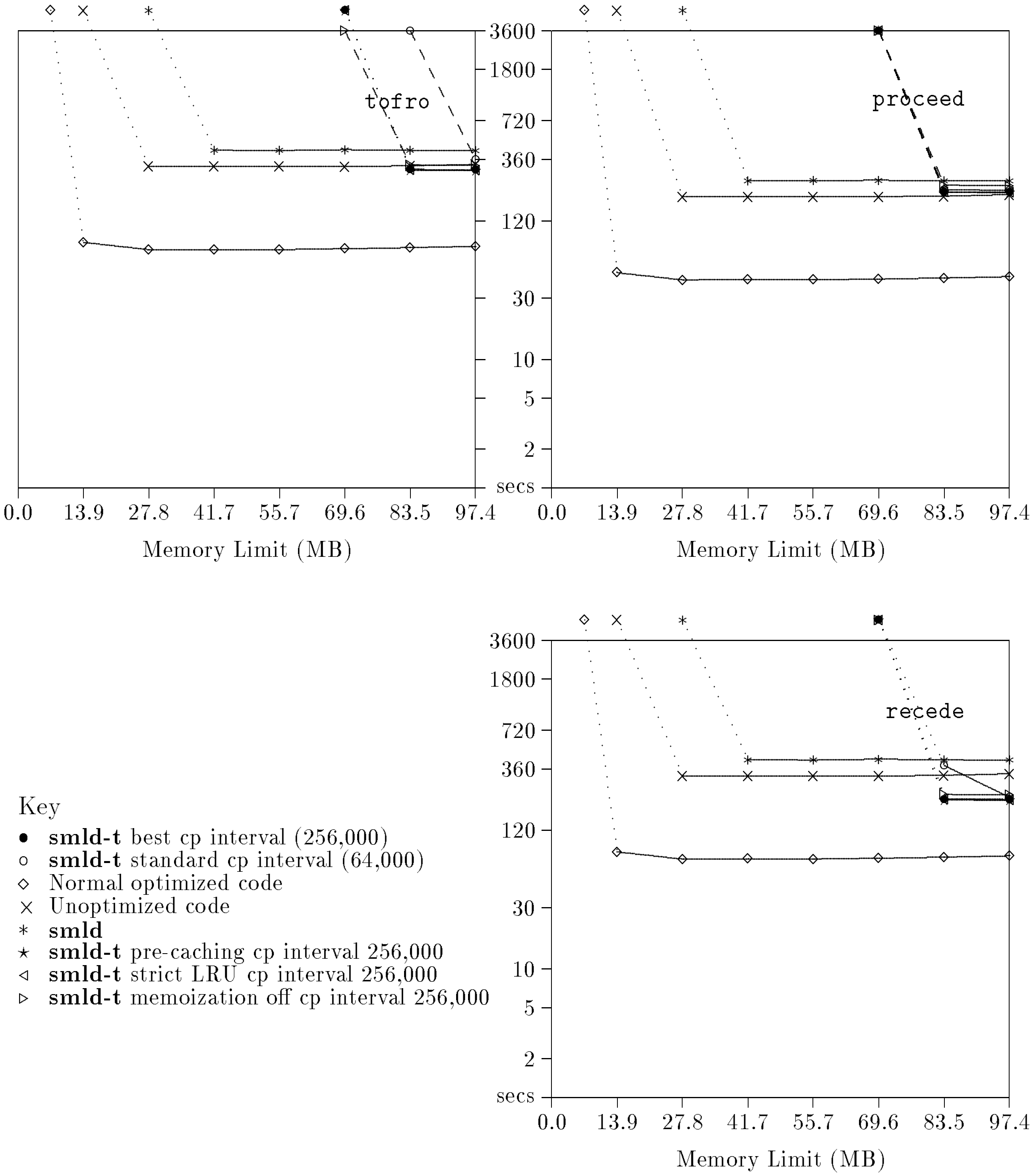

Figure 41: Time vs. space for Simple.

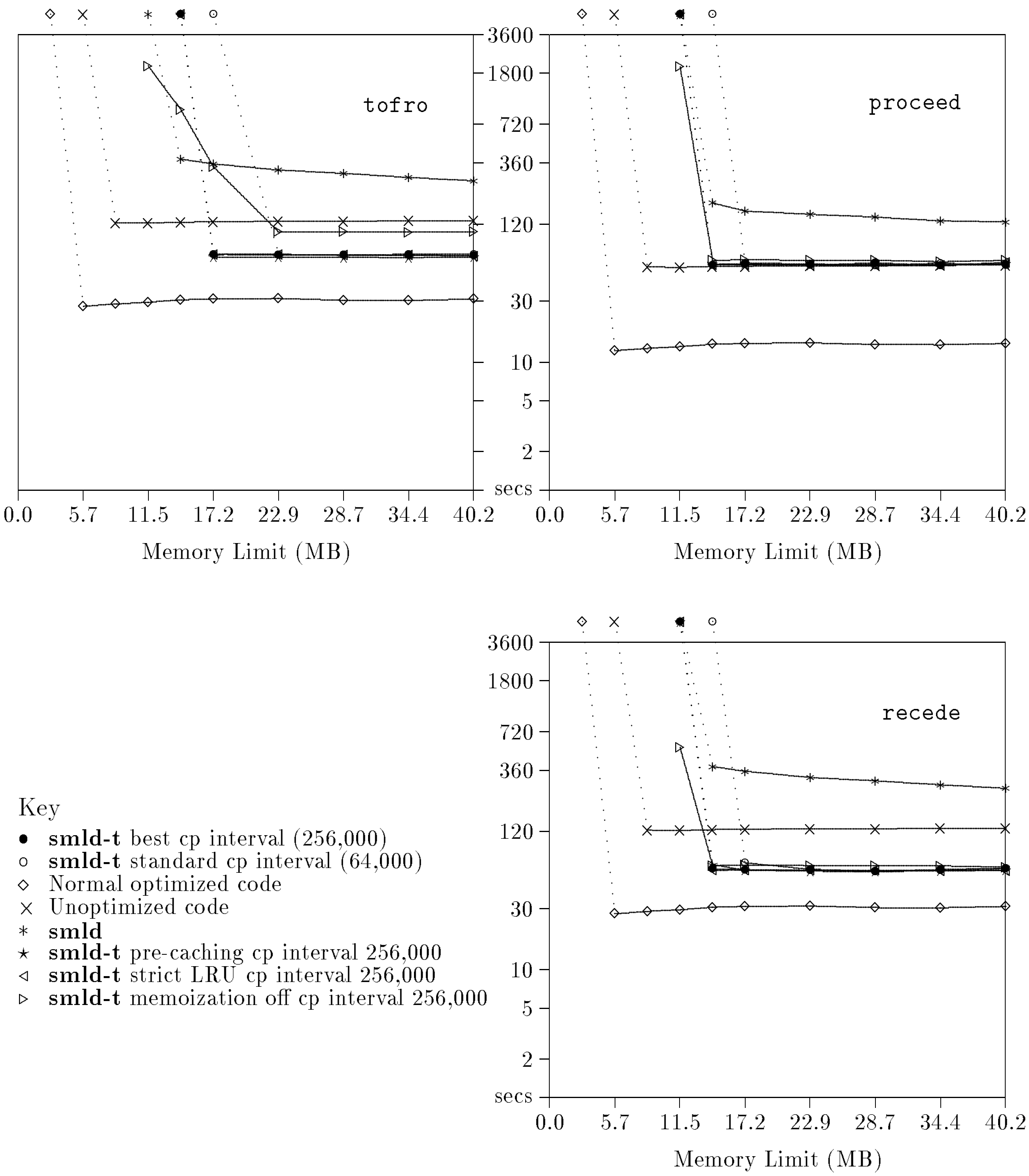

Figure 42: Time vs. space for Knuth–Bendix.

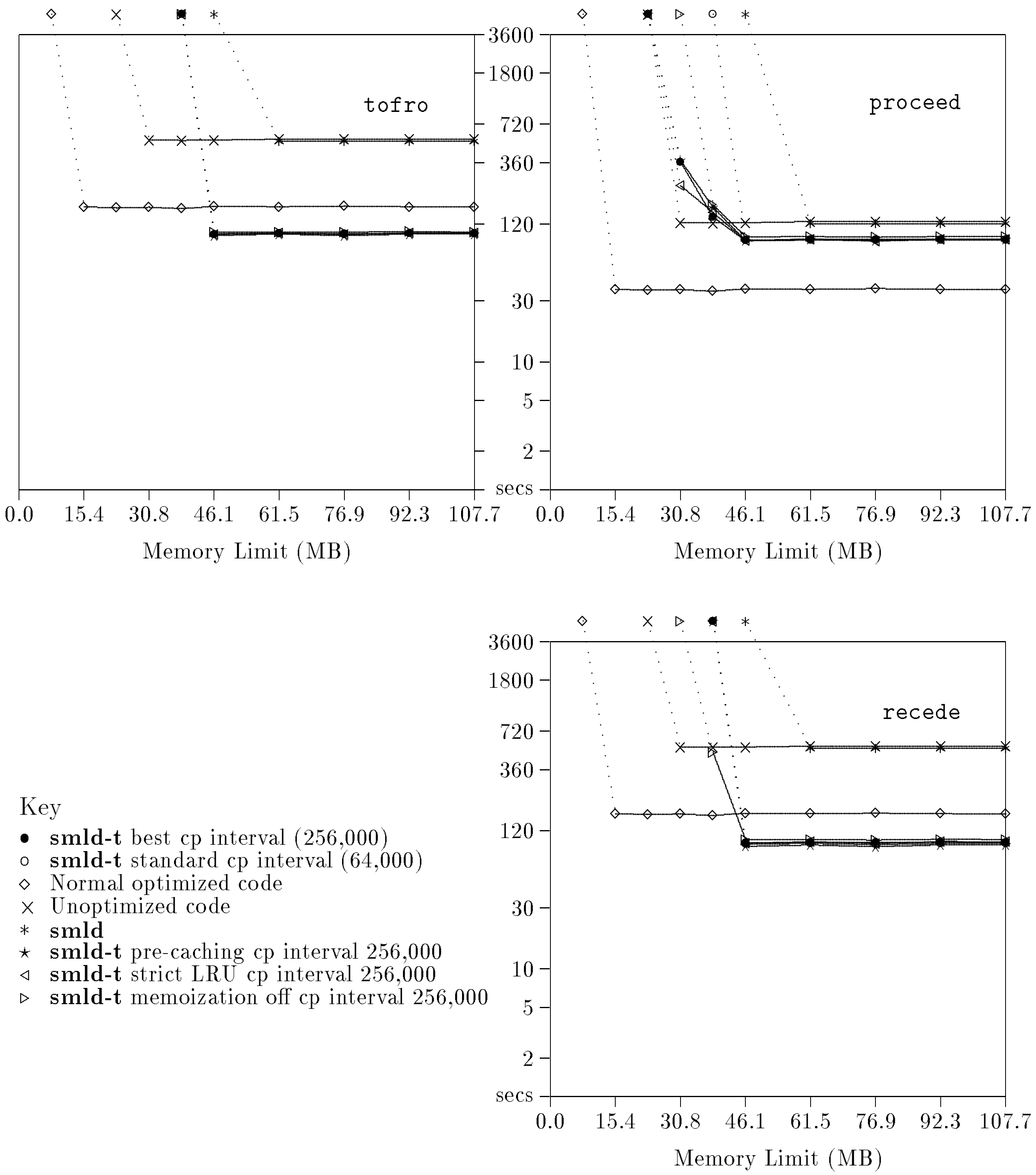

Figure 43: Time vs. space for RedBlack.

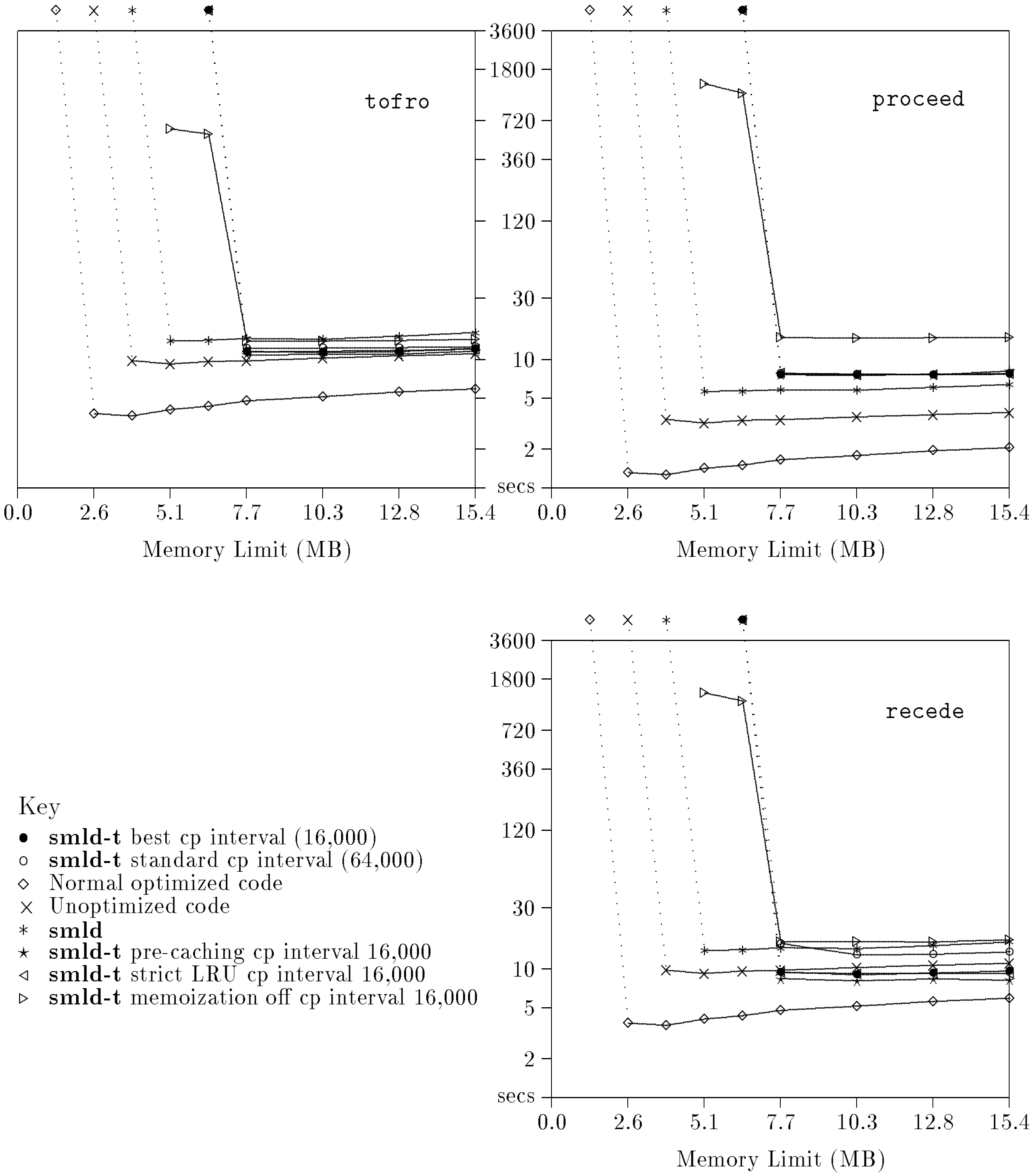

Figure 44: Time vs. space for Lexer.

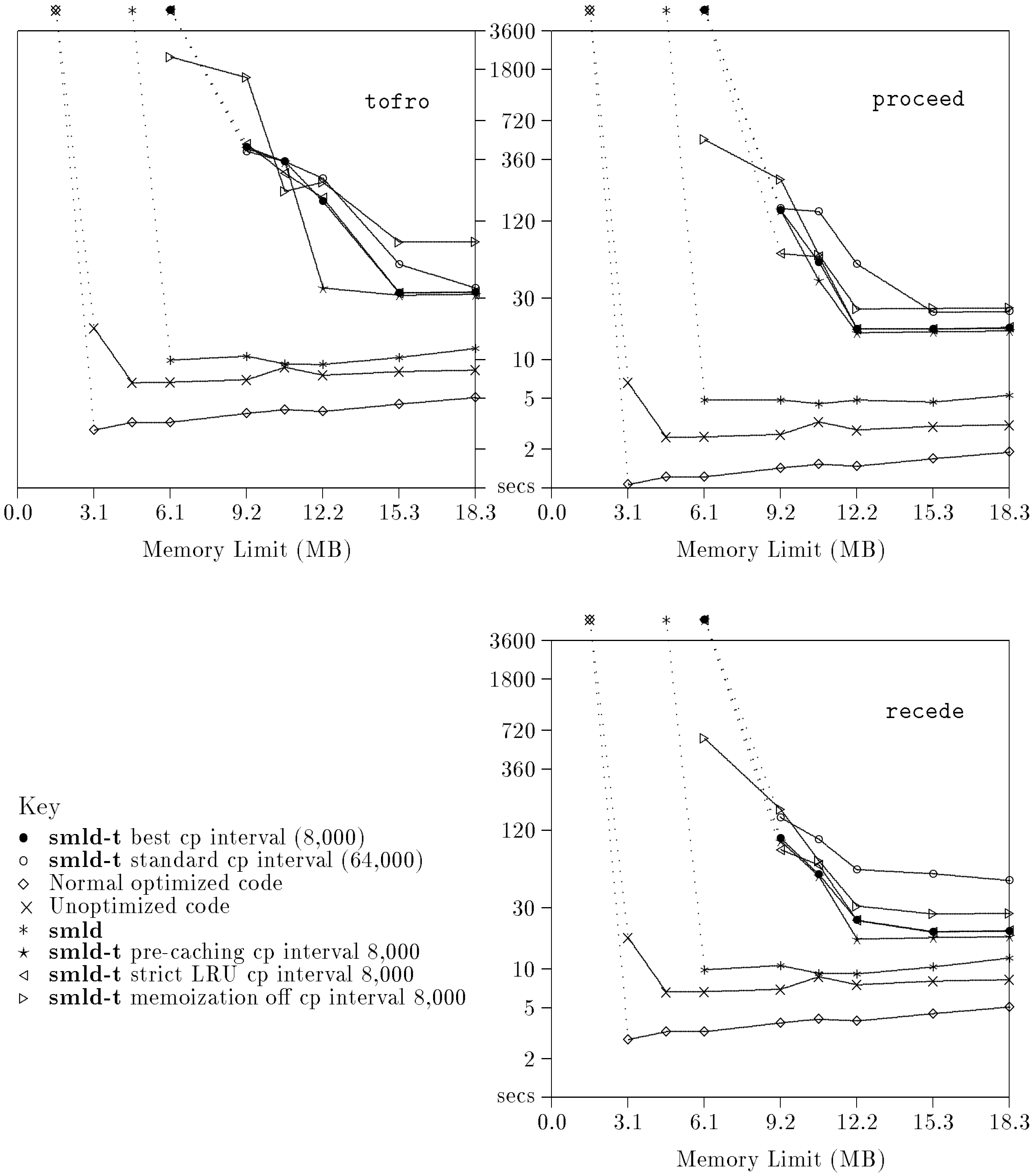

Figure 45: Time vs. space for UnionFind.

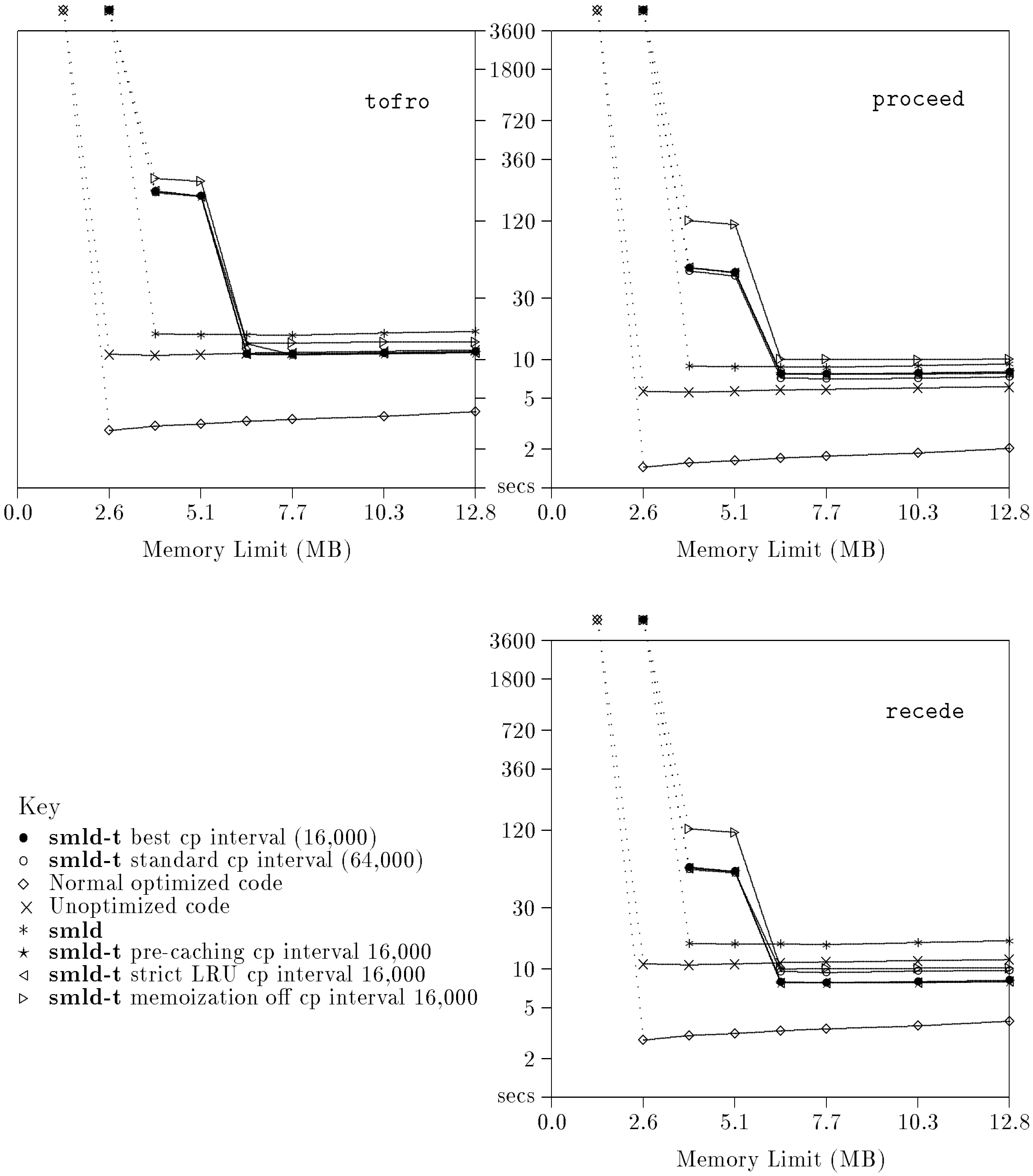

Figure 46: Time vs. space for TermRewriter.

8.9 Query Times

Since **smld-t** 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.

Table 4 shows the cumulative distribution of elapsed times required for **smld-t** to execute the commands listed above for each benchmark. Statistics are combined from at least three runs of all three scripts, executed with the same parameters as the **smld-t** best checkpointing interval curves of Section 8.8 and at the lowest memory size that is "comfortable" for all scripts. Measurements were made inside **smld-t**, so the substantial time (hundreds of milliseconds) required by the user interface to decode each command and display its results is not included. Garbage collection time, if any, is included. **Smld-t**'s `showCalls()` command shows a variable number of frames; the "show frame" entries give the time for the command divided by the actual number of frames shown.

Nearly all the tested commands executed in less than 1 second; many of them ran an order of magnitude faster. Execution time exceeded 2 seconds only for "show frame" under Life and Knuth–Bendix and for all commands except "single-step" under Simple. 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 lengthy garbage collection occurred during the execution of a "show frame." The poor times under Simple may be due to the cost of saving and restoring the program's large mutable state when time-traveling.

Benchmark	Command	< 0.1s	< 1.0s	< 2.0s	< 10.0s
Life	lookup var	63	92	100	
Cp 128,000	change scope	63	100		
Mem 23.1 MB	show frame	38	64	88	92
	single-step	100			
	single-step back	88	100		
Simple	lookup var	93	90	95	100
Cp 256,000	change scope	5	35	65	90
Mem 83.5 MB	show frame	20	45	55	90
	single-step	85	90	100	
	single-step back	0	33	43	100
Knuth–Bendix	lookup var	97	100		
Cp 256,000	change scope	53	100		
Mem 17.2 MB	show frame	40	93	93	100
	single-step	100			
	single-step back	80	100		
RedBlack	lookup var	82	94	100	
Cp 256,000	change scope	67	100		
Mem 46.1 MB	show frame	78	100		
	single-step	100			
	single-step back	99	100		
Lexer	lookup var	82	100		
Cp 16,000	change scope	76	100		
Mem 7.7 MB	show frame	86	100		
	single-step	100			
	single-step back	100			
UnionFind	lookup var	60	95	100	
Cp 8,000	change scope	59	100		
Mem 15.3 MB	show frame	57	100		
	single-step	100			
	single-step back	82	100		
TermRewriter	lookup var	98	100		
Cp 16,000	change scope	94	100		
Mem 6.4 MB	show frame	100			
	single-step	100			
	single-step back	100			
ALL	lookup var	80	96	100	
Best Cp	change scope	65	96	98	99
Mem	show frame	66	92	95	98
"Comfortable"	single-step	99	100		
	single-step back	88	97	98	100
.tex					

Table 4: **Smld-t** user-command execution times. Each row shows the cumulative distribution of execution times for a single benchmark and command type. The column labeled $< 0.1s$ shows the percentage of commands executing is less than 0.1 seconds, $< 1.0s$ shows the cumulative percentage executing in less than 1 second, etc.

Chapter 9

Debugging ML Threads

ML Threads is an extended version of Standard ML that supports concurrent computation. This chapter describes how our instrumentation and time-travel techniques can be applied to ML Threads, producing a debugger called **smld-c**. **Smld-c** works by adding debugging instrumentation to user code and providing special debugger versions of ML Threads' concurrency primitives and data types. **Smld-c** is completely independent of the underlying implementation of Threads, so it can be ported among implementations.

Smld-c supports the interactive debugging commands of Section 2.2. It also supports reverse execution and replay of the entire concurrent program; forward and reverse execution can be interleaved, as in **smld-t**.

The major challenges in the design of **smld-c** include dealing with non-deterministic computations, defining a suitable temporal model, and efficiently controlling a multi-threaded computation without special implementation knowledge or privileges. This chapter describes our general approach to meeting these challenges; details of the implementation and performance measurements appear in Chapter 10.

9.1 ML Threads

9.1.1 Definition

ML Threads extends Standard ML with a library of concurrency primitives. This library, initially developed at Carnegie Mellon University [32], resembles C Threads [31] and the threads facilities of Modula-2+ [18, 19] and Modula-3 [90], which are derived from Mesa [76] and monitors [65]. **Smld-c** provides debugging support for an enhanced version of ML

Threads, whose signature is given in Figure 47. The principle enhancements over the library described by Cooper and Morrisett [32] are "safe" references and write-once arrays, described in more detail below.

New threads are created dynamically by evaluating `fork` *function*, which executes *function* in a new thread of control running in parallel with the parent thread. The thread lives until it executes `exit` or until *function* returns. Threads are scheduled preemptively on as many processors as are available; they can also `yield` a processor voluntarily. The model provides no guarantees about the relative progress of concurrent threads. No assumptions are made about the granularity of parallelism in applications; the practical minimum granularity depends on how efficiently the thread primitives are implemented.

A thread inherits the environment of its parent, which typically consists of immutable values that can be shared safely without synchronization. Threads can also communicate via mutable shared memory using any `ref` cell or `array` in their common environment. Message passing primitives can be implemented on top of this model. There is no `join` primitive; it can be synthesized out of shared variables if needed.

The user is expected to protect shared memory with mutexes, although this discipline is enforced only partially by the model. Mutexes—mutual exclusion locks—are created dynamically by `mutex`. Mutexes are acquired by `acquire` and `try_acquire`. If the mutex is already locked, `acquire` blocks until the mutex's owner issues a `release`. Multiple outstanding acquisition requests are granted in arbitrary order; there are no fairness guarantees. If the mutex is free, `try_acquire` acquires it and returns true; otherwise, it immediately returns false. The `owner` function returns true if the caller currently owns the mutex. If a thread tries to release a mutex that it does not own, exception `Mutex` is raised.

In many programs, synchronization can be organized by associating a mutex with each shared-memory cell; a thread must hold the mutex to access the cell. ML Threads supports this programming style with "safe" references, created by calling `sref` with the initial value and associated mutex as arguments. Attempting to read (`sget`) or write (`sset`) a `sref` without holding the associated mutex triggers a `Mutex` exception.

Since it can be awkward or inefficient to associate a unique mutex statically with each shared-memory cell, ordinary `ref`s and `array`s are also permitted in ML Threads programs. However, their use is discouraged, both because it permits synchronization errors and because it exposes the underlying hardware consistency model in an otherwise machine-independent environment. For example, some machines maintain sequential consistency [75]

```
signature Threads =
sig
  val fork : (unit -> unit) -> unit
  val exit : unit -> 'a
  val yield : unit -> unit

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

  type 'a sref
  val sref: '1a * mutex ->  '1a sref
  val sget: 'a sref -> 'a
  val sset: 'a sref -> 'a -> unit
  val mutex_of_ref: 'a sref -> mutex

  type 'a sarray
  val sarray: int * '1a * mutex -> '1a sarray
  val ssub: 'a sarray * int -> 'a
  val supdate : 'a sarray * int * 'a -> unit
  val mutex_of_array: 'a sarray -> mutex

  type 'a woarray
  exception Empty
  exception Full
  val woarray: int -> '1a woarray
  val wosub: 'a woarray * int -> 'a
  val woupdate: 'a woarray * int * 'a -> unit

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

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

Figure 47: ML Threads primitives.

in hardware at all times, whereas others rely on user programs to use explicit synchronization primitives to enforce consistency where needed.

ML Threads also includes a "safe" `sarray` data type; a single mutex is associated with the entire `sarray`. If the individual array entries need separate locks, a vector (immutable array) of `sref`s can be used. In addition, ML Threads supports shared arrays in which each entry can be written at most once. These "write-once" arrays are created by calling `woarray` with an array size. All entries of a `woarray` are initially empty; entries are written with `woupdate` and read with `wosub`. Reading an empty entry triggers exception `Empty`; writing an already-written entry triggers `Full`. Use of write-once arrays is often combined with barrier synchronization: threads cooperate to fill a write-once array and then synchronize to ensure that the array is ready to be used by the next phase of the computation.

Threads can arrange their own synchronization and scheduling using conditions, which are created by `condition`. Each condition has an associated mutex and typically corresponds to some computable predicate on shared `ref`s having the same associated mutex. To block until some such predicate is true, a thread acquires the associated mutex and evaluates the predicate; if the predicate is false, the thread invokes `wait` on the corresponding condition. The semantics of `wait` are as follows: the calling thread atomically releases the associated mutex and blocks until a `signal` occurs on the condition; it then reacquires the associated mutex and returns. Treating the release and block operations as an atomic unit prevents "lost wakeup" problems: if the operations were separate another thread could obtain the mutex after the waiting thread released it, alter the value of the predicate, and signal the condition before the waiting thread has a chance to block on it. `signal` normally releases one thread blocked on the condition, but is allowed to release more. The `broadcast` operation releases all threads blocked on the condition. Receipt of a signal is only a hint that the mutex should be reacquired and the predicate retested.

An ML function can normally maintain private mutable data by defining local `ref` variables, but in ML Threads, multiple threads can execute the same function concurrently, implicitly sharing these `ref`s. Data private to a thread is provided by *per-thread variables*. Creating a per-thread variable (using `var`) conceptually allocates a separate mutable cell in each thread. All these cells are accessed by the same variable name (using `vget` and `vset`), but the interpretation of the name depends on which thread performs the access; each thread sees only its own cell. The type of a per-thread variable is fixed across all threads, but the value of each thread's cell is initially undefined; if a thread attempts to `vget` from a `var` before it has `vset` that `var`, exception `Undefined` is raised. Per-thread variables are useful for storing user-level thread identifiers and other state information.

The `callcc` operator is valid in ML Threads. Continuations captured in one thread can be invoked from another thread. **Smld-c** makes significant use of this feature to capture and restore program state. The sequential ML signal mechanism also works under threads; in particular, control can be transferred to a user-specified handler when a `CTRL/C` signal occurs.

Since ML Threads relies heavily on communication via the mutable store, it is not a particularly idiomatic extension to ML. However, Threads can be used as the basis for a number of "cleaner" higher-level communication mechanisms. As an example, Figure 48 shows a ML Threads implementation of Multilisp-style "futures" [60]. Evaluating the declaration

```
val r = future f x
```

starts a new thread to calculate `f x` in parallel with the current thread. The result of this calculation is obtained by `touch r`, which blocks, if necessary, until the calculation is complete; a result may be touched any number of times.[1] The general-purpose function `with_mutex` is provided in the standard ML Threads library. A typical use of futures is illustrated in Figure 49, which serves as a running example for the remainder of this chapter. Executing `min_g(m,n)` evaluates the function `g:int->int` (not shown) on each integer from `m` to `n` and returns the minimum value. The evaluation constructs a tree of parallel computations.

9.1.2 Implementation

ML Threads is implemented on uniprocessors and multiprocessors. The uniprocessor version in implemented entirely within SML/NJ. It uses `callcc` to capture thread state [116]; management of blocked threads is based on queues of continuations. Timer interrupt signals are used to preempt compute-bound threads [98]. An important feature of this implementation is that a permanently blocked thread e.g., a thread forked by a call to `future` whose result is discarded, can be garbage collected. The uniprocessor implementation runs on all current SML/NJ hardware platforms.

The multiprocessor version reimplements the uniprocessor code on top of machine-independent extensions to the SML/NJ runtime system that include support for obtaining

[1] A more complete implementation of `future` would handle exceptions that occur during `f x` and forward them to the touching threads.

```
fun with_mutex (m:mutex) (body:unit -> 'a) : 'a  =
    (* execute body while holding mutex *)
    (acquire m;
     let val result = body()
     in release m;
        result
     end)

abstype 'a future = FUTURE of 'a cell sref
and     'a cell = BUSY of condition | DONE of 'a
with
  fun future (function:'a -> 'b) (arg:'a) : 'b future =
        let val m = mutex()
            val c = condition m
            val f = sref (BUSY c,m)
            fun w () =
              let val result = DONE(function arg)
              in with_mutex m (fn () => (sset f result; broadcast c))
              end
        in fork w;
           FUTURE f
        end
  fun touch((FUTURE f):'b future) : 'b =
        let fun touch' () =
              case sget f of
                BUSY c => (wait c; touch' ())
              | DONE x => x
        in with_mutex (mutex_of_ref f) touch'
        end
end
```

Figure 48: Implementing `future` in ML Threads.

```
fun min_g (m:int,n:int) : int =
   if m = n then g m
   else
     let val b = future min_g ((m+n) div 2 + 1, n)
         val a = min_g (m, (m+n) div 2)
     in min(a, touch b)
     end
```

Figure 49: Function `min_g`.

and releasing physical processors, managing mutual exclusion locks with hardware support, maintaining per-processor data regions, and performing garbage collection across processors. Prototypes incorporating this set of extensions are currently implemented for Silicon Graphics' MIPS-based multiprocessors running Irix and Omron multiprocessors running Mach.

The safe memory access features of ML Threads depart from earlier Threads systems, whose design emphasized efficient implementation [19, 31]. For example, these systems typically use a single bit for a mutex and do not record the mutex's owning thread; thus, they don't even have enough information to determine if a thread attempts to release a mutex it doesn't own. ML Threads implements safe memory access checks at runtime. Each mutex holds the (internal) thread id of its current owner, and each `sref` points to its associated mutex. On each `sref` access, the system fetches the mutex pointer, fetches the thread id of the mutex's owner, and checks that it equals the current thread's id; if not, `Mutex` is raised. Checking ownership is analogous to checking array bounds. It provides extra safety at extra cost (two or three fetches and a compare) per access. In principle, static analysis techniques can be used to elide some checks at compile time; for example, in a straight-line sequence of accesses to an `sref`, only the first access requires a test. Our current implementations do not perform such optimizations. Finally, checking can be disabled.

9.2 Temporal Model

Smld-t, the sequential replay debugger, provides a simple abstract model of program time as a single line segment (see Section 5.1). This definition of program time has three key properties.

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

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

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

In defining time for ML Threads programs, we maintain as many of these properties as possible. The *local* time for a thread is maintained by associating a counter with the thread and incrementing it at each event. Events are located at the same sites as in **smld-t**, and also at *synchronizing operations*, i.e., lock acquisitions and releases and memory reads and writes, performed on *shared objects* in the mutable store, i.e., `mutex`, `ref`, and `array` objects. Local times have all the above properties with respect to events within a single thread.

There are several ways to extend the definition of time across threads. Our approach emphasizes the execution structure induced by `fork`s. The *faceted* time (*f-time*) of an event in thread t is a list of integers defined recursively as

$$\text{local time of event} :: \text{f-time of fork event that created } t$$

where the root event is forked at f-time `nil`. These faceted times have property 1, although it is complicated to determine which lists of integers constitute legal times.

Since threads have no intrinsic user-visible identifiers, **smld-c** identifies a thread by the f-time of the fork event that created it. With this understanding, faceted times can be interpreted as

$$\text{local time within thread} :: \text{identifier for thread.}$$

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

Just as sequential program execution builds a time line, a multi-threaded execution builds a *fork tree*. An example fork tree for the evaluation of `min_g(0,3)` is shown in Figure 50. Each branch of the tree represents a single thread; events within the thread correspond to points on the branch, with `fork` events at each branch junction.

We define a partial order $<$ on f-times as follows. Let t and t' be f-times, with

$$t = t_j :: t_{j-1} :: \ldots :: t_1 :: \texttt{nil},$$

$$t' = t'_k :: t'_{k-1} :: \ldots :: t'_1 :: \texttt{nil}.$$

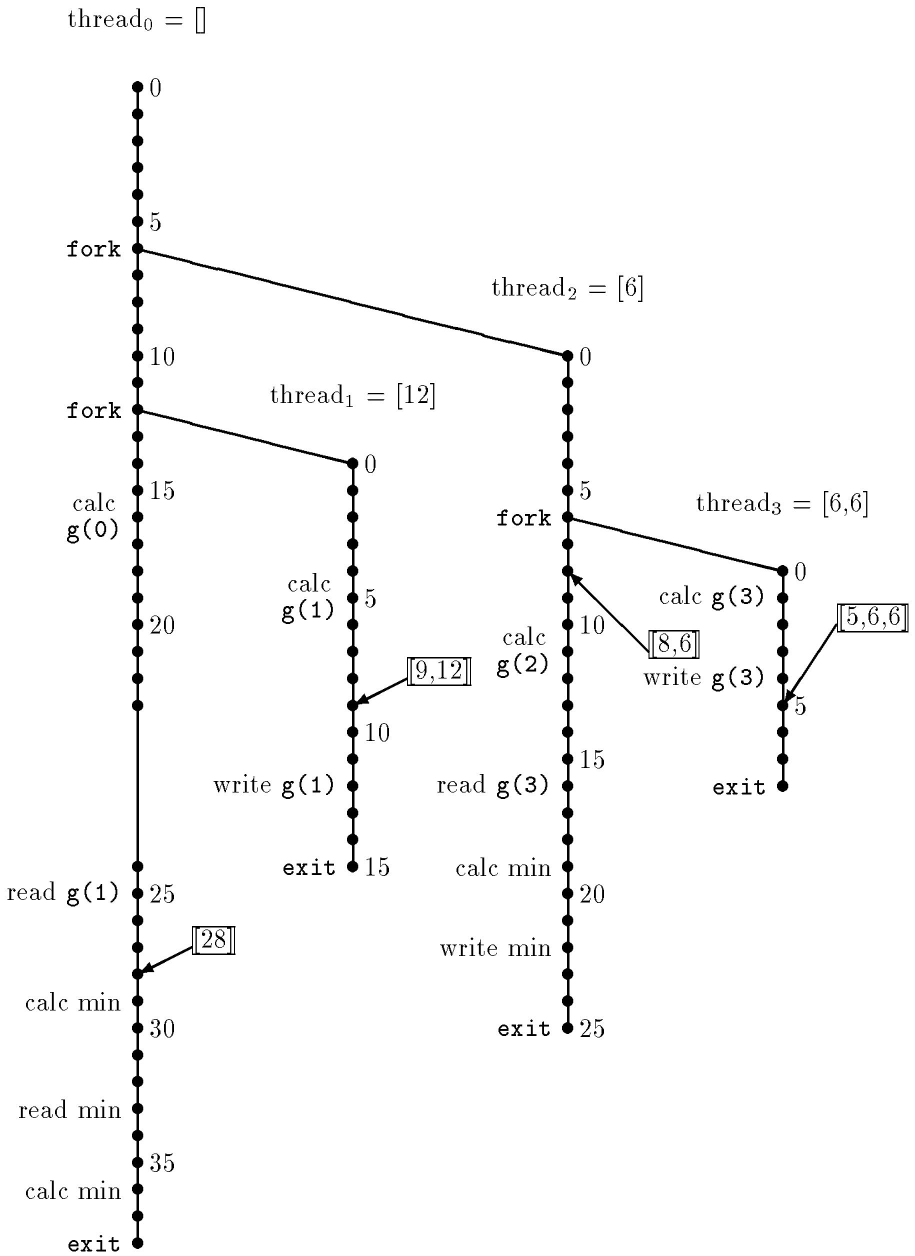

Figure 50: Fork tree for `min_g(0,3)`. Local times are only illustrative and do not correspond exactly with the program text. Selected f-times are shown boxed. Selected events are annotated.

Then

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

In terms of the fork tree, $t < t'$ implies that t lies on the path from t' to the tree root, and thus certainly implies that t occurred before t' during execution.

Defining a summing operator on the facets of a faceted time

$$\mathcal{S}(t_j :: t_{j-1} :: \ldots :: t_1 :: \texttt{nil}) = \sum_{i=1}^{j} t_i$$

produces a single integer *summed* time (*s-time*) representing total computation effort on the path from the root to the event, which approximates property 3. Section 9.4 describes how $<$ and s-times can be used to manage checkpoints for re-execution.

F-times do not reflect synchronizations resulting from shared-memory operations, so property 2 does not hold. The partial order $<$ is only a proper subset of Lamport's "happened before" relation [74]; we cannot deduce $t < t'$ from the fact that t occurs before t' in a particular execution, or even in *every* execution, if this ordering results from synchronization operations. Similarly, the s-time of an event measures computation effort only along the path from the root to the event, neglecting effort in other threads on which events on that path may depend.

9.3 Non-determinism

ML Threads programs are non-deterministic: their behavior can depend on the order in which threads perform synchronizing operations on shared objects. To obtain a complete representation of the causality relationships among threads during a particular execution, we add directed *synchronization edges* to the fork tree. The resulting *execution dag* is just a Lamport-style "space-time diagram" and exactly represents his "happened before" relation for an execution [74] (see Section 9.6.2).

In general, a synchronization edge is added from event a to event b if a and b lie in different threads, both threads access the same synchronization object x, a accesses x before b, and no other synchronization event accesses x after a and before b. Specifically, we add an edge from every mutex `release` to the next `acquire` of that mutex; from every mutex `acquire` to each subsequent `try_acquire` that fails because of that acquisition and from each such `try_acquire` to the `release` matching that acquisition; from each read or write of an unsafe `ref` or `array` element to the next read or write of that element; and from each write of a `woarray` element to a subsequent read of that element and vice-versa. Since waiting on a condition implicitly releases the mutex associated with the condition, we add an edge from the `wait` to the next `acquire` of that mutex, just as for an explicit `release`. No edges are needed for `signal` or `broadcast`, since the synchronization activity induced by these operations is described fully by `acquire`s and `release`s of the associated mutex. Safe memory element accesses do not generate edges for the same reason.

A possible execution dag for `min_g(0,3)` is shown in Figure 51. This execution performs three touches. The touch performed by thread_2 and the second touch performed by thread_0 occur after the future results have already been calculated by thread_3 and thread_2, respectively. The first touch by thread_0 occurs before the future result has been calculated by thread_1, so the touch initially reads `BUSY` from the result cell and `wait`s on the future's condition until the calculation is complete. `wait` implicitly releases the associated mutex; when the condition is signaled, it reacquires the mutex. There is no synchronization edge from the `signal` in thread_1 because thread_0 cannot proceed until the `acquire` succeeds.[2]

Non-determinism may be intentional, e.g., when several threads come to a barrier synchronization point in unpredictable order, or it may represent an error, e.g., when the programmer incorrectly believes that a data dependence prevents two threads from simultaneously accessing a shared object. In either case, **smld-c** must reproduce synchronization behavior on every re-execution of the non-deterministic program. Moreover, **smld-c** must guarantee repeatability for even for faulty programs; for example, it cannot rely on the user's adherence to mutual-exclusion protocols unless these are enforced by the language itself.

9.3.1 Logging

The simplest way to obtain repeatability is to use an initial non-deterministic execution of the program to build an execution dag, and then use the dag to control synchronization order in subsequent deterministic re-executions. If we are willing to replay the execution as a whole, and do not require the ability to replay an individual thread in isolation, it suffices to record the *order* of synchronization events; the *values* read or written are correctly

[2] If **smld-c** understood futures, it could log the synchronization effect of this touch operation more efficiently, since the initial `acquire`–read–`release` cycle does not affect program behavior.

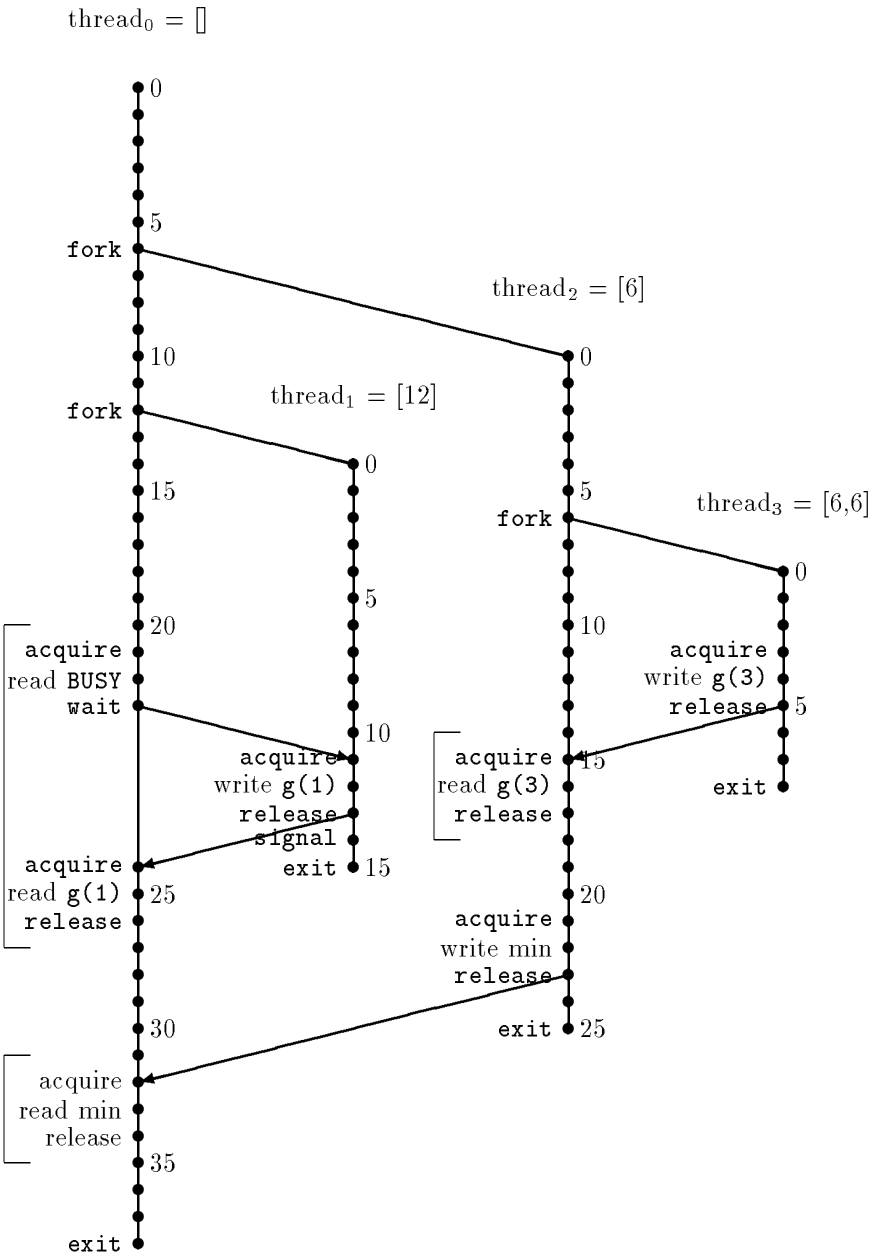

Figure 51: Execution dag for `min_g(0,3)`. Synchronization events are annotated. Events composing `touch` operations are bracketed.

reproduced by re-executing [77]. To support logging of synchronization events, **smld-c** uses special debugging versions of the ML Threads functions and data types, themselves written entirely inside the ML Threads model, that read or write the logs.

Maintaining a single log for the entire program would cause unnecessary serialization and represent a potential bottleneck during both recording and replay. Instead, **smld-c** maintains per-thread logs linked together into a tree isomorphic to the fork tree. In the simplest logging protocol, a field is added to each synchronization object to hold the f-time of the most recent access to the object. This field must be protected by a mutex. When a thread accesses a synchronization object during initial execution, it obtains the mutex, reads and remembers the last access field, and records its own f-time there. It then appends an entry containing the last access f-time to its log. Before replay, the log pointer is reset; during replay, it is advanced by one entry each time the thread attempts a synchronization operation. The operation is delayed until the last access field of the object matches the current log entry. This logging protocol requires a rigid separation between recording and replay execution phases, since the two phases interpret the last access field in different ways.

Replay delays could be implemented as busy-waiting loops. To improve performance, we associate a condition with each object. Threads that arrive at a synchronization prematurely wait on the condition; threads that enable a synchronization by accessing the object broadcast on the condition. Upon being released, each blocked thread checks to see if its f-time now matches the current log entry. If so, it performs the access; otherwise, it reblocks on the condition. If many threads are competing for the same object, this delay protocol is inefficient. To avoid this problem, we could generate separate conditions for each log entry so that only the correct thread is signaled. Alternatively, we could log read values instead of orderings, so that read operations on different versions could proceed concurrently. We have not implemented either of these variations.

The logging protocol linearizes accesses to each synchronization object. This protocol is adequate for ML Threads, since accesses to safe memory are already linearized by mutex acquisitions in any case. However, the protocol is unnecessarily strict for unsafe memory-element accesses, since adjacent reads in the linear ordering could be safely interchanged on replay without changing program execution. If ML Threads supported concurrent-read-or-exclusive-write (CREW) locks, a non-linearizing protocol would also be desirable for replaying read-mode acquisitions.

Once generated, logs cannot be discarded until the end of the debugging session. Since logs are maintained on a per-thread basis, every thread created has an entry in the log database, which is unfortunate since ML Threads implementations can ordinarily garbage collect dead threads. However, thread history serves other useful purposes, e.g., checking the validity of user-specified f-times (see Section 9.4). It would be possible to maintain logs on a per-object rather than per-thread basis; doing so would make it easier to collect dead threads, but harder to collect dead objects. Moreover, per-object logs would need to be protected by mutexes during update, which might increase synchronization overheads [84, pages 43–44].

Most parallel debuggers that support replay attempt to minimize "probe effects," i.e., the possibility that instrumenting for replay will eliminate timing-dependent errors or introduce new ones. In **smld-c**, the delays of synchronization logging are insignificant compared with the overhead of executing instrumented code in sequential sections. Therefore, we have not tried to avoid probe effects in **smld-c**, and we do not expect the system to be very useful for reproducing timing-dependent errors. In fact, **smld-c** records an initial non-deterministic execution not because it is "realistic" but because it is a convenient and efficient way to get *some* legal execution. An alternative approach is discussed in Chapter 11.

9.3.2 Making Logs Smaller

Since logging consumes space and time, it is important to minimize the number of shared mutable objects that must be logged. ML Threads allows any mutable variable to be shared among threads without any distinguishing declaration; moreover, because ML supports first-class functions, compile-time analysis to determine which mutable variables are actually shared would tend to be conservative. However, mutable variables must be declared explicitly in ML and typically constitute only a small fraction of all identifiers. In this respect, ML compares favorably with Lisp-based systems in which every object is potentially mutable [60, 61].

Logs can be made smaller by omitting "expected" cases. For example, **smld-c**'s logs omit unsuccessful calls to `try_acquire` or reacquisitions of a mutex by the same thread that last owned it. It must be possible to detect "missing" log entries, which is done by adding a local time stamp to each log entry.

Safe memory objects can be accessed only when their associated mutex is held, so mutex acquisition order fully determines the access order for safe objects. Thus, if the order of mutex acquisitions is logged, the order of `sref` and `sarray` accesses need not be. Use of safe memory may reduce log requirements, since a single mutex acquisition can cover multiple safe object accesses. Amusingly, since mutex operations have no effect on the synchronization semantics of unsafe objects, there is no point in logging these operations unless safe memory is being used.

Write-once arrays can be logged efficiently using a sparse log: only unsuccessful access attempts, which raise exceptions, are recorded in the log. On replay, if a log entry is found, the appropriate exception is raised. If there is no log entry, writes proceed immediately; reads block on a condition until the entry is written by another thread. The condition can be global to the array, local to each entry, or shared among multiple entries.

9.4 Global States and Execution Control

When a program is halted under **smld-c**, its entire global state is visible to the user. The global state consists of the individual thread states and a description of the mutable store, including the ownership status of all shared objects.

A *global time* is any set of f-times that are mutually incomparable under $<$. Given a global time $T = \{t_1, t_2, ..., t_n\}$ and an f-time t, we define an extension of relation $<$ as follows:

$$T < t \iff (\exists i) t_i < t \quad \text{or} \quad n = 0.$$

Every possible global state can be assigned a global time consisting of the set of f-times of its threads. Graphically, a global time is a cut across the fork tree that intersects each path from root to leaves at most once; it need not intersect all such paths. Figure 52 shows several global times for the execution of `min_g(0,3)`. Since shared objects cannot be altered without some thread incrementing its local time, the mutable store associated with a given global time is uniquely defined.

Smld-c always presents the user with *consistent* global states, i.e., states that actually occur during initial execution, or that obey the synchronization constraints on re-execution. Since f-times do not reflect synchronization relationships, there is no numerical characterization of consistent states in terms of global times. Graphically, any synchronization edge that originates below the cut line of a consistent global state must also terminate below the cut line. Thus, in Figure 52, global states A and C are consistent, but B is not.

Smld-c supports breakpoints at source text sites, which can be triggered by any thread

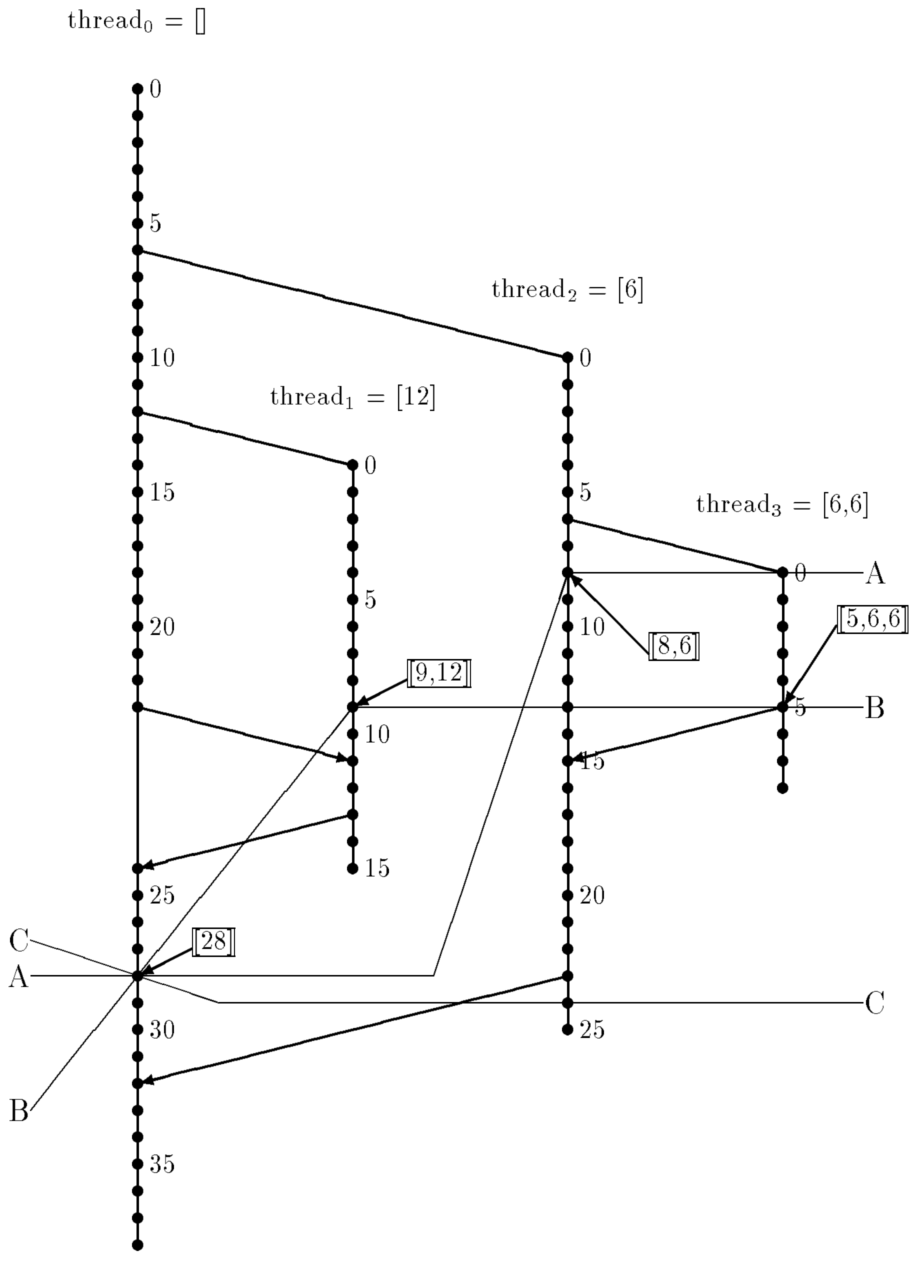

Figure 52: Sample global states and times for `min_g(0,3)`. A = {[28],[8,6],[0,6,6]} and C = {[28],[24,6]} are consistent states; B = {[28],[9,12],[13,6],[5,6,6]} is not, since the source of the synchronization edge from [13,12] to [24] is below B but the target is above B.

during forward execution; any number of breakpoints can be enabled. When an enabled breakpoint is encountered by a thread, that thread is halted immediately. All other threads are halted promptly but not immediately; however, threads cannot execute past a site with an enabled breakpoint. When all threads have halted, query commands can be applied to the local state of any thread or to the global state. By convention, terminated threads are omitted from global states. Because threads not involved in triggering a breakpoint stop at unpredictable points, repeated executions through a breakpoint may halt at different global times.

Smld-c could easily be extended to support breakpoints specific to some particular thread or group of threads. Reversing execution to a breakpoint within a particular thread would also be straightforward to implement, using a per-thread array of site times. However, it is not clear how to define reverse execution to a breakpoint if any thread is allowed to trigger the breakpoint: if two threads execute events at the same site concurrently, which one came "last"?

There are also user commands to "jump" to a specified f-time t. If $T < t$, where T is the current global time, **smld-c** executes all threads forward until t is encountered; otherwise, **smld-c** resets the global state to an earlier global time $T' < t$ and re-executes until t is encountered. To reset a global state, **smld-c** reverts the contents of the mutable store and restarts each thread described in the state. Every thread must be re-executed even if it is not part of the path from the program root to t, since execution along that path may require synchronizing with other threads. As with location-based breakpoints, threads other than the one containing t stop at unpredictable points, so the global time resulting from a successful jump is unpredictable. In Figure 52, both A and C are possible global times corresponding to a breakpoint at f-time $\boxed{28}$.

Many t values do not correspond to reachable local states. If t lies within a region of the fork tree already explored by initial execution, reachability can be determined by checking `fork` entries in the synchronization logs; otherwise, t's impossibility will be discovered when all threads have terminated or deadlocked. **Smld-c** could be extended to support jumping to a wholly or partially specified global time; if the target were an inconsistent time, the user program would deadlock attempting to reach it.

A global time can contain many f-times that are irrelevant to the user. **Smld-c** maintains a current *focus* on a single thread, which is set and altered automatically. At a breakpoint, the focus is set to the thread that triggered the break; if there is more than one, the focus is left unchanged if possible. After a jump, the focus is set to the thread containing the target f-time; a jump to an f-time already contained in the current global time can be used to change focus without causing execution.

To improve reverse execution performance, **smld-c** maintains a global checkpoint cache, similar to that of **smld-t**. Checkpoints are taken periodically by halting all threads. A checkpoint is a copy of the global state, including the continuation and synchronization log of each live thread, and a description of the global store; the checkpoint is tagged with a global time.

To reset to a particular target f-time t, **smld-c** re-installs a suitable checkpoint and re-executes. Candidate checkpoints are those having global times $T < t$; the beginning of the program, which has global time $\emptyset$, is always a candidate. For each candidate checkpoint j having global time $T^j = \{t^j_1, t^j_2, t^j_3, ...t^j_n\}$ with associated entry $i_j : t^j_{i_j} < t$, **smld-c** calculates $S^j = \mathcal{S}(t) - \mathcal{S}(t^j_{i_j})$, which gives a lower bound on the computation required to execute from the checkpoint at T to the event at t. **Smld-c** reinstalls the checkpoint having the lowest value of S^j. Since S^j measures computation only along the path from t to the root, this choice may not be the best one. For example, consider the execution dag in Figure 52, and suppose that there are stored checkpoints with global times A and C and that the target f-time is $\boxed{35}$. Under the above criterion, A and C look like equally good candidates for reinstallation, since $S^A = S^B = 7$. If we consider computation in other threads, however, C is clearly the better checkpoint to reinstall, since the program cannot reach $\boxed{35}$ without obtaining the result of the calculations in thread_2.

Smld-c uses **smld-t**'s lazy time-travel techniques (Section 5.2) without modification. The reconstruction of event records may involve replay of multiple threads. Speculative search (Section 5.3) is less useful in the Threads setting; it cannot be used to find global times corresponding to predicates, since these times do not form a linear space suitable for binary search. Speculative search could be used within a single thread, e.g., to find the thread-specific breakpoints described above, but we have not implemented this technique.

9.5 Implementing Control

The mechanics of stopping, starting, and reversing a distributed computation are complicated because **smld-c** has no special privileges with respect to the concurrency model. An ML Threads program has the same basic structure under the debugger as under normal execution: each user thread runs as a distinct thread in the debugged version. The debugger itself runs as a separate thread.

The fundamental debugger control operation is "stopping the world," i.e., halting all user threads at a breakpoint or a checkpoint. Since there are no asynchronous signals or alerts in ML Threads, the debugger thread has no way to interfere with a running user thread. Instead, it defines a global `pleaseStop` flag and augments the instrumentation code at user-thread events to test the flag periodically; if the flag is set, the instrumentation code writes the thread's continuation and other local state into a global location that can be read by the debugger thread and exits.

This self-halting method doesn't work for threads that are blocked on a mutex or condition. Such threads need to be artificially unblocked long enough to stop themselves. But a user thread blocked on a mutex will be unblocked only when the mutex is freed, and the debugger thread cannot free mutexes it does not own. It might appear that this problem could be solved by having each running thread free its own mutexes before it stops; threads blocked on these mutexes would then be released, allowing them to notice `pleaseStop` and stop, after freeing *their* mutexes to allow other blocked threads to proceed, and so forth. Having threads free their own mutexes before stopping would also permit the debugger thread to copy the contents of the safe shared-memory cells associated with these mutexes into a checkpoint. But this unwinding solution never reaches mutually deadlocked threads. Such threads cannot affect the remainder of the computation, so they could be omitted from checkpoints, but the mutexes they own need to be released so that **smld-c** can reset the system to an earlier global time when these mutexes might have been held by other threads.

Smld-c solves the blocked thread problem by preventing user threads from accessing primitive mutexes directly. Instead, the debugger emulates mutexes using a data type `Dmutex` with associated operations `Dacquire`, `Drelease`, etc. When user code is compiled to run under the debugger, these emulation routines are automatically substituted for the standard ML Threads primitives by linking with a special version of the ML Threads library. Simplified versions of the mutex emulation routines are shown in Figure 53; the actual routines, which must also deal with support for time-keeping, breakpoints, logging, and replay, are described in more detail in Section 10.6. The emulation code makes use of the primitive ML Threads condition mechanism; it also uses primitive mutexes, but only as spin locks to protect the fields of the `Dmutex` record. When the emulation routines are in

```
val noOne : threadId            (* debugger internal data type *)
val myThreadId : threadId var   (* id of current thread *)

type spinlock = mutex

datatype Dmutex =
  DMUTEX of
    {slock : spinlock,
     owner : threadId sref,
     cond : condition}

fun Dmutex() =
  let val slock = mutex()
  in DMUTEX{slock=slock,
            owner=sref(noOne,slock),
            cond=condition slock}
  end

fun Dacquire (DMUTEX{slock,owner,cond}) =
  (acquire slock;
   while (sget owner <> noOne) do
     guardedWait cond;
   sset owner (vget myThreadId);
   release slock)

fun Drelease (DMUTEX{slock,owner,cond}) =
  (acquire slock;
   if (sget owner) = (vget myThreadId) then
     (sset owner noOne;
      signal cond;
      release slock)
   else
     (release slock;
      raise Mutex))
```

Figure 53: Debugger versions of mutex primitives.

place, a user thread never actually blocks on a primitive mutex; instead it blocks on the `cond` field of the `Dmutex` structure. Since this condition field is known to **smld-c**, it can be signaled when the debugger stops the world. Moreover, since the debugger has access to the `owner` field, it can lock and unlock `Dmutex`es at will, thus solving the problem of accessing safe shared variables to take checkpoints.

`guardedWait` encapsulates support for stopping the thread when `pleaseStop` is set. Before allowing the thread to block on a condition, it adds that condition to a global set `blocked`; when the debugger thread stops the world, it sets `pleaseStop` and signals all the conditions in `blocked`. When the user thread is signaled in `guardedWait`, it inspects `pleaseStop` to see whether the signal was due to a debugger request; if so, the thread stops itself. If the thread is later restarted from this state, it simply returns from `guardedWait`. From the caller's perspective, `guardedWait` behaves like ordinary `wait` except that it will return after being unblocked either by a user program `signal` or because the debugger has stopped and restarted the world; the caller cannot tell which of these cases has occurred. Thus, like ordinary `wait`, `guardedWait` should only be used in contexts where the underlying predicate for the condition can be retested explicitly; in `Dacquire`, for example, the `owner` field is retested. Implementing `guardedWait` and `stopWorld` correctly and efficiently is complicated; details are deferred to Section 10.4.

Smld-c must preserve the execution behavior of a user program when it replaces the threads primitives in the program by debugger library versions. Since multi-threaded programs are non-deterministic, equivalence of behavior corresponds to equality between sets of *possible* execution dags. Reasoning about equivalence can become intricate, as proofs often depend on the detailed semantics of threads primitives. We have informal equivalence proofs for **smld-c** versions of ML Threads primitives, but we lack a suitable approach for formalizing such proofs. At the very least, formalization would require a specification for ML Threads, such as exists for other threads packages [19].

Smld-c's emulations for threads primitives must pass information from user threads to the debugger thread, which increases synchronization overhead. It is important to minimize this overhead and to distribute it to avoid bottlenecks. Ideally, adding debugging instrumentation should not destroy any scalable speedup properties enjoyed by the original ML Threads program; i.e., it should slow the program down by a constant factor independent of the number of processors in use. We examine experimental results on this question in Section 10.7.

9.6 Related Work

Instrumentation and reverse execution techniques are well-established in parallel debugging systems. McDowell and Helmbold [82] provide a general survey of tools for parallel and distributed debugging, and recent workshops on debugging [85, 86] have focused on such tools.

In this section, *process* means a thread of control that may share all, some, or none of its memory with other processes, depending on the system under discussion.

9.6.1 Event Recording and Replay

Debugging concurrent programs by recording program behavior for subsequent replay was first developed in the context of distributed message-passing systems. Curtis and Wittie's BugNet [35] logs the arrival time and contents of all inter-node communications, which must be made explicitly using system calls. The logs are built on a per-node basis and can be used to replay execution of a single node independently.

Miller and Choi's PPD system [26, 27], described more fully in Section 5.5.1, applies the value-logging approach to shared-memory parallel programs. Programs are divided into units separated by synchronization operations; the values of variables accessed within each unit are recorded in a per-thread log. The access sets are determined using dataflow analysis; this method produces compact logs except when the analysis encounters arrays or pointers. If the program obeys a synchronization protocol that prevents data races, the logs suffice to support independent, repeatable execution of each thread. PPD also collects sufficient information to detect data races; see Section 9.6.3.

Pan and Linton [94] suggest an approach to value logging of shared-memory programs that provides re-execution even for programs lacking proper synchronization. Their proposed RECAP system records the results of *every* shared-memory read into a per-process log; processes can be replayed independently. To reduce replay times, periodic per-process checkpoints are taken by performing a Unix-style fork on the process's address space, using copy-on-write to minimize copying. To save log space, the authors suggest that only a recent window of execution history be replayable. RECAP has not been implemented.

All these systems build logs containing communicated values; such logs can quickly grow large, especially in shared-memory systems having high communication rates and capable of passing large structures by reference. By contrast, **smld-c** logs only the relative order of inter-thread communications; to recreate the communicated values during replay, all threads are re-executed simultaneously. This approach was introduced by Carver and Tai [25], who describe how to reproduce the behavior of programs based on semaphores or monitors by logging "SYN-sequences" of synchronization events. They use a single global log, which serializes all synchronization operations during both recording and replay. For semaphore systems, the log is an ordered list of processes that perform P operations. During replay, a process delays each P operation until its process identifier is encountered in the log; the delay can be implemented by busy-waiting or blocking on a global condition semaphore. Monitor systems can be handled by logging P operations on a global semaphore governing entry into monitor bodies; alternatively, if the monitor mechanism supports condition variables, these can be used to block and reschedule processes waiting their turn to enter the monitor on replay. Use of one global log represents a bottleneck during recording and unnecessarily serializes replay.

Mellor-Crummey and LeBlanc's Instant Replay system [77, 84] records per-process logs of synchronization events; the logs are organized into a fork tree just as in **smld-c**. To avoid logging all shared-memory accesses, programs debugged under Instant Replay must adhere to a suitable synchronization protocol. Both CREW and mutual-exclusion protocols are supported; the former permits more efficient replay because the order of concurrent reads need not be reproduced. To log the CREW protocol, each shared object maintains a version number, incremented each time the object is written, and a read count for the current version. Log entries for reads record the version number; for writes, the version number and read count of the previous version are recorded. For the mutual-exclusion protocol, a version number incremented at each mutex acquisition suffices. On replay, logged synchronization operations are delayed until the current object state matches the recorded state.

Instant Replay emphasizes efficient recording to minimize probe effects. Breakpointing and trace commands are available only during replay. Performance experiments (using a 64-node BBN Butterfly multiprocessor) indicate the importance of tuning logging protocols to the communication style of a particular application; for example, recording overhead for a Gaussian elimination program was reduced from 200% to 1% by adopting a special protocol. Replay was less than 5% slower than recording on the two applications measured. Since Butterfly applications are typically configured with one process per processor, the replay mechanism implements synchronization delays by busy-waiting, which introduces minimal overhead.

Halstead and Kranz's MulTVision [61] is a recording and re-execution system supporting debugging and performance visualization for Mul-T [73], an implementation of Multilisp [60]. Like **smld-c**, MulTVision records all non-deterministic side-effects of a program, and assumes that most programs are mostly functional and thus have small logs. Unlike ML, Multilisp does not statically distinguish mutable from immutable cells; however, MulTVision dynamically distinguishes values written as side-effects by wrapping them in special records, and logs only those reads that return such records.

MulTVision is implemented by modifying the compiled code for reads, writes, and touch operations; it strives for minimal probe effect and performance impact during recording. Multilisp performs implicit touches on all operands for strict operations, which allows a programmer to wrap `future` around an expression without worrying about where the result is used. In Mul-T, futures are marked using a special type tag; implicit touches check for this tag in-line and transfer control to an out-of-line subroutine if a future is found. MulTVision takes advantage of this check. Since a side-effecting write can affect program execution only when the value written is used in a strict operation, logging read operations during recording can be deferred until the value read is implicitly touched. Side-effect values have the same type tag as futures, so the existing touch check detects them at no extra cost; only the out-of-line subroutine needs to be modified. Under this approach, the time to read an immutable value is only doubled by logging, but true side-effects can slow down by a factor of more than 100 when logged [61].

Side-effect values contain a version number; on replay, all versions of a side-effected cell are simultaneously available, so all reads can proceed immediately and writes are not needed. MulTVision reorganizes the log between recording and replay, so interleaving the two modes is not practical.

An ideal logging method would cause no probe effect or performance degradation. In support of this goal, Bacon and Goldstein [15] propose hardware assistance for logging shared-memory interactions in a cache-coherent multiprocessor. Since any communication between processors requires a bus transaction, non-deterministic behavior can be logged by bus snooping and recording the processor number and hardware instruction counter at each transaction. The log can be stored either as a total or partial order; the latter allows more parallelism during replay but requires more complicated hardware to maintain multiple queues of entries. The hardware has not been built.

9.6.2 Clocks and Time

Our basic model for describing time in a distributed system is due to Lamport [74]. Events occurring in different processes in a distributed system can affect one another only when there is communication between them. Temporal order assumes that "earlier" events can causally affect "later" ones, so temporal relationships in such systems should be represented as a partial order on events. This "happened before" relation ($\rightarrow$) is the smallest relation satisfying three conditions: (1) if a and b are events in the same process and a occurs before b then $a \rightarrow b$; (2) if a is a message send by one process and b is the receipt of that message by another process, then $a \rightarrow b$; (3) if $a \rightarrow b$ and $b \rightarrow c$, $a \rightarrow c$. Two events a and b are *concurrent* if neither $a \rightarrow b$ nor $b \rightarrow a$. Our definition of consistent states (see Section 9.4) is equivalent to mutual concurrency under $\rightarrow$. "Space-time diagrams," such as Figure 51, can help visualize $\rightarrow$ for a particular program execution.

Lamport describes a distributed algorithm for extending $\rightarrow$ to a consistent total ordering using scalar-valued logical clocks at each node; however, this total order masks concurrency [74]. It is also possible to maintain a time system fully reflecting $\rightarrow$ in a distributed fashion [48, 55]. Times are represented as vectors with one entry for each process. Each process maintains a current vector time and uses it to timestamp every message it sends; the receiving process compares each entry in the message timestamp with the corresponding entry in its current time, and resets the latter to the maximum of the two. Current times can be used to timestamp arbitrary events and determine whether one "happened before" the other: if event e occurs on processor p with timestamp $[t_1, t_2, \ldots, t_n]$ and e' occurs on p' with timestamp $[t'_1, t'_2, \ldots, t'_n]$, then $e \rightarrow e'$ if and only if $t_p < t'_p$. These timestamps can be used to test for consistent states using the mutual concurrency criterion described above. Goldberg, *et al.* [55] propose a replay facility for distributed message-passing systems that uses vector times to maintain global consistency while allowing processes to be replayed independently but in parallel.

Lamport-style scalar global times or vector times could be implemented for shared-memory systems as well, e.g., by treating shared locations as virtual "processes." However, doing so would increase the overhead of shared-memory operations, especially in the case of vector times, which would need to be written, read, and compared at each synchronization event. Vector times would be particularly awkward to implement if the number of processes varies. **Smld-c**'s f-times are less expensive to maintain and provide a useful if limited temporal ordering.

Elapsed times can also be useful for understanding distributed system behavior. For example, MulTVision [61] stamps log entries with elapsed times and specifies replay targets in the same way. BugNet [35] logs use local elapsed-time stamps to achieve accurate replay of processes handling real-time interrupts.

9.6.3 Detecting Race Conditions

Race conditions, or access anomalies, occur when two processes concurrently perform conflicting data accesses, i.e., they both access the same shared object and at least one of the accesses is a write. Race conditions typically arise when programmers fail to use appropriate synchronization primitives. Race-condition detection is important for several reasons. First, a race condition ordinarily indicates a bug. Second, replay debuggers often rely on correct use of synchronization primitives by the debugged program to keep log sizes small [26, 27, 77, 84]; a program execution that fails to adhere to the synchronization protocol cannot be replayed accurately. Finally, recently proposed multiprocessor architectures use weakly consistent memory models that rely on correct program synchronization to give the illusion of sequential consistency [2, 3, 53].

Efficient detection of race conditions has been the subject of much research; Netzer [91] provides an excellent summary and proposes a standard formalism and terminology for the problem. Detection can be performed for any single program execution by recording a synchronization dag for the execution (as in Section 9.3), including read, write, and synchronization events. The dag is then analyzed; a race is reported whenever a pair of data-conflicting events is concurrent under $\rightarrow$. Analysis can either be conducted postmortem on an event log [3, 26, 27] or on-the-fly during program execution [38, 39, 40, 101, 106]; on-the-fly methods typically store only part of the dag at any one time and report only a subset of races. A variety of compact encodings have been proposed for on-the-fly detection, but the method remains expensive in time and space [39].

These detection methods operate on a single program execution, and ordinarily do not provide information about other possible executions, although in some cases such information can be deduced by omitting selected synchronization edges from the graph [40] or using conservative approximations when analyzing array accesses [26, 27]. Moreover, instrumenting the program for detection can induce probe effects that hide data races. Thus, even repeated execution with detection instrumentation in place provides no guarantee of race-free production execution. Also, these detection methods don't provide much information about the origin of the races, although Netzer [91] proposes methods for isolating informative sets of races.

ML Threads provides safe memory access primitives that cannot be subject to races. The runtime cost of these primitives can be substantial, but might be comparable to the measured costs of on-the-fly methods. If a safe primitive attempts an unsafe access, the raised exception points the user directly to the offending code.

If unsafe primitives are used, **smld-c** logs and replays them accurately. Moreover, when an unsafe program is instrumented for execution under **smld-c**, all unsafe accesses to a given memory cell are serialized using proper synchronization. This serialization permits reliable replay of the instrumented program even on weakly consistent hardware [53]; on the other hand, it may also mask errors that might appear in the production version.

9.6.4 Debugging in the Target Language

Some existing systems support parallel debugging in the target parallel language. Carver and Tai [25] implement logging of semaphore or monitor sequences using emulation routines that rely only on the constructs being logged. Gait [52] describes a portable system for debugging C programs consisting of Unix processes that communicate using system calls and files. He inserts debugging instrumentation in the C source code, which passes control to the debugger at the end of each source line and at least once per loop iteration. He does not address the problems of sharing address space with a faulty program in an unsafe language.

Source-code instrumentation has also been used to implement several tracing and debugging tools for Ada [37, 78]. Goldszmidt, Katz, and Yemini [57] describe a set of portable debugging tools, designed for concurrent languages such as OCCAM, NIL, or ADA, that rely on source-code instrumentation to send tracing data to a separate monitoring process.

One defect of source-language instrumentation is that it causes large probe effects. Probe effects are recognized as a problem, but there has been little study of how they destroy the behavioral equivalence of normal and instrumented versions of a program. Malony [81] studies how instrumentation perturbs performance measurement; he seeks to deduce performance behavior of the production program by analyzing measurements of the instrumented version, but he can deal with only a restricted class of synchronization primitives. Gait [52] found that his instrumented programs could reproduce most of the race conditions found in the production versions; we conjecture that this surprising finding may be due to the very high overhead of his synchronization primitives.

Chapter 10

Implementing Concurrent Debugging

The implementation of **smld-c** is similar to that of **smld-t**, but substantially more complex. Much of the debugger's state information must be replicated in each thread, and distributed mechanisms are required to stop threads at breakpoints and subsequently restart them. In addition to the usual instrumentation, **smld-c** relies heavily on a special version of the ML Threads library to implement logging and deterministic replay, and to permit the debugger to control blocked threads. This chapter describes key features of the **smld-c** implementation and its performance under the multiprocessor ML Threads prototype system.

10.1 Threads, Times, and Logs

Smld-c permanently associates a `threadInfo` record with each logical user thread. This record is created by the thread's parent when the thread is forked and subsequently updated by the thread itself. The record, shown in Figure 54, includes the identity of the thread's parent, the local time (within the parent) and corresponding s-time at which it was forked, the furthest time and event to which it has been executed during this program invocation, and a pointer to the beginning of each per-thread log.

Smld-c uses several time representations. Local times (`ltimes`) are integers. Externally, **smld-c** identifies threads by the f-times at which they were forked (see Section 9.2); `ftimes` are lists of `ltimes`. Since f-times are cumbersome for internal use, **smld-c** generates a unique, unchanging reference to each `threadInfo` record, of type `thread`, to use as an

```
type ltime = int           (* local time *)
type ftime = ltime list    (* f-time *)
datatype itime = ITIME of ltime*thread (* internal version of f-time *)

and threadCursors =
   CURSORS of
      {fork: thread cursor,
       synch: itime cursor,
       create: object cursor}

and threadInfo =
   THREADINFO of
      {parent: thread,
       forkTime: ltime,         (* in parent when this thread was forked *)
       cummForkTime: int,       (* s-time when this thread was forked *)
       maxKnownTime: ltime ref, (* highest time reached in thread *)
       maxKnownEvRec: eventRecord ref,  (* corresponding eventRecord *)
       initialCursors: threadCursors}  (* pointers to logs *)
 | NULLTHREADINFO

and thread = THREAD of threadInfo ref

val nullThread = THREAD(ref NULLTHREADINFO)
```

Figure 54: Time and thread data types in **smld-c**. Data type `eventRecord` is defined in Figure 55. Many of the data types described in this chapter are mutually recursive, so the actual code must use a single `datatype` declaration to define them simultaneously; they are presented in separate figures here for clarity.

internal thread identifier. Internal `itime`s are `ltime*thread` pairs; an `itime` pair can be converted to an `ftime` by recursively consulting the `parent` and `forkTime` fields of `threadInfo` records starting from the initial `itime`'s `thread`; the reverse conversion can be made for legal `ftimes` by recursively consulting the record of `fork`s executed by each thread, starting from the root thread.

Each debugged user thread maintains three logs: a *fork* log listing the child `thread`s forked by this thread, a *synchronization* log describing this thread's synchronization dependencies (see Section 10.6), and a *creation* log of all mutable objects generated by this thread (see Section 10.5). Each log is an instance of a abstract log data type parameterized by log entry type. Logs are accessed sequentially using *cursors*, also parameterized on entry type; each cursor contains a mutable pointer into a particular log. A single log may be referenced by multiple cursors. Log operations are specified as operations on cursors; they include read at current cursor pointer, append at current pointer (implicitly deleting later entries in the log), and advance current pointer, which affects the cursor itself but not the log. To support sparse logs, each log entry is tagged with the `ltime` at which it was appended. Sets of cursors for the three per-thread logs are grouped together as the `threadCursors` data type. Function `createThreadCursors` creates a new set of logs and associated cursors; function `copyThreadCursors` copies the pointers from an existing set of cursors into a new, independent set.

10.2 Instrumentation

Smld-c's instrumentation code, shown in Figure 55, combines characteristics of both **smld** and **smld-t**. `event` is always executed in the context of a user thread; it accesses both global and per-thread information. Since the current time, target time, and thread identifier differ for each thread, they are defined as per-thread variables. By convention, the names of these variables begin with `my`, to emphasize that each thread sees a different copy of them. As in **smld-t**, event records are generated lazily and `binding` fields contain event times, but these times are now `itime` pairs. As in **smld**, location-based breakpoints are supported using a boolean array `breakWanted`, indexed by site number; this array is globally shared among all threads, but user threads only read it. The debugger is invoked via `break` if the current time matches the target time or if the relevant `breakWanted` entry is set. **Smld-c**'s instrumentation placement algorithm is the same as for **smld** and **smld-t** (see Figure 9),

```
type siteNumber = int
type value
datatype eventRecord =
   EVENT of {siteNumber: siteNumber,
             binding: itime,
             boundValues: value list}

(* global data *)
val breakWanted : bool array
val nullEvRec = EVENT{siteNumber=0,
                      binding=(0,nullThread),
                      boundValues=nil}

(* per-thread data *)
val myCurrentTime : ltime var
val myTargetTime : ltime var
val myThread : thread var

fun event(siteNumber,binding,boundValues) : itime =
   let val newTime = vget myCurrentTime + 1
       and currThread = vget myThread
   in vset myCurrentTime newTime;
      if newTime = vget myTargetTime orelse
           breakWanted sub siteNumber then
        break(EVENT{siteNumber=siteNumber,
                    binding=binding,
                    boundValues=boundValues})
      else ();
      ITIME(newTime,currThread)
   end
```

Figure 55: Events in **smld-c**.

```
datatype threadState =
     THREADSTATE of
      {thread: thread,
       time: ltime,
       cursors: threadCursors,  (* current pointers into logs *)
       cont: unit cont,         (* per-thread continuation *)
       evRec: eventRecord}
```

Figure 56: Thread state.

with the understanding that `event_`n identifiers are pairs rather than scalars. However, the combination of per-thread variable accesses and use of pairs for times makes this code run considerably slower than the equivalent instrumentation in the sequential debuggers. If **smld-c** knows the internal representation of `vars`, it can in-line calls to `vget` and `vset` to make `event` faster.

The `event` routine is also called directly from the debugger versions of the ML Threads library functions at "pseudo-events" with artificial site numbers designated by `FORKsite`, `EXITsite`, and `SYNCHsite` (covering synchronization operations on mutexes and conditions).

10.3 Thread State and Control

It would be most straightforward for **smld-c** to associate each logical thread in the user program permanently with a single actual thread, i.e., an ordinary thread created by the underlying ML Threads implementation. However, this association would be inconvenient for supporting flexible time travel. Instead, **smld-c** maps each logical thread in the user program to a series of actual threads over the course of a debugging session. When initially forked, a logical thread executes within a new actual thread. When the program halts at a breakpoint or checkpoint, each actual thread records its current logical `threadState` (shown in Figure 56) on a global `currentThreads` list and exits. Since threads are lightweight, this is an efficient process; most of the state is contained in a continuation that can be captured using `callcc`. A global checkpoint of program state consists of the contents of the `currentThreads` list and a copy of the contents of the mutable store (see Section 10.5). To restart the computation from a given checkpoint, the debugger passes each state on the `currentThreads` list to a newly forked thread, which restarts the computation described by that state. The debugger itself is a permanent actual thread at the root of **smld-c**.

Figure 57 shows **smld-c**'s basic control functions, all executed in the context of the debugger thread. Function `setCompilationUnit` initializes `currentThreads` to contain a `threadState` for the root user thread. Execution is initiated by calling `execTo`, which forks an actual thread to run each entry in `currentThreads`, and then bocks until all threads have halted and placed themselves back on `currentThreads`. `execTo` is invoked by `recordTo` and `replayTo`, which first set the global `userMode` appropriately; at any given point during execution, either all threads are recording or all are replaying. Before passing control to the user continuation, each forked actual thread calls `setupState` to initialize per-thread variables including `myLimitTime`, and `setTargetTime` to initialize `myTargetTime`. These functions, which are always executed in the context of a user thread, are shown in Figure 58. `myLimitTime` is a per-thread variable containing the thread's "long-term" goal time: if the global limit f-time specified to `execTo` lies in the current thread, `myLimitTime` is set to the corresponding local time; otherwise, it is set to infinity. `myTargetTime` is the thread's short-term goal; it depends on the "stop-the-world" testing interval (described below), the limit time, and, if replaying, the thread's maximum known time. The code shown for `setTargetTime` is simplified slightly; the actual code checks whether the calculated target time has already been reached and, if so, returns a flag to the caller indicating that the thread should stop immediately.

An individual thread stops when it executes an explicit `exit` or the function it is executing returns; when it reaches its maximum known time while replaying, the global limit f-time, or a breakpoint specified in `breakWanted`; or when it determines, at a periodic test point (every 1000 events), that a stop-the-world request has been made by another thread or by the CTRL/C signal handler.[1] Checks for stopping are performed in two stages. `event` (see Figure 55) invokes `break` at location-based breakpoints or when the thread's local time matches `myTargetTime`. Function `break`, shown in Figure 59, rechecks whether the thread is at a breakpoint site or has reached `myLimitTime`. In either of these cases, the thread generates a stop-the-world request and then stops itself. Otherwise, `break` tests for a stop-the-world request (`worldStopping()`). If no request is outstanding, `myTargetTime` is reset to the next test point and `break` returns, continuing normal thread execution. Function `stop` is called with the current `eventRecord` and continuation, captured via `callcc`; it generates a `threadState` and appends it to `currentThreads`, updates various thread data

[1] A thread also stops if it raises an uncaught exception; for simplicity, exception handling code is omitted from the figures in this chapter.

```
val currentThreads: threadState list sref = (nil,mutex())  (* global *)

fun setCompilationUnit (f: unit -> 'a) : unit =
    let val rootThread = ref (THREADINFO{ ... })
        val fcont:unit cont =
                    callcc (fn c => (callcc (fn c1 => throw c c1);
                                     f();
                                     deregister();
                                     exit()))
        val rootThreadState =
              THREADSTATE{thread=rootThread,
                          time=0,
                          cursors=initial cursors from rootThread
                          cont=fcont,
                          evRec=nullEvRec}
    in initialize currentThreads to [rootThreadState];
    end

datatype limitTime = FTIME of ftime | INFINITY
val globalLimit: limitTime ref  (* global *)

fun execTo(limit:limitTime) : unit =
  let fun restart (THREADSTATE{thread,time,cursors,cont,evRec}) () =
        (setupState(thread,time,cursors);
         setTargetTime();
         throw cont ())
      val threads = contents of currentThreads
  in clear currentThreads;
     globalLimit := limit;
     initRuncount (List.length threads);
     app (fork o restart) threads;
     allThreadsStopped()  (* block until all threads halt *)
  end

datatype mode = RECORD | REPLAY
val userMode: mode ref (* global *)

fun recordTo(limit:limitTime) = (userMode:=RECORD; execTo limit)
fun replayTo(limit:limitTime) = (userMode:=REPLAY; execTo limit)
```

Figure 57: Basic control functions for **smld-c**. All functions are executed within the debugger root thread; all variables are global to all threads but `globalLimit` and `userMode` are updated only by the debugger thread. Certain syntactically complex operations are represented by pseudo-code in *italics*; in particular, the details of obtaining and releasing the mutex associated with `currentThreads` before accessing the `sref` are elided.

```
val myLimitTime: ltime var  (* per-thread *)

fun setupState (thread: thread,
                time:ltime,
                cursors:threadCursors) : unit =
    (vset myThread thread;
     vset myCurrentTime time;
     register();
     vset myLimitTime
       (case !globalLimit of
          FTIME ft => if ft is in thread then ltime for ft else infinity
        | INFINITY => infinity);
     vset myCursors (copyThreadCursors cursors))

val checkDelta: time ref = ref 1000  (* global (read-only) *)

fun setTargetTime() : unit =
  let val checkLimit = vget myCurrentTime + !checkDelta
      val newTarget = min(vget myLimitTime, checkLimit)
      val useTarget =
          case !userMode of
            RECORD => newTarget
          | REPLAY =>
             min(value of maxKnownTime from myThread, newTarget)
  in vset myTargetTime useTarget
  end
```

Figure 58: Set-up control functions for **smld-c**. Functions are executed in a user thread's context. Per-thread variables `myThread`, `myCurrentTime`, and `myTargetTime` are defined in Figure 55. Pseudo-code is in *italics*.

```
fun break(evRec as {siteNumber,...}:eventRecord) : unit =
  if (vget myCurrentTime = vget myLimitTime) orelse
      breakWanted sub siteNumber then (* at real breakpoint: stop everyone *)
    (stopWorld();
     callcc(stop evRec))
  else if worldStopping() then (* global stop requested *)
    callcc(stop evRec)
  else setTargetTime()  (* return to user program *)

fun stop (evRec:eventRecord) (cont:unit cont)  =  (* stop myself *)
    let val state = THREADSTATE
                    {thread = vget myThread,
                     time = vget myCurrentTime,
                     cursors = copyThreadCursors (vget myCursors),
                     cont = cont,
                     evRec = evRec}
    in append state to currentThreads;
       update maxKnownTime and maxKnownEvRec for myThread;
       deregister ();
       exit()
    end
```

Figure 59: Functions `break` and `stop`. Functions are executed in a user thread's context.

structures, and exits.

The code in Figures 57–59 invokes several housekeeping functions: `initRuncount` initializes the count of running threads, `register` and `deregister` add and remove the current thread from a global list of running threads, and `allThreadsStopped` blocks until there are no more running threads. The implementation of these functions is described in the next section.

10.4 Stopping the World

As described in Sections 9.5 and 10.3, the debugger "stops the world" at breakpoints or checkpoints by having all threads periodically test a global predicate. Threads blocked on a condition cannot perform the test, so they must be handled specially. A distributed algorithm for implementing `stopWorld` in the presence of blocked threads is shown in Figures 60 and 61. Each running thread has a safe reference `status`, associated with a unique mutex, that indicates whether the thread is `RUNNING` or `BLOCKED` on some condition. A thread has access to its own status through the per-thread variable `myStatus`, and to the statuses of other threads via the global list `threadStatuses`, guarded by a global mutex `threadm`. The mutexes that guard thread `status` flags also serve as a distributed `pleaseStop` flag: if a thread cannot obtain its own status's mutex, a stop-the-world operation must be in progress. This test is encapsulated by `worldStopping`. `stopWorld` can be invoked either from a user thread, e.g., when it reaches a breakpoint, or via an ML signal handler invoked when the user types CTRL/C. It first acquires `threadm`, in order to freeze the `threadStatuses` list, and then acquires all the mutexes of `sref`s on the `threadStatuses` list. It also sets the global flag `stw` to prevent multiple simultaneous stop requests.

The housekeeping functions described in Section 10.3 manipulate the data associated with `threadm`. `initRuncount(n:int)` is called from the debugger thread to initialize `runcount` to `n`, `threadStatuses` to `nil`, and `stw` to `false`. `allThreadsStopped` waits on `allStoppedCond` until `runcount` reaches 0. `incRuncount` is called from the debugger version of `fork` (see Section 10.6) to increment `runcount` before a new thread is forked. `register` creates a new status `sref` and mutex for the current thread, places it in `myStatus`, and appends it to `threadStatuses`. To avoid a race condition in which a new thread registers after another thread has initiated a stop request, `register` checks the `stw` flag; if it is set, the new thread acquires its own status mutex, so that it will halt correctly the first time

```
datatype status = RUNNING | BLOCKED of condition weak
val myStatus: status sref var   (* per-thread *)

(* global data *)
val threadm = mutex()
val threadStatuses : status sref list sref = sref(nil,threadm)
val runcount = sref(0,threadm)
val allStoppedCond = condition threadm (* signaled when runcount = 0 *)
val stw = sref(false,threadm)

fun stopWorld () : unit =  (* stop everyone else *)
    let fun lockThreads (s::r,cacc) =
             (acquire (mutex_of_ref s);
              case sget s of
                RUNNING => lockThreads(r,cacc)
              | BLOCKED wc => if  target of wc has been garbage-collected then
                                (sset runcount (sget runcount - 1);
                                 lockThreads(r,cacc))
                              else
                                (wc points to some condition c;
                                 lockThreads(r,c::cacc)))
           | lockThreads (nil,cacc) = cacc
        fun alertBlocked c =
             let val cm = mutex_of_condition c
             in acquire cm; (* see text *)
                release cm;
                broadcast c
             end
    in with_mutex threadm (fn () =>
          if not (sget stw) then
           (sset stw true;
            let val conds = lockThreads(sget threadStatuses,nil)
            in app alertBlocked conds;
               if sget runcount = 0 then
                 broadcast allStoppedCond
               else ()
            end)
          else ())
     end

fun worldStopping() : bool =
    let val m = mutex_of_ref (vget myStatus)
    in if not (try_acquire m) then
         true
       else (release m; false)
    end
```

Figure 60: Functions `stopWorld` and `worldStopping`. These functions are normally executed in a user thread's context; `stopWorld` can also be invoked from **smld-c**'s `CTRL/C` handler.

```
fun guardedWait cond evRec =
    let val s = vget myStatus
        val sm = mutex_of_ref s
        val cm = mutex_of_condition cond  (* we hold this *)
    in if not (try_acquire sm) then (* stop-the-world in progress *)
         (release cm;
          callcc(stop evRec);
          (* returned from stop-the-world *)
          acquire cm)
       else
         (update maxKnownTime and maxKnownEvRec for myThread;
          sset s (BLOCKED (weak cond));
          release sm;
          (* dangerous window here; see text *)
          wait cond; (* atomically release cm and do the actual wait *)
          (* we have been signaled and have reacquired cm *)
          if not (try_acquire sm) then (* stop-the-world in progress *)
            (release cm;
             callcc(stop evRec);
             (* returned from stop-the-world with status reset to RUNNING *)
             acquire cm)
          else  (* we were signaled normally *)
            (sset s RUNNING;
             release sm))
    end
```

Figure 61: Function `guardedWait`. This function is executed in a user thread's context.

it calls `worldStopping`. `deregister` decrements `runcount` and broadcasts on condition `allStoppedCond` if it reaches 0.

In addition to testing for stop-the-world periodically during normal execution, user threads must also use `guardedWait`, shown in Figure 61, to perform this check before and after blocking on a condition. If no stop request is pending, `guardedWait` successfully acquires the status mutex, changes the thread's status to `BLOCKED`, and records a pointer to the condition. If `stopWorld` encounters a `BLOCKED` status while acquiring status mutexes, it broadcasts on the condition. Before broadcasting, `stopWorld` acquires the mutex for the condition to ensure that any thread that is executing `guardedWait` and is going to perform an actual `wait` has already done so. When signaled, the waiting thread retests its status mutex and discovers that a stop is requested. Programmers often make threads block permanently on a condition that will never be signaled, relying on the garbage collector

```
fun jumpTo (target:ftime) =
      if target is not the f-time of some thread in current state then
        let val bestCheckpoint = findPrevCheckpointInCache target
        in resetState bestCheckpoint;
           if target is not a current f-time then
            (if target is within range of known times then
               replayTo (FTIME target)
             else
               (replayTo INFINITY;
                recordTo (FTIME target))
             putInCache (getState()))
           else ()
        end
      else ()

  fun proceed () (* from current state until breakpoint reached *)
    (replayTo INFINITY;
     if no breakpoint reached then
       recordTo INFINITY
     else ();
     putInCache(getState()))
```

Figure 62: Time-travel functions for **smld-c**. Functions are executed in the debugger thread's context.

to remove them; to avoid keeping such threads alive unnecessarily, the condition field in `BLOCKED` holds a weak pointer. When `stopWorld` encounters an invalidated weak pointer, it decrements `runcount` directly, since the blocked thread will never execute `deregister`.

The distributed representation of the `pleaseStop` flag guarantees that polling for a stop request and waiting on a condition are local functions. However, thread fork and exit require global synchronization, controlled by `threadm`, to update `runcount` and `threadStatuses`. Thus, programs using fine-grained parallelism may run disproportionately slower than coarse-grained programs under **smld-c**.

10.5 Time-Travel Functions and Checkpointing

Figure 62 shows the basic time-travel functions for **smld-c**; these functions are invoked by the debugger in response to user commands. `jumpTo` attempts to reach a specific target f-time, first resetting the global state to the best available checkpoint. A preliminary check (not shown in Figure 62) can be made on the argument f-time to eliminate obviously invalid targets, i.e., those specifying a time beyond the known termination time of a thread. `proceed` executes forward from the current state until a site breakpoint is reached or the program terminates.

Smld-c uses a simple checkpoint cache, with a fixed number of entries and a LRU replacement strategy. Functions `putInCache` and `findPrevCheckpointInCache` perform the obvious tasks; the latter chooses the previous checkpoint according to the minimal s-time strategy described in Section 9.4. Unlike **smld-t**, **smld-c** does not generate checkpoints at regular intervals, only at user-specified breakpoints.

Checkpoints are generated by function `getState` and reinstated by `resetState`. A checkpoint consists of a copy of the `currentThreads` list and a description of the contents of the mutable store. **Smld-c** handles checkpointing of the mutable store using a distributed variant of the scheme described in Section 6.5. Lists of updated `ref` cells and `array` entries are maintained on a per-thread basis. When threads stop at a breakpoint, the debugger thread processes the entire collection of lists as if it were a single "global" list, removing duplicates and recording current values. For simplicity, the code managing these lists is omitted from figures in this chapter. Logs of mutable object creations used to avoid recreating pointer objects during replay (see Section 6.5) are maintained on a per-thread basis; the `create` components of `threadCursors` are cursors into these logs. As in **smld-t**, `create` logs contain weak pointers to the objects, to conserve space when the objects are short lived.

All mutable fields of debugger versions of threads data types, e.g., `Dmutex`, `Dcondition`, `Dsref`, etc. (see Section 10.6), are treated as ordinary mutable `sref`s, so they can be saved and restored using the standard mechanism described above. In particular, the lock state and ownership of mutexes are described by the mutable store checkpoint. Since the library code for these functions is not instrumented, instructions to place these `sref`s on the appropriate update lists are included explicitly.

10.6 Emulation of Threads Primitives

Smld-c provides a version of each ML Threads primitive that emulates the behavior of the normal version using debugger-controlled data structures. Debugger primitives support logging of forks and synchronizations and implement operations such as stop-the-world.

The debugger versions of the primitives are distinguished from the standard primitives by prefixing them with the letter `D`, e.g., `Dfork`, `Dacquire`, etc.

Figure 63 shows code for `Dfork` and `Dexit`. If recording, `Dfork` generates a new `thread` descriptor for the child thread and appends it to the parent's fork log; if replaying, it reads the original descriptor from the log. `Dfork` then forks a child process that invokes the argument function `f` after calling `setupState` and `setTargetTime` (see Figure 58). If `f` terminates, `Dexit` is called explicitly to deregister the child thread.

The debugger versions of synchronization objects and functions are typified by the data type `Dmutex`, shown in Figure 64, and the corresponding functions `Dacquire` and `Drelease`, shown in Figure 65. These are more complete versions of the code shown in Figure 53. This code and the fields of `Dmutex` serve several purposes. First, they must correctly emulate normal mutex behavior. The `locked` field represents the state of the mutex; if it is `true`, the `owner` field contains the owning thread. This field is checked by `Drelease` to make sure that the releasing thread is indeed the owner; otherwise, the `Mutex` exception is raised. During recording, if an acquire must be delayed because another thread already owns the mutex, the acquiring thread blocks on `cond`, which is signaled by the releasing thread. All these fields are associated with `slock`.

Second, the code must support logging of mutex acquisitions when recording and adhere to the logged order when replaying. Each thread's synchronization log contains one entry for each `Dacquire` operation, holding the `itime` of the previous `Drelease` operation on the mutex. To generate the log entry, `Dacquire` reads the contents of the `owner` and `access` fields of `Dmutex`. The former is set only by successful `Dacquire`s, so after a release it contains the identity of the releasing thread; the latter is set by `Drelease`.

On replay, the log entry is interpreted as a requirement to delay until the specified `Drelease` occurs. As in recording mode, the acquiring thread blocks on `cond`, which will be alerted when the mutex is released. `Drelease` uses broadcast instead of signal for this purpose because it cannot tell which of several blocked threads should replay next. The code shown in Figure 65 is simplified somewhat. In practice, log entries are recorded only when the releasing thread differs from the acquiring one; on replay, if there is no entry with the current timestamp, it is safe to proceed with the `acquire` immediately.

Finally, the code must block on `cond` using `guardedWait` rather than ordinary `wait`, so that the thread can respond to a stop-the-world request. As with an ordinary `wait`, a return from `guardedWait` is only a hint: the underlying predicate (i.e., the value of `locked`

```
fun Dfork f =
  let val thread =
         case !userMode of
           RECORD =>
              let val newThread =
                      THREAD (ref (THREADINFO
                        {parent= vget myThread,
                         forkTime= vget myCurrentTime,
                         cummForkTime= (cummForkTime from myThread) +
                                       (vget myCurrentTime),
                         knownTime= ref 0,
                         knownEvRec= ref nullEvRec,
                         initialCursors= createThreadCursors()}))
              in append newThread to fork log in myCursors;
                 newThread
              end
         | REPLAY =>
              let val oldThread = read thread from fork log in myCursors
              in advance log;
                 oldThread
              end
       fun child() =
         (setupState(thread,initial cursors from thread,0);
          setTargetTime();
          event(pseudo-event at FORKsite);
          f();
          Dexit())
  in incRuncount();
     fork child
  end

fun Dexit () =
   (event(pseudo-event record at EXITsite);
    update maxKnownTime and maxKnownEvRec for myThread;
    deregister();
    exit())
```

Figure 63: Functions Dfork and Dexit.

```
datatype Dmutex =
     DMUTEX of {slock:mutex,        (* spin lock for these data *)
                locked:bool sref,   (* state of mutex *)
                owner:thread sref,  (* current or previous *)
                access:ltime sref,  (* last release time *)
                cond: condition}    (* signaled on release *)

fun Dmutex() : Dmutex =
  let val slock = mutex()
      val dm = DMUTEX{slock=slock,
                      locked=sref(false,slock),
                      owner=sref(nullThread,slock),
                      access=sref(0,slock),
                      cond=condition slock}
  in append locked,owner,access to create log in myCursors;
     dm
  end
```

Figure 64: Function `Dmutex`.

when recording or of `owner` and `access` when replaying) must be retested. In addition, the code must cover the possibility of `userMode` changing during a `guardedWait`. Since the predicate-testing code in `do_acquire` may be called an arbitrary number of times before the acquisition succeeds, it must be idempotent; in particular, on replay the synchronization log must not be advanced until the acquisition is assured.

The techniques shown here for mutexes extend to the other synchronization primitives. In general, the synchronization log contains an entry for every operation that is the target of a synchronization edge in the execution dag, representing the operation (in some other thread) that is the source of of the edge (see Section 9.3). Where possible, sparse log techniques are used for specific operations; for example, `Dtry_acquire` logs only successful acquisitions.

Treating return from `guardedWait` as only a hint requires that the state of each synchronization object be made explicit so that it can be retested. To achieve this requirement for conditions, **smld-c** represents each `Dcondition` by an integer counter protected by a spin lock. Each time the condition is signaled, the counter is incremented. Before blocking on the condition, a process remembers the value of the counter; when unblocked, the process compares the current and remembered counter values to determine whether a signal actually occurred.

```
fun Dacquire (dm as DMUTEX{slock,locked,owner,access,cond}) =
  let val evRec = pseudo-event at SYNCHsite
      fun set_owner () =
        (sset locked true;
         sset owner (vget myThread);
         append locked, owner to thread's list of updated refs)
      fun do_acquire () =
         case !userMode of
           RECORD =>
             if not(!locked) then
               let val releaser = sget owner and releaseTime = sget access
               in set_owner();
                  release slock;
                  append ITIME(releaseTime,releaser) to synch log in myCursors
               end
             else
               (guardedWait cond evRec;
                do_acquire() (* try again *))
         | REPLAY =>
             let val ITIME(releaseTime,releaser) =
                        read entry from synch log in myCursors
             in if not (sget locked) andalso
                     releaser = sget owner andalso releaseTime = sget access then
                    (set_owner();
                     release slock;
                     advance synch log in myCursors)
                  else
                    (guardedWait cond evRec;
                     do_acquire() (* try again *))
             end
  in event evRec;
     acquire slock;
     do_acquire()
  end

  fun Drelease (DMUTEX{slock,locked,owner,access,cond}) =
    (acquire slock;
     if (sget locked) andalso (sget owner = vget myThread) then
       (sset locked false;
        sset access (vget myCurrentTime);
        release slock;
        append locked, access to thread's list of updated refs;
        case !userMode of
          RECORDING => signal cond        (* alert one waiter *)
        | REPLAYING => broadcast cond)    (* alert all waiters *)
     else
       (release slock;
        raise Mutex))
```

Figure 65: Functions Dacquire and Drelease.

10.7 Performance

This section describes preliminary performance results for a prototype implementation of **smld-c**. The prototype lacks support for I/O logging and for several user commands, including "skipping." We measure the overhead of controlled execution and compare the speedup properties of normal and debugger execution.

Four programs were benchmarked with and without debugger instrumentation. AllPairs solves the all-pairs shortest-paths problem on a directed linear graph of 100 nodes using a parallel version of Floyd's algorithm, originally written in Mul-T [89]. Execution consists of 100 phases separated by barrier synchronizations; every phase forks 100 threads, each of which computes a `woarray` forming one row of the intermediate result matrix, itself a `woarray`. MaxSpanTree finds the minimum spanning tree of a fully connected 100-node graph using a parallel version of Prim's algorithm, also originally written in Mul-T [89]. The tree is built in 100 steps; at each step, the tree is augmented with the closest point not yet in it, as determined by exhaustive parallel search. Tree nodes are implemented using safe memory cells. MatMult multiplies two 100×100 integer matrices, represented as vectors; it forks one thread to compute each row of the result matrix, which is represented as a `woarray`. Simple is a parallel version of the algorithm described in Chapter 8, run for one step on a 30×30 grid. Each phase of the original algorithm is parallelized on a row-by-row basis and produces a `woarray`; the phases are separated by barrier synchronizations.

Benchmarks were run on a 8-processor Silicon Graphics 4D/380S multiprocessor running version 3.3 of the IRIX System V operating system. Each processor is a 40MHz MIPS R3000 with a 64KB direct-mapped primary instruction cache, a 64KB direct-mapped write-through primary data cache and a 1MB secondary snooping data cache. The processor also supports specialized spin-lock hardware [103].

Benchmarks were conducted under SML/NJ version 0.70, which supports multiprocessing with ML primitives for obtaining processors from the operating system, accessing hardware spin locks, and accessing per-processor data areas. Each processor allocates in a separate region of memory. When a processor runs out of heap space it first attempts to borrow memory from another processor's region; if this fails, it halts all the other processors and initiates a sequential garbage collection covering all regions. Benchmarks used a variant of the version 0.70 runtime system, modified to allow precise control of heap size and to support various monitoring functions. All benchmarks were run using 100MB of main memory, except for Simple, which was given 200MB. Threads were scheduled from a global FIFO ready queue. Thread execution was preempted by timer interrupts every 100ms. All figures are taken from benchmark runs having the minimum elapsed time over a sample of at least three runs; in all cases, the benchmark was the only user process on the machine.

Only controlled execution overhead was measured. Each benchmark was executed from beginning to end with no query operations or checkpoints. Figure 66 shows times for normal execution, debugger recording, and debugger replay, for one to eight processors. The graphs show both total elapsed time and elapsed user time, i.e., total elapsed time minus elapsed time for sequential garbage collections; the two times roughly coincide for normal execution.

The basic overhead of debugging can be judged by comparing execution times on a single processor. The comparison varies considerably among benchmarks. Elapsed times for debug recording are 4–18 times slower than for normal execution, and debug replay times are typically about 30% slower than recording, although they are almost the same for MatMult. Omitting garbage-collection times improves relative debugger performance for MaxSpanTree and AllPairs by 15–20%, but makes little difference for the other benchmarks, at least in the single processor case.

Controlled debugger execution under **smld-c** is relatively slower than under **smld-t** (see Section 8.5). We attribute this poor performance primarily to the greater cost of using `var`s and maintaining times as pairs (see Section 10.2). The overhead of recording or replaying fork and synchronization logs also degrades performance, but separate measurements (not shown here) indicate that the cost of logging is small in comparison with the overall cost of instrumentation. We have made no attempt to improve **smld-c**'s instrumentation speed, except to in-line `var` accesses as described in Section 10.2.

Performance on multiple processors is disappointing even for the normal system. Figure 67 shows relative speedups for the different disciplines. Each curve shows performance of a given discipline relative to its own performance on a single processor, *not* relative to the best sequential implementation. These curves indicate that the normal discipline is typically unable to gain any advantage from using more than 3 processors (except MatMult, which can use up to 4).

The most obvious reason for poor speedup is the sequential garbage collector. However, the dotted curves in Figure 67, which represent speedup for elapsed user times without garbage-collection time, show that garbage collection is not the principal problem. An

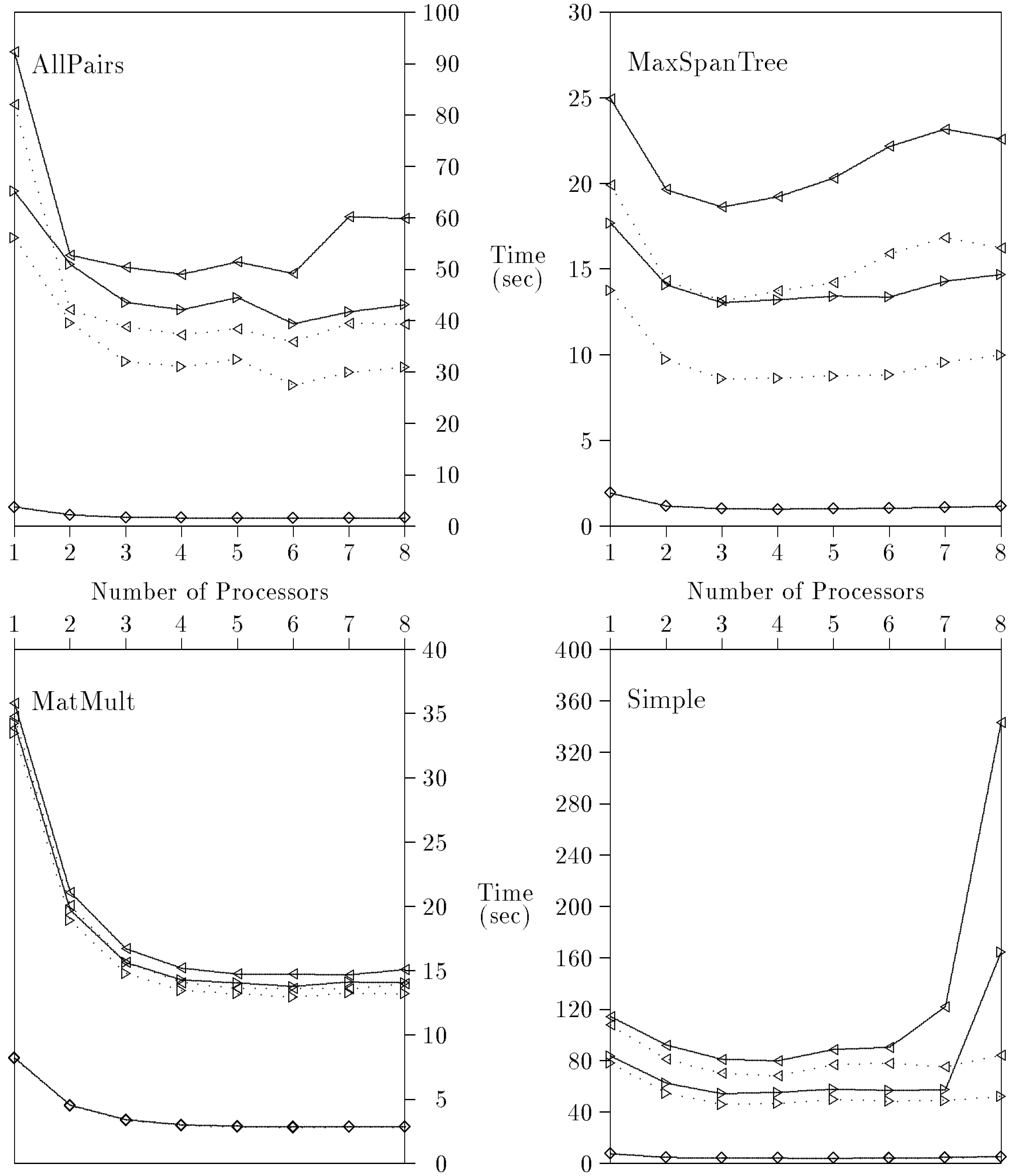

Figure 66: Threads execution times under various disciplines. Total elapsed time (solid line) and elapsed user time (dotted line) are shown for normal execution (⋄), debugger recording (▹), and debugger replay (◃).

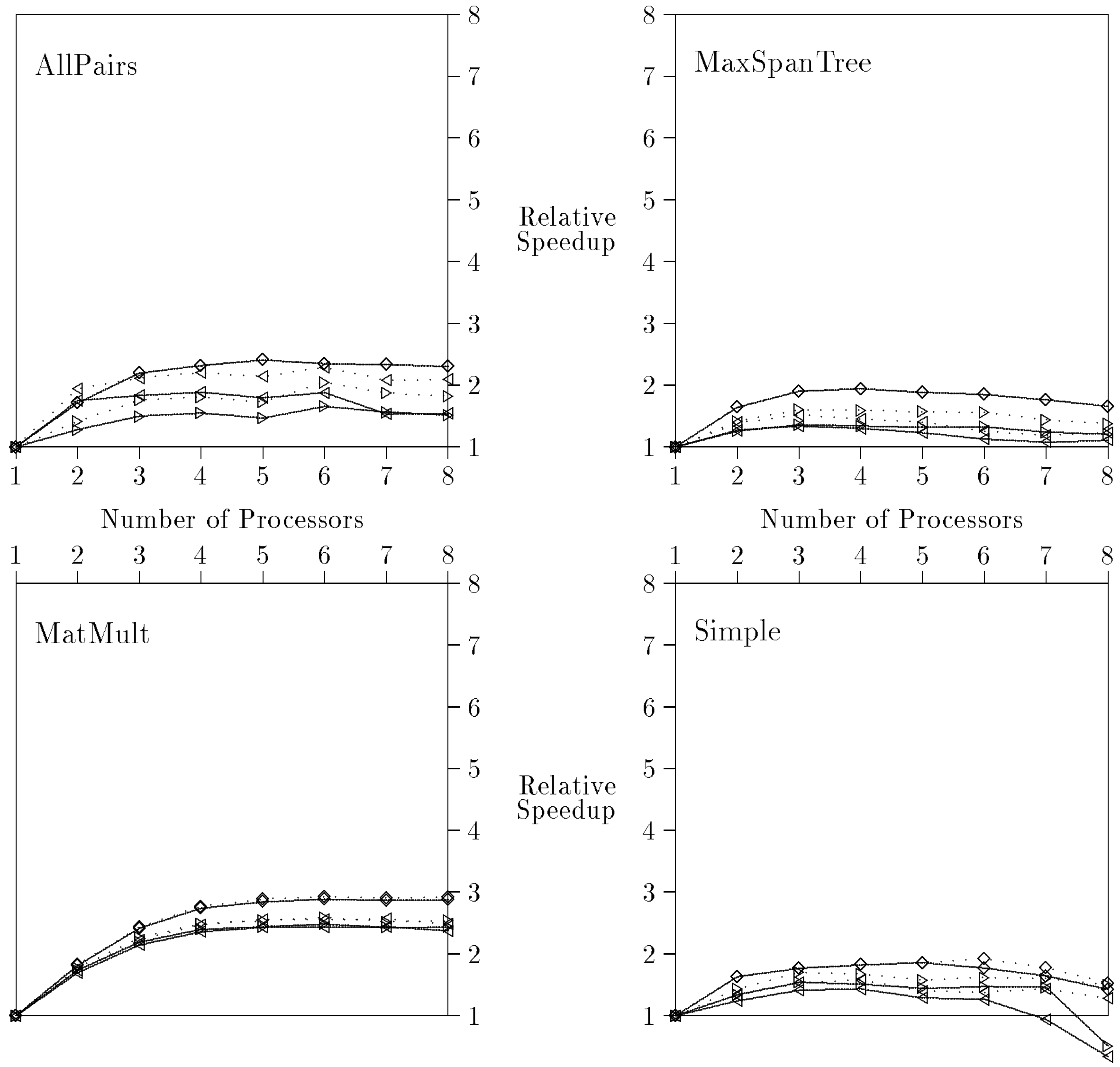

Figure 67: Relative speedup under varying disciplines. Ratios of total elapsed time (solid line) and elapsed user time (dotted line) to single-processor times are shown for normal execution (◇), debugger recording (▷) and debugger replay (◁).

additional overhead caused by synchronizing for collections (not illustrated here) is measurable, but also small.

The most important limitation on speedup may be SML/NJ's heavy allocation rate, typically 7–10 MB/sec on a single processor. This rate is high because SML/NJ uses the heap for all data, including function closures, which a conventional runtime system would allocate on a stack. Most allocation writes miss in both primary and secondary caches, requiring the processor to access main memory. Measurements indicate that, for this hardware, the maximum achievable bandwidth between secondary cache and main memory is 30 MB/sec. For our benchmarks, as the number of processors increases, total rate of allocation by all processors peaks at roughly the point where speedup stops, at 17–28 MB/sec. These figures do not include cache misses on reads and writes to memory that is already allocated, particularly shared memory, which also contribute to the load on the main memory bus and could easily saturate it. Thus, it appears likely that this bus is the primary system bottleneck, although we lack hardware support to test this conjecture.

If **smld-c** succeeded in introducing no additional synchronization overhead, the debugger disciplines might have the same speedup curves as normal execution. In fact, the debugger's curves are somewhat worse than the normal ones, but it is difficult to determine whether this problem is due to increased synchronization overhead or to increased main memory traffic. Although the debugging disciplines allocate more slowly than normal code, they might make more non-allocating memory references and thus saturate the main memory bus sooner.

The memory bus bottleneck could be removed by improving the locality of SML/NJ's allocation strategy or by using a faster bus. To estimate the debugger's speedup in such a setting, we independently measured components of performance that can affect speedup, including garbage-collection time, idle time, context-switching overhead, and lock contention.

The debugging disciplines maintain more live data than normal code, in the form of emulation data structures and logs, so garbage collection takes longer. Since collections are performed sequentially, longer collections reduce speedup; thus, the speedup curves for the debugging disciplines improve slightly when garbage-collection time is disregarded, as shown by the dotted curves in Figure 67. Note that the solid and dotted curves usually coincide for normal execution, which, for these benchmarks, was too brief to require any major collections.

Figure 68 shows the percentage of user time on all processors spent idling while waiting for a runnable thread to be placed on the ready queue. For most benchmarks, this percentage does not differ much between normal and debugger execution; indeed, the debugger disciplines often have more available work than the normal execution. MatMult is the exception: debugging appears to disturb its even load balance.

Thread context switches occur when threads complete, are preempted by timer interrupts, or block on a user-level mutex or condition. We measured the number of thread context switches performed under each discipline; these numbers were too small to contribute significantly to execution time under any of the disciplines, so they are not shown.

Overhead for obtaining user-level mutexes is primarily reflected in hardware spin-lock contention time. Figure 69 shows the approximate percentage of non-idle user time on all processors spent contending for spin locks. These percentages were obtained by counting the number of spins required for each lock acquisition and multiplying by the time for a single spin (about 1μsec). Only spin counts in excess of 100 were recorded, so the displayed figures underestimate contention time. Total times include contention for user-level mutex structures and write-once arrays, and for the thread ready queue; the latter time is also shown separately. For most benchmarks, contention time increases under the debugger disciplines; most of the increase is for accessing user-level structures. The increased contention time indicates that the debugging disciplines do synchronize significantly more often than normal code. Contention in AllPairs and MaxSpanTree increases more than in Simple, and much more than in MatMult. This pattern is consistent with the hypothesis that most of the increased synchronization involves the `threadm` mutex (see Section 10.4), since AllPairs and MaxSpanTree create 40 times more threads per second than Simple, and 240 times more than MatMult. Thus, we might decrease synchronization overhead by further distributing the thread registration mechanism, e.g., by keeping multiple `threadStatuses` lists.

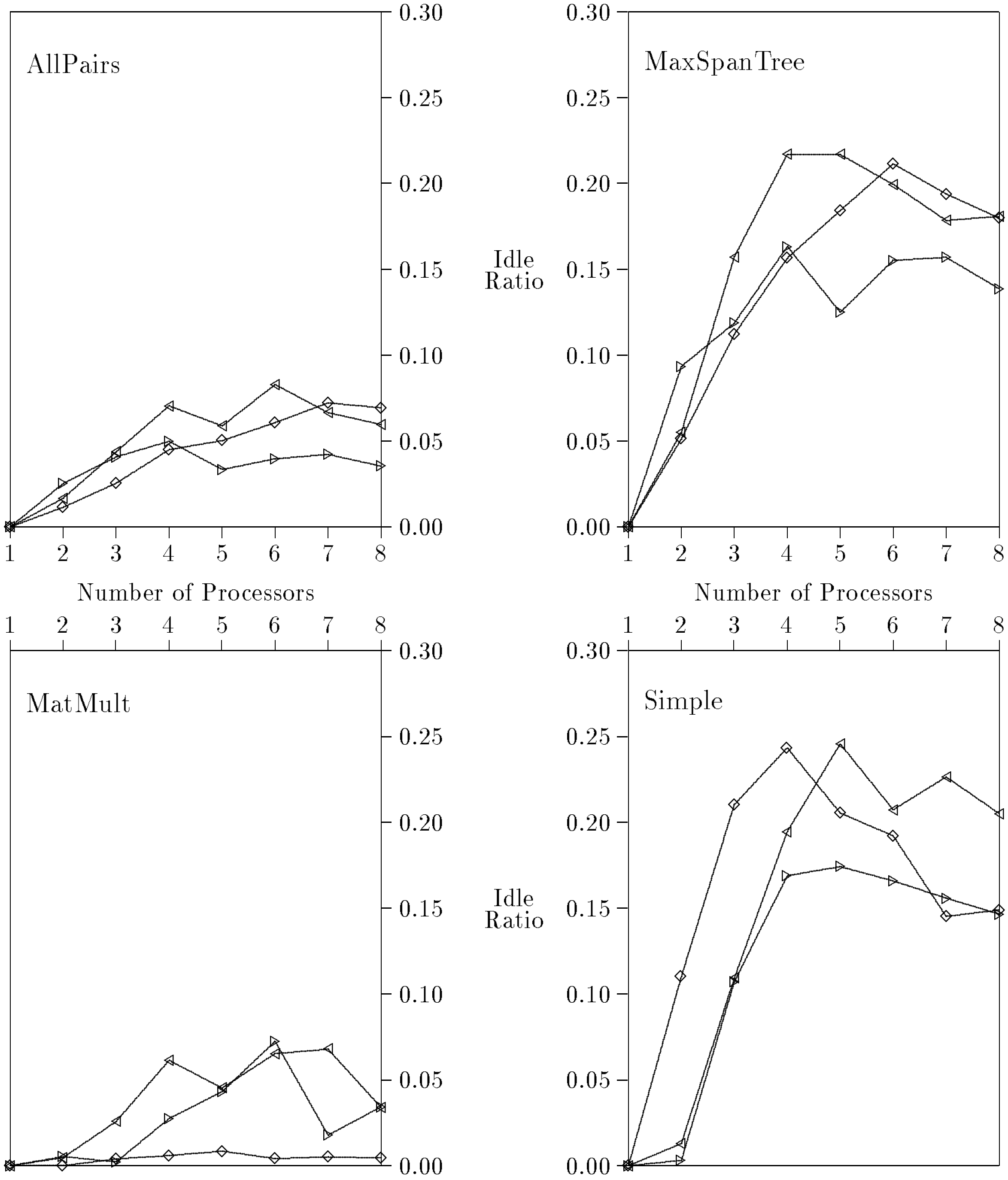

Figure 68: Ratio of idle time to total user processor time (elapsed user time times number of processors) under normal execution (◇), debugger recording (▷) and debugger replay (◁).

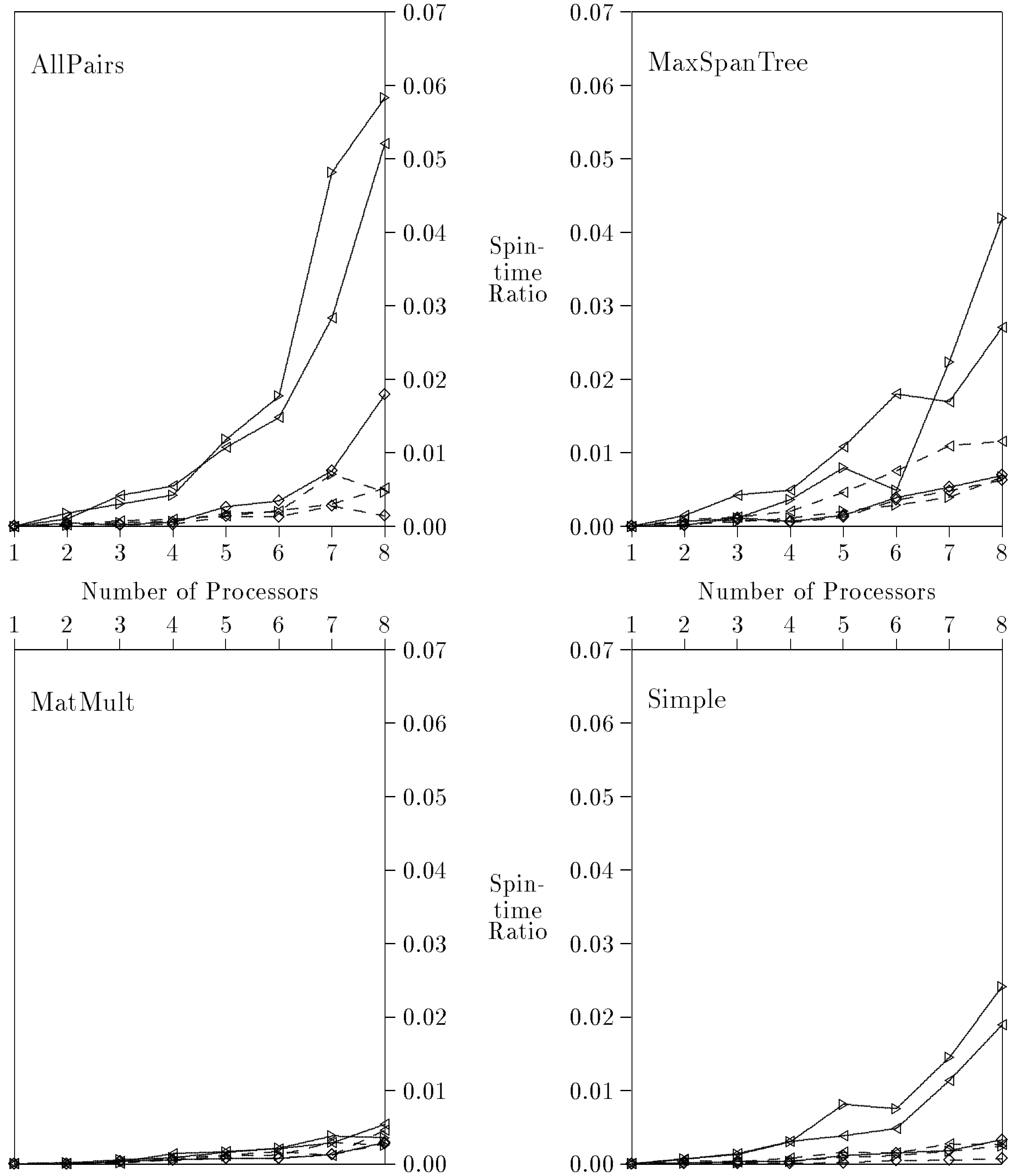

Figure 69: Ratio of total spin-lock delay time (solid line) and ready queue spin-lock delay time (dashed line) to non-idle user processor time (elapsed user time times number of processors less idle time) under normal execution (◇), debugger recording (▷) and debugger replay (◁).

Chapter 11

Conclusions and Future Work

Smld demonstrates that source-code instrumentation can be used to build a simple, portable debugger for a complex language with an optimizing compiler. **Smld-t** demonstrates that time travel, in addition to being a valuable user-level feature, can be used to implement conventional debugger features elegantly and efficiently. **Smld-c** demonstrates that the concept of portable, target-independent debugging "in the source language" can be extended to concurrent languages.

All of the debuggers are independent of the rewriting, optimization, and code-generation phases of the SML/NJ compiler. During the development of this dissertation, these phases have been modified several times, without requiring any changes to the debuggers. The debuggers have also become available on several new target architectures as a result of back-end ports, without changing any debugger code. Currently, the debuggers do 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.

Smld-t, the largest of the debuggers, is about 8,500 lines of ML, making it about 20% of the compiler's size. Supporting debugging is much simpler than compiling because the debuggers are 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 debuggers stems from the complexity of ML itself, rather than from elaborate debugger algorithms.

One important source of simplicity in **smld-t** and **smld-c** is the use of `callcc` to capture most of the program's state at a checkpoint. `callcc` is efficient in SML/NJ, but even if it were slower, it would remain an elegant way to gather checkpoint information without requiring the debugger to understand the compiler's back end. The portability of the debugger would be improved further if the compiler provided an analogous mechanism for capturing the current contents of the mutable store, perhaps as a byproduct of garbage collection.

The debuggers reconstruct the dynamic types of polymorphic variables at runtime, using a lazy mechanism for reconstructing instantiating substitutions for type variables. Dynamic types are formally defined using the structural operational semantics style employed by the ML definition. We are currently developing a formal proof of the soundness of this definition. More efficient methods for reconstructing types from minimal runtime information and for integrating the reconstruction process more cleanly into the compiler are also subjects for further research.

Instrumented code is not quite standard ML because it violates the typing rules when passing program values to the debugger. This kind of type violation seems inevitable when writing "metaprograms" under ML-style polymorphism. Fortunately, the instrumented code itself does not compute with values whose types have been coerced, so the program's type-safety is unaffected. Adding instrumentation should not alter the behavior (or, for **smld-c**, the set of possible behaviors) of the program being debugged. Our debuggers appear to meet this condition, but a formal demonstration of behavioral equivalence between normal and instrumented programs would be useful, especially for **smld-c**.

Performance of the sequential debuggers is adequate. Execution speed under debugger control is typically about three times slower than normal execution for **smld-t** and five times slower for **smld**; these ratios are competitive with the ratios we estimate for hypothetical machine-based debuggers and much lower than for interpretation. Debugger-controlled execution generates more live data than normal execution. **Smld** can require an arbitrary amount of additional live data in order to hold event records; **smld-t** avoids this problem by generating event records on demand. On the other hand, **smld-t** requires an indefinite amount of space for logs and uses any available memory to hold checkpoints; however, it can continue to operate at slower speed even when few checkpoints are cached. The benchmarks we measured required three to eight times normal memory to run "comfortably" under **smld-t**.

The most significant drawback to using the debuggers 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.

Smld-c is considerably less developed than the other debuggers, and many of its ideas remain to be tested on practical applications. The fork-based temporal model may prove to be too clumsy or too uninformative for practical use. A possible alternative would be to use vector times (see Section 9.6.2), if we can find an efficient way to compute them and a compact way to specify them.

Performance of the **smld-c** prototype is less satisfactory than that of the sequential debuggers. Preliminary measurements, taken before any significant performance tuning, indicate that debugger-controlled execution is an order of magnitude slower than normal execution. The costs of synchronization logging are insignificant compared with the overall effect of instrumentation, so we cannot measure whether our strategies to minimize log entries save time (although they do save space), or compare **smld-c**'s logging performance with that of other systems. Indirect measurements indicate a substantial increase in synchronization overhead under debugger execution, but major improvements to the underlying ML Threads implementation will be needed before the performance of **smld-c** can be addressed in detail.

The log-based approach to replaying non-deterministic programs, which **smld-c** shares with many other parallel debuggers, is clearly unsatisfactory. Synchronization logs can quickly grow very large, and they must be kept for the duration of program execution even for threads that do not otherwise appear in checkpoints; they need to be kept even longer (and written into an external format) if the user wants to preserve a particular execution sequence for debugging at a later time. Moreover, because of the intrusiveness of **smld-c**'s instrumentation, debugged execution may fail to capture the faulty aspects of the program's synchronization behavior.

An alternative approach would be to make programs behave deterministically on their initial run. One simple way to force determinism would be for the **smld-c** library to tag each synchronization operation with the s-time of the current thread (see Section 9.2), and to guarantee that operations on each mutable location are executed in tag order (an ordering on thread identifiers can be used to break ties). The execution could be made more realistic by integrating an explicit scheduler into the library and adjusting the tag times accordingly. By varying the scheduler's parameters, the user could search for faulty behavior under different execution orderings. This approach would be a form of discrete event simulation. To execute such a simulation on a multiprocessor, we could apply optimistic distributed simulation techniques [50, 67], which would be particularly appropriate for **smld-c** because they rely on reverse execution to undo execution when over-optimism leads to a violation of the simulation's ordering constraints. This kind of reverse execution also requires synchronization logging, but old log entries can be discarded after a bounded time. A simulation can be reproduced simply by remembering the values of its parameters. The key question to be settled by experiment is whether the optimistic simulation approach is efficient enough to be practical.

All the debuggers should provide better support for modular programming. 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 total side-effect of the function rather than its internal, individual side-effects. The implementation of futures in terms of ML Threads primitives (see Figure 48) exemplifies both these motivations. The user wants to treat `future` and `touch` as atomic functions; the debugger could avoid instrumenting their internals and could log their synchronization effects atomically.

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. This facility would be most useful if the status of a given unit could be switched dynamically, but such a feature would prevent the debugger's taking advantage of trusted status to avoid instrumenting, unless the compiler were extended to support dynamic recompilation of selected functions. 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 programmers to specify the net side-effects of trustworthy units is a harder problem, but it represents the key to making the debugger genuinely extensible. One interesting application of extensibility would be to add support for other concurrent extensions to ML, such as CML [99], by implementing debugger versions of the primitive concurrency operations (`callcc`, processor control, spin-lock operations, etc.) and using them to build debugging support for user-level concurrency operators.

The techniques applied to ML in this dissertation should also work for other languages. Instrumentation is a suitable debugging method for any type-safe language, particularly when the compiler makes significant transformations of the source code. Time travel can be implemented in any language, with an appropriate choice of checkpointing mechanism.

Bibliography

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, Apr. 1991.

[2] S. V. Adve and M. D. Hill. Weak ordering–a new definition. In *Proc. 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[3] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. Netzer. Detecting data races on weak memory systems. In *Proc. 18th Annual International Symposium on Computer Architecture*, pages 234–243, May 1991.

[4] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Efficient debugging with slicing and backtracking. Technical Report SERC-TR-80-P, Software Engineering Research Center, Dept. of Computer Sciences, Purdue University, Oct. 1990.

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

[6] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, June 1990. Published as *SIGPLAN Notices*, 25(6), June 1990.

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

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

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

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

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

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

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

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

[15] D. F. Bacon and S. C. Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 194–206, May 1991. Published as *SIGPLAN Notices* 26(12), Dec. 1991.

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

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

[18] A. D. Birrell. An introduction to programming with threads. Research Report 35, DEC Systems Research Center, Palo Alto, CA, Jan. 1989.

[19] A. D. Birrell, J. V. Guttag, J. J. Horning, and R. Levin. Synchronization primitives for a multiprocessor: A formal specification. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 94–101, Nov. 1987. Published as *Operating Systems Review,* 21(5), Nov. 1987.

[20] D. Bowen, L. Byrd, F. Pereira, L. Pereira, and D. Warren. *Prolog-20 User's Manual*, Feb. 1984.

[21] B. Bruegge. *Adaptability and Portability of Symbolic Debuggers.* PhD thesis, Carnegie Mellon University, Sept. 1985. Also Carnegie Mellon Univ. Dept. of Computer Science Tech. Report CMU-CS-85-174.

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

[23] L. Cardelli. Compiling a functional language. In *Proc. 1984 ACM Conference on Lisp and Functional Programming*, pages 208–217, Aug. 1984.

[24] T. A. Cargill and B. N. Locanthi. 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, Oct. 1987.

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

[26] J. Choi. *Parallel Program Debugging with Flowback Analysis.* PhD thesis, University of Wisconsin–Madison, Aug. 1989. Also Univ. of Wisconsin–Madison Computer Sciences Tech. Report 871.

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

[28] J.-D. Choi and J. M. Stone. Balancing runtime and replay costs in a trace-and-replay system. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 26–35, May 1991. Published as *SIGPLAN Notices* 26(12), Dec. 1991.

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

[30] Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA. *dbx – debugger (UNIX User's Manual).*

[31] E. C. Cooper and R. P. Draves. C threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1988.

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

[33] D. S. Coutant, S. Meloy, and M. Ruscetta. 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, June 1988. Published as *SIGPLAN Notices*, 23(7), July 1988.

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

[35] R. Curtis and L. Wittie. BugNet: a debugging system for parallel programming environments. In *Proc. 3rd International Conference on Distributed Computing Systems*, pages 394–399, Oct. 1982.

[36] L. Damas and R. Milner. Principal type-schemes for functional languages. In *Proc. Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[37] A. Di Maio, S. Ceri, and S. C. Reghizzi. Execution monitoring and debugging tool for Ada using relational algebra. In J. G. P. Barnes and J. Gerald A. Fisher, editors, *Ada in Use: Proceeding of the Ada International Conference 1985*, pages 109–123. Cambridge University Press, Cambridge, 1985.

[38] A. Dinning. *Detecting Nondeterminism in Shared Memory Parallel Programs.* PhD thesis, New York University, Nov. 1990. Also N.Y.U. Dept. of Computer Science Tech. Report 526.

[39] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proc. 2nd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 1–10, Mar. 1990. Published as *SIGPLAN Notices*, 25(3), Mar. 1990.

[40] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96, May 1991. Published as *SIGPLAN Notices* 26(12), Dec. 1991.

[41] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Proc. 18th ACM Symposium on Theory of Computing*, pages 109–121, May 1986.

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

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

[44] K. Ekanadham and Arvind. SIMPLE: An exercise in future scientific programming. Technical Report Computation Structures Group Memo 273, MIT, Cambridge, MA, July 1987. Also IBM/T.J. Watson Research Center Research Report 12686, Yorktown Heights, NY.

[45] B. Ezzell. *Using Turbo Debugger and Tools 2.0.* Addison-Wesley, Reading, MA, 1991.

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

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

[48] C. J. Fidge. Partial orders for parallel debugging. In *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 183–194, May 1988. Published as *SIGPLAN Notices,* 24(1), Jan. 1989.

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

[50] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

[51] R. S. Gaines. *The Debugging of Computer Programs.* PhD thesis, Princeton University, Aug. 1969.

[52] J. Gait. A debugger for concurrent programs. *Software—Practice and Experience*, 15(6):539–554, June 1985.

[53] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

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

[55] A. P. Goldberg, A. Gopal, A. Lowry, and R. Strom. Restoring consistent global states of distributed computations. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 144–154, May 1991. Published as *SIGPLAN Notices* 26(12), Dec. 1991.

[56] B. Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 165–176, June 1991. Published as *SIGPLAN Notices* 26(6), June 1991.

[57] G. S. Goldszmidt, S. Katz, and S. Yemini. Interactive blackbox debugging for concurrent languages. In *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 271–282, May 1988. Published as *SIGPLAN Notices,* 24(1), Jan. 1989.

[58] R. Grishman. The debugging system AIDS. In *Proceedings AFIPS 1970 Spring Joint Computer Conference*, volume 36, pages 59–64, Montvale, NJ, May 1970. AFIPS Press.

[59] R. E. Griswold. A portable diagnostic facility for SNOBOL4. *Software—Practice and Experience*, 5(1):93–104, Jan.–Mar. 1975.

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

[61] R. H. Halstead, Jr. and D. A. Kranz. A replay mechanism for mostly functional parallel programs. Technical Report CRL 90/6, DEC Cambridge Research Lab, Nov. 1990.

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

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

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

[65] C. A. R. Hoare. Monitors: An operating system structuring concept. *Comm. ACM*, 17(10):549–557, Oct. 1974.

[66] P. Hudak and P. Wadler. 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, Apr. 1990.

[67] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[68] G. F. Johnson and D. Duggan. 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, Jan. 1988.

[69] M. S. Johnson, editor. *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Pacific Grove, CA, Mar. 1983. Published as *SIGPLAN Notices*, 18(8), Aug. 1983.

[70] S. N. Kamin. *Programming Languages: An Interpreter-Based Approach.* Addison-Wesley, Reading, MA, 1990.

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

[72] A. Kishon, P. Hudak, and C. Consel. 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, June 1991. Published as *SIGPLAN Notices*, 26(6), June 1991.

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

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

[75] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

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

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

[78] C. H. LeDoux and J. D. Stott Parker. Saving traces for Ada debugging. In J. G. P. Barnes and J. Gerald A. Fisher, editors, *Ada in Use: Proceeding of the Ada International Conference 1985*, pages 97–108. Cambridge University Press, Cambridge, 1985.

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

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

[81] A. D. Malony and D. A. Reed. Models for performance perturbation analysis. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 15–25, May 1991. Published as *SIGPLAN Notices* 26(12), Dec. 1991.

[82] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, Dec. 1989.

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

[84] J. M. Mellor-Crummey. *Debugging and Analysis of Large-Scale Parallel Programs.* PhD thesis, University of Rochester, Sept. 1989. Also Univ. of Rochester Computer Science Tech. Report 312.

[85] B. P. Miller and T. J. Leblanc, editors. *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988. Published as *SIGPLAN Notices,* 24(1), Jan. 1989.

[86] B. P. Miller and C. McDowell, editors. *Proc. ACM/ONR Workshop on Parallel and Distributed Computing*, May 1991. Published as *SIGPLAN Notices* 26(12), Dec. 1991.

[87] R. Milner and M. Tofte. *Commentary on Standard ML.* MIT Press, Cambridge, MA, 1990.

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

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

[90] G. Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[91] R. H. B. Netzer. *Race Condition Detection for Debugging Shared-Memory Parallel Programs.* PhD thesis, University of Wisconsin–Madison, Aug. 1991. Also Univ. of Wisconsin–Madison Computer Sciences Dept. Tech. Report 1039.

[92] J. T. O'Donnell and C. V. Hall. Debugging in applicative languages. *Lisp and Symbolic Computation*, 1(2):113–145, Sept. 1988.

[93] J. L. Ophel. *AIMLESS: A Programming Environment for ML.* PhD thesis, The Australian National University, Jan. 1991.

[94] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel debuggers. In *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124–129, May 1988. Published as *SIGPLAN Notices,* 24(1), Jan. 1989.

[95] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, Cambridge, 1991.

[96] Projet Formel, INRIA-ENS. *CAML Reference Manual (Version 2.3)*, June 1987.

[97] C. Reade. *Elements of Functional Programming.* Addison-Wesley, Reading, MA, 1989.

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

[99] J. H. Reppy. CML: a higher-order concurrent language. In *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 293–305, June 1991. Published as *SIGPLAN Notices* 26(6), June 1991.

[100] E. H. Satterthwaite, Jr. *Source Language Debugging Tools.* PhD thesis, Stanford University, May 1975.

[101] E. Schonberg. On-the-fly detection of access anomalies. In *Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 285–297, June 1989. Published as *SIGPLAN Notices*, 24(7), July 1989.

[102] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, MA, 1983.

[103] Silicon Graphics, Inc. POWER series technical report, June 1991.

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

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

[106] G. L. Steele Jr. Making asynchronous parallelism safe for the world. In *Proc. 17th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 218–233, Jan. 1990.

[107] J. L. Steffen. Experience with a portable debugging tool. *Software—Practice and Experience*, 14(4):323–334, Apr. 1984.

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

[109] W. Teitelman. *Interlisp Reference Manual.* Xerox Palo Alto Research Center, 1978.

[110] M. Tofte. *Operational Semantics and Polymophic Type Inference.* PhD thesis, University of Edinburgh, 1987.

[111] A. P. Tolmach and A. W. Appel. Debugging Standard ML without reverse engineering. In *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, pages 1–12, June 1990. Also Princeton Univ. Dept. of Computer Science Tech. Rep. CS-TR-253-90.

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

[113] D. A. Turner. 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, 1985.

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

[115] D. Wall, A. Srivastava, and F. Templin. A note on Hennessy's "Symbolic debugging of optimized code". *ACM Transactions on Programming Languages and Systems*, 7(1):176–181, Jan. 1985.

[116] M. Wand. Continuation-based multiprocessing. In *Proc. 1980 LISP Conference*, pages 19–28, Aug. 1980.

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

[118] M. Zelkowitz. *Reversible Execution as a Diagnostic Tool.* PhD thesis, Cornell University, Jan. 1971.

[119] M. Zelkowitz. Reversible execution. *Communications of the ACM*, 16(9):566, Sept. 1973.

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

[121] L. W. Zurawski and R. E. Johnson. Debugging optimized code with expected behavior. *ACM Transactions on Programming Languages and Systems*, 1992. To appear.