Portable Debugging and Profiling

Mikael Pettersson*

INRIA Sophia Antipolis, France
E-mail: mikpe@sophia.inria.fr

Abstract. This paper addresses the problem of implementing portable debuggers for compiled or translator-based language implementations. We describe a general strategy based on viewing application execution as event generation, and debugging as event processing. The implementation approach relies on instrumentation of the compiler's intermediate code. We give examples of portable and efficient implementations of several common debugging primitives, and also show how profiling support can be added using similar ideas.

1 Introduction

Support for debugging and profiling is an important quality-of-implementation issue for most programming languages. Even very high level languages, executable specification languages, and application domain specific languages need varying forms of such support.

Traditionally, debugging mechanisms such as breakpoints, single-stepping, and catching hardware faults, have been implemented primarily using special hardware and operating system facilities. For source-level debugging, compilers have been relied upon to provide the mapping between source-level objects and their machine-level counterparts, usually by generating less optimized code and additional system-dependent symbol tables (e.g. "stabs") in object modules. Thus, debugging support has required significant amounts of non-portable code in both debuggers and compilers.

This is a problem when portability is desired, in debuggers, compilers, or both. Another problem is that the traditional debugging mechanisms may be inadequate for effective debugging of very high level languages.

In this paper we address the problem of implementing portable debuggers for compiled or translator-based language implementations. After outlining some possible implementation strategies, we describe a general strategy based on viewing application execution as event generation, and debugging as event processing. The implementation approach relies on instrumentation of the compiler's intermediate code. We give examples of portable and efficient implementations of several common debugging primitives, and also show how profiling support can be added using similar ideas. Related work is discussed.

* Visiting researcher, on leave from Linköping University, Sweden. This work was supported by a grant from the Swedish Research Council for Engineering Sciences, TFR.

There are several papers on *specific* implementations of more-or-less portable debuggers in the literature. We believe our contribution to be the amalgamation of several ideas and techniques spread throughout the literature, augmented with our improvements:

- A general event-based model of execution and debugging.
- An intermediate-code level instrumentation approach to compiler support. By performing the instrumentation at an early stage in the compiler, the back-end can remain mostly ignorant of the debugger. Code optimizations do not prevent precise debugging.
- A collection of debugging primitives with portable implementations. Compared to some previous work [6, 5], we emphasize both portability and efficiency, and describe generalizations for richer languages and higher-level debugging primitives.

2 Approaches to debugging support

2.1 Native debugging

In this approach the compiler generates system-dependent debugger symbol tables (e.g. Unix "stabs"), and the the debugger uses system-dependent debuggee-control mechanisms (e.g. the Unix `ptrace` system call or `/proc` file-system). The advantages include improved performance for simple debugger actions, such as breakpoints, and the possibility of perhaps using the system's debugger instead of writing a new one. The disadvantages include: non-portable code in the compiler to generate system-specific data structures, non-portable and machine-specific code in the debugger, difficulties debugging optimized code, and difficulties bridging the semantic gap between a high-level language and the limited low-level debugging features normally supported.

2.2 Wrapper on top of existing debugger

A native debugger is used to control the debuggee. Commands are translated to commands for the native debugger, and its responses are translated back. There is less porting effort to write the debugger wrapper, but it becomes difficult to implement features that the underlying debugger does not support.

The compiler can either generate machine code with system-dependent symbol tables, or it can translate via a lower-level language, like C, and compile that code with debugger support. A problem is that additional data structures have to be generated to explain (to the debugger wrapper) the relationships between the high-level language's concepts and those of the intermediate code.

2.3 Our approach: code instrumentation

The compiler instruments its intermediate code with data structures, polling code, and calls to a debugging monitor. The debugger executes either in the monitor itself, or as a separate process that communicates with the monitor.

1. The features (e.g. breakpoints) and constraints (e.g. compatibility between debugged and non-debugged code) of the debugger are determined.
2. Runtime events that should be monitored to support the features are identified, as are the code sites where those events can occur.
   Thus, the program generates potentially interesting events at certain sites, and the debugger processes these events, performing suitable actions as a result. The application and debugger monitor are assumed to execute as coroutines rather than as concurrent processes.
3. The compiler is modified to instrument its code to perform the necessary event generation at the possible event sites, and to emit supporting data structures, subject to the design constraints.
   This is the most complex step, the details of which depend heavily on the source language and the chosen debugger features. Our main focus in this paper is the realization of standard debugger features for sequential languages with notions like procedures, variables, and call stacks.

3 Mechanisms

In this section we discuss several low-level debugging mechanisms. Our emphasis is on efficient portable implementations; to this end, we occasionally use C to describe data structures and instrumentation code.

3.1 Breakpoints

To support breakpoints, a debugger must be able to map from a source-code position to the corresponding event site in the code. Then it must instrument that site to invoke the debugger when control reaches it. Traditionally, debuggers have mapped a source-code statement line number to the first instruction of that statement, and then replaced that instruction either by a trap instruction or by an out-of-line call to a debugger hook.

A portable debugger cannot actually change the code dynamically; what it *can* do is to have the compiler generate instrumentation code that *polls* the breakpoint condition for that site, and if true invokes the debugger.

Position-to-site map. For simple languages, positions might just be source-code line numbers, and sites be placed at the start of statements.

For a more fine-grained and flexible solution, the HLL compiler can record the character position of every newline in an array, and annotate every syntax tree node with its source-code *region*, i.e. start and stop character positions. Assuming debugger instrumentation takes place at the syntax-tree level, every event site is associated with the corresponding source-code region. For every site, the compiler generates a static *site descriptor* that includes the region information, and all descriptors are also placed in an array or list so they can be enumerated. The newline array and the set of site descriptors are inserted into the generated code, as static immutable data objects.

In a debugger session, the user inputs a source code position to specify a breakpoint location, and the debugger searches the set of sites for a site whose region includes the indicated position. Inversely, given a site, its region information plus the newline array allows the debugger to highlight the corresponding source region.

Polling and suspending. To implement the polling code it suffices to associate a breakpoint flag with every site. The instrumentation code calls the debugger library with its site number as a parameter; this corresponds to generating a "reached site $\#i$" event. The debug monitor checks the breakpoint flag: if set, the debugger is invoked, otherwise it returns immediately to the debuggee.

For performance reasons, since most sites will not have breakpoints most of the time, the test and possible invokation of the debugger should be inlined in the instrumentation code. This saves the overhead of many procedure calls.

The breakpoint flag can be placed in the site's descriptor. This makes the flag easy to find from the debugger, but has the disadvantage of making the entire descriptor mutable. (Since this data structure is included in the debuggee's address space, we would like it to be immutable, in order to protect it from pointer/array bugs.) Making the flag a separate variable allows the site descriptors to remain immutable, but causes a slight performance problem; with separate flags, the breakpoint test code at some site i becomes:

```
if( flag_i ) generate_event(&descriptor_i);
```

Since the flag is separate from the descriptor, the compiler has to emit additional code to compute a new global address when the actual breakpoint occurs. By placing the flags in one array and the descriptors in another, both indexed by site number, the code can be changed to:

```
if( flag[i] ) generate_event(&flag[i]);
```

The debugger uses the flag pointer to derive its index in the flag array, and then uses that index to locate the corresponding descriptor.

If the language has separate compilation, i.e. modules, then there is likely to be one flag array and one site descriptor array per module. As described later, every procedure starts by pushing a new shadow stack frame on a shadow stack, and the shadow frame contains a pointer to the static descriptor for that procedure. We let the procedure's descriptor contain a pointer to its module's descriptor, which in turn points to the flags and site descriptors for the module.

Since the debuggee suspends itself by making a recursive procedure call to the debugger monitor, returning from the monitor resumes the debuggee.

In the cdb [5] debugger the compiler emits a "coordinate" word for each site, that includes a 16-bit line number, a 10-bit line offset, and a one-bit breakpoint flag positioned in the sign bit. The instrumentation code tests if the word is negative, and if so, invokes the debugger with the *site number* and *current scope descriptor* as parameters. (These parameters being new values, additional code is generated to compute them.)

Heymann's debugger [6] uses source lines to identify sites. The compiler emits a *breakpoint array*, BPA, with one byte per source line. The instrumentation code for an executable source line stores its site number in a globally-accessible variable, tests its BPA entry, and invokes the debugger (with no parameters) if the value is non-zero. BPA entries that do not correspond to executable source-code lines are pre-loaded with a special marker, to prevent breakpoints from being planted there. The site number is stored unconditionally because asynchronous faults (e.g. pointer errors or division by zero) are caught by Unix signal handlers, which then produce a stack dump.

3.2 Single-stepping

Single-stepping is like breakpointing, except that every breakpoint site should be triggered. A global flag might be added, but this would cause code growth and slowdown; a better solution is to reuse the existing breakpoint mechanism to force breakpoints at relevant sites.

Since each breakpoint flag is usually a byte or an integer for fast access, three different flag values can be used: '0' for no breakpoint, '1' for a real breakpoint, and '2' for a single-step breakpoint. To set single-step mode, all '0' entries are changed to '2'; to leave single-step mode, all '2' entries are reset to '0'.

Since there may be very many breakpoint sites in a complete application, changing to and from single-step mode can be slow. A more fine-grained implementation is possible if every procedure descriptor includes a list of all its breakpoint sites. To enter single-step mode, only the breakpoints in the current procedure are set. Before a procedure exit, the single-step breakpoints are removed from the current procedure, and the breakpoints of the returned-to procedure are set.

Procedure call sites are more challenging: the called procedure may not have been compiled with debugging enabled, but we still want a breakpoint if *it* in turn invokes a debug-enabled procedure; the called procedure may also be a runtime value whose descriptor is unknown at the call site. One solution is to set a global flag indicating single-step mode. As a debug-enabled procedure is entered, it calls the debugger to register its shadow stack record, dynamic scope record, and static procedure and scope descriptors. (This is described later.) The debugger library then sets every breakpoint in the entered procedure if the single-step flag is set.

Heymann's debugger [6] uses all these techniques. The cdb debugger [5] supports breakpoints but not single-stepping. The authors suggest that either a control-flow graph of breakpoints be generated, so that only a few need to be set between each single-step break, or that machine-dependent means be used, but that is non-portable and requires a detailed map between high-level site locations and their corresponding machine instructions.

Step-Over. Step-over is like single-step, except that recursively invoked procedures are not executed in single-step mode. This is easy with the mechanism described here: just clear the single-step flag before making a recursive call.

3.3 Inspecting the call stack

Being able to traverse the call stack, and inspect the state of individual suspended procedure invokations, is a useful standard debugger feature. But the standard implementation techniques are inappropriate for portable debuggers or debuggers for certain kinds of high-level languages. We consider two cases:

1. Source-level procedures and recursions are mapped to target-level procedures and recursions. Since almost no language above the level of assembly code allows programs to do detailed inspection of activation records or call stacks, any such access would have to be made at the machine-level, and be non-portable.
2. Implementations of languages with tailcalls, backtracking, coroutines, or iterators, often choose highly specialized representations for procedures and recursions. In some cases, suspended invokations are represented using *concrete* continuations, and (non-local) control flow using tailcalls [13]. Continuations are created, manipulated, and invoked *explicitly*, instead of relying on C or machine-level procedure boundaries, calls, and returns. The machine stack is either not used at all, or is used in non-standard ways. Compilers often employ aggressive inlining and other code transformations, so source-level procedure boundaries do not necessarily correspond meaningfully to C or machine-level procedure boundaries [11].

In both cases, machine stacks and activation records are either inaccessible, or have no obvious meaning. The suggested solution is to maintain explicit *shadow* stacks and *shadow* activation records that present a *portable* view of the chain of procedure invokations.

Shadow activation records. A shadow activation record should include the following fields:

- A *dynamic link* to the shadow activation record of the previous procedure, to allow traversing the shadow stack. This link might be implicit if records are always located at fixed offsets from each other.
- A *static descriptor link* to the complete static information about the invoked procedure, e.g. name, return type, argument types, list of breakpoint sites, the module it is defined in, etc.
- A *call site link* to allow stack backtraces to locate the descriptor for the site where a procedure was suspended in a recursive call.
- If the language has nested procedure declarations, then there should be a *static link* to the shadow activation record of the most recent invokation of the lexically enclosing procedure.
  Normally, compilers choose between static links or displays to provide access to enclosing scopes, based on overheads, calling conventions, and costs for non-local variable accesses. However:
  1. The debugger may walk up and down the stack, accessing different activation records that occur *at the same scope depth.* A global display alone does not work.

2. If debug-enabled and non-debug-enabled code should be compatible, then the calling conventions cannot in general be changed to pass "debugger" static links as additional parameters, or the representation of procedure arguments be similarly changed.

Our suggestion is to use a shadow display of pointers to shadow activation records, but only for parameter passing. On entry to a nested procedure, it sets its shadow static link by indexing the shadow display.

Maintaining the shadow stack. If the source language or its implementation is *not* properly tail-recursive, then maintaining the shadow stack is simple. A global *shadow stack top* variable always points to the top-most shadow activation record. Source level procedure entries and exits are events: on entry, a new shadow activation record is allocated and initialized, and the shadow stack top is set to refer to the new record; on exit, the shadow stack top is set to the value of the dynamic link in the top-most activation record. A procedure call is an event site; just before the call, the instrumentation code updates the call site link in its shadow activation record to refer to the site's descriptor. (The implementation does become more involved in the presence of non-local exists, though.)

Allocating the shadow activation records as local stack variables in the generated C or machine-level code is simple and fast. The disadvantage is that they become vulnerable to unchecked out-of-bounds array indexing errors. Keeping them in a separate region of the address space should reduce, but not eliminate, the risks of data corruption. (We ignore the possibility of keeping *all* debugger state in separate processes, or even on separate machines, due to the high runtime costs.)

Tailcalls. In the presence of tailcalls, a caller's activation record must be removed before the callee's activation record is pushed. This can be done by marking tailcalls as special events whose instrumentation code perform the necessary pop before the actual tailcall.[1]

3.4 Inspecting variables in scope

To display variables, the debugger must first map a suspended site to the set of variables visible there, their types, and locations, and then access their values. If local variables are assigned to registers, and if extensive optimizations are permitted, then accessing these values from the debugger may be impossible (e.g. when a dead variable's register has been reassigned), difficult (e.g. when a variable is replaced by a derived value), or simply highly machine dependent (e.g. when a variable is assigned to a callee-save register, we have to traverse the

[1] The `psd` Scheme debugger [9] usually preserves tail-recursiveness, except when in single-step mode. In the `smld` SML debugger [14, 15], application code remains tail-recursive, but the debugger's shadow activation record is *not* deallocated at a tailcall.

stack to see where and if it has been saved). Traditional debuggers either require that optimizations be turned off, so that local variables are stored in memory, or become unable to inspect variables in optimized procedures.

If local variables are allocated in memory and their addresses are exported to the debugger, the access problems disappear. Optimizations such as assigning temporaries to registers or scheduling instructions, need not be disabled.

For each procedure, all its local variables are collected in a single *dynamic scope record.* A *static scope descriptor* is built, in which every local variable has a *local variable descriptor* describing its name, type, and memory location *relative the dynamic scope record.* The instrumentation code at a procedure's entry event registers the address of its dynamic scope record and static scope descriptor with the debugger. For example, the Pascal procedure

```
procedure proc;
var x: integer, y: real;
begin ... end;
```

could be instrumented as follows in C:[2]

```
struct local_var_desc {
    char *name;
    TYPE type;
    size_t offset; /* from start of dynamic scope record */
} local_var_desc;
struct proc_dynamic_scope { int x; double y; };
const struct local_var_desc proc_static_desc[2] = {
    { "x", TYPE_INT, offsetof(proc_dynamic_scope,x) },
    { "y", TYPE_REAL, offsetof(proc_dynamic_scope,y) }
};
void proc(void)
{   struct proc_dynamic_scope locals;
    register_locals(2, proc_static_desc, &locals);
    /* the procedure body uses locals.x and locals.y */
}
```

(In reality the dynamic scope record would be combined with the shadow stack frame, and there would be only one event site at procedure entries.)

The cdb debugger [5] has a static "symbol" record for every local variable. On entry to a procedure, *every* local variable has its address taken, and the difference between it and the procedure's shadow activation record (a generated local variable) is stored in the variable's "symbol" record. This forces local variables to memory, but incurs continuous runtime overhead.

The compiler/debugger by Heymann [6] uses a similar approach, but performs the registration only once. At program startup, the debugger sets a global flag and calls every procedure; a called procedure will, when this flag is set,

[2] `offsetof(`*type*`,`*name*`)` is an ANSI-C macro whose compile-time value is the byte offset of the *name* field in *type* records [1, Section 4.1.5].

bypass its normal body and instead generate debugger information for every local variable, with its name, type, and address offset from the shadow activation record. However, this scheme is *broken* by modern C compilers. The code only takes the addresses of the local variables when executing the registration code; during normal execution, there is no control-flow path in which the locals have their addresses exported. Hence, a good C compiler will allocate the locals to registers in the real procedure body. We found that several compilers would break the scheme, unless invoked with no or low optimizations. Furthermore, since the debug flag is always tested, normal execution is still slowed down.

Both Hanson/Raghavachari's and Heymann's schemes use pointer subtraction between the addresses of *different* objects, something the ANSI-C standard [1, Section 3.3.6] marks as undefined. By placing all locals in a single record, our scheme is well-defined.

Nested scopes. If the language has nested scopes, a.k.a. blocks, then the scope information changes *dynamically* as the control point moves around in a procedure. The cdb [5] debugger exports a pointer to the innermost scope descriptor at stopping points, and each descriptor contains a pointer to the descriptor of the surrounding scope. If scope changes are infrequent, it may be faster to register and de-register the new scopes at block entries and exits.

For minimal overhead, we suggest that all local variables be collected in a single procedure-global dynamic scope record, with a union of sub-records for nested scopes. (This is effectively what many compilers do already, since they can then avoid having to dynamically change the size of stack frames.)

The HLL compiler builds an inverted tree of static scope descriptors, where each node describes one block and contains a pointer to the node for the surrounding block. The tree is emitted as a set of static immutable data objects. The static site descriptor for every stopping point is extended to include a pointer to its corresponding node in the static scope tree.

With this scheme, no additional events (block entries/exits) occur, and no additional parameters are passed in calls to the debugger library.

3.5 Data breakpoints

Data breakpoints are used to implement queries such as: "stop when variable x is assigned." Existing efficient implementations use various non-portable techniques, such as replacing store instructions, or write-protecting virtual memory pages and emulating caught write instructions [16]. Pre-filtering techniques have been proposed to improve performance by reducing the number of checked stores [17, 10]. Since a portable software-only implementation cannot access and patch code, we use cheap conditional breakpoints at interesting stores, and pre-filtering to allow many stores to be unchecked.

Checking if an address is watched. Data breakpoints are implemented by maintaining a set of *watched* addresses, and generating debugger events when data at watched addresses is updated. For the set, Wahbe et al [17] recommend using a segmented bitmap instead of a hash table: a hash table would require a loop of memory fetches and comparisons to resolve hash conflicts, while the segmented bitmap can be inspected using only two fetches and some arithmetic.

A flat bitmap with one bit per byte would be very large and very sparse. If a segment table is used to point to smaller bitmaps, all empty segments can share the same empty bitmap. Assuming a fair level of locality among watched addresses, storage requirements should be manageable. If the top-level segment table is too large, the number of bits to index it can be reduced, or perhaps a 3-level segmentation scheme be used.

Another possibility is to use a hash table, and inline the code for computing the hash and indexing the table. Only if the collision list is non-empty is an out-of-line call made to complete the conditional breakpoint. Assuming a very fast hash function, this might be cheaper than the segmented bitmap, partly because it uses less space, and partly because the common case (store at non-watched address, no hash collision) only uses one memory fetch instead of two.

Finally, source-level stores are made into conditional breakpoint sites, and instrumented to perform the appropriate check and event generation:

```
/* site i: x := <exp>; */
if( isWatched(&x) ) storeTrap(i, &x, ...);
x = <exp>;
```

Reducing the number of checked stores. If the `isWatched` test costs more than a few instructions in the common case, then it might be worthwhile to add a per-site `isChecked` flag, and augment the code to perform the test only if the flag is set. Since checked stores now become more expensive, it is important that most stores remain unchecked.

- If the source language is statically strongly typed and data *cannot* be accessed at different types, then any store of a type different from the type of the watched address need not be checked. (This requires that updates via out-of-bounds array indices or invalid pointers are caught, see [8].) The compiler should include the type of the store in the descriptor generated for every store site.
- If the watched address is a named location, and there is no aliasing by name, then any store whose target is a different named location need not be checked. In the store site descriptor, the compiler should include the address of the target, as *global* x, *local* x, or *dynamic pointer*.
- In the presence of pointers or references, range analysis might be performed to eliminate further checks, but it is not clear that the benefits make up for the added implementation complexity [17].

Reusing dynamic type checks. Dynamically typed languages, like Lisp, Scheme, and Prolog, often *tag* runtime data, and perform type checks before accesses and updates [4]. It may be possible to implement data watchpoints by reusing the dynamic type checks already performed. Usually, there is an n-bit type tag in every object's header. We add one more tag bit, and modify the compiler to retrieve and check all $n+1$ bits during type checks before data updates. (The runtime cost should be the same as before.) When an object is watched, the extra bit is set; updates invoke the error handler, which in turn notifies the debugger.

Compilers that perform "soft" type inference in order to eliminate unnecessary type checks should only optimize read accesses, not write accesses.

Some implementations tag pointers and omit headers from common small objects, e.g. cons cells. Our technique will not work for those objects.

Copying GC and VM page protection. An implementation with a copying garbage collector could relocate a watched data object to its own VM page, write-protect that page, and install a store fault handler. (This is reasonably portable in modern Unixes.) Since a portable debugger cannot patch code, the fault handler should unprotect the page, arrange for a breakpoint as soon as possible after the store has completed, and resume the debuggee. At the following breakpoint, the debug monitor re-protects the page, and sends the appropriate event to the debugger.

Every store could update a global "next site" pointer before it executes, in order to allow the fault handler to plant a breakpoint there; unfortunately, this incurs overheads even when no watchpoints are active. If procedure descriptors include a list of their breakpoint sites, then by accessing the top frame of the shadow stack, the handler can set every breakpoint in the current procedure. Alternatively, *all* breakpoints can be set.

3.6 Profiling

Traditional profiling tools collect two pieces of information for every source-level procedure: the number of times it was called, and the percentage of total execution time spent in it. Every procedure is given a call-count variable, and code is added to the procedure's prologue to increment its call-count. A histogram, indexed by *ranges* of code addresses, is also maintained. An operating system service, such as Unix' `profil` system call, is requested to sample the value of the program counter at regular intervals, scale the program counter by the size of the ranges, and increment the corresponding entry of the histogram. At program termination, the call-counts and the histogram are written to a file, which is then processed by the profiling tool.

The code instrumentation for call-counts is traditionally buried in the backend, but could easily be done by modifying the intermediate representation of every procedure to include a declaration of a private call-count variable, and code to increment the call-count at procedure entry.

Sophisticated compilers often perform code transformations such a (full or partial) inlining or cloning which significantly complicate the mapping from code addresses to source locations. Another problem occurs for implementations that compile via another language, say C. The profiling tool uses C procedure names when reporting call-counts and timings, but a single C procedure may correspond to (parts of) the code for several HLL procedures. Garbage collectors that relocate code blocks present further problems.

Appel et al [2] describe a profiling tool for an early version of the SML/NJ compiler. To avoid the problems described above, they instrument the program (at the abstract syntax level) to maintain a global `current` variable, which points to the clock interrupt counter for the currently executing function. The clock interrupt handler increments the cell pointer to by `current`. Now, even if the function is inlined in several different places, all copies will charge the same call-count and interrupt-count variables.

The technique can also be used for profiling smaller program units, such as basic-blocks, or for recording true/false frequencies in conditionals.

Modern workstations and PCs allow applications to install interrupt handlers invoked at regular sub-second intervals, e.g. using the Unix `signal` and `setitimer` system calls. If this support is unavailable, the HLL compiler can explicitly instrument each basic block to update the corresponding histogram entry; the time to charge can be estimated to the size of the basic block.

4 Related work

Heymann [6] developed a "portable" debugger for a simulation language that is translated to C. The translator inserts additional calls to a debugging library at procedure entries and exits, and before each statement. At start-up, the debugger invokes all procedures in a special mode (a global flag is set) that causes their instrumentation code to register various meta-information with the debugger. The instrumentation code maintains a shadow stack and registers the addresses of local variables, but, as we have described, the implementation is non-portable and broken.

Ramsey and Hanson [12] describe `ldb`, a retargetable debugger for the `lcc` compiler. Code is compiled as usual, but the compiler generates additional debugger symbol tables that are encoded as Postscript programs. A "nub" is linked with the debuggee, and it is responsible for reading and writing the debuggee's state. A breakpoint is set by finding the appropriate machine instruction via the symbol tables, and then telling the nub to overwrite the instruction with a trap instruction. Machine-dependencies are minimized, but there are still several hundred lines of machine-dependent code, including some assembly, for every supported platform.

Hanson and Raghavachari [5] describe `cdb`, a debugger for `lcc`. It is inspired by `ldb`, but much smaller and almost completely portable. The difference from `ldb` is that the compiler updates its high-level intermediate representation to include data structures that contain the debugger symbol tables, and instruments code to test breakpoint flags and maintain a shadow stack. The debugger does not implement single-stepping or conditional breakpoints. Their instrumentation code has some unnecessary overheads.

Tolmach and Appel [14, 15] describe a portable event-oriented debugger for the SML/NJ compiler. Instrumentation code is inserted at the abstract syntax level, in order to avoid having to change the complex middle and back-end components of the compiler. The debugger supports execution *replay*. Side-effects are logged and the state is checkpointed at certain intervals. To move back to some previous event, the state is restored from the last checkpoint before that event, and the code is re-executed until the event re-occurs. During re-execution, the results of side-effects are taken from the logs. Breakpoints are triggered by source site or by virtual time. Other features are implemented on top of these two primitives. The implementation suffers from significant overheads, caused partly by the instrumentation code, and partly by the work needed to checkpoint states, log side-effects, and re-execute side-effects.

Appel et al [2] describe how profiling was added to the SML/NJ compiler using simple code instrumentation and a runtime clock interrupt handler.

Kellomäki [9] developed psd, a portable Scheme debugger. A source-level transformation replaces every expression with a call to a debugging library with the expression itself, converted to an anonymous function, as one of the parameters. It supports several standard features, but not call-stack back traces. Debuggable and non-debuggable code modules may be mixed. Source code is expanded by about an order of magnitude, and execution is slowed down by about two orders of magnitude.

Wahbe [16] evaluates several different implementations of data breakpoints. *Code patching*, which replaces stores by jumps to a debug library, is shown to have better performance than either *trap patching* (replacing stores by traps, which are then handled by the debugger) or *VM protection* (using VM page protection to catch updates). Keppel [10] notes that VM protection can be used to perform code patching *lazily*, which greatly reduces the number of store instructions that need to be patched.

Wahbe et al [17] discuss several optimizations to the code patching scheme. They describe a technique based on analysis of binary code that allows many stores to be unchecked. The complexity of the analysis in relation to its benefits is such that the authors recommend to simply check all stores.

Jones and Kelly [8] show how pointer bounds checking can be added to C code without making it incompatible with non-bounds-checked code. They add code to maintain a map of all valid objects and their address bounds, and instrument every pointer operation to consult the map to verify the operation. A modified version of the gcc compiler is used, and some library procedures are replaced with bounds checked versions. Bounds checked code is slowed down 5–6 times in the current implementation.

Ducassé [3] described the Prolog Opium debugger which is based on event-processing: the debuggee generates events at interesting points, and the debugger processes them. Full time travel, i.e. the ability to reference events generated at given time points, is supported. Performance or implementation details are not discussed; however, Ducassé found that for acceptable performance, executing a *pre-filter* in the debuggee itself was necessary.

The *p2d2* debugger from NASA Ames Research Center [7] is a portable debugger for HPF programs distributed over a heterogeneous set of machines. It uses a client-server protocol between a user-interface client and debugger servers executing on the machines. Debugger servers are implemented as wrappers on top of preexisting Fortran debuggers. The application code is also linked with a modified library that reports interesting process and communication events to the debugger server.

5 Comments

At the time of this writing, we do not have a brand new debugger to show off the ideas presented in this paper.

At an early stage, we tested some of the basic ideas in a prototype debugger for Standard ML. This debugger uses source-code instrumentation, static site descriptors, and places events at function entries and recursive calls. It supports breakpoints, single-stepping, variable inspection, and call stack traversal, but does not catch runtime exceptions. The prototype relies on manual code instrumentation and is unable to handle polymorphic variables, but these problems can be overcome [15]. The code size is 1300 lines of SML, with 550 lines implementing the core functionality.

A master's student is currently implementing a portable instrumentation-based debugger for procedural languages. This work is in a very early stage, but we will report on it when it is done.

Since traditional debuggers often require the optimizer to be turned off, it may be that instrumentation-based debuggers do not lose too much performance compared to traditional debuggers. To test this, we experimented with a small C program, using gcc on the Alpha processor. The program was compiled in three versions: *normal* (compilation flag `-O`), *debug* (`-g`), and *instrumented* (instrumentation code for breakpoints and variable inspection manually inserted, compiled with `-O`). Compared to the normal version, the debug version had 35% more object code and was 55% slower, while the instrumented version had 99% more object code and was 64% slower. Thus, it seems that code instrumentation need not be unacceptably slow compared to traditional debugging support.

6 Summary

Code instrumentation is a very powerful technique: it can be used to implement debugger functionality that is otherwise unavailable, and it may be performed in a machine and operating system-independent manner, allowing both the instrumentation phase in the compiler and the debugger to be portable.

On the negative side, code instrumentation may make debug-enabled and non-debug-enabled code incompatible, and it expands the size of generated code and data segments. Executing parts of the debugger in the same process as the debuggee allows for very fast conditional breakpoints and other tests, but makes the debugger vulnerable to memory addressing errors in the debuggee.

The technical report version of this paper will include detailed code examples for the debugging primitives described here.

References

1. ANSI X3.159-1989. *Programming Language – C.* American National Standards Institute, 1989.
2. Andrew W. Appel, Bruce F. Duba, and David B. MacQueen. Profiling in the presence of optimization and garbage collection. Technical Report CS-TR-197-88, Princeton University, November 1988.
3. Mireille Ducassé. Opium: An extendable trace analyser for Prolog. *The Journal of Logic Programming*, 1998. To appear in the special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds).
4. David Gudeman. Representing type information in dynamically typed languages. Technical Report TR 93-27, University of Arizona, Department of Computer Science, October 1993.
5. David R. Hanson and Mukund Raghavachari. A machine-independent debugger. *Software – Practice and Experience*, 26(11):1277–1299, November 1996.
6. Jurgen Heymann. A 100% portable inline-debugger. *ACM SIGPLAN Notices*, 28(9):39–46, September 1993.
7. Robert Hood. The *p2d2* project: Building a portable distributed debugger. In *SPDT '96: SIGMETRICS Symposium on Parallel and Distributed Tools.* ACM, May 1996.
8. Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging, AADEBUG'97*, Linköping, Sweden, May 26–27 1997. `http://www-ala.doc.ic.ac.uk/~phjk/`.
9. Pertti Kellomäki. Psd – a portable Scheme debugger. *Lisp Pointers*, VI(1), 1993.
10. David Keppel. Fast data breakpoints. Technical Report UWCSE TR-93-04-06, University of Washington, Department of Computer Science and Engineering, 1993.
11. Mikael Pettersson. *Compiling Natural Semantics.* PhD thesis, Department of Computer and Information Science, Linköping University, December 1995.
12. Norman Ramsey and David R. Hanson. A retargetable debugger. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, PLDI'92*, pages 22–31. ACM Press, 1992.
13. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, 1972.
14. Andrew Tolmach and Andrew W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, April 1995.
15. Andrew P. Tolmach. *Debugging Standard ML.* PhD thesis, Princeton University, October 1992. Technical Report CS-TR-378-92.
16. Robert Wahbe. Efficient data breakpoints. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'92*, pages 200–212. ACM, 1992.
17. Robert Wahbe, Steven Lucco, and Susan L. Graham. Practical data breakpoints: Design and implementation. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, PLDI'93*, pages 1–12. ACM Press, 1993.