Reprinted from the Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, San Francisco, June 1992, pp. 22-31.

A Retargetable Debugger

Norman Ramsey and David R. Hanson
Department of Computer Science, Princeton University
Princeton, NJ 08544
nr@princeton.edu drh@princeton.edu

Abstract

We are developing techniques for building retargetable debuggers. Our prototype, ldb, debugs C programs compiled for the MIPS R3000, Motorola 68020, SPARC, and VAX architectures. It can use a network to connect to faulty processes and can do cross-architecture debugging. ldb's total code size is about 16,000 lines, but it needs only 250–550 lines of machine-dependent code for each target.

ldb owes its retargetability to three techniques: getting help from the compiler, using a machine-independent embedded interpreter, and choosing abstractions that minimize and isolate machine-dependent code. ldb reuses existing compiler function by having the compiler emit PostScript code that ldb later interprets; PostScript works well in this unusual context.

1 Introduction

Retargeting some programming environment components, such as compilers and editors, is now common, thanks to well developed retargeting techniques. For example, compiler writers use machine-independent intermediate representations and code-generator generators. We are developing techniques for building retargetable, multiple-target debuggers. This paper describes the design and implementation of ldb, a prototype retargetable debugger. ldb is a source-level debugger like gdb or dbx [15, 22]. It can be used with C programs compiled with lcc [11], a retargetable compiler that generates code for the MIPS R3000, Motorola 68020, SPARC, and VAX architectures. Users can set and remove breakpoints, start and stop programs, evaluate expressions, and make assignments to variables.

ldb's contribution lies in implementing these operations in a retargetable way. It does so by using three techniques: getting help from the compiler, using a machine-independent embedded interpreter, and choosing abstractions that minimize and isolate machine-dependent code.

ldb is an experiment in coupling between compiler and debugger; they are separate tools, but ldb depends on and uses existing compiler function as much as possible. The division of labor between ldb and lcc shows that making modest demands on the compiler can simplify the debugger substantially. lcc's assistance in generating machine-independent symbol tables, in implementing breakpoints, and in evaluating expressions are examples.

ldb is connected to the compiler by PostScript programs, which its embedded PostScript interpreter evaluates as necessary. This approach provides a uniform mechanism for representing symbol tables, shields the debugger from irrelevant information, and supports machine-independent expression evaluation.

ldb's design embodies a set of engineering choices that collaborate to minimize and isolate machine-dependent code. For example, it controls target processes with a small "debug nub" that is loaded with the target program. It exchanges messages with this nub using a machine-independent protocol, and it establishes a connection to a nub by connecting to an existing process over the network, by forking the target process as a child, or by being forked by a faulty process asking to be debugged.

The bulk of ldb, including its embedded PostScript interpreter, is written in Modula-3 [18]. The nub is written in C and assembly language, and much of the symbol table and expression evaluation support is written in PostScript. ldb's machine-dependent code depends only on which architecture the target program and its nub run on, not on which architecture ldb runs on. As a result, cross-architecture debugging with ldb is identical to single-architecture debugging, and ldb can change architectures dynamically.

2 PostScript Symbol Tables

Production versions of lcc generate machine-dependent, symbol-table "stabs" for dbx and gdb. We changed this code to generate *machine-independent* symbol tables represented by PostScript programs that build PostScript objects, such as dictionaries. Symbol tables contain code as well as data, e.g., procedures that ldb can interpret to print structured data, so ldb need not know the layout of runtime data structures.

Embedded in ldb is an interpreter for a dialect of PostScript. This dialect omits font and imaging types and operators, but adds new types and operators for debugging. For example, `ldb`'s PostScript dialect supports "abstract memories," which, as described in Sec. 4.1, are a machine-independent representation of target registers and memory.

A *symbol-table entry* is a PostScript dictionary describing a source-language identifier: a variable, procedure, type, or constant. The symbol-table entry for `i`, on line 6 of the sample C program shown in Fig. 1, is associated with the name `S10`:

```
/S10 <<
  /name (i)
  /type << /decl (int %s) /printer {INT} ... >>
  /sourcefile (fib.c) /sourcey 6 /sourcex 8
  /kind (variable)
  /where 30 Regset0 Absolute
  /uplink S8
>> def
```

Brackets (`<<`...`>>`) surround dictionaries; within each dictionary, names preceded by slashes are associated with the values that follow. `name`, `type`, `sourcefile`, `sourcey`, `sourcex`, `kind`, and `uplink` appear in all symbol-table entries; `where` appears only in entries for variables and procedures. The value associated with `where` represents `i`'s location (register 30) and is computed when the symbol table is interpreted.

A source-level type dictionary, associated with `type`, must contain at least two values: a string used to declare variables of the type and a PostScript procedure used to print values of the type. The procedure's arguments are abstract memory, location, and type dictionary; the machine-independent procedure `INT` ignores the dictionary, fetches a 32-bit integer from the abstract memory, and prints it. The ellipsis in the type dictionary shown above stands for information not used by `ldb` proper, but used by PostScript code that supports expression evaluation.

In entries for local symbols, `uplink` is associated with the entry for the preceding symbol in the current or enclosing scope; `i`'s `uplink` value is the symbol-table entry for `a` (Fig. 1, line 2), which is associated with `S8`. The `uplink` values link symbol-table entries in a tree that handles nested scopes; using a tree avoids complications that can arise with the flattened tables emitted by typical assemblers and loaders [6].

Fig. 2 shows the tree for `fib`. The symbol-table entry for a procedure associates `formals` with the entry for the procedure's last parameter and `loci` with a PostScript array of *stopping points*. Each element of this array contains a source location, an object location, and a symbol-table entry; the symbol and its ancestors are visible from that stopping point. For example, the 9th element of `fib`'s stopping-point array contains the entry for the symbol `j`. This point corresponds to the expression `j<n` in line 11 of Fig. 1, from which `j`, `a`, `n`, and `fib` are visible.

A context for name resolution is determined by a particular stopping point in a particular procedure, normally the place where control has stopped. `ldb` resolves names by walking up the tree of entries for local symbols, beginning with the symbol-table entry contained in the stopping point. When it reaches the root, it searches two PostScript dictionaries; one contains entries for symbols local to this

```
 1  void fib(int n) ⁰{
 2    static int a[20];
 3
 4    if (¹n > 20) ²n = 20;
 5    ³a[0] = a[1] = 1;
 6    { int i;
 7      for (⁴i=2; ⁵i<n; ⁷i++)
 8        ⁶a[i] = a[i-1] + a[i-2];
 9    }
10    { int j;
11      for (⁸j=0; ⁹j<n; ¹¹j++)
12        ¹⁰printf("%d ", a[j]);
13    }
14    ¹²printf("\n");
15  ¹³}
```

Figure 1: Example program.
Superscripts show stopping points.

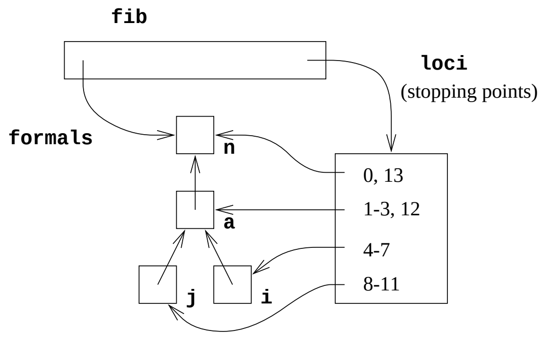

Figure 2: The tree structure of `fib`'s symbol table.

compilation unit (C `statics`) and the other contains entries for global symbols (C `externs`). The first is associated with `statics` in the current procedure's symbol-table entry; the second is associated with `externs` in the "top-level dictionary" for the program.

`a`'s symbol table entry is associated with `S8`:

```
/S8 <<
  /name (a)
  /type <</decl (int %s[20]) /printer {ARRAY} ...>>
  /sourcefile (fib.c) /sourcey 2 /sourcex 13
  /kind (variable)
  /uplink S7
  /where {(_stanchor__V2935334b_e288a) 8 LazyData}
>> def
```

`uplink` is associated with `n`'s symbol-table entry, named `S7`. `a` is `static`, so its location is not determined until link time. The value associated with `where` is a PostScript procedure that is interpreted at debug time; it computes `a`'s location by calling `LazyData`. `LazyData` gets the location of the generated *anchor* symbol, `_stanchor__V2935334b_e288a`, from `ldb`'s linker interface (see Sec. 3) and fetches the address of `a` from the 8th word following that location. This anchor-symbol technique is also used to compute the object-code locations of stopping points.

`ldb` prints values using the procedure associated with `/printer` in the value's type dictionary. The machine-independent PostScript procedure `ARRAY` prints C arrays. Like `INT`, its arguments are an abstract memory, a location, and a type dictionary. From the type dictionary, it fetches the type of an element, the element size, and the size of the whole array. This information, which may be machine-dependent, is placed in the type dictionary by the compiler. It is elided in the example above because it is used only by PostScript code like the `ARRAY` procedure, not by `ldb` proper.

The part of `ARRAY` that prints `a`'s value is

```
({) Put 0 Begin
0 &elemsize &arraysize 1 sub
{ dup 0 ne { (, ) Put 0 Break } if
  dup &limit ge { (...) Put pop exit } if
  &machine &loc 3 -1 roll Shifted &elemtype print
} for
(}) Put End
```

This code prints an opening brace and loops through the offsets of the array elements. At each iteration, it adds the offset to the location of the array and prints the element at that location. Each element except the first is preceded by a comma and a potential line break. If the number of elements exceeds an adjustable limit, an ellipsis is printed and the loop terminates. Finally, a closing brace is printed.

A *top-level dictionary* describes a single compilation unit or any combination of compilation units, up to an entire program. It contains an array of symbol-table entries for procedures, a dictionary associating external symbol names with their symbol-table entries, a dictionary associating file names with arrays of symbol-table entries for the procedures defined in the files, and an array of the names of all anchor symbols used. The anchor-symbol names are compared with the anchor-symbol names in the loader table (see Sec. 3) to ensure that the top-level dictionary matches the object code. A top-level dictionary also contains the name of the architecture for which the program was compiled, which `ldb` uses at debug time to find its machine-dependent code and data.

Assuming that `S1` and `S6` represent the symbol-table entries for procedures `main` and `fib`, the top-level dictionary for `fib.c` is

```
<<
  /procs [ S1 S6 ]
  /externs <<
     /main S1
     /fib  S6
  >>
  /sourcemap <<
    /fib.c [ S1 S6 ]
  >>
  /anchors [ /_stanchor__V2935334b_e288a ]
  /architecture (sparc)
>>
```

The structure of top-level dictionaries is determined by the operations `ldb` needs to perform. When given a program counter, `ldb` must find the symbol-table entry of the corresponding procedure. `ldb` uses the `procs` array to build a table mapping procedure addresses to symbol-table entries. (Mapping from program counters to procedure addresses is done by the linker interface described in Sec. 3.) `ldb` uses the `externs` dictionary to resolve names of global symbols. `ldb` uses the `sourcemap` dictionary to build a map from source locations to stopping points, making it possible to set breakpoints by source location. Because of the C preprocessor, a single source location may correspond to more than one stopping point.

3 Division of Labor

`ldb` gets link-time information by cooperating with `lcc`'s compiler driver. After linking a program, the driver uses the UNIX program `nm` to generate PostScript that, when interpreted, builds a *loader table*, which is a PostScript dictionary. The loader table contains the program's top-level dictionary, a dictionary associating the names of anchor symbols with their addresses, and an array of (address, name) pairs for each procedure in the program. `ldb`'s linker interface hides machine dependencies and provides access to this table as a Modula-3 object. `fib`'s loader table is

```
<<
    /symtab << top-level dictionary >>
    /anchormap <<
      /_stanchor__V2935334b_e288a 16#000023d8
      ...
    >>
    /proctable [
      16#00002270 (_fib)
      16#00002374 (_main)
      ...
    ]
  >>
```

The loader table supports debugger operations that need information not available until link time. `ldb` needs the locations of variables and procedures; Sec. 2's anchor-symbol technique reduces the problem to finding the locations of the anchor symbols, which are stored in the `anchormap` dictionary. The `proctable` enables `ldb` to find the address of the procedure containing a given program counter, which is the first step in mapping a program counter to the symbol-table entry of the corresponding procedure.

To plant breakpoints, users specify source locations or procedure names; `ldb` computes the locations of the corresponding instructions. `ldb` plants a breakpoint at an instruction *I* by overwriting *I* with a trap instruction. To resume execution, it interprets *I* out of line, then continues with the next instruction. For now, `ldb` can set breakpoints only at no-op instructions, which can be skipped instead of interpreted. The implementation is machine-independent, but it manipulates machine-dependent data: the bit patterns used for break and no-op, the type used to fetch and store instructions, and the amount to advance the program counter after "interpreting" the no-op. This interim scheme relies on compiler support: `lcc` already places labels at stopping points, so putting no-ops there requires no extra effort. The no-ops increase the number of instructions by 16–19%, depending on the target.

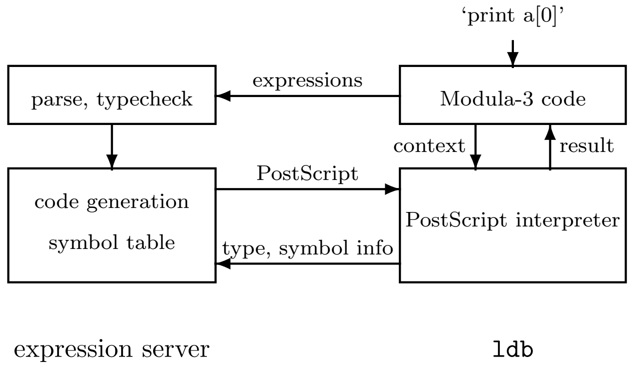

Figure 3: Communication paths between `ldb` and an expression server

The possibility of stopping the program before any top-level expression reduces the benefit of instruction scheduling because the scheduler may rearrange instructions only within such expressions, not within basic blocks. `lcc` does not do instruction scheduling, but the MIPS assembler does. When `lcc` compiles for debugging, the MIPS code size increases by 13%, because there are load delay slots that the assembler is unable to fill using the more restricted scheduling. This penalty is independent of the cost of the explicitly inserted no-ops.

Assignment and expression evaluation use an "expression server," with which `ldb` communicates via pipes, as shown in Fig. 3. To evaluate an expression, `ldb` sends it to the server, which is a variant of the compiler. The server attempts to parse and type-check the expression and to produce an intermediate-code tree. When it fails to find an identifier `a` in its symbol table, it does not print an error message and stop; its symbol-table code has been modified to send "`/a ExpressionServer.lookup`" back to `ldb`. The PostScript procedure `ExpressionServer.lookup`, when interpreted by `ldb`, finds the PostScript dictionary representing `a`'s symbol-table entry and sends information from that dictionary back to the expression server. The server's modified symbol-table code uses that information to reconstruct `a`'s symbol-table entry on the fly, and it returns the newly created entry to the parser just as if the entry had always been present. The expression server discards new symbol-table entries after the evaluation of each expression, but it saves type information until the user switches to a different target program.

The server's intermediate-code tree is not passed to the usual compiler back end; instead it is rewritten as a PostScript procedure. This PostScript procedure is sent to `ldb`, followed by "`ExpressionServer.result`", which tells `ldb` that the procedure is on the stack and that it can stop listening to the expression server. `ldb` then takes the procedure from the stack and interprets it, which has the effect of evaluating the expression.

Rewriting intermediate code into PostScript takes little effort, and doing so instead of interpreting intermediate code directly should make it easier to support different languages, compilers, and intermediate codes. One interpreter supports code in symbol-table entries and expression evaluation.

The expression-server architecture has two benefits: reuse of the compiler and simplicity in the debugger. The changes that make the compiler front end act as a server affect only the input, lexical analysis, symbol, and type modules. The debugger treats each expression as a string; it sends the string to the expression server, then interprets PostScript code until the expression server tells it to stop. The operation of interpreting until told to stop is implemented by applying "`cvx stopped`" to the open pipe from the server.

Reusing the compiler has a limitation: the debugger cannot provide an extended language for debugging. Such facilities could be provided by implementing an extended language as an expression server.

Because it puts the expression server in a separate address space, `ldb` can reuse a range of compiler implementations. The compiler and debugger need not be written in the same language, need not support the same data types, and need not agree on how to manage storage or share input and output. The cost of this flexibility is that the compiler writer must devise PostScript procedures that take symbol-table data and send it to the expression server over a byte stream. The `lcc` expression server receives sequences of C tokens that represent type and symbol data. If the `lcc` front end could easily have shared an address space with `ldb`, `ldb` could have made procedure calls to expression-server code instead of sending messages between processes.

4 Design for Retargetability

Three aspects of `ldb`'s design contribute most to its retargetability: abstract memories, the debug nub, and the use of Modula-3 subtyping.

`ldb` is written in Modula-3 and uses Modula-3 interfaces and subtypes to minimize and isolate machine dependencies. Each target architecture requires Modula-3 objects that contain machine-dependent data, methods, or both. When possible, machine-dependent function is specified by Modula-3 objects containing machine-dependent data but no code.

Machine-independent classes[1] define the important abstractions in `ldb`; machine-dependent code is isolated in machine-dependent subtypes of such classes. (A similar technique has been used in implementations of I/O streams [18, p. 143].) One such class is `ldb`'s stack-frame abstraction. The machine-independent class includes the value of the program counter, the symbol-table entry of the corresponding procedure, and several machine-independent methods, which perform such tasks as computing scopes for name resolution. Machine-dependent instances of the class supply only two methods: one that walks down the stack and one that restores registers from the stack.

4.1 Abstract Memories

Abstract memories represent the registers and memory of a target process. An abstract memory is a collection of *spaces*, denoted by lower-case letters, e.g., `d` for data space, `r` for registers, etc. Locations within a space are determined by an integer offset. `ldb` provides several "addressing modes"

[1]A *class* is a Modula-3 type that has no exact instances; all instances have types that are proper subtypes of the class.

to refer to locations, including an immediate mode. Given a memory and location, `ldb` can fetch and store three sizes of integers (8, 16, and 32 bits) and three sizes of floating-point values (32, 64, and 80 bits).

`ldb` assumes that every machine has code and data spaces. Other spaces can be added to the abstract memory model as necessary for a particular machine. On the MIPS, for example, we add `r` for general-purpose registers, `f` for floating-point registers, and `x` for "extra registers." The extra registers are the program counter and the virtual frame pointer.[2] The code and data spaces may refer to the same locations or different locations depending on the target architecture.

Abstract memory is another Modula-3 class; `ldb` combines several different instances to represent the state of memory and registers during a particular procedure activation. The instances form a directed acyclic graph, shown in Fig. 4. The *wire* is an abstract memory that holds a connection to the nub; it forwards fetch and store requests to the nub, which executes them and returns the results. The nub, located in the target process, can respond to requests only for locations in the code and data spaces. Registers are saved either on the stack or in a *context*, which is an area in memory that holds the state of a stopped program. The *alias* memory translates requests for locations in register spaces into requests for locations in the code or data spaces (or for immediate locations).

The *register* memory solves a problem that occurs when fetching the least significant byte from a register. The alias memory records the location in memory of the whole register, but the location of the byte depends on the target's byte order. The register memory transforms fetches and stores into full-word operations on the underlying memory, making byte order irrelevant. For example, if `ldb` fetches a character from a 32-bit register, the register memory fetchs the whole register, but returns only the least significant 8 bits. Register memories enable `ldb` to execute the same code whether debugging a little-endian or a big-endian MIPS, for example.

The *joined* memory combines memories that serve different spaces, routing fetch and store requests to the appropriate underlying memory. The joined memory is the instance presented to the rest of the debugger as representing the abstract memory for a stack frame.

`ldb` uses abstract memories to print the values of variables. If a breakpoint is planted at stopping point 7 of `fib` (line 7 of Fig. 1), `ldb` can access `i`, `a`, `n`, and `fib` when it stops there. For example, it prints `i` by executing the PostScript procedures shown in Sec. 2, which generate fetch requests that travel through Fig. 4's abstract memories. `i` is located at offset 30 of space `r` (register 30), so the joined memory fetches `i` from the register memory. The request is for a full word, so the register memory fetches the word from the alias memory, which notes that register 30 is an alias for a location in the data space 92 bytes after the beginning of the context and asks the wire to fetch the word at that location. The wire sends a message to the nub, which fetches `i` using the target's byte order and sends the value back to the debugger in little-endian order, where it gets returned up the dag of abstract memories and put on the PostScript operand stack, whence it is printed. Printing the value of `a` is similar, but its elements are located in the data space, so the fetch requests are routed from the joined memory directly to the wire, which sends them to the nub.

[2]The MIPS has no actual frame pointer, but `lcc` uses the virtual frame pointer to address local variables. The other targets have other idiosyncrasies of similar complexity.

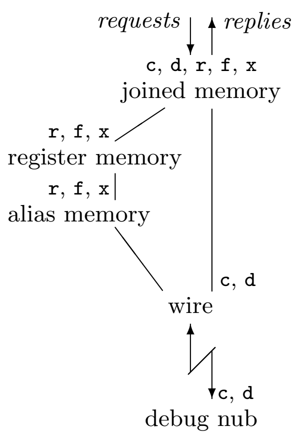

Figure 4: Abstract memory for a frame.

In the abstract memory dag, machine-independent code manipulates machine-dependent data: the aliases recorded in the alias memory. Fetches that use the immediate addressing mode return immediate values; all other fetches and all stores are eventually done on target memory by the nub. Only the data is machine-dependent, so it doesn't matter what architecture the code runs on; except for floating-point data, cross-debugging is free.

`ldb` creates abstract memories using machine-dependent stack-walking procedures. On the MIPS, for example, `MipsFrame.New` takes the context from the nub and creates a machine-dependent instance of the stack-frame abstraction, a representation of the topmost frame on the target call stack. The stack frame includes an abstract memory, which is a joined memory that is part of a dag like the one shown in Fig. 4. Since the general-purpose and floating-point registers are saved in the context, those locations are made aliases for the appropriate locations in the target. The extra registers (program counter and virtual frame pointer) are aliases for immediate locations, not for locations in target memory. `MipsFrame.New` uses the program counter to find the procedure's dictionary, then computes the virtual frame pointer by adding the size of procedure's frame to the stack pointer. The machine-dependent frame size is stored in the dictionary by the MIPS implementation of `ldb`'s linker interface (see Sec. 4.3).

The stack-frame object created by `MipsFrame.New` has two machine-dependent methods. When a user walks the stack, one of these methods finds the calling frame, and the other constructs a new abstract memory for that frame. The aliases in the new alias memory usually stand for locations on the stack, not in the context, but when callee-save registers are not modified by the called procedure, the aliases from the called frame are reused.

4.2 The Debug Nub

The nub executes in user space; at program startup it installs a signal handler that gets control when the target process faults or encounters a breakpoint. When such an event occurs, the nub uses a socket to notify `ldb` of the signal, passing a signal number, an associated code, and a context that holds the values the registers had when the signal arrived. It then services `ldb`'s fetch and store requests until told to continue execution, to terminate, or to break the connection. Normally, when a connection is broken, even by a debugger crash, the nub preserves the state of the target program and waits for a new connection from another instance of `ldb`. The nub may be told to continue execution instead. The little-endian communication protocol between `ldb` and the nub has been used on all combinations of host and target byte orders and has been validated [13].

The nub supports several connection mechanisms, including debugging over the network. Since the nub is always loaded with the target program, it can catch unexpected faults and wait for a connection from `ldb`; the target program need not be a child of the debugger.

The nub isolates dependencies on the host operating system. Using sockets and signal handlers makes it easier to retarget the nub; these mechanisms are more uniform on different Unix implementations than are mechanisms that support debugging more directly, such as system facilities permitting one process to control another.

The principle guiding the design of the nub has been to keep it as small as possible. The nub is loaded with every program, so the cost for programs that are never debugged should be small. The smaller the nub is, the simpler it is, and the easier to retarget. It is easier to make a simple nub reliable, and the nub must preserve the state of the target program even if the debugger crashes.

In user space, the debug nub is vulnerable; if a faulty program destroys the nub's data structures it becomes impossible to debug that program. Moving the nub into the operating-system kernel would solve this problem; it could also reduce the number of context switches needed to perform a nub operation. The smaller and simpler the nub is, the more reasonable it is to consider putting it in the kernel. Finally, the nub *interface* should be small as well. A small interface will be easier to implement in a variety of environments, e.g., PCR [24] or the Topaz TeleDebug server [21].

4.3 Retargeting `ldb`

The debugger proper, the PostScript, and the nub all have machine-dependent parts, and each must be retargeted for a new machine. This section lists all such parts.

Machine-dependent Modula-3 code plants breakpoints, communicates with the nub, gets machine-dependent data bound at link time, and walks the call stack. `ldb`'s interim breakpoint implementation requires four items of machine-dependent data (see Sec. 3). The code that communicates with the nub is machine-independent, but it uses three machine-dependent procedures or objects: a procedure that distinguishes breakpoint faults from other faults, an object used to fetch and store the program counter located in a context, and a procedure that takes a context and creates the top frame on the call stack (see Sec. 4.1). Likewise, the code that fetches and stores fields of a context is machine-independent, but is parameterized by a machine-dependent description of those fields.

The VAX, SPARC, and 68020 share a single, machine-independent implementation of the linker interface. The MIPS cannot use this implementation because it has no frame pointer. To be able to walk past a MIPS stack frame, `ldb` needs to know its size. That information is available, even for procedures without debugging symbols, in the MIPS runtime procedure table [17]. The MIPS implementation of `ldb`'s linker interface uses the runtime procedure table to find the addresses and frame sizes of procedures, as well as other machine-dependent data.

The top-of-stack procedure, for example, `MipsFrame.New` of Sec. 4.1, hides the rest of the machine-dependent stack-walking code, i.e., the machine-dependent methods of the stack-frame object. The implementations for all four targets are similar.

`ldb` uses machine-dependent PostScript to address local variables and to enumerate a target's registers.

Most of the nub is machine-independent, but it has a few machine dependencies. The target architecture determines the type of the signal handler and whether, as on the MIPS and SPARC, the `struct sigcontext` can serve as a context or, as on the VAX and 68020, another representation must be used [8]. On a big-endian MIPS, nub code for doubleword fetches and stores of saved floating-point registers must swap the words.[3] Each machine has a different one-line "pause" procedure, which stops the target program before it calls `main`. The VAX and 68020 require assembly code to save and restore registers, and the 68020 requires assembly code to fetch and store 80-bit floating-point values.

To give the nub initial control, the system-dependent startup code must be modified to call the nub instead of `main`. It is most easily modified by editing the object code.

The following table lists the number of lines of machine-dependent Modula-3, ANSI C, and PostScript code that collaborate to implement the machine-dependent parts of `ldb` on each of its targets. The amount of machine-independent code is listed in the rightmost column.

	MIPS	68020	SPARC	VAX	shared
Debugger (M3)	476	187	206	199	12193
PostScript	15	18	18	13	1203
Nub (C, asm)	34	73	5	72	632

A few numbers stand out. The MIPS requires an extra 250 lines of Modula-3 code in the linker interface because the machine has no frame pointer. There is very little SPARC-dependent code in the nub because the operating system provides most of the registers and there is no other machine-dependent dirt. The nub sizes for the 68020 and VAX are large because of the assembly language used on those machines.

The effort required to retarget `ldb` is minimal. The authors' ports to the 68020 and VAX architectures each took

[3]On a big-endian MIPS, doubleword floating-point values are stored with the most significant word first, except that when the kernel saves floating-point registers in a `struct sigcontext`, it stores the least significant word first.

less than a week of one programmer's time. A colleague independently retargeted the Modula-3 code for the SPARC; he spent about two weeks working half-time.

5 Why PostScript?

The language embedded in the debugger should be easy for people to read and for tools to manipulate, so bytecode and machine code are inappropriate. Designing a new language is best avoided, so we considered PostScript, Scheme, FORTH, and Tcl [20]. The latter two offer too few data types. Although most of the benefits of using PostScript would also be obtained with Scheme, there are a number of reasons to prefer PostScript.

Both the compiler and the expression server have to generate code for the debugger, and PostScript was designed to be generated by other programs [2, p. 2]. Our experience confirms that it is easy to generate PostScript; for example, the expression server code that rewrites `lcc`'s intermediate representation into PostScript is only 124 lines of C, even though the intermediate representation has 112 operators.

PostScript dictionaries provide a convenient notation for symbol-table entries. Dictionaries are easily extended with machine-dependent data by adding entries; we have done so for two targets. For example, the compiler adds register-save masks when compiling procedures for the 68020. Most of `ldb` ignores these masks, but they are used by the machine-dependent stack-walking code.

Every PostScript object has an attribute that tells explicitly whether the object is literal or executable; the distinction need not be inferred from context. Because attempts to execute a literal object put that object on the stack, procedures that are interpreted at most once can be replaced with their results. We use this technique when fetching addresses relative to anchor symbols (see Sec. 2).

We can defer not only the interpretation but also the lexical analysis of PostScript code by quoting it with parentheses; the scanner reads the resulting string quickly. This deferral technique reduces by 40% the time required to read a large symbol table.

PostScript names are bound dynamically, and the "dictionary stack" used to resolve names is both distinct from the call stack and explicitly controlled by the PostScript program. When `ldb` changes architectures, it rebinds machine-dependent names by placing a dictionary on the dictionary stack; we supply one such dictionary for each target architecture.

As noted in Sec. 2, `ldb`'s PostScript implementation has extra types and operators to support debugging, and it omits types and operators that support fonts and graphics. The debugging support includes abstract memory and location types and operators. It includes an interface to a prettyprinter supplied with Modula-3; the prettyprinter procedures are called by the PostScript code that prints structured data. We made several other changes that make PostScript work better when embedded in a Modula-3 program. Strings are immutable, for compatibility with Modula-3 `TEXT`s. There are no save and restore operators; memory is reclaimed by the Modula-3 garbage collector. There are no substrings or subarrays. Interpreter errors raise Modula-3 exceptions. Files are readers or writers.

6 Related Work

Implementation strategies for debuggers vary. In some experimental systems, the compiler and debugger are tightly coupled. DEC SRC's Vulcan debugger executes in the same address space as the compiler, sharing its annotated abstract syntax trees. The DICE debugger cooperates with an incremental compiler as part of an integrated environment [12]. Most debuggers, like VAX DEBUG [4] and `dbx` [15], are separate tools which communicate with the compiler only through symbol-table information. This information is placed in the object file in a machine-dependent format. `ldb` is like these debuggers, but it uses PostScript as a machine-independent, language-independent, symbol-table format.

Most debuggers must re-implement some facilities for every language that they support [4]. Debuggers must recognize expressions and assignments, must be able to print values, including structured data, and must apply the language's scope rules to resolve names. `ldb`'s expression-server architecture and its use of PostScript reduce this kind of re-implementation effort. `ldb` has a fixed name resolution algorithm, but its implementation of printing values and its interaction with the expression server are independent of C. The name resolution algorithm could handle nested procedures, but not overloading.

`gdb` 4.0 [22] supports 20 different target machines and many different versions of UNIX, but of its more than 150,000 lines, over 47,000 are noted in the documentation as machine-dependent. This figure does not include over 10,000 lines that deal with machine-dependent object code formats like `a.out` and COFF. The long delay between the availability of MIPS machines and the availability of `gdb` for them also suggests that substantial effort is needed to retarget `gdb`.

Like parts of the Cedar programming environment [23], `ldb` defines a client interface so that it can be used by other programs, e.g., user interfaces. If `gdb` and `dbx` defined client interfaces, it would simplify the implementation of better user interfaces (e.g., `dbxtool` [1]) and of higher-level debugging tools (e.g., Dalek [19]). Event-action debugging techniques [3, 5, 19] seem well suited for implementation above `ldb`. `ldb` can debug on multiple architectures simultaneously, so it can process events from pieces of client-server applications that execute on different hardware.

`ldb`'s abstract memory model is low-level, close to the hardware. By contrast, the Cedar debugger manipulates an *abstract machine*, whose abstractions are those of the Cedar programming language [23, §6.4]. The abstract machine's interface to Cedar data is used by clients other than the debugger, including a user interface tool that lets users manipulate fields (including procedures) of records chosen dynamically. A similar abstraction could be built on top of `ldb`'s PostScript symbol tables and abstract memories, but it, like the expression server, would need detailed knowledge of the contents of the symbol tables; it would not be isolated from the details of C.

`ldb`'s nub interface is derived from the Topaz TeleDebug protocol, TTD [21], with some simplifications. `ldb` uses code outside the nub to connect to a target process. `ldb`'s nub does not support multiple threads or multiple processors. `ldb` and TTD use different data models; `ldb`'s model makes it easier to write code that is independent of target byte order, but TTD's model makes it easier to cache copies of target memory in the debugger process. The important simplification in `ldb` is that the protocol and nub do not mention breakpoints or single-stepping; breakpoints are implemented entirely in `ldb` using fetches and stores without assuming the ability to single-step. Other remote debuggers use more complex interfaces [16].

One way to resume execution after a breakpoint is to return the overwritten instruction to memory, execute it by single-stepping the target machine, and replant the breakpoint [7]. Single-stepping can be avoided by using a memory-patching implementation [9]; in that case, the overwritten instruction can be interpreted or transformed so that it can be correctly executed out of line [14].

7 Discussion

Work is divided between `ldb` and `lcc` to facilitate `ldb`'s retargeting, even at the cost of sacrificing other features. Having `lcc` insert no-ops at stopping points makes it possible to specify a breakpoint implementation in four lines, but makes target programs bigger and slower. Inserting relocatable addresses into locations known relative to anchor symbols simplifies the debugger's interface to the linker (`ldb` never needs the value of a private or static symbol), but makes target programs bigger. Using `nm` to extract information from the linked program makes `ldb` independent of linker formats.[4] Using PostScript for symbol tables makes the compiler interface machine-independent, but is incompatible with existing compilers, linkers, and debuggers. Having `lcc` emit PostScript procedures that `ldb` interprets to print values makes `ldb` independent of C's type structure and of the details of the runtime layout of data structures; the compiler writer determines both how values are represented in memory and how they are printed by the debugger.

The use of `nm` and the anchor-symbol technique could be avoided if `lcc` used a linker that recorded information about the locations of global and private symbols. This information should be in a machine-independent format and should make it possible to distinguish identically named private symbols from different compilation units. It is unnecessary to have the linker "relocate" the information in the PostScript symbol tables. For external symbols, `nm` is a good compromise; its output is mostly machine-independent and is easily transformed into PostScript.

`lcc` generates PostScript symbol tables using the same internal interface that is used by the production versions to generate machine-dependent symbol-table "stabs" for `dbx` and `gdb`. The implementation is somewhat more complex; about 1000 lines of C to generate PostScript versus about 300 for stabs. The additional cost is reasonable considering that the PostScript contains more information; it must be enough to enable the expression server to reconstruct the compiler's symbol-table and type information at debug time.

[4]`ldb`'s MIPS linker interface gets machine-dependent data from the runtime procedure table located in the target address space [17], not from the object file.

`ldb`'s PostScript symbol tables can be manipulated by PostScript programs. For example, we wrote PostScript code that reads the top-level dictionary for the nub and constructs a Modula-3 description of one of the nub's machine-dependent data structures.

`ldb` takes about 6 seconds to start up on a one-line "hello world" program, and about 9 seconds to start up on a 13,000-line version of `lcc`. The costs of initializing the Modula-3 runtime system and of interpreting PostScript increase the startup time of `ldb`. The following table shows the elapsed time required to execute the initial phases of `ldb`, `dbx`, and `gdb`. All programs were executed on a lightly loaded DECstation 5000/200 with 16M of main memory. Times shown are the average of seven measurements made with a stopwatch; errors are a few percent.

Modula-3 initialization	1.9 sec
Read initial PostScript	1.6
Read symbol table for `hello.c` (1 line)	2.2
Read symbol table for `lcc` (13,000 lines)	5.5
Connect to `hello.c` (one machine)	1.8
Connect to `lcc` (one machine)	5.1
Connect to `lcc` (two MIPS machines)	6.2
Connect to `lcc` (host MIPS, target SPARC)	5.0
`dbx`: start and read `a.out` for `lcc`	1.5
`gdb`: start and read `a.out` for `lcc`	1.1

All PostScript symbol tables include symbol-table information for the nub. The extra time required to debug a MIPS target is spent fetching information from the runtime procedure table in the target address space.

PostScript symbol-table information is about 9 times larger than `dbx` stabs for the same program. The `dbx` information is in a binary format, so it may be fairer to compare the PostScript after compression by the Unix program `compress`, in which case the ratio is about 2.

`ldb` originally used an abstract memory model based on C primitive types; for example, it distinguished signed from unsigned integers. The current abstract memory model (Sec. 4.1) uses three sizes of integers and three sizes of floating-point values. These values and types correspond closely to the values and types manipulated in `lcc`'s intermediate representation [10]. The change simplified both `ldb` and the PostScript symbol tables, and it enables the expression server to share more code with the compiler front end.

Floating point complicates cross-debugging. Different precision in `ldb`'s machine and the target machine is the most obvious problem, but even if both machines use IEEE floating point, each floating-point unit can hold a different state — rounding mode, for example. One possible solution to these problems is to extend the nub and its protocol to do all floating-point operations in the target. Even then, the floating-point state may be correct only for the topmost procedure activation. Other debuggers can suffer from these problems, too, unless the operating system or compiler tracks floating-point state changes.

`ldb` can connect to multiple targets simultaneously, so it must not leave target-specific state in global variables. It stores such state in *target objects*. Dependence on such state is surprisingly pervasive; many procedures require a target object for reasons not immediately obvious. One might expect, for example, that values of variables could be printed without access to a target object, but to print the function name associated with a C function pointer requires the loader table, which is accessed from the target object. A second problem with the target object is that several parts of the debugger must know whether the debugger is connected to a target process and whether that process is running or stopped. These parts include the user interface, the breakpoint implementation, the debugger end of the nub protocol, and the implementation of the client interface.

Both problems are exacerbated by the use of the anchor-symbol technique, because every computation of a location might fetch data from the target address space. This operation not only requires the target object, it requires that it be connected to a stopped process. Fortunately, the fetches from the target address space have little impact on performance because such fetches are performed only on demand and at most once per symbol-table entry.

7.1 Future Work

The expression server is not yet complete; in particular, `ldb` cannot evaluate expressions that include procedure calls into the target process. We expect to use the existing nub protocol when adding procedure-call support. `ldb`'s interface with the expression server is defined precisely by the relevant code, but our experience suggests that we can formulate a more abstract interface analogous to `lcc`'s code generation interface [10].

The PostScript approach may extend gracefully to languages with more semantic complexity, like C++ and Modula-3. If `dbx` "stabs" are extended to describe a richer language, `dbx` itself must also be extended [15], but `ldb`'s capabilities can be extended by changing only the PostScript symbol tables; `ldb` itself need not change. `ldb` may well suit language implementations that compile to C, because the first compiler can emit PostScript code that manipulates the symbols emitted by the C compiler, producing one set of symbols that combines the results of two compilations.

PostScript invites further exploitation; for example, it might help debug optimized code. If an optimizer performs strength reduction and replaces the use of `i` in `a[i]` with an induction variable p, the compiler can emit Postscript that recovers `i` from p.

The similarity of the stack-walking procedures on the four targets suggests that formal descriptions of stack-frame layout and calling conventions might be devised and used to generate all or part of the stack-walking code. Such descriptions might also be useful for providing a compiler back end with information about register usage or calling conventions.

`ldb`'s model of breakpoints should be replaced with one based on instruction-level single-stepping. Single-stepping lends itself to simpler machine-dependent implementation than does interpretation, and it would eliminate the no-ops emitted by `lcc`. Implementing breakpoints entirely in the debugger keeps the nub simple, but runs the risk of losing important state in a debugger crash, and of being unable to resume debugging with a new debugger. We can solve this problem by enriching the protocol with a special store operation used only for planting breakpoints and by making the nub capable of reporting to a new debugger the instructions overwritten by such stores, in case the connection to the original debugger is lost.

`ldb` needs facilities like source-level single stepping. Implementing such facilities on top of breakpoints is complex, because the event that is expected may not be the one that occurs; a fault may occur when a breakpoint is expected. One solution is to make the debugger internals event-driven [15]. Exporting the mechanisms used to make the debugger event-driven would simplify the implementation of event-driven clients. Event-driven debugging subsumes conditional breakpoints as a special case.

Sec. 4.2 describes the potential reliability and performance benefits of moving the nub into the kernel. Reimplementing the nub interface on top of operating-system support for debugging might have similar benefits, but at some cost in retargetability. Using operating-system facilities has an additional advantage: they may provide access to instruction single-stepping, breakpoints, threads, signals, or exceptions. `ldb` could be designed to take advantage of these kinds of extensions to the nub protocol when they are available, but should continue to function correctly when they are not available.

8 Acknowledgements

Mike Spreitzer helped clarify retargeting issues by independently writing Modula-3 code for the SPARC. Anne Rogers insisted that we write more about ideas and less about code. John Ellis's thorough reading of an earlier draft uncovered many important omissions.

This work was supported by a Fannie and John Hertz Fellowship, an AT&T Bell Laboratories Fellowship, and a summer research internship at Xerox PARC.

References

[1] E. Adams and S. S. Muchnick. Dbxtool: A window-based symbolic debugger for Sun workstations. *Software—Practice & Experience*, 16(7):653–669, July 1986.

[2] Adobe Systems Incorporated, Reading, Ma. *PostScript Language Reference Manual*, 1985.

[3] P. C. Bates and J. C. Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. *Journal of Systems and Software*, 3(4):255–264, Dec. 1983.

[4] B. Beander. `VAX DEBUG`: An interactive, symbolic, multilingual debugger. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging,* in *SIGPLAN Notices*, 18(8):173–179, August 1983.

[5] B. Bruegge. *Adaptability and Portability of Symbolic Debuggers.* PhD thesis, Carnegie Mellon University, September 1985.

[6] T. A. Cargill. Pi: A case study in object-oriented programming. *Proceedings of OOPSLA'86, SIGPLAN Notices*, 21(11):350–360, Nov. 1986.

[7] D. Caswell and D. Black. Implementing a Mach debugger for multithreaded applications. In *Proceedings of the Winter USENIX Technical Conference*, pages 25–39, Washington, DC, Jan. 1990.

[8] G. V. Cormack. A micro-kernel for concurrency in C. *Software—Practice & Experience*, 18(5):485–491, May 1988.

[9] Digital Equipment Corporation, Maynard, MA. *DDT—Dynamic Debugging Technique*, 1975.

[10] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. *Software—Practice & Experience*, 21(9):963–988, Sept. 1991.

[11] C. W. Fraser and D. R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, Oct. 1991.

[12] P. Fritzson. Symbolic debugging through incremental compilation in an integrated environment. *Journal of Systems and Software*, 3(4):285–294, Dec. 1983.

[13] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[14] P. B. Kessler. Fast breakpoints: Design and implementation. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 25(6):78–84, June 1990.

[15] M. A. Linton. The evolution of Dbx. In *Proceedings of the Summer USENIX Conference*, pages 211–220, Anaheim, CA, June 1990.

[16] P. Maybee. pdb: A network oriented symbolic debugger. In *Proceedings of the Winter USENIX Conference*, pages 41–52, Jan. 1990.

[17] MIPS Computer Systems, Mountain View, CA. *MIPS Assembly Language Programmer's Guide*, May 1989.

[18] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[19] R. A. Olsson, R. H. Crawford, and W. W. Ho. A dataflow approach to event-based debugging. *Software—Practice & Experience*, 21(2):209–229, Feb. 1991.

[20] J. K. Ousterhout. Tcl: An embeddable command language. In *Proceedings of the Winter USENIX Conference*, pages 133–146, Washington, DC, Jan. 1990.

[21] D. D. Redell. Experience with Topaz TeleDebugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging,* in *SIGPLAN Notices*, 24(1):35–44, January 1989.

[22] R. M. Stallman and R. H. Pesch. Using GDB: A guide to the GNU source-level debugger, GDB version 4.0. Technical report, Free Software Foundation, Cambridge, MA, 1991.

[23] D. C. Swinehart, P. T. Zellweger, R. J. Beach, and R. B. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, Oct. 1986.

[24] M. Weiser, A. Demers, and C. Hauser. The portable common runtime approach to interoperability. *Proceedings of the 12th Symposium on Operating Systems Principles,* in *Operating Systems Review*, 23(5):114–122, Dec. 1989.

In a multipass program, the earlier passes must transmit information to the later passes. This information is often transmitted most efficiently in a somewhat machine-like language, as a set of instructions for the later pass; the later pass is then nothing but a special purpose interpretive routine, and the earlier pass is a special purpose "compiler." This philosophy of multipass operation may be characterized as telling *the later pass what to do, whenever possible, rather than simply presenting it with a lot of facts and asking it to* figure out *what to do.*

— D. E. Knuth, *The Art of Computer Programming*, vol. 1, 2nd ed., p. 198. Addison-Wesley, 1973.