The Evolution of Dbx

Mark A. Linton
Center for Integrated Systems, Room 213
Stanford University
Stanford, California 94305

Abstract

Dbx is the standard source-level debugger on most Unix[1] workstations. Over the past six years Dbx has grown from a debugger for interpreted Pascal programs to a debugger for compiled C, C++, FORTRAN, Pascal, and Modula-2 programs. Dbx also has been retargetted to a variety of architectures, including VAX[2], Motorola 68000[3], MIPS, IBM RT-PC[4], IBM 370, Sun SPARC, and Intel 80386.

This paper describes the evolution of Dbx and examines how the organization of Dbx has enhanced its portability and extensibility. The structure of Dbx is based on a set of abstractions that define what a debugger must do, not on a decomposition by language or machine. These abstractions provide greater flexibility in handling the unexpected problems associated with retargetting a program.

1 Introduction

Dbx is a source-level debugger that runs on most Unix systems. Compared with earlier Unix debuggers such as Adb [1] and Sdb [4], Dbx contains significant advances in the areas of portability, multilingual support, and ease of use. A port to a new machine architecture takes about one week. Dbx can be used with several programming languages, even in the same application. The Dbx user interface provides a high-level set of commands that can be extended using macros.

Dbx did not start out as a portable, multilanguage debugger. The original version of Dbx, which was called Pdx, was the author's Master's project at the University of California at Berkeley [5]. Pdx was written to add debugging support to the Berkeley Pascal system [3]. The implementation was targetted specifically at the Pascal language and the Berkeley Pascal interpreter ("px").

Despite the limited scope of the original project, Dbx has evolved into a much more general debugger without major changes to the original structure. This evolution has been possible because the implementation is decomposed into logical components that encapsulate debugger functionality.

This paper describes the evolution of Dbx from a single-language debugger for an interpretative Pascal system to a debugger that supports five languages and seven architectures. Following an overview of source-level debugging, the paper presents the major components of the implementation and how they have evolved, relating the experiences porting to Dbx different machines. Future debugging tools should expose these components so that several applications can access them, leading to an "open" debugging system that is simple, extensible, and powerful.

2 Source-Level Debugging

The purpose of a debugger is to help the programmer understand why a program, called the **debuggee**, is producing incorrect results. A debugger can show the debuggee's execution state, including values of variables and a list of currently active functions. The programmer can either view a snapshot of execution state saved in a **dump** file or examine a debuggee process as it is running. On Unix, a debugger normally runs in a separate address space from the debuggee to guarantee that an error in the debuggee cannot affect the debugger's state.

A programmer can set a **breakpoint** to suspend execution of the debuggee process when a specified event occurs. The most common kind of event is execution of code at a specified location in the program. When a breakpoint is reached, the programmer can view the execution state and continue execution to the next breakpoint or to the end of the program. Execution is also suspended when a

[1] Unix is a trademark of AT&T.
[2] VAX is a trademark of Digital Equipment Corp.
[3] 68000 is a trademark of Motorola Semiconductor Corp.
[4] IBM is a trademark of International Business Machines.

hardware trap occurs, as would be caused by dividing by zero or dereferencing an invalid pointer.

A *source-level* debugger provides these execution views and controls in terms of the program source. For example, the user can display a variable's value by specifying the variable's name and can set a breakpoint by specifying a line in a source file.

Figure 2 shows a Dbx debugging session. The "% " indicates a Unix shell prompt; "(dbx) " is the Dbx prompt. The first command the user gives is **run**, which starts the debuggee process. Execution stops when a trap called a "segmentation fault" occurs. Dbx determines where the error occurred and lists the offending source line. The user gives the **print** command to display the value of the suspect pointer ("ptr") and finds that it is nil. The user then gives the **where** command to list the stack of currently active functions, from the most recently called to the first (main).

The user decides to run the program again, this time stopping execution before the error occurs. The user sets a breakpoint at line 10 in the source file with the **stop** command and then runs the program. Execution stops at the breakpoint, and Dbx lists the associated source line. The user prints "ptr", and its value seems reasonable. However, "ptr" is about to be set to the value of "badptr", which is nil. The user then exits Dbx with the **quit** command.

3 Dbx organization

The design of Dbx was guided by the principles of data abstraction, in which a problem is decomposed into types of objects. For a debugger, the fundamental data types are commands, symbols, the source code, the object code, the run-time stack organization, the machine architecture, and the debuggee process. The Dbx organization reflects this set of types. Figure 2 shows the Dbx modules and how they fit together.

3.1 Commands

The command interpreter parses and interprets user input. The goal of the command interface is to provide a simple, high-level set of commands that match what the user wants rather than what the machine or operating system provides. For example, a common debugging technique is to set a breakpoint, print some variables when the program stops, and then continue execution. Dbx supports this technique with the **trace** command. The command **trace** *function* displays a message each time *function* is called and another message each time it returns. Although the command syntax was designed for novice users, a simple macro expansion capability has been sufficient to satisfy experienced users. Unfortunately, the documentation does not emphasize this capability; many users have complained about the long command names, unaware that they can define their own abbreviations.

The implementation of command parsing and interpretation follows the normal structure of a compiler: a scanner reads tokens from input; a parser built with YACC recognizes commands as specified by a grammar; an abstract syntax tree is built during parsing; a pass over the tree checks for correctness; another traversal executes the tree using an evaluation stack.

Symbol lookup is a more complex problem in a debugger than a compiler. The difficulty arises from the need to access variables that are not statically visible. For example, suppose execution is suspended in function f, which was called by function g. The user may wish to display the values of variables local to g. Furthermore, it should be possible to access values in recursive calls.

The Dbx symbol lookup algorithm consists of three steps. First, Dbx looks for the name in a static context, starting with the current function and searching outward as a compiler would. If that search fails, it then looks for the name in a dynamic context, starting with the current stack frame and searching up the call stack. If both searches fail, the symbol could still be defined as local to a compilation unit not in the static context. However, Dbx cannot distinguish between symbols with the same name in several units. Several choices are possible here; it currently picks an arbitrary symbol and prints a warning message.

Dbx also provides a syntax for qualifying names to identify symbols in an outer scope. For example, if execution is suspended in f, which was called by g, and both f and g have local variables named i, the user can use $g.i$ to identify the i local to g.

It took three attempts at writing the lookup algorithm before all the cases were recognized and handled correctly. The problem was not so much getting the implementation correct as understanding all the possible cases.

```
% dbx a.out
dbx version 3.4 of 7/27/87 11:24 (lurch).
Type 'help' for help
(dbx) run
Segmentation fault in P at line 12 in file "source.c"
   12        *ptr = param;
(dbx) print ptr
nil
(dbx) where
P(param = 3), line 12 in file "source.c"
main(argc = 1, argv = 0x7fffe210), line 105 in file "main.c"
(dbx) stop at 10
[1] stop at 10
(dbx) run
[1] stopped in P at line 10 in file "source.c"
   10        ptr = badptr;
(dbx) print ptr
0x7fffe218
(dbx) print badptr
nil
(dbx) quit
%
```

Figure 1: Sample Dbx session

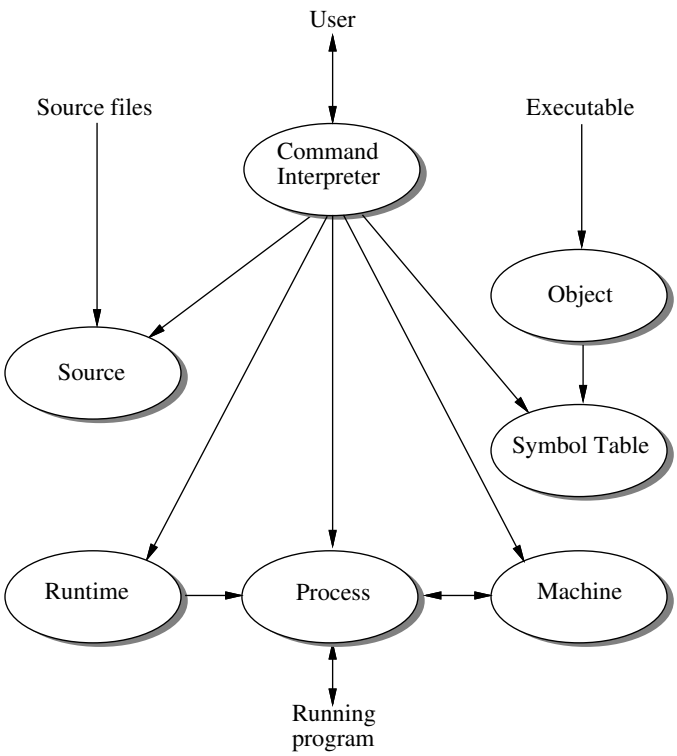

Figure 2: Organization of Dbx modules

3.2 Breakpoints and tracing

Most debuggers provide the ability to set a breakpoint that suspends execution when the program reaches a certain location. Such breakpoints support "binary search" debugging, in which one finds an erroneous condition, sets a breakpoint "halfway" between the beginning of execution and the fault point, and then restarts the program. If the execution state looks correct at the half-way point, then the user sets another breakpoint at a three-quarters point and continues execution. The user often checks whether execution is correct by examining the value of one or more variables.

Dbx provides a set of **stop** commands for suspending execution at a breakpoint and a set of **trace** commands for displaying a value during execution. Stop and trace points can be specified at a source line, within a procedure, or when an arbitrary condition becomes true. Trace information can also be requested as each line is executed.

The events module in Dbx implements breakpoints and tracing. The first implementation interpreted stop and trace commands directly. This approach was not flexible enough to handle many cases; in fact, tracing hardly worked at all. The problem was that internal breakpoints had to be created for commands such as **trace** *f*, where *f* is a function. Because the user could interrupt execution at arbitrary locations while debugging, Dbx frequently got confused about what breakpoints should be present.

3.2.1 Event representation

The solution was a major overhaul of event management. The current approach represents all user commands uniformly in terms of events and actions. Dbx translates this representation into low-level breakpoints and interprets associated primitive actions. For example, the representation for the command **trace** *f* could be expressed as

```
when ($proc = f) {
    printcall(f);
    once ($pc = $retaddr) {
        printrtn(f);
    }
}
```

The symbols "$proc", "$pc", and "$retaddr" denote the entry point for a procedure, the value of program counter, and the return address for the current function, respectively. The **when** and **once** keywords distinguish an event of general interest from a one-time-only event, respectively. In this example, the execution sequence consists of the following steps:

1. A breakpoint is set for the entry to *f*.

2. When *f* is called, Dbx suspends execution and displays a message about the call, including parameter values. A second breakpoint is set for the current return address, which is the instruction following the call to *f*. Dbx then continues execution without user intervention.

3. When *f* returns, Dbx suspends execution and displays a message about the return, including the return value. Dbx deletes the return breakpoint and continues execution.

The advantage of this representation is that Dbx can stop execution when a function returns without knowing the address of the return statement(s). Therefore, Dbx can trace functions that do not have debugging information.

3.2.2 Conditional breakpoints

Dbx implements conditional stops and traces correctly, but the algorithm is so slow that using this capability is often not practical. The representation of "stop at 10 if *condition*" is

```
when ($line = 10) {
    if (!$condition) {
        continue;
    }
}
```

The straightforward implementation of this event is to suspend execution at line 10, check the condition from within Dbx, and continue unless the condition is true. Unfortunately, this can create an execution cycle with frequent context switches between the debuggee and Dbx, thereby slowing execution significantly. A far better approach would be to patch the debuggee so that it checks the condition, thus eliminating the context switches.

Conditional tracing in Dbx is often very slow and therefore rarely used. Dbx users have not voiced a strong desire for this situation to be improved, perhaps because they do not realize what they are missing.

3.3 Symbols

Dbx uses a uniform representation for symbols. This format is a union of all the types of symbols

in different programming languages. The three aspects of symbol management are the external representation that is generated by compilers, the internal representation, and how language-dependent operations are implemented. The Dbx implementation contains a separate module for each aspect.

3.3.1 External representation

Berkeley Unix stores symbol information in a format called "stabs" (for *s*ymbol *tab*le entrie*s*). Following the object code for the functions and data, the symbol information appears in two parts: an array of symbol records and a string table containing the names of all the symbols. Each symbol record contains the offset of the symbol's name into the string table.

The symbol records contain some fields with type information; however, the language information that can be represented in the fields is very limited. For example, there is no way to represent array subscript bounds. To change the symbol table format would have required modifications to many other system tools, including the assembler, linker, and library archiver. Such a change would also have required modifications to any user programs that rely on the format. To modify as few programs as possible, the choice was made to keep the existing format and rely on an important attribute of symbol names—they can be of arbitrary length. Therefore, compilers that generate information for Dbx append an encoding of type information at the end of each symbol's name. This hybrid name is called a "stabstring".

For example, the C declaration

```
typedef struct {
    int x;
    char* y;
} a;
```

generates the stabstring

```
a:t13=s8x:1,0,32;y:14=*2,32,32;;
```

Dbx recognizes the ":" as separating the variable name "a" from its type. The character immediately following the ":" indicates what kind of symbol it is. In this example, the "t" means "typedef". A number is associated with every type to avoid replicating information for types that are referenced more than once. This number is unique within the compilation unit. In the example above, the type number is 13. The "s" specifies a structure definition, which is followed by its size in bytes and a list of fields separated by semicolons. Each field has a name, a ":" separator, a type, bit offset, and bit length.

In the example above, the field x is defined by the string

```
x:1,0,32
```

Type number 1 refers to the builtin C type **int**. The value of x starts at bit 0 in the record and is 32 bits long. The field y is defined by the string

```
y:14=*2,32,32
```

A new type with number 14 is defined to be a pointer to type number 2, which is the builtin C type "char". The value of y starts at bit 32 in the record and is 32 bits long.

The original motivation for encoding information in names was a short-term desire to avoid modifying the assembler, linker, and other tools. It was also quick—it took about one week to modify the C compiler to generate the encodings. In retrospect, this approach has had many long-term advantages. As Dbx evolved to support additional languages, stabstrings have been easy to extend. For Modula-2, for example, it was necessary to add information for modules, opaque types, and nested procedures. Because names are arbitrary length and other tools are not concerned with the contents of a name, this information was added by changing only Dbx and the Modula-2 compiler. Thus, what began as a "quick-and-dirty" solution turned out to be a wise design decision.

3.3.2 Internal representation

The Dbx object file module builds a symbol table from the information stored in the executable image. Dbx represents symbols internally with a structure containing a fixed-length section and a varying-length section for information specific to a symbol. The fixed-length section contains the following fields:

name	identifier
language	pointer to procedure vector
class	variable, function, type, ...
storage	register, stack, heap
type	pointer to another symbol
chain	link to next symbol in a list
level	lexical depth
block	pointer to outer scope
next_sym	hash table chain

The varying-length section contains information such as the stack offset for a local variable, code address for a function, and range bounds for a subrange type. Figure 3 shows the representation for several C declarations.

This representation was based on the symbol table organization used by the Berkeley Pascal translator, Pi. Because type equivalence in Pascal is determined by a type's name, it is necessary to represent the entire type graph. This requirement led to a general symbol representation in Pi, which was easy to adapt to other languages. If Dbx had started with the traditional C symbol representation, which packs type information into a single word, it would have been more difficult to support other languages.

A shortcoming of the symbol representation is that size information should be present for *all* symbols. For one or two languages and compilers, symbol sizes can be computed correctly. However, when Dbx was retargetted to machines requiring word alignment or compilers that used different strategies for choosing the size of an enumerated type, it became clear that the compiler should generate size information for all symbols.

3.3.3 Language-dependent operations

The Dbx **print** command evaluates and displays the value of an expression. The display format can vary depending on the language associated with the symbols in the expression. The syntax of Pascal and Modula-2 sets is different, for example. Dbx also provides the **whatis** command to display the declaration of a symbol using the syntax of the symbol's language.

Dbx uses an array of procedure variables to implement language-dependent operations. Every symbol contains a pointer to a language-specific data structure. This pointer is assigned as the symbol is read from the external representation. The language data structure contains the name of the language and an array of procedure variables. A language-dependent operation is called on a symbol by dereferencing the symbol's pointer, indexing for the desired operation, and calling indirectly through the procedure variable. The reason every symbol contains a language pointer instead of having a global language mode is to make sure an operation is always defined. For example, displaying a Pascal set would make little sense in C language mode.

As Dbx has evolved to support new languages and features, several language-dependent operations have been added. When FORTRAN was added, for example, it was necessary to define multi-dimension array subscripting as language-dependent because Pascal and FORTRAN use row-major and column-major addressing, respectively. Another addition arose from the implementation of the **call** command, which allows the user to execute a procedure in the debuggee program. A language-dependent operation was added that tests whether to pass a parameter implicitly by address, because arrays are passed by address in C and by value in Pascal.

Although the support for multiple languages in Dbx is adequate for the current set of languages, several problems remain to be solved. Inline procedures are only partially-supported: Dbx can recognize where execution has suspended within an inline call and display a "virtual" stack frame entry, but there is no support to set a breakpoint inside an inline. Language features such as overloading, generics, and inheritance are not supported at all.

3.4 Source-address mappings

When a user wants to stop execution at a line in a source file, Dbx must compute the address in the object code that corresponds to the given line. Conversely, whenever execution stops, Dbx must compute the source file and line number that corresponds to the program counter. These mappings are implemented using two binary-search tables, one for files and one for lines. Each element in the file table contains a file name, object-code address, and an index into the line table. Each element in the line table contains a line number and object-code address. Figure 4 shows how the tables are organized.

The tables are sorted by address, from smallest to largest. To compute the source file and line number for a given address, an algorithm similar to a binary search is performed on each table to find the element with the largest address less than the given address. To compute the address for a given source file and line number, first a linear search is performed to find the file's name in the file table. Then a second linear search is performed on the region of the line table to find the given line number. A binary search cannot be used on the line number because a source line may be associated with several object code addresses. For example, a C for-loop typically has two addresses, one corresponding to the beginning of the loop and one

```
typedef struct A {
    char id[50];
    struct A *next;
} *B;

int f(x)
B x;
{
    ...
}
```

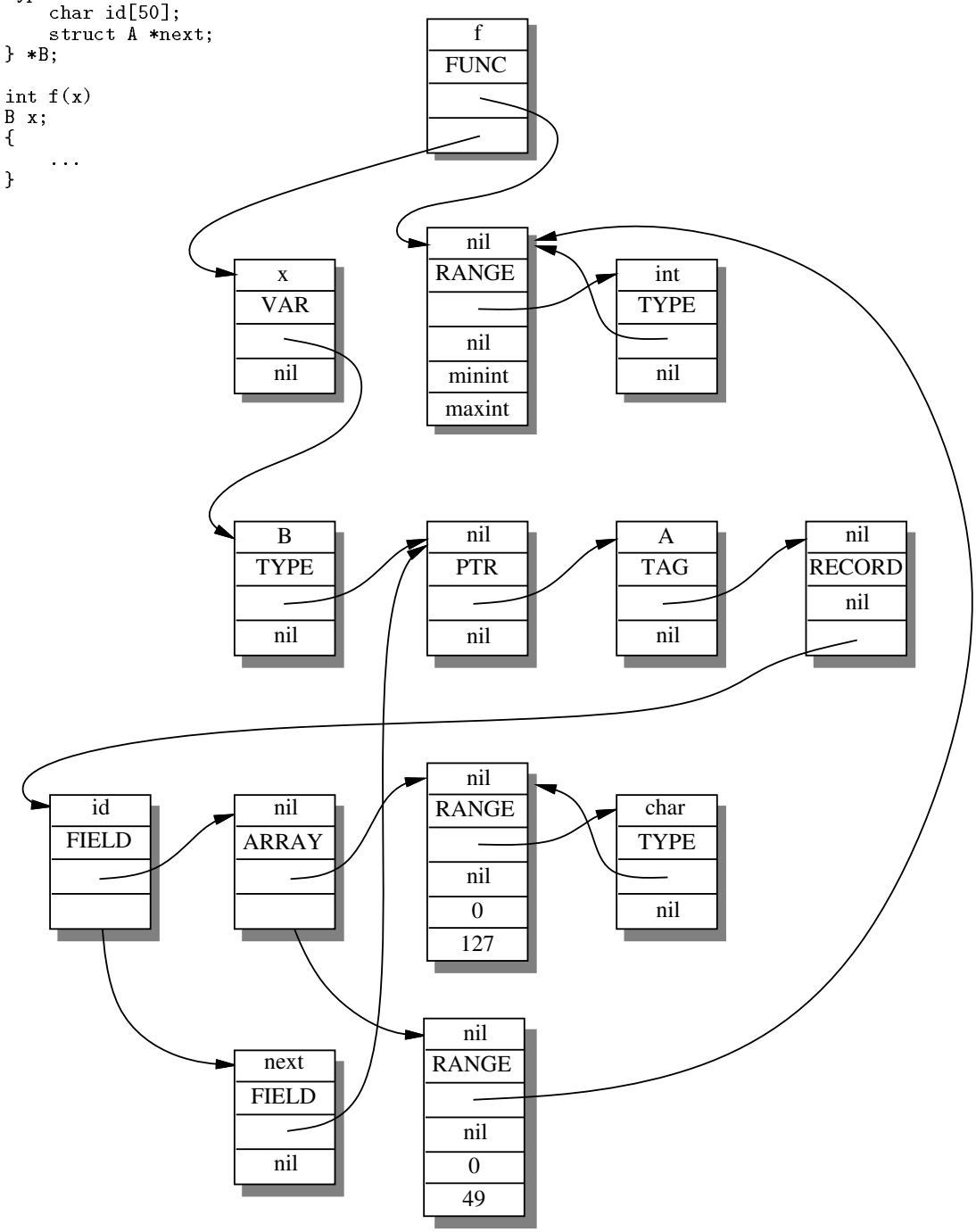

Figure 3: C declarations and symbol representation

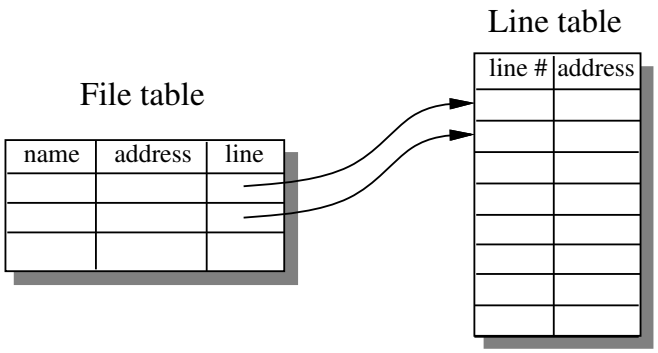

Figure 4: Source-address mapping tables

corresponding to the increment at the end of the loop.

3.5 Process management

The process module encapsulates the operating system mechanisms for controlling execution of the program being debugged. The module is divided into two layers: a high-level interface used by the command interpreter and a low-level interface to machine-specific data such as the values in registers.

The top layer includes operations to run, continue, and single-step a process, as well as to read and write the process' memory. This layer automatically continues execution after handling an implicit breakpoint associated with a trace command. When a dump is being examined (instead of a suspended process), the top layer implements memory access with the appropriate file access; otherwise, it passes the request down to the bottom layer.

The bottom layer provides operations to create, destroy, continue, step, read and write memory, and read and write registers. The memory interface can access an arbitrary section of the process' address space. Some versions of Unix support this kind of access directly; other versions allow only single-word accesses.

The bottom layer caches register and memory information to avoid unnecessary accesses to the process' address space. These accesses can be quite expensive on some systems (equivalent to several context switches).

Although the number and kind of registers is machine-dependent, they are accessed through the process layers because of system dependencies. On some Unix systems, for example, it is necessary to use the kernel's per-process data structure to find the register values.

3.6 Run-time information

The major function of the run-time module is to "walk the stack" from the current function through each call back to the start of the program. There are three distinct uses of stack walking. The most obvious use is to list all the active calls when execution is suspended. The second use is to find the stack frame associated with a given function. This search is needed when the user attempts to display a variable local to a function up the call chain. For example, consider the following C code:

```
int g()
{
    int x = 3;
    f();
}

int f()
{
    int y;
     ...
}
```

If execution is suspended in f and the user wants to display x, Dbx must find g's activation record on the stack.

The third use of stack walking is for commands that set the frame used to start the search for a function. The **up** command moves the current frame up the stack to the caller of the current function, and the **down** command moves down to the callee of the current function. Originally, Dbx only had a concept of a current function, not a current frame. Consequently, Dbx could only print the values of local variables associated with the most recent call of a recursive function.

Defining a current frame also solves a subtle problem with the **next** command when stepping

through recursive functions. Consider the following code:

```
f(x)
int x;
{
    if (x > 0) {
        f(x-1);
        g(x);
    }
}
```

The **next** command single steps across a function call. That is, if a statement is a function call, **next** continues execution up to the statement following the call. The **step** command, in contrast, continues execution to the first statement in the function that is called. In the code above, a naive implementation of a **next** command at the call to *f(x-1)* would continue to the call to *g*, but in the recursive call. To implement the proper semantics for **next**, Dbx continues execution in a loop that terminates when the current frame matches the starting frame.

Stack walking can be more difficult if a function is encountered that does not have compiler-generated information. The difficulty depends on the target architecture and compiler. On a VAX, which has a call instruction that sets up a standard stack frame, it is always easy to walk the stack. On a 68000 or 80386, however, Dbx must decode the instructions at the beginning of the function to determine which registers were saved and to check for optimizations, such as omitting the stack pointer inside a leaf function (a function that does not call other functions).

Some debuggers assume that all functions have debugging information and these debuggers might fail or produce incorrect results when they encounter a function without any information as they walk the stack. In practice, functions are often compiled without symbol information to reduce disk storage or speed up compilation. Therefore, although it is reasonable to give limited information in the presence of such functions, it is unacceptable to terminate or give incorrect information.

Unix signals also complicate stack walking. A signal is an asynchronous event that transfers control to a specified function, called the *signal handler*. When a signal occurs, the operating system stores a special record on the stack before transferring control to the signal handler. Thus, Dbx must detect and skip over this signal record. This logic is not difficult conceptually, but is often time-consuming to implement because the structure of the record is not well-documented on many systems.

3.7 Machine interface

The machine module decodes instructions, formats data, and implements breakpoints. No attempt has been made to encapsulate machine dependencies completely in this module. Some machine characteristics, such as the layout of the stack frame, are implemented more easily within the run-time or process modules.

3.7.1 Decoding instructions

Instruction decoding has two uses. First, although Dbx is a source-level debugger, it is occasionally useful to display machine instructions. This feature was originally added to help debug Dbx itself, but it has continued to be valuable.

The second use of decoding is to implement the Next-address function. Given an address A, this function determines what value the program counter would have if the instruction at A were executed. For branch instructions, Next-address returns the target address; otherwise, it returns A plus the size of the instruction at A.

Dbx uses the Next-address function to single-step execution. If execution is suspended at A, Dbx sets a breakpoint at Next-address(A), continues execution (which will stop after executing one instruction), and then removes the breakpoint.

Some machines have a special CPU flag that when set causes a trap after a single instruction is executed. Even when a machine has a special flag, however, using Next-address can eliminate many of the context switches between Dbx and the debuggee. One way to implement single-stepping by source line is to single-step by machine instruction until execution reaches an address that corresponds to the beginning of a source line. Instead of actually single-stepping, Dbx uses Next-address to compute where execution would be. When Next-address finds a conditional or indirect branch, it continues execution up to the instruction before trying to compute the new address. This approach reduces the number of context switches from two per machine instruction to two per branch, which can be significant when tracing.

3.7.2 Formatting data

Dbx can read from any address in the process' memory and can display the data in decimal, octal, hexadecimal, character string, or floating point formats. This capability is useful in examining variables that do not have type information.

Formatting data requires knowledge of the machine's bit and byte ordering as well as its floating point representation. The machine module encapsulates this knowledge with a function that extracts a range of bits from a data word. This function is also used to display the value of a C bit field.

3.7.3 Setting breakpoints

To set a breakpoint at location A, Dbx overwrites the instruction at A with an instruction that will cause a trap. The particular instruction, its length, and the kind of trap all vary across machine architectures. Before overwriting the location, Dbx copies the instruction at A into a local data structure. To remove a breakpoint, Dbx writes the copied value back into the target location.

Two distinct events may require setting a breakpoint at the same location. For example, suppose the user entered the following commands:

```
trace f
stop in f if x < 0
```

The first command asks Dbx to display a message whenever function f is called. The second command asks to suspend execution when f is called and the value of x is negative. Both requests are implemented by setting a breakpoint at the beginning of f. Using the implementation of breakpoints described above, the following sequence is possible:

1. Set a breakpoint for the trace command at the beginning of f:
   (a) Save the old instruction
   (b) Write a breakpoint instruction
2. Set a breakpoint for the stop command at the beginning of f:
   (a) Save the old instruction (now a breakpoint instruction)
   (b) Write a breakpoint instruction
3. Continue execution
4. Remove the breakpoint for the trace command by writing the saved instruction.
5. Remove the breakpoint for the stop command by writing the saved instruction (a breakpoint instruction!).

This sequence makes it possible to leave a breakpoint instruction at the beginning of f instead of the original instruction. To avoid this problem, Dbx keeps a list of all breakpoints to search when a breakpoint is set to see if the location has already been set.

Dbx sets all breakpoints every time execution continues and removes all breakpoints every time execution stops. This approach is simple but inefficient; many unnecessary accesses can occur, particularly when tracing. A better approach would be to have the instruction decoder check in the breakpoint list before accessing the program's memory.

4 Porting experiences

Porting Dbx to a different machine takes about one week, where roughly half the time is spent coding and the other half testing and debugging. A collection of test cases that were added for regression testing have become the driving force behind the porting procedure. A port is defined to be complete when all the tests run successfully, and the central activity during a port is to get successive test cases to work.

Every port consists of modifying Dbx to handle (1) the machine's instruction set, (2) differences in the site's version of Unix (the most common differences are in accessing process information), and (3) the calling sequence used by compilers. Dbx's organization has "bent" slightly with each port to handle newly-found machine dependencies, but it has never needed a major restructuring.

Regardless of how many times Dbx had been ported or how much was known about the target machine, every port included at least one surprise. Sometimes the surprises were bugs in the target system. For example, in one port the operating system crashed as a result of Dbx reading a nonexistent address in the debuggee.

Usually the bugs revealed the need for new functionality in Dbx because the port was to a new machine with new compiler or operating system technology. For example, the RT-PC compiler always passes arguments to a function in registers. Inside the function, an argument value may stay in its register, be moved to a different register, or be copied onto the stack. Thus, the location of a variable's

value may depend on where execution is within the program. For example, a value could be in r2 at a function entry point and on the stack after the function prolog. Because debugging traditionally has meant disabling compiler optimizations, Dbx assumed that every symbol had a statically-assigned address, stack offset, or register number. With the arrival of RISC machines, debugging (at least partially) optimized code is a requirement.

Despite problems that would have been impossible to anticipate, Dbx has been able to accommodate new features with a relatively small number of changes. In the case of register parameters on the RT, it was necessary to replace an access to the field that contained a parameter's address with a function call that computed the address based on the current program location.

5 Future directions

Dbx has been relatively easy to adapt to different languages and machines, but it needs additional functionality to meet the needs of more integrated programming environments and the diverse requirements of parallel, distributed, and real-time applications. At the same time, Dbx is already too big and difficult to maintain and extend. The solution is to split Dbx into several tools.

The first step is to remove functionality from Dbx that should be handled by existing programming tools. Showing source upon reaching a breakpoint or setting a new breakpoint should be handled directly by an editor. Parsing expressions should be handled by a compiler, especially when supporting languages like C++ with operator overloading and inheritance.

Integrated environments such as Smalltalk [2] combine the editor, compiler, and debugger into a monolithic system. Unfortunately, this coupling makes it difficult to support multiple editors, languages, or debuggers. Our solution to the problem of integration is to have tools communicate with other tools directly by sending messages. For example, a debugger could send an expression to a compiler, forward the generated code to a linker for relocation, load the code into an executing program, and have the program execute the code in the proper context. The overall architecture of this kind of environment is described in a separate paper [6].

For the component of debugging that handles program execution, we are building a separate application for monitoring and controlling processes. This application, called Ndb, will be a *server* that provides access to one or more processes and address spaces. Figure 5 shows an example of how Ndb might be used.

Ndb will support a standard protocol that defines operations on processes, address spaces, and breakpoints. It is critical that this protocol be simple yet powerful enough to support debugging and monitoring of sequential, parallel, or distributed applications. In particular, Ndb will *not* resolve program names or interpret language-specific expressions.

The primary advantage of a server is that several clients can share access to the same target through Ndb. For example, a performance monitor could show execution time information during a debugging session to help find performance problems. By adding new client applications, the debugging environment can be extended without changing current applications or the Ndb server. A second advantage is that the server can run on a different machine from the clients.

The implementation of Ndb can be customized to the target environment. If possible, Ndb would map the target's address space into Ndb's memory. When a request is sent to set a conditional breakpoint, Ndb would patch the test into the debuggee if sufficient memory were available; otherwise Ndb would set a trap at the point and check the condition itself.

6 Conclusions

The organization of Dbx is the key to its portability and extensibility. The relative ease in handling the surprises associated with a port demonstrates that a good design leads to portability. The converse is not true; a portable design is not necessarily good. Software designed for specific portability requirements is likely to fail because it is impossible to predict future targets to which one will want to port.

The use of stabstrings to represent type information illustrates a paradox of software systems: the best design for an product's lifetime might not be the best technical design at a particular time. For a fixed set of languages and compilers, there are many formats that are superior to stabstrings in clarity and performance. However, because of the continual changes to the environment around Dbx, stabstrings were the right choice for the long term.

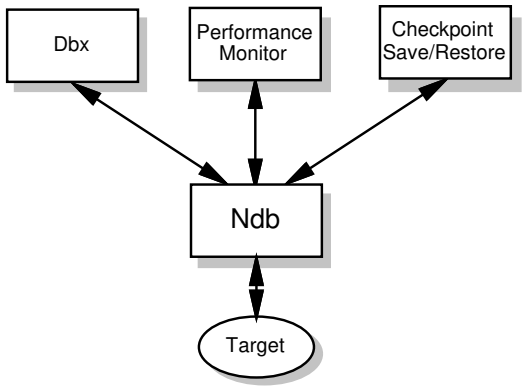

Figure 5: Using the Ndb server

In the future, Dbx should be split into several applications so that portions of its functionality can be more easily used and extended. This partitioning will expose the internal abstractions in Dbx and make it easier to incorporate debugging into an integrated programming environment. Ndb is a first step to this partitioning in the context of the execution environment.

7 Acknowledgments

Mike Clancy was my Master's project advisor while I worked on Pdx at Berkeley. Mike Powell contributed to the stabstrings design, especially to support Modula-2. Jim Terhorst and Mark Himmelstein helped with the IRIS and MIPS ports, respectively. Larry Breed helped me understand the compiler issues for the RT-PC port. Doug Pan is largely responsible for the design of Ndb. Throughout Dbx's evolution, I have had many productive discussions about its design and implementation with Evan Adams.

References

[1] S. Bourne and J. Maranzano, "A Tutorial Introduction to Adb", AT&T Bell Laboratories, Murray Hill, New Jersey, 1977.

[2] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Massachusetts, 1984.

[3] W. Joy, S. Graham, and C. Haley, "Berkeley Pascal User's Manual", Version 2.0, EECS Department, University of California at Berkeley, October 1980.

[4] H. Katseff, "Sdb: A Symbolic Debugger", AT&T Bell Laboratories, Holmdel, New Jersey, 1979.

[5] M. Linton, "A Debugger for the Berkeley Pascal System", Master's report, Computer Science Division, University of California at Berkeley, June 1981.

[6] M. Linton, "Distributed Management of a Software Database", *IEEE Software*, Vol. 4, No. 6, November 1987, pp. 70-76.