

Debugging Support For Dynamically Generated Code

by

Jonathan Zachary Litt

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

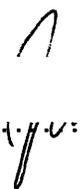
at the

Massachusetts Institute of Technology

May 1997

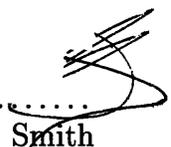
© Jonathan Zachary Litt, MCMXCVII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly
paper and electronic copies of this thesis document in whole or in part, and to grant
others the right to do so.

Author 
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by 
M. Frans Kaashoek
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Certified by
Massimiliano A. Poletto
PhD Candidate, Electrical Engineering and Computer Science
Thesis Co-Supervisor

Accepted by 
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

OCT 29 1997



Debugging Support For Dynamically Generated Code

by

Jonathan Zachary Litt

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1997, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes a framework for the debugging of dynamically generated code. Dynamic code debugging differs from normal debugging in that there must be additional run-time support for the generation, storage, and processing of debugging information. We discuss the issues involved in this process, which we call Dynamic Debugging-data Generation (DDG), and present the design and implementation of a DDG framework for 'C (*tick-C*), an extension of ANSI C that supports dynamic code generation. This approach supports seamless debugging of 'C programs in a C debugger, thus contributing to the usefulness of the 'C programming environment.

Thesis Supervisor: M. Frans Kaashoek

Title: Associate Professor of Computer Science and Engineering

Thesis Co-Supervisor: Massimiliano A. Poletto

Title: PhD Candidate, Electrical Engineering and Computer Science

Acknowledgments

I extend a sincere and heartfelt thanks to Massimiliano Poletto for being both a mentor and a good friend throughout the course of this project. I wish him the utmost success in his future academic and professional careers. I also thank Frans Kaashoek for supporting this research and providing valuable guidance and editorial input.

I thank Joann Yeh for her love and support and for putting up with my crazy hours. I also thank all of my friends for making my undergraduate years so enjoyable. Last but not least I thank my parents and my sister for being the wonderful and loving people that they are.

Contents

1	Introduction	7
1.1	Motivation	7
1.2	Contribution	8
1.3	Thesis Overview	8
2	Background	9
2.1	Dynamic Code Generation	9
2.2	The 'C Programming Language	11
2.2.1	'C Overview	11
2.2.2	'C Syntax	11
2.2.3	'C Implementation	13
2.3	Debugging	14
2.3.1	Debugging Overview	15
2.3.2	Stabs Overview	16
2.3.3	Discussion	17
3	Design	20
3.1	Constraints and Criteria	20
3.2	DDG Design Issues	21
3.3	DDG Design for 'C	23
3.3.1	Interface	23
3.3.2	Data Storage	24
3.3.3	Ordering and Naming	26
3.3.4	Summary	27

4	Implementation	28
4.1	VCODE-Stabs	28
4.1.1	v_stabs	29
4.1.2	v_stabd	30
4.1.3	v_stabs_setfile	31
4.1.4	v_stabs_global_types_{begin,end}	31
4.1.5	v_stabs_func	32
4.2	'C Modifications	32
4.3	Debugger Modifications	33
4.4	Integration	34
5	Results	36
5.1	Debugger Examples	36
5.2	Performance	41
6	Conclusions and Future Work	43

List of Figures

- 2-1 Several common stab types 17
- 2-2 Flow of information in a non-interpreted debugging environment. 18
- 2-3 Flow of information in an interpreted debugging environment. 18

- 4-1 VCODE-stabs run-time invocations. 35

- 5-1 Compile-time performance (seconds) 41
- 5-2 Instantiation-time performance (seconds) 41
- 5-3 Run-time performance (seconds) for dynamic functions of various sizes, each
executed one million times. 42

Chapter 1

Introduction

Dynamic code generation (DCG) is the creation and execution of binary machine instructions in the run-time environment of a computer program. Dynamically generated code is “tailor-made” to the specific instance of a running program, and is thus able to benefit from optimizations that would not otherwise be possible during the static compilation of source code.

Unfortunately, use of DCG has been restricted to application-specific implementations, mostly due to the lack of high-level software tools that support it [12]. Just as the development of high-level languages such as Fortran and Pascal gave programmers more leverage in managing software complexity, the development of high-level languages and tools for DCG could allow programmers to more easily take advantage of it in everyday programming. A few such languages have recently been developed, most notably ‘C (*tick-C*), an extension of ANSI C that supports various DCG constructs [5, 21].

1.1 Motivation

Debugging can be a difficult and time-consuming process. DCG-based languages such as ‘C make debugging even more difficult, because they introduce a level of indirection by allowing for code that specifies the creation of further code. Thus, there is all the more need to be able to examine and control the run-time execution environment with respect to its corresponding source code, especially when it comes to building large, complex applications.

Another motivation for this research is that a debugger is a valuable teaching device. Often, the best way to learn a language or to understand pre-existing source code is to

step through it in a debugger. Once again, this is especially true in the case of DCG-based languages. A tutorial that could allow the user to step through DCG examples would give valuable insight into the nature of the run-time environment, insight that might be difficult to discover from simply compiling and executing example source code. This is analogous to other engineering disciplines. For example, a mechanical engineer studying engine design spends a significant amount of time examining the components of a running engine.

1.2 Contribution

The goal of this research is to identify and solve the problems associated with providing debugging support for high-level DCG-based programming languages. Much like DCG support in general, this requires transferring what used to be a static framework into a run-time system. Whereas standard compilers generate debugging data at compile time, DCG-based debugging is akin to DCG – because the dynamically generated code is not known until it is generated at run time, only at that point can debugging information itself be generated. The phrase we have coined for this process is “dynamic debugging-data generation,” or DDG.

To implement DDG, we have augmented the ‘C compiler to support dynamic debugging in a manner backwards-compatible with the C debugger. This allows the ‘C programmer to utilize the familiar features of the C debugger, such as breakpointing, single-stepping, and variable printing, on dynamically generated code. The backwards-compatible approach is preferable because it reuses much of the current C debugging framework, thus supporting integrated debugging of C and ‘C source code. The result is a version of the ‘C programming environment that is much more amenable to complex software engineering projects.

1.3 Thesis Overview

This thesis is organized as follows: Chapter 2 provides a thorough background on DCG, ‘C, and debugging. Chapter 3 discusses the issues relevant to debugging high level languages for DCG and presents a design for a ‘C debugging framework. Chapter 4 describes the implementation of this framework. Chapter 5 analyzes the results of the implementation and introduces several examples of its usage.

Chapter 2

Background

This thesis discusses the convergence of two largely overlooked topics in programming language design: debugging and dynamic code generation. Debugging is paid little attention by the research community [15], while dynamic code generation has only emerged in recent years as a subject of considerable interest [12]. For these reasons there is no known background research that is directly applicable to the combination of the two. (Interestingly enough, however, DCG has been used to optimize the implementation of conditional breakpointing in a debugger [13].)

This chapter instead presents background information for the two topics individually. Enough information is presented for understanding subsequent chapters. Section 2.1 gives an overview of dynamic code generation, and Section 2.2 describes ‘C in particular. Section 2.3 explains how debuggers work, and how they are supported by source language compilers.

2.1 Dynamic Code Generation

The purpose of DCG is to improve program speed by taking advantage of run-time information. Its goal could be stated as, “Never evaluate something more than once.”[18] Techniques used to achieve this goal include run-time optimization, partial evaluation, and executable data structures. We define these below. The benefit of DCG depends on several things, including the amount of generated code, the nature of the optimizations performed, and the number of times the resulting code is executed.

To produce fast executable programs, standard compilers perform various optimizations

while transforming the input source language into machine instructions. These optimizations work by eliminating redundant, constant, or unused information in the source code. For example, constant folding is the transformation of an expression such as “1 + 1” into the expression “2”. This comes at the cost of an extra add instruction during the compilation phase, but ensures that no add will ever be performed during the run-time execution of that expression. Other such optimizations include constant propagation, dead code elimination, common sub-expression elimination, strength reduction, and loop unrolling [1, 2].

These optimizations have limited potential for performance enhancements when applied to static code because most useful programs are parameterized for run-time data. If instead they could be performed at run time, then more data would be available for optimization. This process is referred to as *run-time optimization*.

A technique closely related to run-time optimization is *partial evaluation*. Partial evaluation is the transformation of a general-purpose procedure of N arguments to a specialized procedure of K arguments (where $0 \leq K < N$), where the remaining arguments are held constant. The resulting procedure is optimized for the specified constants. This technique has been used heavily in the context of DCG [14, 13, 18, 5], although it is also commonly used as a source-to-source optimization method. For instance, it has been used to boost the performance of a graphics shading language [11].

So-called *executable data structures* take this one step further by using specialized data structure traversal functions or by embedding the data values into the code. For example, a general purpose function that implements binary search on a sorted integer vector could be replaced by a specialized search function for every vector instance, each function having the values from the input vector hardwired into the instruction stream [7].

There are time and space tradeoffs for all of these techniques. If the overhead of performing run-time optimizations and emitting code outweighs the performance benefits of the run-time optimized code, then there is no reason to create the code dynamically. Thus, it is most beneficial when the generated code is used a significant number of times. In this sense, DCG could be considered a general form of memoizing, with all the performance tradeoffs thereof. The difference is that by hardcoding the instruction stream, DCG avoids a level or more of indirection and additionally allows for further performance benefits via the techniques mentioned above (constant folding, constant propagation, strength reduction, etc.).

Another domain of applications that benefit from DCG are compilers for small languages and compiling interpreters. There are myriads of small, application-specific languages that are used in run-time critical environments such as databases and computer graphics rendering systems. In some cases, high-level access to DCG can provide more than just improved performance; it can provide ease of implementation. For example, a language that supports arbitrary boolean and arithmetic circuit descriptions was implemented in fewer than 100 lines of ‘C [20]. The ease of implementation results from the fact that the compiler’s work is done as soon as it emits the dynamic code; it need not worry about implementing an opcode interpreter or some other form of run-time support.

2.2 The ‘C Programming Language

‘C is a superset of ANSI C that allows flexible, high level, and efficient specification of dynamically generated code [5]. We describe ‘C in enough detail to provide context for the discussion of ‘C debugging. Section 2.2.1 gives a brief overview of the language, Section 2.2.2 defines the syntax of the language, and Section 2.2.3 discusses its implementation. A more complete discussion can be found in [5, 7, 21].

2.2.1 ‘C Overview

Consider the typical steps taken by a programmer writing a simple program. First he might write a few low-level procedures, and then some high-level procedures that call the low-level procedures. When finished writing all the code, the programmer compiles everything to produce an executable. Finally, he runs the executable.

The idea behind ‘C is to allow similar versions of these steps to be performed at run time, thereby allowing ‘C programs to generate code on the fly. The corresponding terminology is defined as follows: dynamic code is *specified* at run time; these specifications can then be either *composed* to build larger specifications or *instantiated* (compiled at run time) to produce executable code. Finally, the dynamically generated code may be executed.

2.2.2 ‘C Syntax

‘C supports this functionality with the introduction of two unary operators, ‘ (backquote, or “tick”) and \$, and two postfix-declared type constructors, `cspec` and `vspec`. `cspec` stands

for “code specification,” and `vspec` stands for “variable specification.” Programmers use these extensions to denote code that should be generated at run time. The units of code specification are ANSI C expressions, statements and variables.

A backquote expression indicates a single unit of dynamic code. Applying a backquote to an expression of type `T` results in an object of type `cspec T`. For example, the tick-expression `'4` specifies code to generate the integer constant 4, and has the type `int cspec`. Applying a backquote to a compound statement produces an object of type `void cspec`; this type can also indicate a specification for arbitrary code. Backquote expressions are statically typed, meaning that some code generation costs can be pushed to compile time.

The special form `compile()` takes an object of type `void cspec` and returns a pointer to a dynamically generated function. This is shown in the ‘C version of “hello world”:

```
/* Create the specification for the dynamic code. */
void cspec hello = '{ printf("hello world\n"); }';
/* Compile the expression into a function which returns void. */
hellofunc = compile(hello, void);
/* Execute the function. */
(*hellofunc)();
```

In the same way that a `cspec` denotes a specification for dynamic code, `vspec` denotes the specification for a dynamic variable. The most common reason for declaring a variable as a `vspec` is that it needs to appear across a range of discrete `cspecs`. A `vspec` can be initialized outside the bounds of a backquote expression with use of the special forms `param()` and `local()`, which initialize the `vspec` to be a function parameter or local variable, respectively.

Dynamic code composition is central to ‘C. References to objects of type `cspec` appearing in the body of a tick-expression are automatically converted to their corresponding evaluation types and incorporated into the code of the `cspec` in which they occur. For example, in the code below, compiling `c` has the same effect as compiling the expression `'(4+5)`:

```
/* Compose c1 and c2. Compiling c yields code '4+5' */
int cspec c1 = '4, cspec c2 = '5;
int cspec c = '(c1 + c2);
```

A more sophisticated example involves compositions of `void cspec`s:

```

/* This code does not do anything useful. */
void cspec c1 = '{int x = 1 + 1;}', cspec c2 = '{int y = 3 * 2;};
void cspec c = '{c1;c2;};

```

Support for arbitrary dynamic code composition will play an important role in the design of a debugging subsystem.

2.2.3 'C Implementation

The 'C compiler `tcc` works by compiling every `cspec` into a *code generating function* (CGF), that is invoked at run time to generate code for that `cspec` and all the `cspecs` nested within it.

Since 'C is statically typed, basic instruction selection can occur at compile time. Instructions are emitted in CGFs in the form of an abstract machine language called VCODE [6], a portable library of C functions for emitting binary machine instructions. Using an abstract machine layer allows `tcc` to support multiple architectures by ignoring machine-specific details of code generation; new platforms are supported (for the most part) by swapping in a new back-end for VCODE.

This is best explained with an example that compares normal C code compilation to 'C tick-expression compilation. Consider the following C code:

```
a = b + 2;
```

It might be compiled into assembly code that looks like this:

```
addii r1, r2, 2 /* add 2 to register 2, and place the result in register 1 */
```

Now consider the corresponding dynamic version using 'C:

```

/* Create the specification for the dynamic code. */
void cspec dyncode = '{a = b + 2;};
/* Compile the expression. */
func = compile (dyncode, void);
/* Execute the expression. */
(*func)();

```

The dynamic code specification represented by `dyncode` is statically compiled by the 'C compiler into something that looks like this:

```
_code_generating_function () {  
    v_addii (r1, r2, 2);  
}
```

`v_addii()` is a VCODE macro that, when executed at run time, emits the binary machine instruction to perform the addition. `compile()` then calls `_code_generating_function()` to produce the executable code that is later called by the `func` function pointer. The emitted code is placed on the heap using standard memory allocation techniques.

The symbols `r1` and `r2` point to handles returned from the VCODE register allocation function `v_getreg()`. The function `v_putreg()` is used at the end of the live range of the corresponding variable, allowing the register to be reused by `v_getreg()`. The VCODE run-time system keeps track of the number of available registers, and aborts with a run-time error if `v_getreg()` is called when no more are available. This presents a problem for the 'C compiler because it cannot determine at compile time how many registers will be needed.

This could be solved if the VCODE run-time system abstracted away the concept of register versus stack variables. However, in this case the VCODE client would lose explicit control of loads and stores. This may or may not be a good thing, depending on the application, but for the moment VCODE has been designed for maximum control over emitted code (while still maintaining a portable interface).

The 'C compiler solves this problem by implementing a wrapper around VCODE. This wrapper, called ICODE, does not emit code in place but instead builds up an efficient intermediate representation, allowing it to perform run-time optimization and register allocation before using VCODE for actual code emission. Thus, ICODE emits better code, but increases the cost of DCG.

2.3 Debugging

A debugger allows the user to control the run-time execution of a program and to examine and modify run-time data. Section 2.3.1 provides an overview of how debuggers work, Section 2.3.2 describes one debugging interface in particular, and Section 2.3.3 discusses the limitations of most current debugging interfaces.

2.3.1 Debugging Overview

Support for debugging comes from three distinct sources: the compiler, the operating system, and the actual debugger.

The debugger acts as the liaison between the program being debugged and the user. It keeps track of the status of the program, including all of the defined symbols and the current stack frame, and it provides an interface for the user to control the operation of the program.

Some debuggers do not require any further support because they are implemented in the same address space as part of the run-time system and not as a separate process. For example, the PCLU compiler [4], which compiles CLU [17] to C, implements debugging by inserting debugger callbacks into the emitted C code. These callbacks handle such things as symbol and function call registration. The debugger callback library is then linked in with the final executable. The usefulness of such a design as applied to DCG is considered in Chapter 3.

Support from the operating system comes in the form of primitives that are used to control the execution of a running process and to examine and possibly modify values in a process's address space. For example, the `ptrace()` system call found in many versions of Unix allows the caller to step through the instructions of another process, set breakpoints, and examine and modify the process' address space [22]. The advantage of `ptrace()` (or an equivalent) is that the process being debugged runs exactly as it would if it were not being debugged, except for the fact that it is periodically interrupted and resumed.

Support from the compiler comes in the form of special debugging annotations that are placed in the target code. Typical use of a debugger includes such things as stepping through lines of source code and examining data based on symbolic names. In order for the debugger to associate symbols with actual memory locations and lines of source code with machine instructions, the compiler must annotate the generated assembly language with additional semantic information. For example, if a user types "print foo" into a debugger, then in order for the debugger to print the value of foo, it needs to know the type of the variable foo, the amount of storage taken up by a variable of that type, and the location of that variable in memory.

2.3.2 Stabs Overview

One common format for these debugging annotations is the stabs format [24, 10]. Stabs stands for “Symbol TABLE entrieS”, referring to the fact that the annotations are placed in the symbol table of the executable image file.

There are three overall formats for stab assembler directives, differentiated by the first word of the stab. The name of the directive describes which combination of four possible data fields follows. It is either `.stabs` (string), `.stabn` (number), or `.stabd` (dot). The overall format of each class of stab is:

```
.stabs "STRING" ,TYPE,OTHER,DESC,VALUE
.stabn TYPE,OTHER,DESC,VALUE
.stabd TYPE,OTHER,DESC
```

For `.stabn` and `.stabd`, there is no `STRING`. For `.stabd`, the `VALUE` field is implicit and has the value of the current file location. The number in the `TYPE` field gives some basic information about which type of stab this is. Each valid type number defines a different stab type. The stab type defines the exact interpretation of, and possible values for, any remaining `STRING`, `DESC`, or `VALUE` fields present in the stab.

Here is an example of a stab assembly annotation:

```
LM6:
    .stabn N_SLINE,0,8,L1
L1:
    move  $sp,$fp
```

In this example, the stab entry describes the source code line number that corresponds to the next line of assembly code. The `TYPE`, `N_SLINE`, is a symbolic constant for the type of the stab that denotes the source line number. The `OTHER` entry is unused. The `DESC` entry, ‘8’, indicates that the eighth line of the current source file (as described in an earlier stab) generated the following lines of assembly. The `VALUE` entry indicates the address of the corresponding instruction. Figure 2-1 lists several commonly used stabs.

One major purpose of stabs is to describe the types of the variables, parameters, or function return values. This is accomplished using the string passed to the `.stabs` directive (often with the `N_LSYM` stab type). Information about the size, maximum value, and

Stab type	Description
N_SO	Source code filename
N_SLINE	Source code line number
N_FUN	Symbol corresponding to procedure name
N_RSYM	Symbol corresponding to variable in a register
N_LSYM	Symbol corresponding to variable in on the stack
N_LBRAC	Beginning of new scope (left bracket)
N_RBRAC	End of scope (right bracket)

Figure 2-1: Several common stab types

minimum value are conveyed in this string. For aggregate types such as structures, the name, type, and location of all sub-fields are also included in the type description string. Also, with stabs, there are no predefined types; every type used by a program must be explicitly described by the compiler. An example of such a string might be “unsigned short:t6=r1;0;65536;”, which defines the unsigned short type as allowing values from 0 to 65536. A complete description of stabs types specifiers can be found in [24, 10].

The assembler places all stabs into the symbol table of the output file (as well as placing all .stabs strings in the string table). The stabs are not actually loaded into memory when the file is executed because the operating system has no need for any symbol or debugging information. Instead, the debugger has to look at the executable file on disk in order to glean symbol and debugging information for an executable that is running in memory. The debugger parses the symbols and stabs from the executable file, and uses them internally to implement its various interface features.

2.3.3 Discussion

The fundamental problem with debugging is that it requires all of the useful information from source code files be encoded into a compact representation that binds high-level concepts (such as procedures, types, and compound data structures) to low-level concepts (such as machine registers and addresses). This is a thorny issue, since one purpose of high-level languages is to avoid low-level machine dependencies in the first place. Additionally, the encoding and decoding process is prone to loss of information, especially if the encoding technique used is poor, or just poorly documented. This debugging model is illustrated in Figure 2-2.

Some debuggers run as interpreters, allowing them to avoid this issue entirely. (This is,

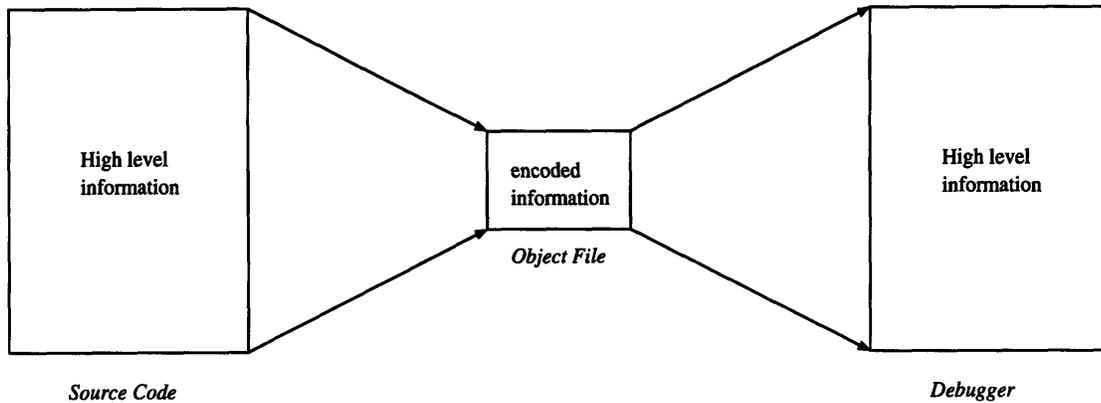


Figure 2-2: Flow of information in a non-interpreted debugging environment.

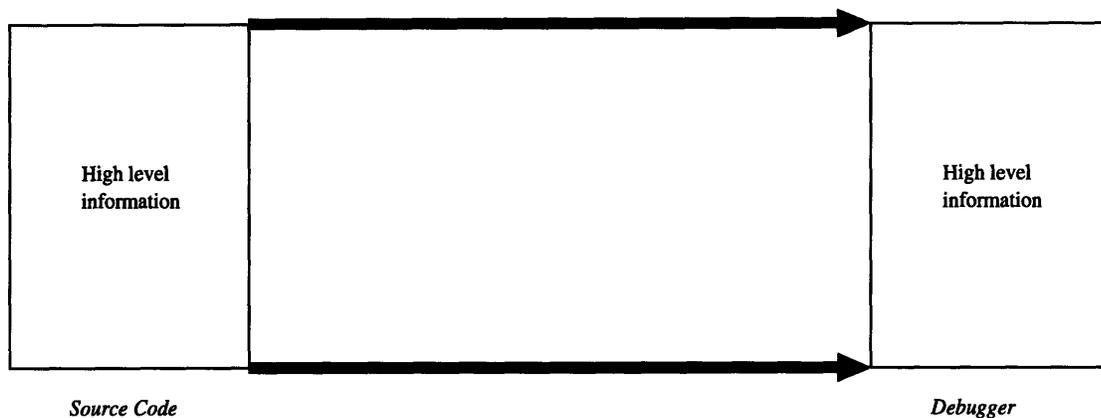


Figure 2-3: Flow of information in an interpreted debugging environment.

of course, also true for languages that are interpreted in the first place.) Since interpreters use the original source code as the representation for high-level information, they do not suffer from this deficiency, although they pay the cost in terms of both performance and complexity. Consider the fact that there are very few C interpreters on the market, and that they all have very expensive single user licenses. The interpretive debugging model is illustrated in Figure 2-3.

As a result, the best non-interpreted debugging environments tend to be the ones where the entire end-to-end system is developed by the same team or company. This includes the compiler, the debugger, and usually even the operating system, since debugging formats are often highly dependent on the operating system's binary object file format. For example, Sun offers a very nice C/C++ compiler and debugger package that runs on their Solaris operating system. Likewise, the freely available GNU compiler and GNU debugger work very well together on Solaris. Yet, the individual components (e.g., the Sun compiler and

the GNU debugger) do not work together, despite the existence of compiler-independent documented interfaces such as stabs, which both of the two development environments claim to use on Solaris. The problem is that these interfaces are underspecified, and thus are inconsistent and incompatible from implementation to implementation of both compilers and debuggers.

One reason for this complexity is the platform-dependence of binary object file formats. For instance, some stabs directives contain offsets to addresses in the text segment of the given object file. These offsets must be relocated at link time, meaning that the linker must have some inherent knowledge of debugger symbols. One attempt to set a standard for binary object files is the development of the platform-independent ELF (Executable and Linkable) format [23], used by all System V, Release 4 versions of Unix. Unfortunately, the extensibility of ELF makes the problem even worse; since ELF supports arbitrary sections (e.g., data, bss, etc...), some vendors have chosen to place debugging information in their own user-defined sections.

One of the most promising file formats in regards to debugging is the “.class” file format used by the Java Virtual Machine [16]. Since everything in Java is a class, all class members and methods are automatically stored using a well-defined naming system. Likewise, since the Java run-time system supports automatic stack trace dumps with corresponding line numbers (e.g., in case of an unhandled exception), line numbers are also part of the inherent file format. The only debugging-specific information that needs to be stored in a class file is information about local variables. This information can be optionally stored in the file if debugging is turned on in the compiler.

Unfortunately, in this project we do not have the resources to design a complete end-to-end system. Instead we must make do with the system components that already exist, adding and modifying only the components necessary to achieve our goal. This limitation affects various design decisions, as discussed in Chapter 3, as well as the implementation and results, as discussed in Chapters 4 and 5, respectively.

Chapter 3

Design

This chapter discusses the design of a DDG system for ‘C. Section 3.1 defines the criteria of the system we are interested in building. Section 3.2 discusses the high-level issues concerned with DDG support, and Section 3.3 presents a DDG design for ‘C.

3.1 Constraints and Criteria

Since code can be generated and combined at run time, debugging information must be generated and combined at run time as well. We define this process, which we call DDG, to be necessary only for “high-level DCG-based languages.” To further qualify this statement we define such languages to have two fundamental properties.

First, for the purposes of this project we are only interested in languages that support direct emission of binary machine code, excluding systems that use run-time compilation to generate data or data structures that can be passed to an evaluator or interpreter. Interpreted environments are not of interest, mostly because of the distinction made in Section 2.3.3. Unlike such environments, we do not have the luxury of being able to modify a virtual machine; we need to be able to examine the behavior of instructions running on a CPU.

For example, perl [25] supports single step debugging of code passed to the eval procedure. This is implemented by annotating the resulting opcodes with debugging information during the source-to-opcode compilation phase. This method is very similar in spirit to the method of DDG discussed in this paper, but it is not directly applicable since the back end for eval is the perl interpreter itself. Therefore, debugging support for eval is no dif-

ferent than debugging support for static perl code – the virtual machine (the perl opcode interpreter) can simply pause for debugging in between individual opcode evaluations. The opcodes have all the information necessary for debugging since they were created in an earlier pass from the source code itself. A similar argument could be made for other interpreted languages such as LISP.

Second, there must be support for the run-time composition of static source code text. In other words, there must be an inherent run-time correspondence between clusters of machine instructions and lines of source code.

Counter-examples of this are systems that support only run-time partial evaluation [13, 14]. The machine code corresponding to a partially evaluated function does not map to any actual source code (although the source code of the original function could always serve as a general purpose mapping). With pure partial evaluation, it is never the case that a bug would appear *only* in the partially evaluated version of a function (short of a bug in the partial evaluation system itself), so debugging the function by turning off partial evaluation would be sufficient.

Code composition, on the other hand, allows for the possibility of a direct, run-time-only correspondence between dynamic code and specific lines of source code text. ‘C is the only language currently known to support these properties.

3.2 DDG Design Issues

The dynamic generation of debugging information requires a high-level run-time representation of the properties of the original source code. For example, symbol names, type definitions, and source code file names and line numbers are among the most important items that must be passed along to the debugger. The three main problems associated with carrying along this information in the context of DDG are ordering, naming, and data storage.

The problem of *ordering* arises because source code order is unknown until run time. We therefore need to consider what it means to debug non-sequential source code, and what kind of support we can expect from the debugger for making the task more intuitive.

Naming is an issue because the run-time system adds new complexity to the meaning associated with symbols. In ‘C, this is especially tricky because the three special forms

compile(), param(), and local() all create variable references at run time. With compile(), for example, we would like to be able to assign a name to our newly created dynamic function so that we can set breakpoints in it and see it in a stack trace.

The *data storage* problem refers to the question of where and how the debugging meta-data is stored. As discussed in Section 2.3, this data is typically stored in non-essential sections of binary executable files. This means that programs have no knowledge or control over access to debugging information. However, in the case of DDG, debugging information must be “planted” into the object code at compile time in a format that can be understood and manipulated by the run-time system.

A good example of a system that already does something like this is PCLU [4]. Since the PCLU debugger runs as part of the process itself, it needs to be able to interpret and keep track of its own symbol names. Take the following CLU program:

```
start_up = proc()
  i:int := 3
  j:int := i + 4
end start_up
```

The corresponding C code generated by the compiler contains the following fragments (several comments have been added for readability):

```
struct {
  struct dbg_info *DBG.INFO; /* Storage for debugging structure, defined below. */
  int DBG.LINE;
  errcode err;
  errcode ecode2;
  CLUREF i; /* Storage for variable i */
  CLUREF j; /* Storage for variable j */
} locals;

/* Debugging information data structures. */
/* This first structure represents a name table. */
static Vlist4 locals_start_up = {0, 4, { {0, 2, "err", &int_ops},
                                         {0, 2, "ecode2", &int_ops},
                                         {0, 2, "i", &int_ops}, /* information for variable i */
                                         {0, 2, "j", &int_ops}}}; /* information for variable j */
/* This second structure contains auxiliary information, such as the current file and procedure names. */
dbg_info_start_up.actual dbg_info_start_up.data = {0, 12,
  "start_up", "test.clu", 0, 0, (int)start_up, 0,
  &vals_start_up, &sigs_start_up, &locals_start_up,
  &owns_start_up, &NO_VALS, &NO_VALS, &NO_PARMS, &NO_PARMS};
```

The data structure locals_start_up contains a run-time symbol table that is used by the “print” command during a debugger callback. Likewise, stack traces are made possible by

adding to the beginning and end of each function a callback which registers or de-registers the function name stored in the `dbg_info_start_up_data` data structure.

The fundamental difference between the PCLU example and what is necessary for a DDG system is that support for run-time manipulation of debugging information other than in a read-only context is not necessary in PCLU; the issues of ordering, naming, and data storage are all taken care of statically by the time the resulting C code is emitted. A DDG system, on the other hand, requires a more flexible run-time design to handle these issues.

3.3 DDG Design for ‘C

The requirements for the ‘C debugger are rather straightforward: we want to ensure that the programmer can pinpoint source code bugs using an intuitive and familiar interface. To achieve this goal we consider several approaches, the tradeoffs of which are discussed here.

3.3.1 Interface

The highest level design tradeoff is whether to support use of existing external debuggers or whether to build a PCLU-style run-time system debugger. The main benefit of using an external debugger is that we can reuse its base functionality, augmenting it with additional code (if any) to support dynamic loading of debugging symbols. This approach is backwards-compatible with current debuggers, meaning that the familiar interface of the debugger is preserved (and possibly augmented with some new commands).

If we choose to support external debuggers, then we also must consider whether we want to support arbitrary external debuggers or just a single external debugger in particular. This decision in turn depends on whether dynamic debugging requires modification of the debugger itself. In particular, the widely used GDB is the only C debugger that has freely available source code. In this regard it is the only possible candidate for usage if changes to the external debugger are necessary to support ‘C. The problem with using GDB, however, is that it is bloated. It has countless features and supports over 100 platforms, but consists of over 200,000 lines of complex code.

This situation is somewhat parallel to the development of the ‘C compiler itself, `tcc`, which was created by modifying a lightweight, well-documented ANSI C compiler called

lcc. The other options were to implement a ‘C (and thus C) compiler from scratch, or to modify the extremely complex code base of GCC. Unfortunately there does not exist an “lcc” equivalent of GDB – a lightweight, moderately portable, and freely available debugger that we could use for any ‘C modifications.

The PCLU-style approach requires that all functionality be built from scratch and incorporated into the run-time system. This design requires a large amount of overhead, but it need not worry about interfacing with outside systems. From an implementation point of view it requires more work, but it is easier to maintain since it does not require sorting through 200,000 lines of previously written code. The built-in debugger design is also a lot more likely to be portable, since it need only be consistent with itself and not with an external debugging format.

A more technical disadvantage of the built-in debugger is that it does not provide easy access to printing values of variables in registers. This is not a problem in a language like PCLU because every variable is a reference to a full-fledged object. However, support for the printing of register variables in ‘C with a built-in debugger would require either using only stack variables when in debug mode, or using a protocol for saving every register in a well-known location before entering the debugger callback function. Neither of these methods is very desirable, especially when compared to the external debugger’s method of access via `ptrace()` or equivalent, which supports examination (and modification) of arbitrary memory or processor state.

Lastly, the backwards-compatible approach of using an external debugger means that there is an integrated and familiar debugging environment for both static and dynamic code. Otherwise, the user would have to switch between the external debugger for normal C code and the internal debugger for dynamic code. This would mean, for example, that breakpoints in dynamic code could not be set from the external debugger.

Since we are most concerned with the debugging of C and ‘C in a seamless, integrated environment, we choose the external debugger design.

3.3.2 Data Storage

The next important high-level issue to address is how the debugging information should get from the ‘C run-time system into the external debugger. In particular, what technique should be used to transfer information to the debugger, and how should the transfer of

information actually take place? The different approaches offer different results in terms of performance, portability to other platforms, and portability to various debuggers.

One possible solution is to place the debugging information somewhere in memory. However, the drawback of writing the data to memory is that it can change the behavior of the program when debugging is turned on, since memory usage will differ from the non-debugging case. This can result in hard-to-find “heisenbugs” – bugs that appear or disappear when debugging is turned on [19]. Note that a PCLU-like system automatically suffers from the heisenbug problem, although it is somewhat mitigated in the case of PCLU by the fact that the language has a safer memory model (e.g., lack of pointers, garbage collection). It is important to avoid the heisenbug problem with languages such as C and ‘C since bugs in those languages are commonly related to memory allocation.

The alternative solution is to write the information to a file. With this method heisenbugs become less likely. This approach also eases development of the debugger since it is much easier to examine the contents of a file than the contents of a data structure in memory. Additionally, it is not clear that we care all that much about performance when debugging is turned on, although we do not want debugging performance to be so slow that it is burdensome.

There is also the question of what format the information should be stored in. (This issue is essentially independent of the storage location issue, especially with the advent of the `mmap()` system call which can make a file look like a block of memory.) One option is to write to the native binary file format for object files on the particular platform of the given ‘C compiler. This has the extremely valuable benefit of requiring few, if any modifications to a given debugger; the debugger can merely load the symbols from the dummy object file. Since most debuggers can do this on demand anyway, it would theoretically be possible to debug dynamic code with a third-party or vendor-provided debugger.

The drawback to writing in the native file format is that it is less portable since almost all machines use a different format for object files and debugging information. Even worse, as stated previously, these formats are extremely fickle and inconsistent from implementation to implementation.

The other option is to create our own format for debugging information, thus gaining portability. The downside is that the source code for any debugger must be modified specifically to deal with our format. In the case of GDB, this could be extremely burdensome.

For example, the section in GDB which parses stabs information consists of over 6,000 lines of code. Over 4,000 lines of that code are used to parse stabs type information. Most of this code would need to be duplicated for a specially designed ‘C format, meaning that the marginal cost of designing our own format would still be very large.

Another possibility would be for the ‘C run-time system to simply build up GDB data structures directly and then pass them along to GDB at the appropriate time. Unfortunately, GDB data structures are extremely complex and point to a large amount of nested GDB state. It would be preferable if GDB supported an interface for dynamically querying symbols. As it turns out, such an interface is currently under development as part of a project to add Java support to GCC and GDB [3]. Unfortunately, that project is still in the development phase.

In summary, the debugger can be either external or internal to the program itself. Likewise, debugging information can be transferred through a file or in memory using either a pre-existing or specially designed debugging format. We choose the approach of an external debugger using a pre-existing file format. We choose the stabs format as described in Section 2.3. Despite its inconsistencies and shortcomings, it is still the most intuitive and well-documented format. It is also the format most compatible with GDB.

3.3.3 Ordering and Naming

We need to consider any problems that might be posed by the support for arbitrary composition of ‘C code specifications. First and foremost, a dynamic function can be the result of cspecs that were defined in any part of any file in the entire program. One way to approach this problem is to handle it with a method similar to inline functions. We can simply allow the user to “hop” around the various cspecs that correspond to a given dynamic function. Due to the ease of design, this is the method that we choose.

However, it would be interesting to consider the possibility of synthesizing a C file for every dynamic function and using it as the primary source location for single stepping. Unfortunately, this would be rather difficult to implement, and quite possibly even less intuitive to use than the “hopping around” approach. The problem is that upon finding a bug, it might not be clear which is the originating cspec that needs modification.

As stated previously, the naming issue relates mostly to the ‘C special forms `compile()`, `param()`, and `local()`. Consider the following code:

```

void cspec c = '{}';
int vspec result;
for(i=0; i < n; i++) {
    int vspec a = param(i,int);
    c = '{c; result += a;}';
}

```

This code creates a specification for a function that sums its n arguments. At run time we would like to be able to step through this code and print each of the arguments by name. However, if we are restricted to printing the value of the run-time variable `a`, then we are restricted to printing one argument at a time, in order. Similarly, if we call `compile()` a repeated number of times, we would like to be able to refer to each resulting function with a different name.

There are two possible solutions to this problem. The first is to assign names via a hard-coded naming scheme, such as calling all function parameters “p1”, “p2”, and so forth. The other option is to extend the syntax of `compile()`, `param()`, and `local()` to accept an extra string argument that indicates what the name of the resulting variable should be. That way, a loop such as the one above is responsible for creating its own set of unique strings to be used as names. The approach used by our implementation is discussed in Chapter 4.

3.3.4 Summary

We choose the design of an external debugger that reads symbol information from dynamically generated stabs files. Use of stabs works extremely well with the paradigm of compiling backquote expressions into VCODE. Just as a normal compiler emits stabs directives into the emitted assembly code, we can create a VCODE stabs-like interface for use by the code generating functions. This interface can then translate run-time arguments into actual stabs file output for use by a debugger. This design could easily be extended to ICODE as well.

Chapter 4

Implementation

The DDG implementation for 'C requires three main changes to the current system:

1. Addition of stabs macros to VCODE.
2. Modification of the 'C compiler to emit stabs macros in the code generation functions.
3. Modification of GDB to support dynamic importation of symbol and debugging information.

These modifications are described in the following three sections, respectively.

4.1 VCODE-Stabs

In the same way that VCODE is an interface to an abstract machine, the VCODE-stabs library is an interface to an abstract debugger. The interface is based on the stabs interface itself, as defined in Section 2.3.2, but as with normal VCODE, any back-end could be substituted for actual emission of debugging information.

Currently, there is only one implemented back-end – one that outputs stabs assembly directives. This is somewhat of a departure from the design as stated in Chapter 3. The original intent was to output information directly to the native binary object file format. We choose instead to emit assembly code and then invoke the assembler on it. We lose even more performance with this implementation, but in return gain portability across any platform with an assembler that supports stabs directives (which is the case for all platforms currently supported by tcc).

The assembly back-end is also extremely useful for development purposes. When the emitted stabs do not result in the proper behavior from the debugger, we can easily examine the assembly file to look for possible sources of errors. The assembly file can even be hand-modified and reassembled, thus allowing us to test a possible fix without having to recompile VCODE.

Another departure from the intended design is that the current implementation only supports debugging of variables declared directly within cspecs, but not variables assigned by `local()` or `param()`. We hope to support this in future implementations.

The VCODE-stabs interface consists of six functions, derived from the three types of stabs directives. In the following sections, we explain each function and provide its prototype.

4.1.1 `v_stabs`

```
void v_stabs(char *string, int type, int desc, int value);
```

The `v_stabs` function is modeled directly after the `.stabs` directive. The only difference is that the `OTHER` field is not supported, since it is not currently used by any of the supported stabs types. An example use of this function is to describe a variable:

```
v_getreg(&i122,V_I,V_VAR); /* request a register from VCODE. */  
...  
v_stabs ("i:r1",N_RSYM,0x4,i122); /* describe the register variable "i" */
```

The type of this stab is `N_RSYM`, meaning that it describes a local register variable or function parameter. The first argument to the function is the string `"i:r1"`, which indicates that the variable name is `i`, and that it is located in an integer register. The third argument indicates that the size of the variable is four bytes. The final argument, `i122`, is an identifier that resulted from an earlier call to `v_getreg` (as described in Section 2.2.3). The value assigned to `i122` by `v_getreg` is simply the number of the allocated register, which is the same number that the `N_RSYM` stab expects for its fourth argument.

This example clearly shows the expressive power of run-time debugging directives. Since the debugging interface is procedural, and not static (as with assembler stabs), we can pass it arbitrary identifiers (such as `i122`) that have run-time dependent values. Yet, since `tcc`

knows the name and type of the variable at compile time, it can emit the static string “i:r1” as part of the `v_stabs()` function call. This design solves one aspect of the naming problem – the issue of how to bind symbolic names to run-time storage locations.

4.1.2 `v_stabd`

```
void v_stabd(int type, int desc);
```

Once again, this function is modeled directly after one of the assembler stabs directives, in this case the `.stabd` directive. It is important to note that in an assembly file, the `.stabd` directive is equivalent to the `.stabn` directive plus a label:

```
/* This directive: */  
    .stabd TYPE,OTHER,DESC  
/* Is the same as this directive: */  
LABEL: .stabn TYPE,OTHER,DESC,LABEL
```

In fact, some assemblers do not even support `.stabd` because it is simply a notational convenience. However, the `v_stabd()` function provides some unique functionality. Consider this C code, which uses a hypothetical `v_stabn()` function:

```
LABEL: v_stabn(N_SLINE, 10, LABEL); /* The current source code line number. */
```

This would simply not make any sense, even if it were legal C syntax (which it is not, since labels cannot be used as identifiers). We are not interested in the static location of the `v_stabn()` function call, we are interested in the run-time address of the next instruction to be emitted by VCODE. Procedural access to the next VCODE address is exactly what the `v_stabd()` function provides. Therefore, an example usage would be:

```
v_stabd(N_SLINE, 10); /* The current source code line number. */
```

In this case, `tcc` knows statically that the corresponding source code line number is 10, but it leaves it up to the run-time system (as implemented underneath `v_stabd()`) to emit the correct address in memory.

4.1.3 `v_stabs_setfile`

```
void v_stabs_setfile (char *filename);
```

The purpose of `v_stabs_setfile()` is to address the ordering issue with regard to the current source code filename. The key issue is that code specifications can be arbitrarily composed at run time, and thus can originate from any file. Therefore, stepping through lines of a dynamically generated function might result in hopping around across discrete source code files. (This is similar to inline functions or `#include`'ed source code.)

We choose a protocol whereby each CGF must set its own originating filename. `v_stabs_setfile()` is called once at the beginning of each CGF as well as immediately after any calls to nested CGFs, thus “restoring” the state of the current file, which may have been over-written by the nested CGF. `v_stabs_setfile()` caches the name of the most recent filename, only emitting an actual stabs directive (`N_SO`) if the filename has changed.

4.1.4 `v_stabs_global_types_{begin,end}`

```
void v_stabs_global_types_begin (void);  
void v_stabs_global_types_end (void);
```

The stabs format requires that all global types be defined at the beginning of each object file. These stabs inform the debugger about the nature of all globally defined types, including built-in types and globally defined enumeration types and structures. In an assembly file, these descriptions are contained in `.stabs` directives; in `VCODE`, they are contained in `v_stabs()` invocations.

Our design uses one object file per dynamic function, so therefore we need to emit global type information once at the beginning of each emitted object (assembly) file. However, there is no way for `tcc` to determine at compile time the top-level CGF for any given dynamic function. Thus, every CGF needs to be prepared for top-level invocation by including information for all globally defined types.

We achieve this by surrounding the code to emit global types with calls to `v_stabs_global_types_begin()` and `v_stabs_global_types_end()`. These functions set a static flag within the `VCODE` library, a flag that is reset at the beginning of every invocation to `compile()`. This means that all CGFs can contain the necessary code to emit global type

information, but the information will only be emitted at run time by the top-level CGF.

In summary, these functions solve the ordering problem as it pertains to global type information.

4.1.5 `v_stabs_func`

```
void v_stabs_func (char *funcname);
```

This VCODE-stab is used to assign a name to the current dynamic function. `tcc` emits calls to `v_stabs_func` with the current filename (without the `.c` suffix) as the `funcname` argument. Thus, dynamic functions are named after the file in which they are compiled. `v_stabs_func()` also appends a monotonically increasing integer to the name, guaranteeing uniqueness.

Once again, we never know which CGF will be at the top level, so every CGF must contain a call to `v_stabs_func`. To ensure that it is only emitted by the top-level CGF, this stab is inserted (along with the global type information) in between calls to `v_stabs_global_types_begin()` and `v_stabs_global_types_end()`.

4.2 ‘C Modifications

As stated previously, `tcc` is derived from the `lcc` [8] ANSI C compiler. `tcc` uses two back ends to generate code. One is the original `lcc` back end, which emits assembly corresponding to static code. The other is referred to as the “dynamic” back end, used to emit code generation functions that contain VCODE. The dynamic back end has a similar structure to the static back end – it implements a set of callbacks used by the `lcc` front end to translate `lcc` data structures into a target language.

`lcc` supports debugging by allowing back ends to implement a set of debugging callbacks. We use this infrastructure to emit VCODE-stabs in the dynamic back end. The five callbacks are defined as follows:

- **`stabinit()`**: Called by the `lcc` front end to emit any directives that must appear at the beginning of an object file (or CGF in this case). For VCODE-stabs we use it to emit the original file name, global types, and function name.

- **stabline():** Called by the front end to emit line number stabs. For VCODE-stabs, we emit line numbers that correspond to the (possibly multi-lined) specification of a cspec.
- **stabblock():** Called whenever a block begins or ends. Emits an N_LBRAC or N_RBRAC (left bracket or right bracket) stab, respectively.
- **stabsym():** Called whenever a new variable is declared.
- **stabtype():** Called whenever a new type is defined.

4.3 Debugger Modifications

As stated in Chapter 3, one goal of this design is to make it portable to all external debuggers that support loading of specified symbol files. In principle, this design goal has been met, although no debuggers have been tested apart from GDB. At the very least, the VCODE-stabs interface should be a sufficient front end for generating the necessary debugging data, regardless of the data format generated by the VCODE-stabs back-end.

GDB supports loading of symbol files with the `add-symbol-file` command. The syntax of this command is as follows [9]:

`'add-symbol-file FILENAME ADDRESS'`

The `'add-symbol-file'` command reads additional symbol table information from the file `FILENAME`. You would use this command when `FILENAME` has been dynamically loaded (by some other means) into the program that is running. `ADDRESS` should be the memory address at which the file has been loaded; GDB cannot figure this out for itself. You can specify `ADDRESS` as an expression.

The symbol table of the file `FILENAME` is added to the symbol table originally read with the `'symbol-file'` command. You can use the `'add-symbol-file'` command any number of times; the new symbol data thus read keeps adding to the old. To discard all old symbol data instead, use the `'symbol-file'` command.

We use this command under slightly different circumstances than those intended. Nonetheless, a dynamically generated function looks no different to the debugger than a statically compiled function. The trick is to create the “fake” object file so that it appears to be the source of the function.

Here is how the command is used in the context of ‘C’:

```

(gdb) next
26   func = compile (c, void);
(gdb) next
28   func ();
(gdb) print func
func = (void (*)()) 0xac620 <end+204848>
(gdb) add-symbol-file file.o 0xac620
(gdb) print func
func = (void (*)()) 0xac620 <tccfunc>

```

The fact that this works in GDB without any debugger modification is nice because it means that if we want to freely distribute the ‘C compiler then people do not also have to worry about installing a customized version of GDB.

However, we still support an optional modification to automate the process of loading the dynamic symbols. If configured to support this modification, the VCODE-stabs run-time system writes the symbol file name and function address to a temporary file with a known name. Then it uses the kill() system call to send itself the user defined signal SIGUSR1. GDB intercepts this signal (since it intercepts all signals raised by the attached process), and understands that it should update its symbol table using the information in the temporary file.

4.4 Integration

Figure 4-1 illustrates how VCODE-stabs are used in combination with the tcc run-time system. The run-time function compile() initializes the VCODE-stabs subsystem, starts a chain of calls to code generating functions, and then closes the VCODE-stabs subsystem. The initialization code is responsible for opening the assembly file for that dynamic function and resetting the global types flag. The shut-down code closes the assembly file, invokes the assembler on it, and optionally informs GDB of the new symbols.

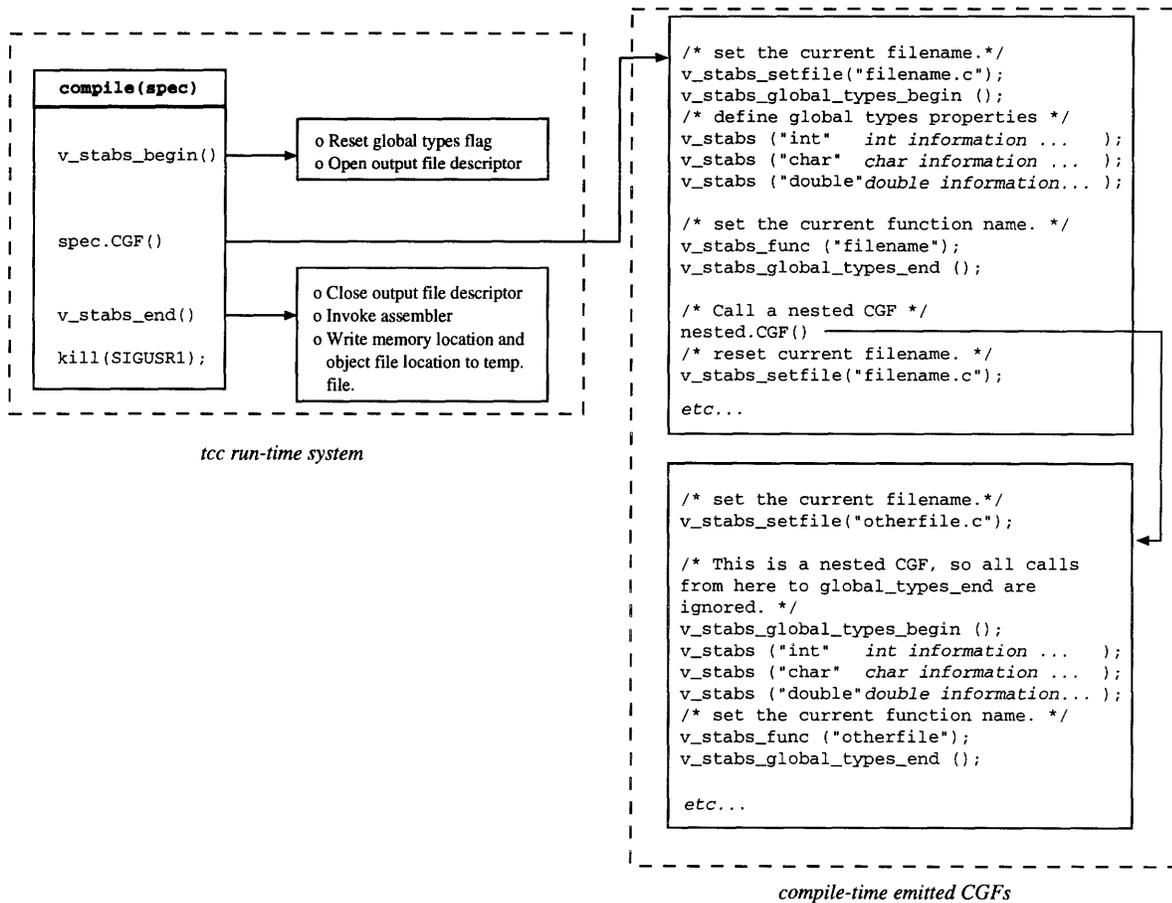


Figure 4-1: VCODE-stabs run-time invocations.

Chapter 5

Results

Previous chapters outlined the design and implementation of a debugging framework for dynamically generated code. This chapter discusses the functionality of the resulting implementation. Section 5.1 presents examples of the debugger at work, and Section 5.2 evaluates its performance.

5.1 Debugger Examples

We start off with an example that shows how the debugger supports single stepping of dynamic code and printing of dynamic variables. Consider the following C function:

```
main ()
{
  int i;
  typedef int (*printfunc) (void);
  printfunc func;
  void cspec a = '{
    int i = -1, c = -4;

    i = c + 4;
    c = i + 5;
    return c;
  };
  func = compile (a, int);
  i = func ();
  printf("i: %d\n", i);
}
```

The following debugging session shows what happens when we step through this function:

```

% gdb test
(gdb) break main
Breakpoint 1 at 0x25d8: file test.c, line 11.
(gdb) run
Starting program: test
Breakpoint 1, main () at test.c:11
11    void cspec a = '{
(gdb) next
19    func = compile (a, int);
(gdb) next
Caught SIGUSR1 ... loading dynamic symbols to address 0xac614
20    i = func ();
(gdb) print func
$1 = (int (*)( )) 0xac610 <test1>

```

We confirm that the debugger has loaded the dynamically generated symbols by printing the value of the function pointer `func`. It resolves to the actual function `test1`, which is correct because `test.c` is the name of the file containing the code specification for the function. (Remember that we derive dynamically generated function names from the name of the originating file.)

At this point the dynamically generated function is about to be invoked. We set a breakpoint there and continue with execution:

```

(gdb) break test1
Breakpoint 2 at 0xac614: file test.c, line 12.
(gdb) cont
Continuing.
Breakpoint 2, test1 () at test.c:12
12    int i = -1, c = -4;
(gdb) next
14    i = c + 4;
(gdb) print i
$2 = -1
(gdb) print c
$3 = -4
(gdb) next
15    c = i + 5;
(gdb) print i
$4 = 0
(gdb) next
16    return c;
(gdb) print c
$5 = 5
(gdb) next
main () at test.c:21
21    printf("i: %d\n", i);
(gdb) print i
$6 = 5
(gdb) cont

```

```
Continuing.  
i: 5  
Program exited with code 01.
```

By stepping through the code, we confirm that the debugger is properly loading the emitted filename and line numbers. The print statements confirm that the run-time specified symbols `i` and `j` are also being properly loaded.

Note that when we set the breakpoint at `test1`, the debugger responds by saying, “Breakpoint 2 at 0xac614: file test.c, line 12.” Unfortunately, it would not be possible to set the same breakpoint by saying “`break test.c:12`”, as can be done normally. The problem is that line 12 of `test.c` now corresponds to two executable portions of the program – the `main()` function and the dynamically generated function whose code is contained in the `cspec`. A quick fix for this problem is to copy the `test.c` file to another file such as `test2.c`, and then modify the emitted stabs to point to this file for the dynamic function. This is somewhat related to the discussion in Section 3.3.3 about synthesizing C files for dynamic functions.

The next example demonstrates the use of code composition. Consider the following C function, which computes $base^{exp}$ by repeated multiplication and squaring [7]:

```
double pow(double base, int exp) {  
    int t, bit = 1, square = base;  
    double result = 1.0;  
    while (bit <= exp) {  
        if (bit & exp)  
            result *= square;  
        bit = bit << 1;  
        square *= square;  
    }  
    return result;  
}
```

Given a fixed exponent, we can improve on this by using ‘C to unroll the main loop and perform only the necessary multiplications:

```
typedef double (*dptr)();  
dptr mkpow(int exp)  
{  
    /* 1 argument: the base */  
    double vspec base = param(0, double);  
    /* 1 local: the running product */  
    double vspec result = local(register double);  
    /* Initialize the running product */  
    void cspec squares = (1&exp) ? '{ result=base; } : '{ result=1.; };  
    int bit = 2;  
    /* Multiply some more, if necessary */
```

```

while (bit ≤ exp) {
    squares = '{ squares; base *= base; };
    if (bit & exp)
        squares = '{ squares; result *= base; };
    bit = bit << 1;
}
/* Compile a function which returns the result */
return compile('{ squares; return result; }, double);
}

```

We now use the debugger to examine the run-time structure of the function returned by `mkpow()`:

```

% gdb pow
(gdb) break main
Breakpoint 1 at 0x25d8: file pow.c, line 38.
(gdb) run
Starting program: pow
Breakpoint 1, main (argc=1, argv=0xefff8dc) at pow.c:38
38      f = mkpow(7);
(gdb) next
Caught SIGUSR1 ... loading dynamic symbols to address 0xb0628
39      res = (*f)((double)2.);

```

At this point `mkpow()` has generated a dynamic function to compute a floating point value to the power seven. GDB has also automatically loaded up the dynamically generated symbols for this function. We can now set a breakpoint at this function, which has the name `pow1()` since it is the first function generated from the file `pow.c`.

```

(gdb) print f
$1 = (double (*)()) 0xb2798 <pow1>
(gdb) break pow1
Breakpoint 2 at 0xb279c: file pow.c, line 15.
(gdb) cont
Continuing.
Breakpoint 2, tccfunc () at pow2.c:15
15      void cspec squares = '{ @result = 1.; };
(gdb) next
19      squares = '{ @squares; @result *= @base; };
(gdb) next
21      squares = '{ @squares; @base *= @base; };
(gdb) next
23      squares = '{ @squares; @result *= @base; };
(gdb) next
21      squares = '{ @squares; @base *= @base; };
(gdb) next
23      squares = '{ @squares; @result *= @base; };
(gdb) next
26      return compile('{ @squares; return @result; },double);

```

These statements confirm that `mkpow()` is working as planned, since *result* (initially set to 1.0) is first multiplied by *base*, then by *base*², then by *base*⁴. This results in *base*⁷ as the final return value, which is what we expect.

Support for printing of `local()` and `param()` values would make this example even more powerful. One would be able to examine the progression of values by typing “print base” or “print result” at any of the above lines. As stated earlier, we expect to support this feature in a future version of VCODE.

Finally, one of the most helpful features of the C debugger is that it can be used to find the source code line responsible for a fatal error such as a segmentation violation. This works equally well when debugging dynamically generated functions. Consider the following code:

```
main ()
{
  int i;
  typedef void (*printfunc) (void);
  printfunc func;

  void cspec a = {
    int *i = 0;
    int a;

    a = *i;
    a += 1;
  };

  func = compile (a, void);
  func ();
}
```

Finding the source of the segmentation violation is the same as with normal C code:

```
% gdb testseg
(gdb) break main
Breakpoint 1 at 0x2294: file testseg.c, line 11.
(gdb) run
Starting program: testseg
Breakpoint 1, main () at testseg.c:11
11     void cspec a = '{
(gdb) next
19     func = compile (a, void);
(gdb) next
Caught SIGUSR1 ... loading dynamic symbols to address 0xaa610
20     func ();
(gdb) cont
Continuing.
```

	1 cspec	10 cspecs	100 cspecs
Debugging off	1.9	6.5	56.4
Debugging on	2.2	8.9	80.5

Figure 5-1: Compile-time performance (seconds)

	1 cspec	10 cspecs	100 cspecs
Debugging off	0.02	0.02	0.23
Debugging on	0.13	1.33	13.63

Figure 5-2: Instantiation-time performance (seconds)

```

Program received signal SIGSEGV, Segmentation fault.
testseg1 () at testseg.c:15
15     a = *i;
(gdb) list
10
11     void cspec a = '{
12         int *i = 0;
13         int a;
14
15         a = *i;
16         a += 1;
17     };
18
19     func = compile (a, void);
(gdb) print i
$6 = (int *) 0x0

```

This functionality could also be extended to support examination of core dump files. All that would be needed is an option to `compile()`, indicating that dynamically generated symbol files should be created with unique names and left in a well-defined location. Upon starting `gdb`, the user would have to manually load all necessary symbol files using the `add-symbol-file` command. Dynamically generated functions could then be examined just like any other function.

5.2 Performance

Debugging support affects performance at both compile time and instantiation time. At compile time we emit a greater number of `VCODE` functions (in the form of `VCODE-stabs`) that must be compiled by the dynamic back end, and at instantiation time we pay the price of calling those extra functions. With our design in particular, the bulk of the performance

	5 line function	10 line function	20 line function
Debugging off	0.26	0.33	0.47
Debugging on	0.26	0.33	0.47

Figure 5-3: Run-time performance (seconds) for dynamic functions of various sizes, each executed one million times.

hit comes from invoking an assembler every time we call `compile()`.

We measure performance by timing the compilation phase and the instantiation phase with debugging turned both on and off. Figures 5-1 and 5-2 list the results of these measurements for various numbers of cspecs. In each figure we measure in units of 5-line code specifications, each of which contains two local variables.

Clearly there is a performance hit from enabling debugging. For instance, instantiation time in the 100-cspec case increases by a factor of about 60. However, these numbers only represent various stages of compilation. The important thing to note about our design is that by using an external debugger, we do not lose any run-time execution performance since the dynamically generated code runs exactly as it would without the debugger. This is shown in Figure 5-3. A PCLU-like debugger, on the other hand, can lose up to an order of magnitude of performance with debugging turned on because the debugger callbacks must check for breakpoints at regular intervals.

Chapter 6

Conclusions and Future Work

This thesis has presented a framework for the debugging of dynamically generated code. In brief, we have devised a technique, called Dynamic Debugging-data Generation, that allows us to generate symbolic debugging information on the fly. We have implemented this technique for ‘C, a language that supports high-level access to dynamic code generation. By generating debugging data in the format used for C debuggers, we allow the programmer to debug ‘C programs in the familiar environment of the C debugger.

Our implementation is a proof-of-concept of the overall DDG design. Unfortunately, the implementation is somewhat constrained by the underspecified interfaces of current debugging systems. Thus, an important direction for future work is the development of an integrated DDG interface. As DCG usage becomes more and more prevalent in software systems, there will be an increased need for a well-defined interface for dynamically generated debugging information. This thesis can be viewed as a starting point for such work.

Bibliography

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. Technical report, Computer Science Division, University of California, Berkeley, 1992.
- [3] Per Bothner. A gcc-based java implementation. Technical report, Cygnus Solutions, 1997. <http://www.cygnus.com/~bothner/>.
- [4] Dorothy Curtis. Portable clu internals guide. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1993.
- [5] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. ‘C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*, January 1996.
- [6] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the 23rd Annual ACM Conference on Programming Language Design and Implementation*, May 1996.
- [7] Dawson R. Engler and Massimiliano Poletto. A ‘C tutorial. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology, April 1996.
- [8] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. The Benjamin/Cummings Publishing Company, 1995.
- [9] Free Software Foundation. *The GNU Debugger*, 1995.
- [10] Free Software Foundation. *Stabs Interface Manual*, September 1995.

- [11] Brian Guenter, Todd B. Knoblock, and Erik Ruf. Specializing shaders. In *Proceedings of SIGGRAPH*, 1995.
- [12] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report TR 91-11-04, Department of Computer Science and Engineering, University of Washington, November 1991.
- [13] David Keppel, Susan J. Eggers, and Robert R. Henry. Evaluating runtime-compiled value-specific optimizations. Technical Report TR 93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.
- [14] Mark Leone and Peter Lee. A declarative approach to run-time code generation. Technical report, School of Computer Science, Carnegie Mellon University, 1995.
- [15] Henry Lieberman. The debugging scandal and what to do about it. *ACM Communications*, 40(4):27, April 1997.
- [16] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.
- [17] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. McGraw Hill, 1986.
- [18] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Graduate School of Arts and Sciences, Columbia University, 1992.
- [19] V. Paxson. A survey of support for implementing debuggers. Technical report, U.C. Berkeley, 1990.
- [20] M. Poletto. bs: A breitenberg vehicle simulator, May 1997. Final project for MIT course 6.836.
- [21] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the 24th Annual ACM Conference on Programming Language Design and Implementation*, June 1997.
- [22] Sun Microsystems. *ptrace (2)*, *SunOS 4.1 Manual Page*, January 1990.

- [23] Sun Microsystems. *elf (3e), SunOS 5.4 Manual Page*, November 1993.
- [24] Sun Microsystems. *Stabs Interface Manual, Version 3.1*, September 1995.
- [25] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, 1996.