

Debugging Optimised Code Using Function Interpretation

Kim Elms

{elms@fit.qut.edu.au}

Programming Languages and Systems Group
Department of Computer Science
Queensland University of Technology
GPO Box 2324, Brisbane, Queensland 4001 AUSTRALIA

Abstract

Previously the debugging of optimised code has not been possible without recompiling the executable code and preventing the use of code optimisation techniques. Although current research efforts offer partial solutions to a small set of optimisation techniques, no unified approach has been developed to overcome the barriers imposed by a large range of sophisticated optimisation techniques.

The approach taken in the building of the prototype described throughout this paper combines program simulation and interpretation techniques with run-time executable modification techniques to provide an integrated environment for function-level interpretation. This has been achieved without the modification of an existing compiler and also without the enhancement of the compiler-debugger interface (CDI), thereby allowing direct application of the debugger in current operational environments.

This paper describes the construction of `gpdb`, a debugger for the Gardens Point compiler environment. This debugger has proved to be fundamental for the production of an interactive development environment which allows an optimised program to be run, corrected, modified, and even further developed without the need for the recompilation of the executable program, or the resetting of the debugging environment.

1. Introduction

The problems associated with the debugging of programs that have been compiled with the assistance of optimisation techniques are severe. Although there is literature available which presents techniques for the minimisation of such difficulties [Copp93, Henn82] they are limited in scope, and not readily available. This paper introduces `gpdb` (Gardens Point Debugger), a comprehensive debugging platform which allows an optimised program to be run, corrected,

modified, and even further developed without the need for the recompilation of the executable program, or resetting of the debugging environment¹.

Consider the following situation. During a debugging session, it is discovered that a particular function is returning an unexpected value. It is logically suspected that the algorithm of the function may be incorrect for the particular set of data values that have been supplied to that call. Given some understanding as to how the algorithm is supposed to work, a plausible way to test the hypothesis that the algorithm is erroneous is to examine and verify the internal computations of the function. This would commonly be done by “stepping through” the statements of the function and verifying the consistency of the actual state of the execution and the conceptual model of how the algorithm should work. In the case where the code of the function has been optimised, such verification would typically be impossible. Why is this so?

When a compiler produces code to implement a given function, the only constraint for correctness is that the object code produces the same input/output behaviour as that specified by the algorithm in the source code. For example, it is legitimate for a compiler to:

- re-order the operations of the code at both the instruction and the statement levels;
- eliminate redundant computations;
- place different variables in the same storage location; and/or
- remove completely some computations and insert other computations which would produce the same final values.

Thus, the inherent problem of observing the state of optimised code is that:

¹ As this environment replaces functions, heavy inter-procedural optimisations may hinder procedural interpretation efforts and require associated functions to also be interpreted.

... program points in the source code may not have any corresponding point in the execution of the optimised object code.

Logically then –

... optimised code should not be modified except by the substitution of whole procedures.

From a **control** point of view, the query, “*Which line am I at now?*” would be responded to with an open reply such as “*You are not anywhere*” or, alternatively, “*You are in many places.*”

From a **data** point of view, when modern optimisation techniques, such as graph colouring and code motion, have been applied together, the question, “*What is the value of local variable x?*” may also meet with an open response such as, “*x currently has no value*” or, alternatively, “*x has many values.*”

Thus, in this case, verifying the internal workings of the suspect function will require the recompilation of the source code of at least that function, with optimisations turned off, and with full symbolic debugging support turned on. In complex cases, it might also be helpful to edit the source code of the function before recompiling, so that appropriate executable assertions can be applied that will check that the internal states of the computation correspond to the expected behaviour.

In order to address this problem, language tool implementers and researchers have, until now, been considering the following options:

1. Gain more information regarding the optimisation techniques and the resulting code modifications from the compiler (or optimiser) and save them for future analysis; [BHS92, Zell83, Zura91]
2. Provide a purely interpretation system (generally operating directly upon the source code) which simulates the running of the executable code, within a controlled environment; or [CmKe93, RaHa92]
3. Reanalyse or decompile [CiGo93] the executable program file (generally using either data-flow or control-flow analysis techniques) and generate the required mappings to be able to represent the program's source code. [HCU92, Wism94]

The approach outlined in this paper differs from these techniques as it provides, upon demand, intermediate language interpretation of functions contained within the program's own executable

environment. This provides a flexible environment for the ad hoc replacement of program functions for purposes such as bug fixing, enhancement, introspection [Sosi92, Sosi95], or profiling.

Gpdb's optimised code debugging environment was developed for languages such as C, Pascal and Modula-2 which provide a purely executable version of a user's source code without internal language mechanisms such as garbage collection, dynamic linking, or kernel function interpretation (e.g., Self, Lisp and Oberon). Although gpdb could be applied to such environments its primary purpose is to provide such a dynamic environment to languages which deliver autonomous program execution. Therefore gpdb's environment has been developed with due regard for:

- supporting all current debugging facilities (e.g., stepping, break-pointing and variable watching);
- no additional files being 'bundled' with an executable (neither within the symbolic executable file format nor as an accompanying external file);
- no 'specially modified' versions of a compiler and/or optimiser being required; and
- a high level of language & machine independence.

This research has also resulted in the development of a mechanism which allows internal examination and assertion insertion to be carried out without any recompilation. This level of functionality is currently not available in any other debug platform.

The following sections will outline the requirements for an optimal debugging platform to enable non-intrusive debugging of optimised code. The concepts effecting the overall design and implementation of the gpdb environment will be discussed and then used to illustrate how gpdb can be applied to advanced debugging concepts and the development of future work.

2. Local-Agent Technology

To be able to implement such a tool as gpdb, traditional parent-child process control technology (see Figure 1) has been used in a similar way to other debuggers, e.g., gdb, sdb, dbx. This allows the debugger interface within the parent process to occupy a separate area of process memory to that of the user's executable program (contained within the child process) and merely communicate via either a `ptrace()` system call or remotely using

a BSD socket connection. This design has therefore minimised the possibility of side-effect memory modifications, while more easily allowing `gpdb` to be utilised in a distributed environment in future.

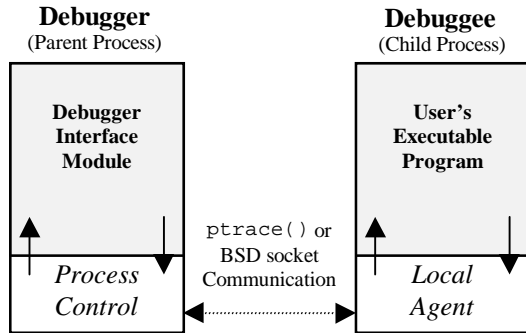


Figure 1: Local Agent Architecture [Pax90]

More importantly, this model uses local-agent process communication architecture in a similar manner to `gdb` (remote) [Pizz93] and `DynaScope` [Sosi92, Sosi95]. This architecture provides a simple and self-contained module that can be incorporated within the user's executable program for the purpose of providing internal information on program execution and variables, as well as interacting with the executable program to modify its execution path. Traditionally, this has been achieved through either:

- static linking the local agent into the user's target program at compile-time; or
- dynamic loading and linking the local agent with the user's target program at runtime when the target program is loaded under parent process control (e.g., through a debugger or a profiler); or
- dynamic loading and linking through a small code routine which has been statically linked into the program. This routine does not disrupt the execution of the user's executable until a signal is received from the debugger (or profiler) process which then instructs it to load the local agent and establish communication between the target executable and the debugger [Sosi95].

(The latter has been invaluable for ad hoc bugs.)

Much of the traditional parent-child interaction overhead has been minimised in `gpdb` by passing many of the debugging tasks to the local agent in the child process. This allows the involvement of the parent to be more limited to the maintenance of the user's debug interface which controls the sending of high-level user instructions to the child process, such as placing and removing of break-

points, loading and replacing functions, and setting variable display points. Thus, all interactions with the 'live' target program are then conducted by the local agent, such as, obtaining current variable values and interaction between the executable image and the interpreted module. This architecture has allowed a far more 'intelligent' and extensible control mechanism, and thereby reduces the amount of kernel traffic produced by the `ptrace()` call.

3. The Gardens Point Environment

The Gardens Point environment (see Figure 2) is a compiler architecture that was designed and developed at the Queensland University of Technology's Programming Languages and Systems research concentration area. It enables the independent maintenance and development of compiler 'front-end's (language parsers and tree builders for languages such as C, Modula-2, Oberon, and Sather) with the 'back-end' instruction selector which produces optimised executable code for a range of machine architectures (e.g., DEC MIPS R3000, DEC Alpha, Sun SPARC, Intel DOS, Intel OS/2, and Linux).

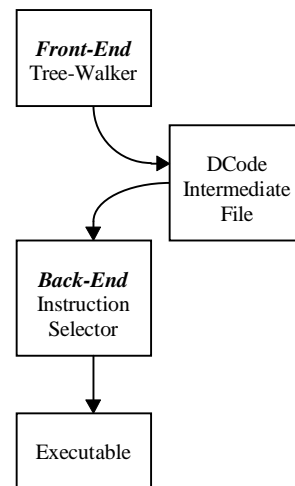


Figure 2: Gardens Point Environment Overview

Although this architecture provides an easier development platform with the separation of the 'front-end' from the 'back-end', it also hinders the collection of information for debugging. For example, the 'front-end' is privileged to information such as the structure and name of types of variables, while the 'back-end' focuses upon whether a local variable is in memory or in a register.

This level of process separation has been aided with the development of an intermediate language called DCode [Goug95] (an example is presented in Figure 3) which is written (in ASCII text format) at the completion of the tree-walker, and then reread by the instruction selector. DCode utilises an abstract stack machine similar to that of P-Code [Wirt71] and U-Code [PeSi79] that enables it to be used for any procedural language. However, DCode offers the advantage of a simplified memory access model that provides faster data access than is able to be achieved through P-Code and U-Code, thus providing a more natural map to RISC architectures.

```

; D-Code output produced by gpm from "hello.mod"
.TITLE Hello
.FILE "hello.mod"
.EXPORT _StartHello
.IMPORT _InOut_WriteLn
.IMPORT _ProgArgs_ArgNumber
.IMPORT _InOut_WriteCard
.IMPORT _InOut_WriteString
.CONST
_mNam: .ASCIIZ "Hello"
.CONST ; [Hello, world.hello needs no args.]
__cDat: .BYTE 048H,065H,06CH,06CH,06FH,02CH,020H
        .BYTE 077H,06FH,072H,06CH,064H,000H,068H
        .BYTE 065H,06CH,06CH,06FH,020H,06EH,065H
        .BYTE 065H,064H,073H,020H,06EH,06FH,020H
        .BYTE 061H,072H,067H,073H,000H
.PROC _StartHello(.NOCHECK,.SIZE=0,.NODISPLAY)
.ENTRY
.ENTRY
lineNum 6
    pshLit 12
    mkPar 4,4
    pshAdr __cDat
    mkPar 4,0
    call _InOut_WriteString,2
    cutPars 8
    call _InOut_WriteLn,0
lineNum 7
    .TRAP __gp_assTrp,labell,__cDat+13,0
    call _ProgArgs_ArgNumber,0
    pshRetW
    pshLit 1
    releQ
    brFalse labell
    exit
    endF
.ENDP

```

Figure 3: DCode produced from Modula-2 "Hello, world"

```

MODULE Hello;
FROM ProgArgs IMPORT Assert, ArgNumber;
FROM InOut IMPORT WriteString;
FROM InOut IMPORT WriteCard, WriteLn;
BEGIN
    WriteString ("Hello, world"); WriteLn;
    Assert (ArgNumber()=1, "Needs no args");
END Hello.

```

Figure 4: Modula-2 Version of "Hello world"

Figure 4 shows a Modula-2 language version of a simple "Hello world" program, while Figure 3 shows its representation in DCode. Although DCode is produced in ASCII format by a language's 'front-end' tree-walker it was necessary to transform it into a more compact binary format to reduce the tasks of parsing and

command validation in the DCode interpreter which is contained within the local agent (this also provided substantial increases in the speed of interpretation).

4. **gpdb Architectural Overview**

Although *gpdb* uses local agent technology to implement the internal modification and control mechanisms, a totally modular approach [Pizz93, Rams93] has been used in its development. This approach allows the easier modification and extension of each component independently, as can be seen in Figure 5.

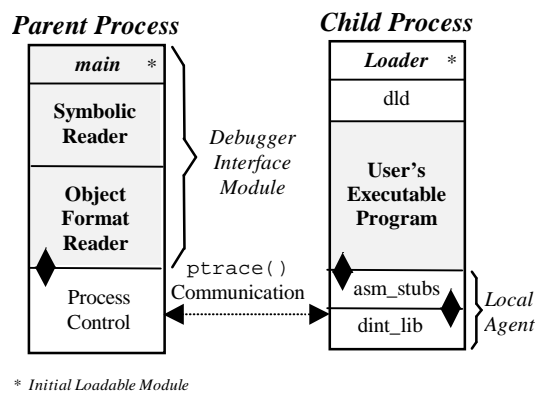


Figure 5: *gpdb* Process Components

As local agent technology has influenced the overall design of *gpdb* much of the low-level program interaction routines have been placed into the child process with the user's executable program. This has allowed the child process to consist of the following components:

- *loader* – This small loading routine is used initially when the executable program is loaded. It establishes the executable environment and invokes *dld* when the local agent is required.
- *dld* – This dynamic linker and loader originally implemented by W. Wilson Ho [Ho91] (ported to the DEC MIPS Ultrix 4.2a environment) allows the supporting local agent routines (*asm_stubs* and *dint_lib*) to be loaded into the same area of process memory as the executable program.
- *asm_stubs* – This small set of assembler routines assist in the transition to and from the standard executable environment into the debug or interpretation mode. The concept of 'context switching' is later discussed in section 5.5 [CmKe93, Kess90].

- *dint_lib* – The DCode Interpreter Library is the main component of the local agent. This component dynamically loads binary encoded DCode code blocks into memory and then interprets them upon demand (interacting with the current variable states contained within the executable program).

Likewise, the parent process is composed of:

- *main* – The main module of the *gpdb* interface consists of several sub-components which provide a user command parser, maintain information about current breakpoints inserted, and display the current source code line. This component forms the hub of the user interface.
- *Symbolic Reader* – This component transforms symbolic information available from within standard COFF executable programs and binary encoded DCode modules into the internal data structures available for function and variable information. This information is then used by *main* for display to the user, and the *process control* module for pertinent memory access.
- *Object Format Reader* – If the user's executable program is available to *gpdb* (i.e., the parent and the child process are contained on the same machine) then this module is available for the obtaining of symbolic information from sections within the executable program.
- *Process Control* – The *process control* component of the parent process has two functions:
 - control of the breakpoint insertion and removal routine; and
 - maintain connection with the child process for the sending and receiving of user commands.

The inclusion of the *symbolic reader* and the *object format reader* within the parent process allows it to directly analyse COFF executable programs, as well as binary encoded DCode intermediate language files which have been generated (upon demand from the user)². With these two components the parent process is able to symbolically represent the data it obtains from the child process, in either the real executable mode or when performing function interpretation. This

² Both the *symbolic* and *object format readers* are also contained within the *dint_lib* component of the child process, however this has been eliminated from Figure 5 for brevity.

reduces the cost of large data transmissions of symbolic information between the processes (although this can be done when *gpdb* is to be used in a remote environment).

5. Implementation of *gpdb*

As stated above, the problems with the debugging of optimised code are severe due to the fundamental problem that:

... program points in the source code may not have any corresponding point in the execution of the optimised object code.

Therefore, this approach assumes that –

... optimised code will not be modified except by the substitution of whole procedures.

This allows existing compiler intermediate code to be utilised for the representation of each source function, and has thus eliminated the need for such concepts as control-flow analysis. [GLE94]

gpdb has been prototyped on a DECstation 3100 machine (using a MIPS R3000 microprocessor) running the Ultrix 4.2a operating system environment with unmodified Gardens Point (GP) language compiler front-ends.

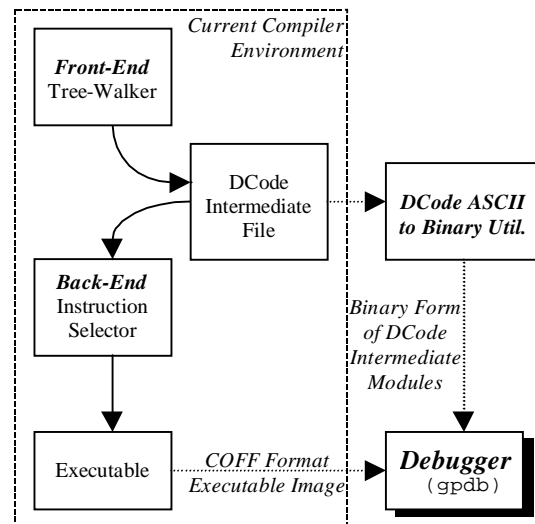


Figure 6: Overview of File Sources

5.1 Implementation of *gpdb* within the Gardens Point Compiler Environment

Figure 6 shows the interaction between the Gardens Point (GP) compiler environment and *gpdb*. No modification was made to either the *Front-End Tree Walker* or the *Instruction Selector* components of the GP compilers in this development.

Gpdb is designed to operate as a total debug environment. The executables are loaded from disk and the source code located and displayed like any other debugger. However, when the source code is modified (with some limitations placed upon the modification of global variable types and lengths) the whole source file is passed through the ‘front-end’ tree-walker to produce the intermediate DCode ASCII file of the module. The DCode intermediate instruction files are then ‘assembled’ into a binary file format to reduce the overhead of ASCII interpretation within the runtime environment – the DCode interpretation library (*dint_lib*). This binary DCode file is then made available to *dint_lib* to load into the child process for interpretation, when individual functions are required, as well as being used by the parent process’s symbolic reader for the meaningful display of program information.

5.2 Utilising Symbolic References

All symbolic references are read from both the current executable file, and any loaded binary DCode modules by both the parent process (through the object and symbol readers) and the symbol reader component of the DCode interpreter within the local agent of the child process.

However, although symbolic information is known in both processes their purpose is totally different:

- The parent uses symbolic information for the meaningful display of information to the user, as well as the command requests to the local agent. This allows the parent to remain at a high level of abstraction and reduces much of the low-level interactions (which have been placed into the local agent contained within the child)³.
- The child process on the other hand uses symbolic information to map the high-level command requests from the parent process into low-level memory addresses, as well as the symbolic references encountered during interpretation of binary DCode.

All machine-dependent knowledge has been encapsulated within the *asm_stubs* module within the local agent thereby allowing *gpdb* to be targeted to new platforms, architectures and symbolic environments more easily.

³ This also includes the easier migration of this model to a remote, socket-based, implementation and even removes the necessity for the parent and the child process to be running on compatible architectures.

5.3 DCode Interpretation

The DCode interpretation module at the heart of the child process local agent is similar, in structure, to many other language interpretation kernels in that it has been written to facilitate the reduction of stack manipulation, and thereby increasing the efficiency of execution/interpretation context switching. This not only reduces the complexity of the implementation but also reduces many of the overheads often found in interpretation systems.

This was achieved by implementing a number of small machine-dependent *jump* code blocks. These code blocks have been able to be developed independently due to the implementation of a ‘register determiner’ routine which calculates which jump code block is required, based on the register rules of the machine and the number and types of the parameters to be passed.

This structure has not only allowed the code of the interpreter, the machine-dependent ‘register determiner’, and the *jump* code blocks to be more readable and more easily maintained, but additional modules will be able to be more easily provided in future within a multi-platform, machine-independent source development environment.

5.4 Breakpoint Mechanism Implementation

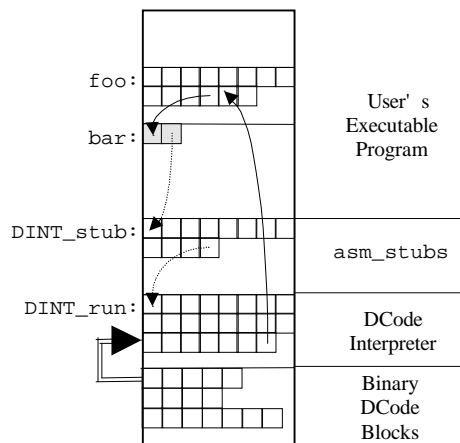


Figure 7: Child Process Execution when *bar* is Interpreted

Figure 7 illustrates the implementation of a breakpoint code patch, placed at the entry point to the function *bar*, which only contains two instructions. One instruction loads a specific breakpoint identifier into a temporary register (so that the DCode Interpreter knows which code block(s) to interpret), while the other performs an unconditional *jump* into one of the routines within the assembler code block set (*DINT_stub*). The function return address is then preserved and the

value of all registers saved into a global structure. Execution control is then passed to the DCode Interpreter (DINT_run). The interpreter then utilises binary DCode files that have previously been loaded into memory, or found on disk.

This method of implementation has allowed the number of assembler code lines to be kept to a minimum (no more than 100), and for the remainder of the system to be implemented in a high level language such as C. This not only has reduced the complexity of the task, but also eliminates much of the low level interactions which make debuggers more difficult and time consuming to retarget to a new architecture.

5.5 Execution/Interpretation Context Switching

The execution/interpretation context switching facility is used at the heart of the breakpoint mechanism described above, and whenever the user wishes to switch the mode of operation (generally for the ease of function replacement). This facility consists of a number of low-level routines contained within the *asm_stubs* and DCode Interpreter modules of the local agent within the child process.

The transition from the executable 'live' program to the interpretation state involves taking a direct copy of the values contained in each of the registers and saving them in a structure for later restoration.⁴

While in the interpretation state, context switching is used for the execution of library and other user defined procedures which are called from within the function being interpreted and are not chosen to be 'stepped into' by the user⁵. In this situation, the registers are allocated according to the procedure's parameters by the 'register determiner', and the stack placed in the state expected from the execution of the 'live' program. After the intended call the interpretation stack is restored with the addition of any returned results, and interpretation is continued.

When the context switch is finally made from the interpretation environment back to the 'live' executable, control can either be passed back to the point from which control came, or to the initial

calling routine (with any return results placed where expected). This flexibility allows either the total replacement of a function or the addition of code before the execution of a function (called a 'function prologue'). Both features allow *gpdb* to meet its primary objective to be able to modify optimised executable code at run-time without recompiling – all without the use of a modified set of tools.

6. Applications of *gpdb*

Although *gpdb* is able to function as a standard symbolic debugger, the full capabilities of the implementation can be more utilised to provide the following additional powerful facilities. The effective use of these concepts can be illustrated through the following typical debugging example.

While debugging a program which only seems to fail after it has been running for quite some time, the user chose to scan quickly the source code and place some "print" statements around some of the routines which were presumed to be causing the problem. The user then recompiled the executable (with debugging information "turned on") and then retraced through the steps of the previous testing process. However, this time a different problem occurred under different circumstances, and so the user again edits the source code and adds even more carefully crafted diagnostic source statements and then recompiles, reruns, and retests the program.

Eventually the user traces down the source of the problem – the updating of a dynamic memory structure. Therefore the user decides to further modify the source with the addition of a suite of routines for the viewing and validation of the dynamic memory structure in question. These routines are then applied at various places throughout the source code, as well as being utilised by the user as ad hoc callable functions within the standard debug environment.

Although this approach may be able to achieve the same end result, it heightens the advantages of such a debug platform as *gpdb*, viz.:

- When using *gpdb*, the run-time environment in which the error was initially discovered would not have been lost, even though the executable was not compiled with debugging information "turned on". *gpdb* allows the evaluation of variable values with a minimum of symbolic information available.

⁴ A temporary register is used for the unconditional jump to the register save area, while the value of the Program Counter (PC) register is passed within another temporary register. The saving of the executable's PC value allows for the continued execution from the point of interruption (later discussed as Function Prologues).

⁵ Only inter-procedural optimisations such as *inlining* are currently handled.

- From within `gpdb` source code modifications could have been made for the addition of the initial “print” statements without interrupting the run-time environment and without recompiling, relinking, and retesting the executable.
- Function prologues could have been built at run-time for the validation of the dynamic memory structure and placed at the start of various functions.
- Totally new whole functions (or just a code fragment) could have been coded for the more complex evaluation of the dynamic memory structure without the need for recompiling or relinking. These routines could then have been interpreted at will, or just applied as function prologues.
- A history could be provided of the values of certain key variables, throughout the execution of the program. This information could be collected and validated by routines, thereby providing tolerance levels of variable value changes, or merely a collection point of variable information that could be later analysed formally.

6.1 Function Prologues

The concept of interpreting modified source code has been extended in `gpdb` to allow code assertion blocks, which are independent from any function, to be utilised throughout a program’s execution. This has allowed a suite of program validation routines to be built at any time throughout the development process and to be used through the DCode Interpreter as and when required.

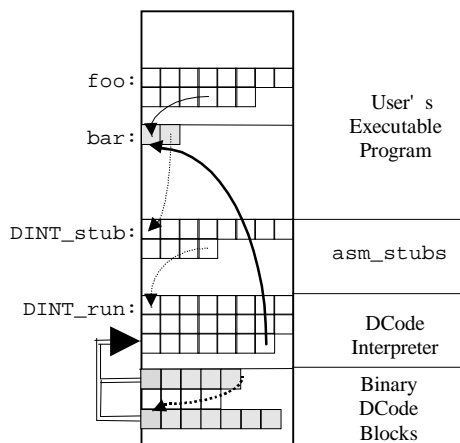


Figure 8: User's View of the Operation of `gpdb`

Figure 8 shows how the execution of the function `bar` from within the function `foo` invokes the DCode Interpreter. Once within the interpreter, the breakpoint identifier (contained within the

patch inserted at the start of the function `bar`) is used to determine which binary DCode block(s) are to be read and interpreted.

Once the interpretation commands have been completed and any interactions with the parent debug interface have been finalised, the environment is ‘cleaned’ to the pre-interpretation state (including the replacement of the original instructions at the start of `bar`) and the point of execution returned to the start of `bar`.

6.2 Tracing Variable States

One of the facilities within `gpdb` which has proven to be invaluable for the application of final debug assertions is *variable tracing* (sometimes referred to as *data watchpoints*). It has been implemented within `gpdb` as an automated conditional function prologue facility.

Variable tracing allows either:

- the collection of a variable’s value at certain points throughout the execution of the program and merely recording it for later evaluation; or
- collection of a variable’s value at certain points throughout the execution of the program and provide a comparison between it and other historical values. This allows for the interpretation of a certain routine upon the successful evaluation of a condition or a tolerance level.

In this example, a variable trace could have been applied by the user to particular key variables; suitable levels of tolerance applied; and a validation routine specified for interpretation (or execution) when these levels are breached. This would have allowed a fast and broad debug/testing approach to be applied which has the potential to more easily and quickly narrow the user’s attention to the problem at hand.

6.3 Automation of Program Testing

One of the largest problems in debugging, besides the finding of the bug itself, is the reproduction of the test environment. `Gpdb` has sought to reduce this problem by allowing the maintenance of the run-time environment and the dynamic patching of the user’s executable program. However, test plans are important as they record not only how a sequence can be retraced, but also as a form of documentation as to the level of completeness of the whole test suite.

Although the variables concerned could be traced and their values recorded, such an approach would only prove to be laborious (recording each variable), and prone to incompleteness. Therefore, a facility has been provided within `gpdb` to allow the collection of all input and output values at a low level (therefore ensuring the capture of all values). This facility not only allows the background recording of a test session but also the loading and application of a previous set of values to the target program, thereby allowing each test session (which may consist of many test plans) to be easily reproduced.

This facility has been found to even have more value when used in conjunction to the variable tracing facility described above – this not only provides an automated testing environment but also an automated validation and recording facility.

7. Future Work

The current implementation of `gpdb` solely utilises local agent technology. In future, `gpdb` will be migrated to a multi-threaded environment. This will allow the interrogation of the target process without the need for a local process modification interface (currently achieved through `ptrace()`). The target process can then be stopped temporarily by the parent, and the results gained through interpretation returned, all without having to insert a function breakpoint. This would also allow `gpdb` to be implemented as a remote debugging tool without the heavy modification of the existing implementation.

8. Conclusions

This research has aimed to allow a user to modify the source code of a function, insert additional debug information, and then make it available for interpretation, while eliminating the need for the recompilation of the executable, and allowing the preservation of the present executable state. This has proven to be extremely useful when tracking errors that are not easily reproduced and usually only surface after many hours of tedious debugging.

While current systems make use of computationally intensive techniques to allow the debugging of specific code optimisation mechanisms, `gpdb` utilises a function-level code replacement concept which allows for the dynamic replacement (and removal) of executable code blocks with interpretable modules. In addition,

`gpdb` does not require the use of specially modified compilers and is language and platform independent.

This approach provides a framework for building run-time interpretation modules and allows for the complex evaluation of run-time states within a comprehensive debugging environment. This environment is already proving to be indispensable for the automation of a comprehensive testing environment and the run-time resolution of program bugs.

9. Acknowledgements

Whenever I required guidance, encouragement and support from my mentors and supervisors, Prof. K. John Gough and Dr. John Hynd, I received it without exception. Without their untiring efforts this project would not have been possible. I would like to especially acknowledge the work Prof. Gough has placed into the development of DCode and the Gardens Point compiler environment.

References

- [BHS92] Brooks, G., Hansen, G.J. and Simmons, S. A New Approach to Debugging Optimised Code. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation; SIGPLAN Notices*, v27(7), pp 1-11, San Francisco, California, June 1992, ACM Press.
- [CiGo93] Cifuentes C. and Gough, K.J. A Methodology for Decompilation. In *Proceedings of the XIX Conferencia Latinoamericana de Informática*, pp 257-266, Buenos Aires, Argentina, August 1993. Centro Latinoamericano de Estudios en Informática.
- [CmKe93] Cmelik, R.F. and Keppel D. *Shade: A Fast Instruction-Set Simulator for Execution Profiling*. TR# 93-06-06. University of Washington, June 1993.
- [Copp93] Copperman, M. *Debugging Optimised Code Without Being Misled*. PhD Thesis, TR# UCSC-CRL-93-21. University of California, Santa Cruz, May 1993.
- [GLE94] Gough, K.J., Ledermann, J. and Elms, K. *Interpretive Debugging of Optimised Code*, Proceedings of ACSC-17, Christchurch, 1994.
- [Goug95] Gough, K.J. *The DCode Intermediate Program Representation Reference Manual and Report*, available from `ftp.fit.qut.edu.au/pub/papers/dcode300.ps.Z`, 1995.

- [HCU92] Hölzle, U., Chambers, C. and Ungar D. Debugging Optimised Code with Dynamic Deoptimization. In *Proceedings of the SIGPLAN '92 Conference on Programming Languages Design and Implementation; SIGPLAN Notices*, v27(7), pp 32-43, San Francisco, California, June 1992, ACM Press.
- [Henn82] Hennessy, J. Symbolic Debugging of Optimised Code. In *ACM Transactions on Programming Languages and Systems*, v4(3). pp 323-344, July 1982.
- [Ho91] Ho, W.W. DLD: A *Dynamic Link/Unlink Editor* (v3.2.3). University of California Davis, 1991.
- [Kess90] Kessler, P. Fast Breakpoints: Design and Implementation. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation; SIGPLAN Notices*, v25(6). pp 78-84, June 1990.
- [Pax90] Paxson, V. *A Local Agent Architecture for Implementing Efficient Breakpoint Debuggers*. CS 262, EECS Department, University of California, Berkeley, 1990.
- [PeSi79] Perkins, S.R. and Sites, R.L. Machine Independent Pascal Code Optimisation. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pp 201-207, 1979.
- [Pizz93] Pizzi, R. *GNU Debugger Internal Architecture*. University of California Davis, December 1993.
- [RaHa92] Ramsey, N. and Hanson D.R. A Retargetable Debugger. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation; SIGPLAN Notices*, v27(7), pp 22-31, July 1992.
- [Rams93] Ramsey, N. *A Retargetable Debugger*. PhD Thesis, Princeton University, Princeton, NJ, January 1993.
- [Sosi92] Sobic, R. Dynascope: A Tool for Program Directing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation; SIGPLAN Notices*, v27(6), pp 12-21, June 1992.
- [Sosi95] Sobic, R. A Procedural Interface for Program Directing. *Software Practice and Experiences*, v25(7), pp 767-787, July 1995.
- [Wirt71] Wirth, N. The Design of the Pascal Compiler. *Software Practice and Experience*. v1(4), pp 309-333, 1971.
- [Wism94] Wismüller, R. Debugging of Globally Optimised Programs Using Data Flow Analysis. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation; SIGPLAN Notices*, v29(4), pp 278-289, June 1994.
- [Zell83] Zellweger, P.T. An Interactive High-Level Debugger for Control-Flow Optimised Programs. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, 1983.
- [Zura91] Zurawski, L.W. *Debugging Optimised Code with Expected Behaviour*, University of Illinois at Urbana-Champaign, 1991.