Interpretive Debugging of Optimised Code

John Gough, Jeff Ledermann and Kim Elms*

Abstract

Traditional debuggers for imperative languages have relied for their operation on information extracted at the time that the modules of the program were compiled. Here, we discuss an alternative model, in which any needed information is extracted by coopting a modified version of the compiler during the debugging process.

The potential advantages of this approach include the ability to debug programs which were not compiled with "debug" options, including those which are globally optimised. The ability to mix interpreted and optimised, compiled code allows arbitrary program animation, and the insertion of arbitrary program assertions specified *after* compilation and *during* the debugging process.

1 Overview

In this paper we introduce a concept which, in principle, allows the capabilities of debuggers for conventional imperative languages to be significantly extended. These concepts are the basis of a project recently started, so we report design data and the preliminary stage of implementation of these ideas. In particular, we are interested in extending the capabilities of conventional debuggers for imperative languages such as *C, Pascal, Modula* and *Ada*, and their object oriented extensions. We restrict ourselves to conventional, uniprocessor but perhaps multi-pipeline machine architectures.

In section 2 we review the history of program debugging, and discuss the ways in which the capabilities of current debuggers fall short of our ideal. In the next section we give an overview of the capabilities of the next generation of language independent debuggers currently under development. We also discuss the internal mechanisms of these debuggers, and some of the alternative proposals which have been experimented with.

In section 4 we introduce the conceptual basis for our extended capabilities, and discuss the implementation of these ideas. We then describe the current level of implementation of a prototype, and discuss the future progress of the project. Finally, we draw some interim conclusions.

2 Why Debugging

2.1 Some History

Computer programs often need to be examined to determine the cause of apparent errors, or to gain a better understanding of their structure and control flow. This examination is called **debugging**, since its usual objective is the location and removal of program errors (bugs). This need arises during both the development phase in the software life cycle, and during subsequent software maintenance. It has been estimated that as much as 70% of annual information systems budgets are spent on maintenance, so that any tool which enhances the efficiency of this process has important practical value.

Since the very earliest days of computing it has been realised that the machine should provide some assistance in this process. In what is possibly the earliest journal article on what we now call debugging[9] this necessity was articulated. This paper already contains the concept of *post-mortem debugging* (analysing memory contents after program abortion), *single stepping* (stepping through the program one instruction at a time so as to animate the effects), and the insertion of *breakpoints* (halting the program execution at specified points to examine the machine state). These concepts are still the mainstays of contemporary debuggers.

During the 1980s there was a movement among some computer scientists to declare debugging to be politically incorrect, claiming instead that software

*All authors are with the Programming Languages and Systems group (PLAS) at the Queensland University of Technology

designs based on correctness proofs, or on mathematical derivation from formal specifications removed the need for such extremities.

A calm reassessment of these claims shows that although formal methods do indeed assist in the production of programs with reduced errors, they do not and will not eliminate the need for tool assisted debugging. Shapiro in his award-winning dissertation[21] states the case well —

> As a program can be proven correct only with respect to another formal description of its intended behaviour, this observation suggests that even if the effort in program verification succeeds, it does not solve the problem of debugging specifications.

2.2 Debugging is difficult

It has been noted that debugging is difficult. There is evidence that it is difficult because it involves both analytical and intuitive processes[13]. However, it is also hard because the tools which are available are, in general, woefully inadequate. It has been realised, essentially since the invention of modern high-level programming languages that debugging should be carried out in the semantic terms of the original source program[19]. However this goal still proves elusive.

To some extent the problems encountered in the use of conventional tools can be attributed to the programming model that they support. The program is viewed as a mechanical artifact that is examined, after failure for instance, run one step at a time or stopped in a particular state. This model facilitates a purely analytical approach to debugging.

The problems of debugging parallel and vectorised programs are even more extreme, and are not addressed by this project. Neither do we address the rather different problems of debugging languages such as *Lisp* or *Prolog*. We are concerned only with imperative procedural languages, such as *C, Modula, Ada* and their object oriented relatives *C++*, *Oberon-2* and the proposed *Ada9X*.

2.3 A Model of Debugging

We have an informal model of the debugging process. This model suggests that debuggers should support certain kinds of interactive reasoning about existent programs, and allow for the *post facto* testing of hypotheses.

We believe that debugging typically proceeds by an alternation of two activities. The first is program examination, the second is hypothesis testing. During the first activity the program is examined in an attempt to understand the phenomenon involved. This may require the examination of the program source, and its input-output behaviour. It may also require examination of the internal state of the program at interesting points of execution. The purpose of this activity is to assist in the formulation of hypotheses regarding the reason for the perceived aberrant behaviour.

The second activity is the testing of the hypothesis formed during the previous stage of program examination. The successful testing of the hypothesis should lead either to a refined focus for the examination activity, or the formulation of a refined hypothesis. Rejected hypotheses also refine the focus of examination, and serve to partition the space of possible causes.

This model suggests that debuggers should provide facilities to–

- examine the internal state of programs at interrupted states of execution and after abnormal termination
- animate the execution of interesting parts of programs, so that the behaviour can be checked against the user's understanding of the source program semantics
- allow for the testing of arbitrary assertions, which are specified *during* the debugging session
- allow for the selective replacement of program parts by alternative code specified *during* the debugging session

In the following we examine the level at which current debuggers support this kind of interaction.

2.4 Compiler optimisations

It is an unfortunate fact that almost all compiler optimisations place obstacles in the path of debuggers. Some optimisations place insurmountable obstacles in the path of conventional debuggers. In particular those optimisations which use data out of machine registers rather than memory, and those that eliminate or move code and procedures, cause particular problems. For some machines, the performance penalty for not performing at least the so-called local optimisations is so great that compilers do not even permit a completely "unoptimised" version of the program to be created. Furthermore, it seems clear that the situation will become even more difficult in the immediate future. In order to exploit the capabilities of the new generation of multi-pipeline architectures it seems that gross code motion will occur in at least two stages of compilation. Code must be moved as a result of dataflow analysis, in an attempt to extend basic block length. Current research on instruction scheduling also suggests that scheduling must treat entities larger than a basic block, allowing instructions to move around the control flow graph in an arbitrary fashion.

Some valiant work has been done in allowing conventional debuggers to find variables in registers[14, 6, 24], warn the user of misleading values[14, 3, 23], and reconstruct the logical values of data which do not have separate existence in the optimised program. Nevertheless production compilers severely restrict the possible optimisations performed when debugging capability is specified at compile time.

The inability of conventional debuggers to deal with optimised programs is a severe handicap. Certainly, it is customary and sensible to perform much testing early in the software development process. However, programs must meet performance criteria prior to release, and this necessitates the introduction of optimisations. The need for testing and debugging is a continuing part of the software life-cycle, and is especially critical to the ongoing maintenance of released, production code[20].

Current work[1, 15] in the debugging of optimised programs is focussed on extensions of Hennessy's seminal paper[14, 23].

2.5 An interpretive interface

Various investigators[4] have pointed out the advantages of *interpretive* execution of programs during interactive debugging. The interpretive approach facilitates stepping through a piece of code and also provides for the interactive entry of additional code, perhaps to compute additional information for the user, or to correct a recognised error. In particular, the interpretive approach allows the debugging commands to be in the same language as the program being debugged. Unfortunately such interpretive approaches are either restricted to those languages which are customarily implemented by interpretation or are based on environments which operate excruciatingly slowly[1]. Much of the advantages of interpretive execution are also obtained for systems which are based on incremental compilation.

2.6 Language neutrality

Since current debuggers rely on information in the symbol file to interpret data, they assist only the users of languages which have their data types built-in at the time of the design of the file format. For example, `dbx` supports only scalars, arrays, pointers, structs and unions. Currently, the debuggers supplied by the major workstation vendors are being enhanced to support the *C++* language. This exercise demonstrates the problem: these enhancements typically only deal with the idioms of a single implementation of *C++*, and provide no help at all for the users of other, even similar object-oriented languages. We believe that the approach of placing additional information in the executable file and enhancing the debugger to interpret the new information is fundamentally flawed, despite its appropriateness for single-language environments[5].

We are particularly interested in the problems of debugging programs written using object oriented languages. These provide problems for current debuggers, since these languages have hidden mappings of programmer specified names to the unique names in the address space of the link-editor.

We believe that debuggers should be language-neutral, and should be able to work conveniently with programs which involve more than one language. In particular any symbolic expressions which the user specifies should be expressed in the syntax of the source, and with source language semantics.

2.7 Coopting the compiler

An interesting approach to language-neutral debugging has been demonstrated in prototype form at the *SIGPLAN* symposium in mid-1992[17]. In this

[1] One of the authors worked on a direct execution project at the University of Maryland[12] in the early 1980s. The debugging capabilities of these systems were the high point of an otherwise misguided concept.

paper the authors introduced the idea of arranging for language neutrality by coopting the assistance of (a modified version of) the compiler at debug time. Since the compiler surely understands the source language, the debugging commands can be expressed in the same language as the source code. The compilers communicate with the debugger process using a modified version of the *Postscript* language.

In this paper, no attempt was made to allow for interpretive execution of the program being debugged. Furthermore, the executable programs which were able to be debugged were not optimised. In fact, the programs needed to be compiled with additional marker instructions leading to even slower execution than the normally "unoptimised" case.

3 An Overview of Current Debuggers

Since we are interested in multi-language and platform-independent debugging, we discuss the state of the art of debuggers which address this challenge. The most widely used debuggers for imperative language programs on workstation and *UNIX*[2] machines are `dbx`, the Gnu equivalent `gdb`, and USL's `sdb`. All of these tools have roughly similar capabilities and limitations. These debuggers rely on the presence of information in the executable file of the program, to allow variables to be located in the memory space of the executing program. In general, if more capability is required at debugging time, then more information needs to be written to the file at compile time. Setting the appropriate compile options, so as to produce this additional information invariably requires the foregoing of many compiler optimisations. There are two reasons for this. Firstly, the program must ensure that all variables are written to memory, so that they are able to be examined by the debugger. Secondly, if the program is to be stepped line by line, the code for different lines should not be interleaved, or hoisted out of loops, and much common subexpression elimination must be left undone.

[2]UNIX is a trademark of Unix System Laboratories.

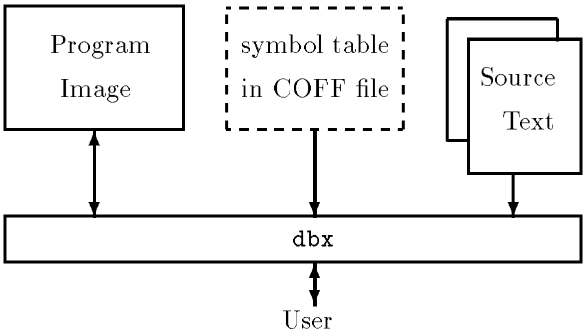

Figure 1: Conventional debugger

3.1 The *DWARF* Format

The *Unix International* organisation has developed an executable file format which specifies debugging information in a source language independent form[22]. The format specification of the debugging information is called *DWARF* and is attached to executable files in the standard *ELF* format. This specification can be used to remove some of the current limitations of debuggers, and defines a set of attributes which are sufficiently rich to allow symbolic debugging of all the common languages.

DWARF provides attributes which specify location information for the variables of the program in question. This location information can be simple (name,location) pairs, or may be an abstract description of the addressing expression which must be executed to find the location of the variable. These abstract expressions are instructions for an abstract stack machine of conventional capabilities. It is expected that full functionality debuggers for *DWARF* will provide an interpreter for this address-expression language. Finally, *DWARF* provides that variables may be located by means of location lists. These lists allow for variables which are found in different locations at different stages of execution. The lists consist of tables of tuples with instruction counter bounds and location expressions. The use of such location lists will, in principle, allow for such things as variables which are never written to memory but are live in different machine registers at different times during program execution.

It may be expected that debuggers based on *DWARF* will (with the cooperation of the compiler) be able to find the values of variables even in the presence of high levels of optimisation. However, out of order execution of instructions in optimised programs will still make line number markers deceptive, and program animation problematical. We note that the level of debugging information desired must still be specified at compile time, and the ability to test assertions will be limited to simple value bounds. In particular it should be noted that the execution of compound assertions specified at debug time, is impossible[3].

4 The Proposed Model

We propose the following overall structure for our debugging environment—

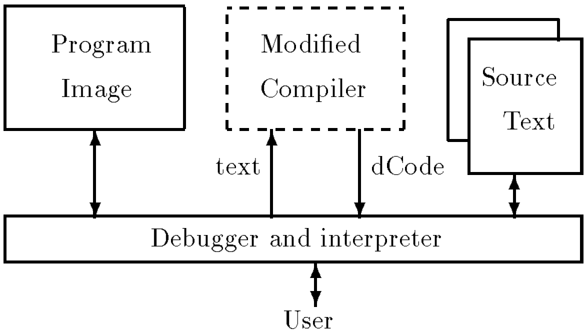

Figure 2: Debugger with coopted compiler

A language independent debugger communicates with the user via one or more windows. The debugger is connected to either a program under test, initially in a suspended state, or to a core-dump image of a faulting program. If a command requires information regarding a particular module of the program under test, a modified version of the compiler responsible for the code is invoked. The compiler will have access to the original source code of the module. The debugger now communicates alternately with the program under test and the compiler. The debugger process contains the usual trap handlers and symbol table mechanisms of conventional debuggers. However it also contains an interpreter for the language independent intermediate representation (*DCode*) which our compilers use.

It is important to see that the interpreter is in a somewhat unusual situation. The interpreter's own data structures exist in the address space of the debugger. However some of the data to which the interpreter refers exist only in the separate address space of the program under test. There are important issues of performance which need to be determined for this approach.

The debugger may be expected to provide all of the facilities of conventional debuggers, using the same mechanisms. In particular, our early implementation efforts are based on *ELF* executables, with debugging information specified in the *DWARF* format. However, our current research is directed towards the innovative aspects of the system, rather than the duplication of facilities which current systems can deliver.

4.1 New Capabilities

We now describe the way in which this new architecture achieves the functionality which we itemised in section 2.3.

4.1.1 Step-by-step execution

If a command requires step-by-step execution of a particular procedure, the compiler creates intermediate code for that procedure. The debugger then places a trap-point in the program image, at the entry point of the optimised code of that procedure. When execution reaches the intended procedure, the trap handler invokes the interpreter, so that the execution of the interpreted code is dynamically substituted for that of the optimised procedure. Since the bulk of the code is unchanged, the program will run at substantially the original speed until the procedure of interest is reached. The execution of the specified procedure may then be examined at any level of detail required. Note that the querying of the values of non-local variables during such intepreted execution is still limited by the conventional debugging information available in the executable file.

4.1.2 Substitution of alternate procedures

Any procedure with an accessible entry point may be substituted by a new procedure specified during debugging. The compiler generates intermediate code for the new procedure, and passes this back to the debugger. The entry point of the original has a trap inserted into it, and the trap handler invokes the *DCode* interpreter with the intermediate form

[3]As an example, consider trying to test a complex assertion such as `Assert(IsBalanced(tree))` if this was not anticipated at compile time.

of the new procedure as data.

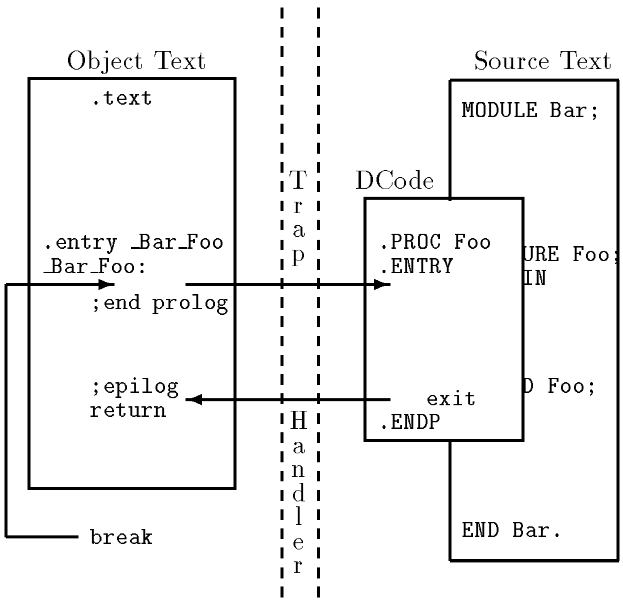

Figure 3: Inserted `break` traps to DCode

4.1.3 Arbitrary assertions at breakpoints

Using the described mechanism, arbitrary computable functions may be specified at debugging time, and attached to the program under test as breakpoint assertions. Whenever such a breakpoint is reached, the trap handler would cause the interpreter to compute the specified function. Execution would resume within the original program unless the function signalled failure of the predicate.

These breakpoint assertions may compute functions on any accessible part of the program state. This is a small limitation, since it relies on the conventional debugger information, but we remark that such information is more readily available at procedure calls than elsewhere in programs. Furthermore, should access to non-local objects become mandatory, the part of the program controlling that part of the state may be replaced by an intepreted version with accessible state as described in section 4.1.2.

4.1.4 Higher level debugging

It is often difficult to forsee what special debug code will be useful to find a particular problem. This can sometimes lead to many edit/compile/execute cycles. Typically after the problem is fixed, the code added for debugging is discarded. A simple but effective technique is to allow the programmer to create the debugging code within the debugger environment. The debugger should provide a mechanism for indexing the code and linking it to the project. We believe that over time a project using these techniques would build up a suite of debugging functions that could be shared amongst the programmers involved in the project, providing them with a higher level of abstraction. In effect these functions would (to a point) provide the ability to debug in the semantics of the application itself.

4.2 Some Observations on the Method

4.2.1 Which compilers can be mixed

There are two cases to consider. The compiler invoked by the debugger is the same as that which produced that particular part of the program which is currently of interest. Alternatively, we need to consider the consequences of trying to debug programs produced by other compilers for the same language.

If the debugger is using (a modified version of) the same compiler which produced the original executable, there are some advantages. Even if the compiler was not originally requested to produce detailed location information for variables, this can be done, after the fact, at debugging time. It is true that a request for the location of a particular program variable may return *"variable is not available"* if high levels of optimisation were invoked. However, we can at least find out which variables **are** available. In particular, in the case of nested procedures, the safety of accesses to variables of enclosing procedures will be able to be determined.

In the case of programs produced by other compilers there are some additional difficulties. At the lowest level, there may be difficulties in locating variables not listed in the symbol table. Even for variables known to the linker, and present in the symbol table, there may be difficulties in reproducing the somewhat arbitrary mapping from source code name to linker name used by the original compiler. Nevertheless, our preliminary investigation shows that some functionality may be able to be retrieved even in this case. In any case, as noted above, we may circumvent any problems by interpreting sufficiently large portions of the original program.

4.2.2 Limitations of the approach

The approach outlined above does not quite solve the problems caused by all possible optimisations, however it does eliminate difficulties caused by the following optimisations —

. local common subexpression elimination,
. global common subexpression elimination,
. value tracking,
. constant folding,
. instruction scheduling,
. induction variable substitution,
. loop unrolling,
. dead variable elimination,
. code motion out of loops,
. loop fusion,
and indeed all code motion within a single procedure.

We do not have a simple solution to the problem of debugging inlined procedures, other than the interpretive execution of the enclosing procedure. We believe that this is not a fundamental limitation, but rather an annoyance.

The debugger will attempt to place breakpoint assertions into procedures without interpreting the whole procedure. This is only possible where the state of the (optimised) computation corresponds to the logical state expected in the source code. For heavily optimised procedures this may mean just the entry point. Furthermore, the expressions may only refer to variables which have values at known locations. These will include procedure parameters and some subset of the non-local variables of dynamically enclosing procedures. We believe that existing deoptimisation techniques may lessen the impact of this limitation, and the evaluation of this hypothesis is an important research outcome of this project.

Almost all debuggers have difficulty assisting in the debugging of programs which rely on false assumptions regarding the semantics of the programming language. For example, most languages do **not** define the order of evaluation of parameters to procedure calls. Compiler writers argue that to do so would force an intolerable inefficiency on their code generators for RISC machines. In any case, when programs rely on such undefined behaviour the result of execution may very well vary with the optimization level which is used during compilation. For our particular compilers, the order of parameter evaluation is determined in the front-end, and would thus be the same for interpreted and compiled code. However, the interpreter would in general be unaware of any reordering of factors in an expression which the code generator might apply to lessen register use.

Finally, we note that our method does not address the needs of compiler writers who wish to debug their code optimisers. Substituting a correctly executing interpreted procedure for an incorrectly executing optimised version of the same procedure may confirm that the optimiser is faulty, but does not directly assist in diagnosing the fault.

4.2.3 Relation to previously published work

The most closely related research of which we are aware, was performed at Rice University, and reported in a paper by Chase and Hood[2] in 1987. These researchers recognised the possibility of mixing compilation and interpretation, and discuss the advantages of such a hybrid scheme. However, their work differs from our project in several ways. Their focus was computationally intensive *FORTRAN* programs, and the possibility of substituting exact real arithmetic for the approximate arithmetic of compiled code. They did not involve the compiler in the debugging process, but instead kept a library of pre-compiled abstract syntax trees in a database. Finally, they did not consider the issues involved in multi-language debugging.

There is also a non-standard "UPS" debugger available on the net, which provides a limited capability to interpret additional language C code inserted into existing procedures[18]. This tool is motivated by many of the same concerns as our project, but does not address either the language neutrality issue, or the introduction of entirely new procedures.

5 Current Status

5.1 Existing technology

This project is substantially based on previous language implementation research by the QUT PLAS team. The architecture of the compiler projects at QUT are described elsewhere[11]. Native code compilers, which form the basis of this project, consist of front-ends based on attributed abstract syntax trees, tree-walking automata, and an abstract stack machine intermediate form called *DCode*. Currently there are front-ends for *Modula-2* and *Oberon-2* languages, and back-ends for MIPS, SPARC and Intel-386. We have a prototype version of the *lcc ANSI-C* compiler front-end[7, 8] which produces *DCode*. Essentially the same compiler technology has been used in a prototype *C++* front-end, and in a Motorola 68040 code-generator.

We have a separate projects which will create a version of the Modula and Oberon compilers for Digital Equipment Corp's *ALPHA* architecture, and the IBM *Power* series, the RS6000. These compilers are now running in prototype form.

Our existing compiler front-ends can produce interpretive code for isolated procedures, and with relatively minor modifications are able to perform incremental compilation as required for some of the aims. However we wish to test the general applicability of our technique by attempting the equivalent modifications on a front-end produced by a different technology. We will be doing this with the `lcc` portable front-end for *ANSI-C*.

One of the important aims is to be able to produce intermediate code for specified procedures (or modified procedures) with minimal delay. We note that our current compilers can produce intermediate code on modern workstations at speeds of several hundred thousand lines of source code per minute.

We have already demonstrated the possibility of directly interpreting *DCode* in our widely available free-ware version of **gardens point modula, gpm-pc**[10].

5.2 Progress so far

We have experimented with prototypes which use facilities of existing debuggers such as `dbx`. These allow programmers to specify procedures for displaying complex structured values using the same format and abstractions as the source program. Although an interesting advance on current capabilities, this feature requires that the display function be specified at program creation time. We would prefer that the display function be automatically produced, and be modifiable if necessary, at debugging time.

For the object-oriented language *Oberon-2*, we have experimented with the automatic production of a *DisplayMe* method for each defined class. This is a relatively simple modification, which overcomes some of the disadvantages listed above. However, it is inapplicable to non-object oriented languages, where (for example) it would need programmer intervention to determine the most semantically meaningful way of displaying a variant record.

We are currently investigating two alternative mechanisms for accessing the address space of the program under test. These are the *ptrace*(2) system call, and the use of the */proc pseudo*-filesystem[16]. Implementation of a target independent version of the *DCode* interpreter is proceeding. We have a proof of concept implementation running at the time of writing this paper, and expect to have most of the functionality demonstrated in early 1994.

6 Conclusions

We believe that the concepts described here represent a significant extension to the state of the art of multi-language debugging of conventional imperative languages. In particular, we claim the following "new" capabilities—

- debugging of fully optimised source code, even when the debugging capability was not specified at the time of compilation
- the ability for the user to add arbitrary breakpoint assertions to programs under test, or to evaluate assertions on core dump images
- language neutrality, allowing data structures to be displayed and expressions entered, using the idioms of the particular source language
- interpretive execution of selected portions of the source code of the program under examination, including immediate testing of modified code

References

[1] Brooks B, Hansen G J, and Simmons S, "A new approach to debugging optimized code", Proceedings of the SIGPLAN Conference on programming language design and implementation, San Francisco July 1992.

[2] Chase B C and Hood R T, "Selective interpretation as a technique for debugging computationally intensive programs", ACM SIGPLAN Symposium on Interpreters and Interpretive Techniques, 1987 (also SIGPLAN Notices **Vol 22 n7**).

[3] Coutant D, Meloy S, and Ruscetta M, "DOC: A practical approach to source-level debugging of globally optimized code", Proceedings of the SIGPLAN Conference on programming language design and implementation, 1988.

[4] Dakin R J and Poole P C, "A Mixed Code Approach" The Computer Journal, **Vol 16 n3**, 1973.

[5] Dearle A and Brown A L, "Safe Browsing in a Strongly Typed Persistent Environment", The Computer Journal, **Vol 31 n6**, 1988.

[6] Ferrante J, "High-level language debugging with a compiler", Proceedings ACM SIGSOFT/SIGPLAN Symposium on High-Level Debugging, ACM 1983 (also SIGPLAN Notices **Vol 18 n8**).

[7] Fraser C W and Hanson D R, "A code-generation interface for ANSI-C", Software Practice and Experience, **Vol 21**, Sep. 1991.

[8] Fraser C W, Henry R R and Proebsting T A, "BURG – fast, optimal instruction selection and tree parsing", Tech Rep 1066 University of Wisconsin, 1991.

[9] Gill S, "The diagnosis of mistakes in programmes on the EDSAC", Proceedings of the Royal Society, Series A, **Vol 206**, 1951.

[10] Gough, Corney, Cifuentes, Hynd and Kolb, "An experiment in mixed Compilation and Interpretation", Proceedings of ACSC-15, Hobart 1992.

[11] Gough K J, "The portable Modula-2 front-end GP2D", QUT technical report 92/1 (also University of Tübingen report WSI-92-2).

[12] Gough K J, "Hardware Name-Mapping for Direct Execution of Block-Structured Languages", Proceedings of the International Workshop in High-Level-Language Computer Architecture, Florida, Dec. 1982 (and references contained therein).

[13] Gramlich, Wayne C, "Debugging Methodology" (session summary), Proceedings ACM SIGSOFT/SIGPLAN Symposium on High-Level Debugging, ACM 1983.

[14] Hennessy J, " Symbolic Debugging of Optimised Code", ACM Trans on Programming Languages and Systems (ToPLAS), **Vol 4 n3**, 1982.

[15] Holzle U, Chambers C, and Ungar D, "Debugging of optimized code with dynamic deoptimization", Proceedings of the SIGPLAN Conference on programming language design and implementation, San Francisco July 1992.

[16] Killian T J, "Processes as Files", Proc. USENIX Association Conference 1984, pps 203–207.

[17] Ramsay N and Hanson D, "A retargetable debugger", Proceedings of the SIGPLAN Conference on programming language design and implementation, San Fransisco July 1992.

[18] Russell M, "The UPS Debugger" (software program). University of Kent at Canterbury 1991.

[19] Satterthwaite E, "Debugging tools for high level languages", Software-Practice and Experience, **Vol 2**, pp 197–217, 1972.

[20] Seidner R, and Tindall N, "Interactive Debug Requirements" Proceedings ACM SIGSOFT/SIGPLAN Symposium on High-Level Debugging, ACM 1983.

[21] Shapiro E Y, *Algorithmic Program Debugging*, The MIT Press 1982.

[22] Unix International programming languages SIG, *DWARF Debugging Information Format*, Version 2, Draft 4 (unapproved draft). Unix International, October 1992.

[23] Wall D, Srivastava A, Templin F, "A note on Hennessy's *Symbolic Debugging of Optimised Code*", ToPLAS **Vol 7 n1**, 1985.

[24] Zellweger, Polle T, "An interactive high-level language debugger for control-flow optimised programs", Proceedings ACM SIGSOFT/SIGPLAN Symposium on High-Level Debugging, ACM 1983.