Extending Compiler Intermediate Languages to Improve Debugger Design

B. B. Chase
P. D. Hahn
R. T. Hood

CRPC-TR89005
April, 1989

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Abstract

Traditional source level debuggers are heavily dependent on both the programming language being debugged and on the target machine architecture. These dependencies place a heavy burden on a programmer who wants to extend the debugger to work on a different language or machine. In order to reduce the effect of those dependencies we propose a design for debuggers that separates them into two parts: one machine independent and the other language independent. The two components communicate via an enhanced compiler intermediate language. This design promotes remote debugging, debugging among heterogeneous machines, ease in porting the debugger to different architectures, and reusability of the machine-specific portion of a debugger. The maps required to convert between source language constructs and machine language addresses can be easily generated by a compilation system that uses an enhanced compiler intermediate language. We discuss the enhancements that are necessary to support this debugger design. In addition, we show how many of the debugging requests that cause interpretive activity on the part of the debugger can be implemented in the extended language. This effectively shifts the burden of execution to the target program.

1 Introduction

Source level debuggers, such as the *dbx* [Com83] debugger that runs on UNIX[1] systems, have been a tremendous boon to programmers. They permit the user to control a machine language program in terms of source level constructs. This is typically accomplished by having the debugger use source $\leftrightarrow$ machine mapping information produced by the compiler during translation and updated by the linker after relocation. Figure 1 shows a simplified view of the traditional relationships between the compiler and debugger. While this design has been reasonably successful, it does have a significant drawback—the resulting debuggers are heavily dependent on both the language being debugged and the architecture of the machine on which the program is run. These dependencies greatly complicate the process of modifying the debugger to accommodate a different source language or target machine.

The standard approach for avoiding half of this problem is to have a different version of the debugger for each target architecture. If the machine-dependent parts of the debugging process are suitably abstracted and isolated into a collection of modules, this can be an effective way to provide portability. To build a debugger for a new machine, one only needs to re-implement the machine-dependent modules.

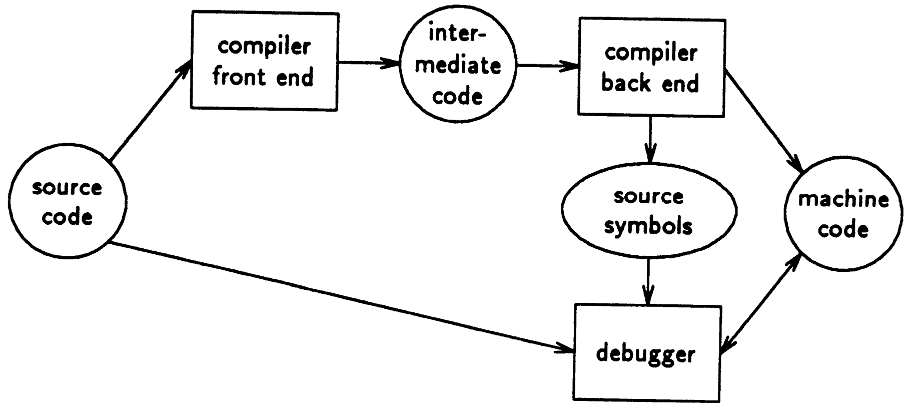

FIGURE 1: Traditional model of compiler-debugger cooperation

[1]UNIX is a registered trademark of AT&T Bell Laboratories.

Unfortunately, this approach begins to break down when the problem of remote debugging is considered. In this scenario, a debugger on one machine needs to control a process running on another machine whose architecture may be different. If that is the case, the debugger obviously needs to have knowledge about the architecture of the remote machine. Furthermore, to provide the correct mapping between machine language and source language, the debugger must know how to interpret the symbol table in the program on the remote machine. Standardizing the symbol table representation from machine to machine would help. It should be noted, however, that this is a very difficult process. Even on UNIX systems that purport to have a common symbol table definition, there are subtle but significant differences in the symbol tables generated by compilers on different machines. To compound the problem, if the compilers are performing optimizing transformations, then the symbol table notations that describe those transformations must also be standardized.

What we propose is that debugging systems be built around the abstract machine model provided by a compiler intermediate language. If a compiler group is responsible for supporting the implementation of a number of languages on several different architectures they are strongly motivated to adopt a common intermediate language. This allows great flexibility, because instead of needing a separate compiler for each language-machine pair, only one front end is needed for each language and only one back end for each architecture. By having the debugging process center on the intermediate language as well, it is possible to split the debugging system up into a small number of components. Such a debugger would consist of a front end that is capable of mapping from source language to intermediate language together with a back end that is capable of mapping from intermediate language to machine language. In response to user actions, the front end generates intermediate code that performs these actions. The back end executes this intermediate code in the current context of the program being debugged. It should be easy to achieve a debugger for any language-machine pair by coupling the appropriate front end and back end components.

The similarity between existing compilation systems and the debugging system that we propose is intentional. Compilers and debuggers need to cooperate extensively in order to do source level debugging. During the translation process, a compiler must produce the source $\leftrightarrow$ machine mapping information necessary to relate the source program to the machine language program. In our view, the source $\leftrightarrow$ intermediate information required by the front end of the debugger would be provided by the front end of the compiler. Similarly, the intermediate $\leftrightarrow$ machine map required by the debugging back end would be generated by the compiler back end. Figure 2 depicts the relationships between the compiler and debugger in our model.

There are other advantages to decoupling the debugging system besides reusability. For example, it would be simple to implement a remote debugging system by inserting a network communication package at the level of the intermediate code. Debugging operations translated by the front end into phrases in the intermediate language can be transmitted over the network to the back end for execution. With a proper definition of the intermediate machine abstraction, problems such as different data representations on the two machines can be avoided.

Such a division of labor also allows interesting research to be done using the independent parts. For example, work could progress on the user interface of the front end without affecting the back end. In addition, prototype front ends for the debugger and compiler supporting new source language constructs can be tested and developed without modifying the back end. If it is possible to express debugging operations in terms of the intermediate language, the set of operations supported at the user level can be enhanced

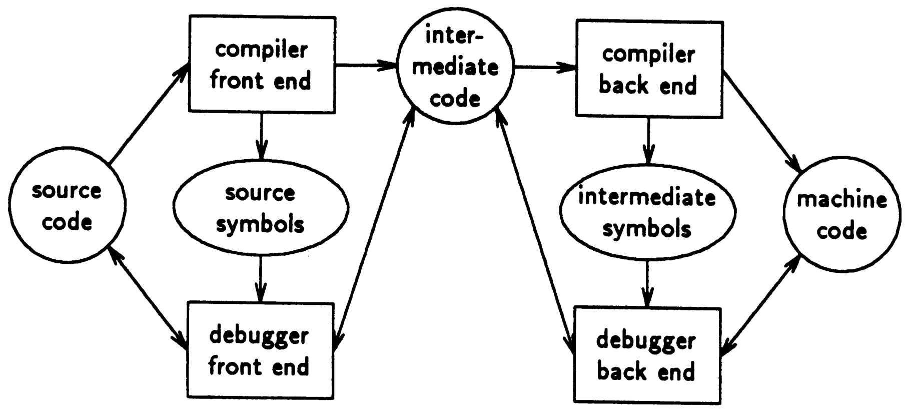

FIGURE 2: Proposed model of compiler-debugger cooperation

without changing the back end of the debugger. Once the intermediate debugging operations have been fixed and implemented in the back end in terms of machine operations, the front end will be free to implement new abstract debugging operations in terms of the intermediate ones. For example, we should be able to extend the capabilities of the debugger so that it could replay program execution, or dynamically monitor the performance of a running program, all via changes to the front end only, and hence in a machine independent fashion. In fact, if the back end of the debugger is clever enough, these debugging operations could be incrementally compiled into the machine language program, thereby reducing the amount of interpretation done by the debugger.

It is also possible to improve the back end independently. For example, optimizing transformations made on the intermediate representation affect only the back end of the debugger. In a monolithic debugger, transformations such as loop unrolling, cross-jumping, and procedure integration can produce a complicated source $\leftrightarrow$ machine mapping [Zel84]. Calling conventions specifying how parameters for procedures are passed, coupled with the requirements imposed on this by hardware, can make the jobs of reconstructing the values of parameters difficult. The use of an intermediate program representation can insulate the front end from these problems.

Under this scheme, debugger design concepts can be tested without writing both a language-specific front end and a machine-specific back end. That is, a machine-independent back end such as an interpreter could be used to explore the degree to which code that has been transformed in complex ways can be debugged. This can be done using the simplest of front ends. For example, the intermediate code from the compiler could be emitted at various stages during compilation, augmented with debugging commands to test various aspects of the debugger language, and then sent to the back end.

In the remainder of this paper we describe the properties that a candidate intermediate language should have in order to support our debugger design. We then detail how such a language could be used to perform standard debugging tasks and how some of the burden of interpretation required of the debugger can be pushed off into the machine code.

2 Design of an Intermediate Language for Debugging

When using our scheme to design a debugger for a given source language, two operations are required. First, there must be a translator from the source language to the intermediate language. Second, there must be a mapping between the source language and intermediate language representations of the program. The requirements of the mapping will depend on the choice of source language, and the sophistication of the user interface provided by the front end. Regardless, the choice of source language to debug should have little impact on the design of the intermediate language.

Similarly, the choice of target machine should only impact the design of the back end of the debugger. If a mapping exists between the intermediate language and the target machine's language, then it should be possible to write a back end for that machine, and thus implement a collection of source level debuggers for it. For a given intermediate language, it may be possible to implement a back ends for a large collection of architectures, but no intermediate language will work efficiently for all architectures. Difficulties may be encountered in accommodating architectural features such as multiple processors, vector processors, and instruction pipe-lines efficiently in one model.

The mappings performed by each portion of the debugger are analogous to the translations performed by the corresponding portions of a compiler. The intermediate language for debugging can be viewed as an intermediate language for a compiler, augmented with a small set of instructions to implement debugging operations. The various intermediate languages used by existing compilers represent starting points for the design of an intermediate language for a debugger. Syntax trees, postfix notation, and three-address code are obvious choices. For the design presented in this paper, we choose to start with a subset of a fairly simple three-address code called ILOC designed at Rice for use in optimizing compilers[BCT87].

Although there is considerable flexibility involved in the selection of and intermediate language to augment, the choice is not arbitrary. It is highly desirable if the language has a "locality of change" quality. That is, if a program written in this language is modified slightly, *e.g.*, an insertion or deletion of a few instructions, the code in other parts of the program should not need to change in order for the program to remain correct and the desired change to occur. An example of this is a relative branch instruction. If code is inserted into the region branched over, operands of the branch instruction will have to be changed to compensate for the insertion. ILOC largely satisfies this "locality of change" quality.

2.1 Debugging Extensions

We want to express as many debugging functions as possible in terms of the compiler's language, introducing new constructs only when necessary. Thus, tracing all executions of a particular point in a program could be implemented by a sequence of ILOC instructions at that point that would inform the front end of the debugger that the trace point had been reached. Thus, it is not necessary to add a "trace" instruction to the intermediate language. Is is clear, however, that some way will be needed to set trace points in the program being debugged dynamically, since we cannot know when the program is compiled which locations will be traced. To accomplish this, we will add an *insert* instruction, to insert some sequence of instructions, which we shall call a *fragment*, at some point in the program.

Also, we might need the sequence of inserted instructions to differ for different trace locations, perhaps to produce different messages to the front end so that it could identify which trace point was being passed.

Again, when the program is compiled we cannot know all the fragments that will be needed to debug it. Thus, we also add a *create* instruction, which takes a list of instructions, and returns some handle to a fragment composed of these instructions, which is suitable for inserting via *insert*.

Of course, if a fragment can be inserted, it will need to be removed as well. Otherwise, the program being debugged will become cluttered with unwanted tracepoints, and the fragments implementing these tracepoints will flood the front end with messages which the front end will have to ignore. So, when a fragment is no longer desired, it may be removed with *remove*, which simply reverses the effects of insertion but does not delete the fragment.

In addition to tracing, we may want to stop the execution of the program at some time, so that its state may be examined. An example of this is setting a breakpoint at a particular point in the program, where execution will stop. Also, once we have stopped the program, we will need a way to resume its execution. For this, we will need to add *suspend* which will cause the execution of the program to pause, and *resume*, which will cause a suspended execution to continue. We will need a *start* request to initiate program execution, and the program will be killed with a *quit*. There are other debugging instructions that will be useful. In Section 2.2, we will discuss some abstractions that promote an efficient implementation of the back end. [2]

In the preceding paragraphs, we have been deliberately vague about what kind of instructions might constitute the fragments mentioned, allowing the reader to assume that we meant ILOC instructions. In fact, it may be useful to include at least some of the debugging extensions within fragments as well. A good example of this is a temporary or one-shot, breakpoint, which is intended to only stop program execution once. Such a breakpoint is often used to implement a "run to here" function in a debugger.

Figure 3a shows one possible implementation of a temporary breakpoint fragment. In this example, we omit the argument to the *remove* instruction, and imply that it operates on the fragment that implements the breakpoint. Using a *remove* instruction, the temporary breakpoint removes itself just before suspending execution. Thus, no interaction with the front end is necessary to remove this breakpoint once it is reached, and in fact once this breakpoint has been set, the front end only needs to keep track of this breakpoint if will be modified before it is reached. Figure 3b shows an example of some intermediate code into which we will insert this fragment. To insert a fragment, we overwrite the instruction following the fragment's insertion point with a branch to the start of the fragment. At the end of the fragment, we add the overwritten instruction, and then a branch to the instructions following it. Figure 3c shows how this particular fragment would be inserted using this scheme. The original instructions are shown in dark grey, the fragment is in white, and the branch instructions added to do the insertion are light grey.

Examining the fragment in Figure 3, we notice that for the fragment to work correctly, the *remove* instruction must allow the flow of execution to return to the instructions in which the temporary breakpoint was inserted. That is, even though the fragment has been removed by the *remove* instruction, the instruction to be executed upon resumption is the one following the point where the fragment was inserted. In Figure 4, we see a possible implementation of this behavior. To remove this fragment, we simply replace the first branch instruction with the instructions that were originally there.

[2] We will defer discussion of others to the full paper.

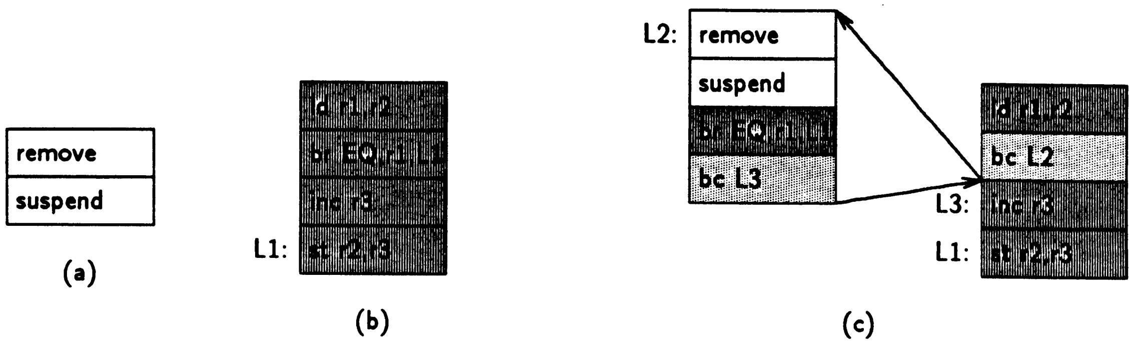

FIGURE 3: An implementation of a temporary breakpoint

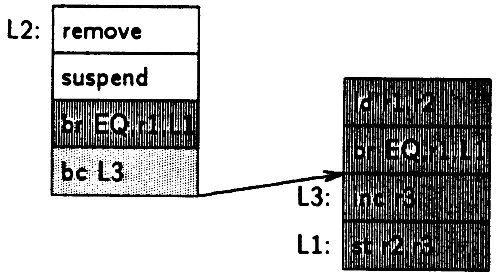

FIGURE 4: A temporary breakpoint after insertion and removal.

2.2 Implementation considerations

There are potential drawbacks to splitting the debugger into separate pieces. One problem is that the back end will not know the "intent" of the front end, and efficiency may suffer as a result. For example, tracing changes to a memory location in a naive way can be very time-consuming. Existing debuggers have used operating system features to protect the memory page containing the monitored location from writing. When the program attempts to write the page containing the monitored location, the debugger is notified. Thus the debugger only checks those memory references in the vicinity of this location, often making this monitoring operation much more efficient.

In our model, if the front end of the debugger specifies that a memory location should be monitored, but does it in a low level way, it may be impossible for the back end of the debugger to realize this, and so will not realize that write-protecting memory pages might be advantageous. The front end will not be able to write-protect memory pages, because our abstraction hides the implementation of the back end. Instead, we need to express this low level operation in an abstract way.

We can do this by including in our design the ability to execute a fragment in response to the modification of a specific memory location. The back end can implement this using whatever method is most efficient, using operating system support, or perhaps data flow information for the intermediate code. In the absence of support from the operating system, the back end could use the naive method of checking the value of the memory location after every execution of any instruction that could modify memory. A similar mechanism of triggering the execution of a fragment when an event occurs could be used to provide other debugging abstractions, such as floating point exception handling. This potentially allows all of the computation associated with the event to be performed by the back end.

Some of the proposed intermediate instructions might correctly be called meta-instructions, since they affect other instructions, including themselves, potentially. That is, they represent self-modifying code, and are easily implemented as such. This may prove to be a problem when debugging programs that are running on machines that were designed with the assumption that self-modifying code is rare or unnecessary. Such an assumption might be made in designing an instruction pipe-line, in order to improve its speed. This should not be a problem only when applying our particular design of debugger to this architecture, but rather a problem when designing any debugger for such a machine.

Because we can specify debugging requests in an intermediate language, there are several possible implementations of them. It is possible to reduce the number of modifications to the instruction stream by instead performing changes to data. The back end allocates a new memory cell to hold a flag, and adds a branch, dependent on flag's value, around the inserted fragment. When a *remove* is performed on this fragment, the action of removing can be performed by changing the value of the flag. If the fragment is re-inserted at the same place, this can be accomplished by changing the value of the flag back. Thus, it may be necessary to only modify the text of the program once for a given fragment inserted and removed some number of times at a given location. Finally, if a fragment can be created and inserted before debugging starts, then no modification of the instructions that make up the program is necessary to perform the insertions and removals of that fragment that might occur during debugging.

Another issue to consider is the implementation of the back end. We have presented it as an abstract machine that understands the intermediate language, perhaps implemented as an interpreter. However, if a fragment doesn't contain any *insert* or *remove* instructions, it is simple to translate this fragment into machine code, and insert it into the program, which could also be compiled into machine code. A prime example of such a fragment would be one that implements a tracepoint, which only needs to send output back to the front end whenever that point is passed during program execution. In existing debuggers, the typical implementation of this function consists of placing at the tracepoint a special instruction which will generate a trap or fault that will be fielded by the debugger. The debugger then gathers the information it needs, and causes the process being debugged to resume execution. Our scheme eliminates this expensive synchronization.

A hybrid scheme involving both a compiled program and fragments, as well as an intermediate language interpreter, could be very useful, and the interpreter would provide a simple way for executing the fragments that are more difficult to compile. Interpreting rather than executing a fragment might be more efficient if it is only used once. As an example, when the user asks to print or change the value of a variable, the front end would create a specially-tailored fragment to obtain the printing representation of the variable, or to change that variable's value, insert that fragment, execute it, remove it, and probably never use it again. Knowing how the fragment will be used is important, and it may be worthwhile to add special versions of the *create* instruction to indicate the expected use of a fragment.

A final example of a back end implementation is of a pipeline of back ends, each implementing the same abstract intermediate machine, each acting as the front end to the next back end in the sequence, and each performing some mapping between its input and output. This scheme might be useful if a sequence of code-improving transformations is applied to the intermediate language, such as might occur in an optimizing compiler. Each stage of the back end would handle the mappings necessary to compensate for a small collection of transformations. In this manner, it may be possible to debug a program that has been highly optimized. Note that it may be desirable to allow the *insert* instruction to fail, especially if we want to debug optimized code. This could occur if the insertion point specified by the front end no longer existed in the back end, or if at the point of insertion the values referenced in the inserted fragment had been eliminated by the back end[Hen82]. This change in semantics will impair the ability to debug the program, but will allow more code transformations to occur.

3 Comparison with Other Work

Adams and Muchnick advocate the use of *dbxtool* [AM86], a window-oriented front end for *dbx*, UNIX's source-level debugger. The front end sends user-level commands to the *dbx* back end in direct response to window events generated by the user. Their scheme places the entire burden of the source $\leftrightarrow$ machine mapping on the back end, making their back end source language dependent. Also, the front end cannot be extended to support new capabilities using the commands provided by *dbx*.

Johnson [Joh81] describes a debugging language called Dispel. Dispel is used as the user interface of a debugger, allowing the user to implement high-level debugging commands in terms of primitives and control structures provided as part of Dispel. Johnson mentions an alternate approach, in which the source language is extended to support debugging by adding the primitives of Dispel to it. A front end supporting Dispel could be implemented in a machine independent fashion by expressing Dispel operations in our intermediate language.

The general design of the programming environment described by Cordy and Graham [CG87] is similar to the model that we propose in Figure 2. They very briefly describe "immediate statements," executed from a statement cache, separately from the list of statements which make up the program. These immediate statements seem to be a rough analog of the intermediate language that we propose. Their immediate statements can be implemented in our system by inserting at the current execution point a fragment formed by a *create* of the list of statements to be immediately executed, with a *remove* appended to the end of that list. We cannot tell from their description if our model can be implemented in terms of their "immediate statements."

4 Future Plans

We intend to replace the existing R^n execution monitor [CCH+87], [CH87] with one that is designed along the lines we have proposed. We will center the system around ILOC [BCT87], the intermediate language used in a compiler for Russell and in the R^n Fortran compiler. Front ends for both of these languages will be considered, and as ILOC code generators new architectures are written, will also consider modifying them to support our extended version of ILOC. We should be able to use an existing low-level remote debugging communication package written for the current execution monitor to quickly support remote debugging on the new system. As the designers of ILOC extend the language to support parallel computation, we will investigate what extensions best suit the debugging of such code.

5 Conclusions

By designing a debugging system around a compiler intermediate language we have shown that it is possible to achieve a high degree of reusability of the debugger code. The same qualities that give the system a great deal of flexibility also make it quite easy to support remote debugging. The design also provides the potential for replacing some of the interpretation performed by a debugger with the execution of machine code. The cost to compiler writers for this design is no higher than traditional debugger symbol table generation; the only difference is that the the front half and the back half of the compiler each produces a symbol map.

References

[ACM87] ACM. *SIGPLAN '87 Conference on Interpreters and Interpretive Techniques*, volume 22, June 1987.

[AM86] Evan Adams and Steven S. Muchnick. Dbxtool: A window-based symbolic debugger for Sun workstations. *Software-Practice & Experience*, 16(7):653–669, July 1986.

[BCT87] Preston Briggs, Keith D. Cooper, and Linda Torczon. Iloc '87. R^n Programming Environment Newsletter 44, Rice University, September 1987.

[CCH+87] A. Carle, K. D. Cooper, R. T. Hood, K. Kennedy, L. M. Torczon, and S. K. Warren. A practical environment for scientific programming. *IEEE Computer*, November 1987.

[CG87] James R. Cordy and T.C. Nicholas Graham. Design of an interpretive environment for Turing. In *SIGPLAN '87 Conference on Interpreters and Interpretive Techniques* [ACM87], pages 199–204.

[CH87] Benjamin B. Chase and Robert T. Hood. Selective interpretation as a technique for debugging computationally intensive programs. In *SIGPLAN '87 Conference on Interpreters and Interpretive Techniques* [ACM87], pages 113–124.

[Com83] Computer Science Division, University of California, Berkeley, CA. *UNIX Programmer's Manual*, 1983.

[Hen82] John Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982.

[Joh81] Mark Scott Johnson. Dispel: A run-time debugging language. *Computer Languages*, 6:79–94, 1981.

[Zel84] Polle T. Zellweger. *Interactive Source-Level Debugging for Optimized Programs.* PhD thesis, University of California, Berkeley, 1984.