Intermediate Languages for Debuggers

Benjamin Chase

CRPC-TR90050
May, 1990

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

Abstract

Existing source-level debuggers are heavily dependent on both the source language of the program being debugged and the architecture of the machine on which the program runs. These dependencies place a burden on a programmer who wants to extend the debugger to work on a different language or machine. Such debuggers also typically offer poor support for sophisticated features such as debugging code that has been radically transformed during compilation.

We propose a new design for debuggers that uses a structure analogous to that of modular compilers. The debugger is separated into stages corresponding to the parser, optimizer, and code generator of a compiler. The various stages of the debugger communicate using an intermediate language which can be derived by augmenting the intermediate language of the compiler with primitives to support debugging.

We expect this design will promote the ability to debug optimized code, ease in porting the debugger to different architectures, and reusability of each portion of a debugger. This design is intended to increase the cooperation between implementers of compilers and debuggers, and to allow efficient implementations of sophisticated debugging operations, while supporting the debugging of optimized programs.

1 Introduction

Source-level debuggers are an integral part of the process of writing and maintaining programs. They permit the user to observe, control, and modify the dynamic behavior of a machine language program in terms of more familiar source symbols and constructs. A debugger maps between the source text of a program, the machine code derived from the source text, and the portion of the user's commands that are expressed with source constructs. This mapping is typically accomplished by having the debugger use a symbol table, produced by the compiler, that describes how source-level constructs map to the machine level. References to the source are retained as the program passes through the various translation stages of the compiler. As the compiler determines the final translation, the machine code generator emits these annotations for later use by the debugger. 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 significant drawbacks. The resulting debuggers depend heavily on both the language being debugged and the architecture of the machine on which the program is run. These dependencies complicate the process of modifying the debugger to accommodate a different source language or target machine. Tailoring such debuggers to follow the transformations of an optimizing compiler reduces the portability of the debugger. Finally, the traditional model does not accommodate program changes during debugging. These limitations are discussed in more detail below.

In an attempt to improve portability, traditional debugger designs abstract the machine dependent parts of the debugging process and isolate them in a collection of modules. To build a debugger for a new machine, one only needs to re-implement the machine-dependent modules. The same technique can be used to provide some measure of portability across source languages. When this approach to portability is used, the debugger implementer is essentially inventing an *ad hoc* intermediate level between the source language and the machine. We will see later that besides portability, other benefits can be realized if we formalize the intermediate level.

Attempting to provide portability across a family of compilers, the implementers of UNIX[1] designed a standard symbol table format that could be shared by the compilers and debuggers on the system. Unfortunately, although their design was feasible, there are ambiguities in the "standard" that result in subtle

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

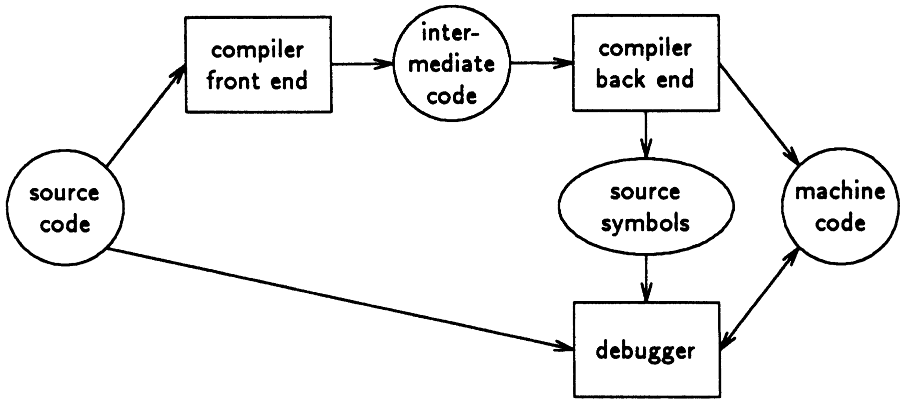

FIGURE 1: Traditional model of compiler-debugger cooperation

but significant differences in the symbol tables generated by different compilers. Also, while in theory this approach ensures portability across compilers, standardization introduces other problems, as detailed below.

Unfortunately, the benefits of standardized symbol tables begin to break down when the problem of optimizing compilers is considered. The notations for a family of compilers sharing a back end must be sufficiently expressive to accommodate the constructs of all of the different source languages recognized. As the compiler more radically changes a program, the debugger must become more tailored to that compiler and its target language, so that the compiler's code transformations may be decoded by the debugger. The communication between the compiler and debugger that describes the transformations, in the form of symbol table notations, must become more complicated. These symbol notations must be able to represent the composition of all of the various mappings induced by each part of the compilation process. This composition can change as different transformations are requested or inhibited by the programmer, and as parts of the compiler are modified.

Modifying the state of a running program is an important part of debugging. Unused or infrequently executed regions of code can be manually tested for bugs by forcing the flow of execution to enter those regions. This technique can be simpler than trying to create the program inputs that will exercise that region of code. Malfunctioning blocks of code can be simply skipped, and correct results can be substituted for the incorrect values that would have been generated by the skipped blocks. This ability allows the user to continue execution, perhaps finding other bugs, and is an important feature when at a point in the execution history that is difficult to reach.

If the debugger permits minor program changes during debugging, how does it accomplish this? The debugger can interpret the inserted source code, but this implementation makes the debugger more dependent on the source language. The debugger can compile the inserted source code and patch the result into the program text, but this makes the debugger more dependent on the machine on which the program is running. As the allowed forms of source-level modification become more elaborate, the debugger has to become more dependent on the source and machine language and the machine state, to translate these modifications of the source-level state into changes in the state of the running process.

2 Related Work

UNIX debuggers such as *dbx*[UNI83] and the GNU debugger *gdb*[Sta88] are typically implemented using separate processes and separate address spaces for the debugger and the process being debugged. The debugger makes calls to the UNIX operating system to perform all operations affecting the controlled process. The separation provided by the operating system allows a low level of intrusiveness, but can be extremely inefficient. Our experiments have shown that in unsophisticated implementations, the use of these system calls causes execution of the controlled process to proceed 20,000 times slower in some cases than the normal execution speed.

Like UNIX debuggers, the VMS debugger VAX DEBUG[DEC88] gets tables of information from the executable that are specifically included to allow symbolic debugging. VAX DEBUG achieves a multilingual capability more effectively than *dbx*, currently handling programs compiled from twelve different source languages. The debugger communicates with the programmer using the syntax, data typing, operators, expressions, and other constructs of the selected source language.

The VMS debugger uses methods similar to those of UNIX debuggers for controlling programs. However, this debugger resides in the same address space as the program being debugged, accessing the instructions and data directly, rather than through operating system calls. VAX DEBUG avoids some of the inefficiencies often found in UNIX debuggers by directly using hardware features specifically designed to support debugging. Thus, the debugger is quite machine dependent, assuming the existence of machine instructions with particular semantics. Also, since VAX DEBUG uses the same address space as the monitored process, the debugger is quite intrusive, potentially affecting the manifestation of the programming error being investigated.

Most production-quality source-level debuggers do not handle optimized code. The debuggers *gdb*, VAX DEBUG, PLITEST[IBM88], and VS/FORTRAN DEBUG[IBM87] are notable exceptions. However, these debuggers provide no way for the user to ignore the confusing effects of optimization.

Navigator[Zel84], a prototype debugger developed for the Cedar programming environment, was used to investigate the effects on debugging of two control flow optimizations, procedure inlining and repeated cross-jumping. These two transformations and their interactions alone produced a complicated mapping from source to object code, but their effects were successfully masked in most cases by the debugger. This hiding was accomplished by careful construction of the symbol tables, and inhibiting optimizations in a small number of cases.

Several research projects have extended source languages with constructs to support debugging. Combined with an incremental programming environment, this can constitute a powerful debugging tool. Although these debuggers are necessarily tied to a specific source language, portability between different architectures is still possible.

DICE [Fri84] is a distributed programming environment for Pascal. The DICE editor and compiler recognize Pascal extended with primitives that support interactive debugging, yielding an effective debugging system. The set of debugging primitives transmitted between the host and target machines is essentially the same set of operations supported by the UNIX *ptrace* system call. The DICE system assumes a straightforward mapping between source statements and machine code, prohibiting optimizations across source statement boundaries.

IPE[MMF81] is an incremental programming environment for GC, a dialect of the C programming language. IPE uses syntax trees as the intermediate program representation. The system allows debugging in the presence of some compiler transformations, although the method for dealing with optimizations during debugging is not described. Execution can continue after program edits if the changed procedures are not currently active. Some edits to active procedures are also allowed.

DOC [CMR88] supports browsing the state of an optimized program. The effects of a collection of transformations are handled by a monolithic mapping. Extending or modifying this collection may be problematic.

Hennessy[Hen82] examined the effects of local and global optimizations on the ability of a symbolic debugger to recover values of variables. Algorithms were developed for detecting variables whose values are inconsistent with those in the optimized model of the program. His analysis indicates that recovery of values blurred by local optimizations is moderately successful, while for some global optimizations this recovery is more problematic, and likely to be less fruitful.

INCROMINT[PS85] keeps an extensive optimization history to incrementally update optimized intermediate code in response to program edits. The information kept in the history is used after a program edit to reverse earlier optimizations that are invalidated and to perform new optimizations that have become valid. The history described is more elaborate than will be needed by a debugger for optimized code, but some of the problems that the history solves will occur in such a debugger.

Cargill described the object-oriented design of the debugger Pi [Car86]. This debugger uses the same methods for supporting user interaction and internal use by the debugger. The power of this framework is demonstrated by the simultaneous support of two different target architectures, achieved by having all of the objects for both back ends present in the debugger. The handling of compiler transformations is not mentioned in Cargill's description, although it is not precluded by the design. However, the structure of the debugger implies that most, if not all, compensation for such transformations would be contained in the *core* or *process* object, requiring the mappings to be composed in a monolithic fashion.

The dynamic state of an abstract machine is defined as part of the Smalltalk-80 [GR83] language. The description of this machine includes information that can be used to construct a fully capable debugger, and the language includes ways of accessing and manipulating that state. To satisfy the language definition, all of this information must be saved or be easily reconstructed when needed.

3 Proposed Solution

We propose that debuggers be built similar to compilers, adopting and extending the abstract machine model provided by a compiler's intermediate language. If compiler implementers are responsible for supporting a number of languages on several different architectures, then they are strongly motivated to use a common intermediate language, and separate each compiler into a front end and a back end. A compiler front end, consisting primarily of a scanner and parser, recognizes a given source language and generates the intermediate language translation. Each back end of a compiler recognizes this intermediate language and generates instructions for a particular machine. This separation reduces the amount of work involved in building compilers, because instead of writing 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 centering the debugging process on the intermediate language, it is also possible to split the debugging system into a small number of components. The debugger would consist of a front end that is capable of mapping the translation from source to intermediate language and a back end that is capable of mapping the translation from intermediate 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. Creating a debugger for any language-machine pair should be easily achieved by coupling the appropriate front end and back end.

The similarity between existing compilation systems and the proposed debugging system is intentional. Compilers and debuggers need to cooperate extensively to support source-level debugging. During the translation process, a compiler must produce the symbol table necessary to relate the source program to its machine language equivalent. In our approach, the information required by the front end of the debugger, mapping between the source and intermediate representations of the program, would be provided by the front end of the compiler. Similarly, the mapping required by the debugger back end would be generated by the compiler back end. Figure 2 depicts the relationships between the compiler and debugger in our model.

The interface between a UNIX debugger and operating system can be viewed as a version of our proposed design. The extended intermediate language is actually the machine language. The extensions added to the language to support debugging are the operations supported by the system calls used by the UNIX debuggers. The "back end" of the debugger is roughly the portion of the operating system that supports these calls, and the front end of the debugger we have designed is the UNIX debugger proper. The information needed by a symbolic debugger to map between source symbols and locations is found in a special part of the UNIX file containing a compiled program.

A more sophisticated compiler might have a number of middle phases that perform transformations on the intermediate language representation of the program before a final code generation phase translates the intermediate code to machine code. The debugging system for such a compiler would have corresponding interior phases, not pictured in Figure 2. Each of these *transformation* stages would compensate for the code-improving transformations performed by the corresponding middle phase of the compiler. The effects of a particular code transformation performed by a stage of the compiler would be handled by the corresponding

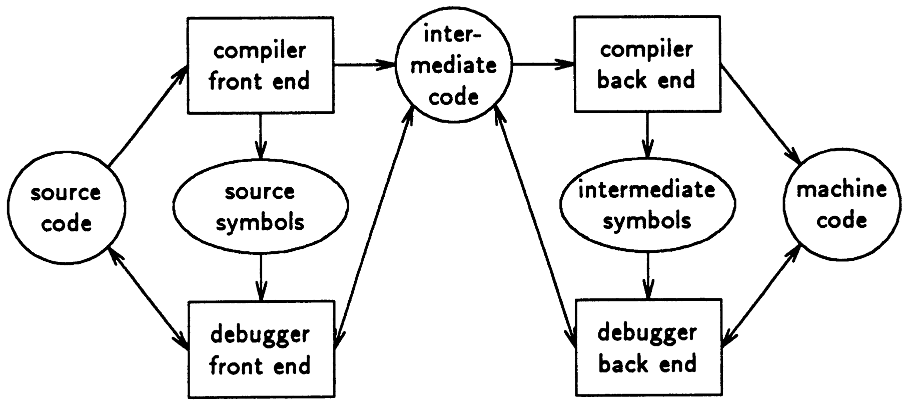

FIGURE 2: Proposed model of compiler-debugger cooperation

transformation stage of the debugger. Each of these debugger stages would be both source language and machine independent, and would only need to be changed if there was a change to the corresponding stage of the compiler.

4 Benefits

The separation of the debugger into stages should simplify the task of designing, implementing, and maintaining a debugger that collaborates with an optimizing compiler. As new optimizations are added to the compiler, our design should accommodate these transformations more easily than the traditional monolithic design. The partitioning of the debugger should encourage modularity, portability, and reuse of code. Yet the debugger can still be efficient, because the abstractions in the intermediate language can be implemented in an efficient way for a particular architecture, without reducing the portability of the rest of the debugger.

Changes to one part of the debugger can occur in isolation from work on the remainder of the debugger. Fast prototyping and implementation are more natural using this design, and new concepts can be tested without writing both a language-specific front end and a machine-specific back end. Such a partitioning of the debugger allows interesting research to be done on each of the independent parts of the debugger. If a team is implementing such a debugger, each part of the debugger can be written independently.

Portability is achieved by freezing the interface between the phases of the debugger, rather than by standardizing the symbol table format. This choice of interface permits the compiler and debugger to cooperate more effectively. In particular, the implementers of corresponding phases of the compiler and debugger can design the communication between those components that is most capable of representing the transformations performed.

Optimizing transformations made on the intermediate representation need only affect the transformation stages of the debugger. For a monolithic debugger, transformations such as loop unrolling, cross jumping, and procedure integration can produce a complicated mapping between source and machine code [Zel84], especially when combined with transformations performed by other stages of the compiler. By compensating for these transformations within a special transformation stage, we insulate the user interface and the machine dependent portions of a debugger from these problems.

Using this design, incremental changes to the program are relatively straightforward. Each stage of the debugger must support a mechanism for modifying its source program and its dynamic state. In all of the phases except for the last one, this modification of state is synonymous with the translation of the request to the next lower level. The stage that maps from intermediate code to machine code has two options: either translate the change into machine code and patch the executable, or arrange for the changed code to be interpreted. The other stages of the debugger are unaffected by which choice is made.

The primitives that are added to the intermediate language to support debugging provide a level of abstraction. These primitives may have a variety of implementations, differing in efficiency, ease of implementation, and intrusiveness[2]. This abstraction allows the implementation of primitives to use special debugging support offered by a particular operating system and hardware, while retaining portability.

[2] *Intrusiveness* is the degree to which the act of debugging can potentially affect the computation in the monitored process. An intrusive implementation of a primitive might require extra object code generated in the program being debugged.

Since debugging operations are built from primitives in the intermediate language, the set of operations supported at the user level can be enhanced without changing the back end of the debugger. For example, given a primitive to watch a memory location, we should be able to extend the capabilities of the debugger, allowing it to continuously update a view of the contents of an abstract data structure, or dynamically monitor the performance of a running program, all by changing only the front end of the debugger.

Although the separation of the debugger should make it easier to design, implement, and maintain a debugger for optimized code, it will not necessarily be able to unravel the effects of optimization any better than a monolithic debugger. On the contrary, the separation should result in a lower level of debugging support than that provided by Navigator[Zel84]. This separation prevents various stages from cooperating with the whole compiler to preserve critical data and program text. The level of support for debugging optimized code provided by the proposed design should be comparable to that described by Hennessy[Hen82].

5 Design of an Intermediate Language for Debugging

Perhaps the most critical single element in the design of a particular compiler is the choice of the intermediate machine model. That choice is equally important to a debugger designed using our framework. Since the language chosen must also support the compilation process, intermediate languages used by existing compilers (*e.g.*, syntax trees, postfix notation, three-address code) are the logical starting points for the search.

Although there is considerable latitude in the selection of an intermediate language to be adapted for debugging, the choice is not arbitrary. If we use an interpreter for the intermediate language, we will prefer intermediate forms that are easily interpreted, such as three-address code. If the debugger is part of an incremental compiling system or an integrated programming environment, it should be easy to perform standard editing operations on the intermediate language, so program edits during debugging can be supported. If a program written in this language is slightly modified, (*e.g.*, an insertion or deletion of a few instructions) then the code in other parts of the program should not need to change to maintain program correctness. An example is a relative branch instruction, which only needs to be changed if there is a change in the size of the section of code being branched over. In contrast, absolute branch instructions must be updated whenever changes to the program alter the absolute addresses used in these instructions. This is usually the case for all absolute branch instructions appearing after the point of change. Also, if instructions in the intermediate language implicitly set condition codes, then program changes will require careful analysis to determine when those values are actually used.

5.1 Extensions for Debugging

We want to express as many debugging functions as possible in terms of the compiler's intermediate language. While this language may be a good start, it will probably prove insufficient. Our intermediate language should be rich enough to represent most debugging operations, allowing them to be compiled. For example, we could add an instruction to the intermediate language specifically to support conditional breakpoints. However, a conditional breakpoint at the source level could be inserted by translating the evaluation of its boolean expression into intermediate code and following that by a "branch-on-false" around a "pause" instruction.

For the sake of economy, we will introduce new constructs only when the compiler's language lacks needed

operations, or when necessary to provide important debugging abstractions that may have many implementations of varying efficiency. The first set of instructions that we will add to a compiler's intermediate language is listed in Figure 3. However, this list is incomplete. For example, this list omits constructs to support certain classes of architectures and constructs that might be needed to improve performance.

In our framework, every interaction with the program, including instrumentation requests, incremental text changes, and advancing the program counter, is expressed with a sequence of instructions in the intermediate language. We view program execution during debugging as two instruction streams, the program itself and an "immediate" command stream. Interaction with the program occurs in the form of instructions appearing on the immediate command stream.

To support interactive debugging, we add an operation that will temporarily install instructions at some point in the program. The need to insert and remove groups of instructions is common for debugging operations that monitor and control execution. In most debuggers, this operation is accomplished by overwriting the monitored point with a special trap instruction and saving the overwritten instruction for later reinstallation when the tracepoint is removed. When the trap instruction is encountered, the generated exception is fielded by the debugger, and the actions for a tracepoint are performed.

A more desirable operation is actually installing the instructions that implement the tracepoint, and subsequently removing them when the tracepoint is removed. Thus, we add two operations, called *patch* and *unpatch*, to the intermediate language. When patching and unpatching in regions where the code has been nontrivially transformed, the same flavor of analysis performed by the compiler is performed for the inserted group of instructions with respect to the patch location to ensure that the patch is reasonable. In some cases, a *patch* operation may fail because the patch location cannot be reconstructed by the debugger, or because the patch refers to values that do not exist in the transformed version of the program. Instances where a debugger might fail to reconstruct information are described by Hennessy [Hen82].

We also need an operation to identify and name a sequence of instructions as belonging to a group that may be patched into a program to support some debugging task, and then later removed. We refer to these groups of instructions as *fragments* and call the operation for forming them *create*.

Many intermediate languages lack a means of accessing all of the memory locations that could be interesting during debugging. For example, it may be impossible to retrieve nonglobal values that are not in the current activation record. Examples include intermediate languages designed for compilers of C[KR78] and FORTRAN[Ame87]. In this case, we will need to modify the intermediate language so that all values in all active procedures can be accessed.

patch	Insert a group of intermediate language instructions into the program.
unpatch	Undo the effects of an earlier patch.
create	Group instructions together for later patching.
pause	Stop a running process, returning control to the debugger.
resume	Undo the effects of an earlier pause, allowing a stopped process to proceed.

FIGURE 3: Basic extensions to support debugging

The programmer may want to examine a single unchanging state of the process. The debugger's intermediate language must include a means of stopping a running program. After examining and perhaps modifying a single state of the process, the programmer may eventually want to proceed to a different process state. A means of resuming execution is also needed. We will call these two operations *pause* and *resume*.

In addition to starting and stopping a process, we will want other more unusual operations. These include the ability to destroy existing processes, start new processes, attach to existing processes to debug them, and detach processes after the user is done examining them using the debugger. Other examples of operations on processes are those that would support debugging in the presence of multiprocessing, using operations for naming and controlling multiple processes. Finally, a sophisticated debugger will want operations to save and restore the state of a process.

5.2 Abstractions for Efficiency

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. Consequently, efficiency may suffer. For example, tracing changes to a memory location in a naive way can be very time-consuming. Existing debuggers [DEC88] have used operating system features to improve the efficiency of this operation. One scheme is to write-protect the memory page containing the monitored location. When the program attempts to write the page containing the monitored location, an exception is generated, which is fielded by the debugger. Thus the debugger only checks memory references in the vicinity of the monitored 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 specifies this action in a low level way, it may be impossible for the back end of the debugger to recognize. The debugger will not realize that write-protecting the page of memory containing the monitored location 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.

To deal with this problem, we include ***assertion-driven breakpoints*** in our design. These are fragments executed when an assertion based upon the program's execution becomes true. An assertion is either a member of a small collection of predicates, or a simple boolean expression built from these predicates. These assertions may be transformed along with the intermediate code into other assertions and then passed to later stages of the debugger.

The back end can implement the boolean functions using whatever method is most efficient. For example, to implement a predicate that becomes true when a particular memory location is modified, the back end could use dataflow information about the intermediate code. Analysis of the program could reveal all of the locations where modification might occur. Fragments would be inserted at these locations. When a fragment is encountered, the predicate is considered to be true if that location were actually modified. In the absence of dataflow information or operating system support for monitoring memory modifications, the back end could resort to the naive method of checking the value of the memory location after every execution of any instruction that could modify memory. This frequent checking is done in existing debuggers, often with bad results when a separate address space is used for the debugger and the monitored process [UNI83].

Other predicates can be added to monitor many events of interest to the programmer, including numerical exceptions, UNIX signals, and memory access violations. This design potentially allows all of the debugging computation associated with exceptions to be performed by the back end. Coupled with the abstraction of insertion and deletion of fragments, this use of assertions on predicates should allow the back end to efficiently implement most debugging operations.

6 Example of Stage Interaction and Fragment Insertion

The following example shows how the transformation stages of a debugger act to compensate for the transformations performed by the compiler on the very simple loop shown in Figure 4. The transformations considered in this example are strength reduction and dead code elimination. These two transformations acting together can replace a loop variable with other induction variables. Specifically, in this example, we will show how the debugger handles a request to print the value of the loop variable i at the beginning of the loop body. At that location, the variable i has been eliminated by the compiler.

The intermediate code is shown in Figure 5 as it is being transformed by the compiler. Some blank rows have been inserted in the first and third columns of code to visually preserve the correspondences between versions of the loop as it passes through the compiler. The first column of code is a straightforward translation of the source loop.

The second column shows the code after it has been subjected to strength reduction. Note the added statements at lines 2 and 9, and the changed code at lines 3, 7, and 10. The compiler has introduced a new induction variable $iv1$, to take over most of the uses of the loop variable i.

The third column is the code after it has been subjected to both strength reduction and dead code elimination. Dead code elimination has removed all the occurrences of i, which were superfluous after $iv1$ had been added. The two occurrences of $n \times 4 + A$ will be cleaned up by a later phase of the compiler that handles constant and shared expressions.

To print the value of i in the loop body, the front end of the debugger will try to insert the fragment **print** i after line 5 of the intermediate code. The correspondence between the body of the loop in the source program and line 5 of the intermediate code is determined by the front end of the debugger, using information recorded by the front end of the compiler. That stage of the mapping is not discussed here, and is assumed to exist. In the Original Code column of Figure 6, the fragment from the front end is shown as it would appear after insertion.

The stage of the debugger handling strength reduction can perform this insertion request by translating the fragment into its equivalent form, mapping the fragment insertion point before strength reduction to

```
do $i = 0, n$
  $A[i] = \dots$
endo
```

FIGURE 4: A simple loop

		Original Code	Strength Reduced	Dead Code Eliminated
1		$i \leftarrow 0$	$i \leftarrow 0$	
2			$ivl \leftarrow A$	$ivl \leftarrow A$
3		$(i \geq n)$?	$(ivl \geq n \times 4 + A)$?	$(ivl \geq n \times 4 + A)$?
4		goto *skip*	goto *skip*	goto *skip*
5	*body* :			
6		$t \leftarrow 4 \times i$	$t \leftarrow 4 \times i$	
7		$*[A + t] \leftarrow \ldots$	$*[ivl] \leftarrow \ldots$	$*[ivl] \leftarrow \ldots$
8		$i \leftarrow i + 1$	$i \leftarrow i + 1$	
9			$ivl \leftarrow ivl + 4$	$ivl \leftarrow ivl + 4$
10		$(i < n)$?	$(ivl < n \times 4 + A)$?	$(ivl < n \times 4 + A)$?
11		goto *body*	goto *body*	goto *body*
12	*skip* :			

FIGURE 5: Intermediate code at three stages of compilation

the equivalent insertion point after that transformation. (Because of the way I have numbered statements and left blank lines in this example, this location mapping is trivial.) Also, this stage of the debugger will translate any data locations named in the fragment, as needed. It appears that no translation of i is needed, because that variable exists before and after translation. The fragment, translated to compensate for strength reduction, is shown in the Strength Reduced column of Figure 6 as it would appear after insertion.

To actually accomplish the insertion request, the strength reduction stage of the debugger in turn asks the next stage of the debugger, the stage that compensates for dead code elimination, to insert the fragment **print** i. However, the compiler eliminated i during dead code elimination. This makes it impossible for the dead code elimination phase of the debugger to translate the fragment being inserted into something meaningful for the next stage of the debugger. Thus, the dead code elimination phase of the debugger reports to the strength reduction phase that the fragment **print** i cannot be inserted. This failure is depicted in the Dead Code Eliminated column of Figure 6.

A particular phase of the debugger attempting to insert a fragment could simply always report failure to its preceding phase, when faced with a report of failure from a subsequent phase. This is a valid action, but better courses of action may be available to that phase of the debugger. Better support of optimized code will result if phases try different translations of an insert fragment operation that has failed. The insert fragment operation requested by the preceding stage succeeds if any of the translations of this operation can be performed by the subsequent stage of the debugger. Thus, for each stage of the debugger, the success of a fragment insertion is the disjunction of the successes of all the translations of that operation.

For instance, when faced with the failure depicted in Figure 6, the strength reduction phase of the debugger could try a different translation for the original fragment. The strength reduction phase of the compiler created synonyms of the loop variable, new induction variables that are affine functions of the original loop variable. Using information recorded by the compiler, the strength reduction phase of the debugger can react to the failure of its first attempt at insertion by restating the fragment in terms of the

		Original Code	Strength Reduced	Dead Code Eliminated
1		$i \leftarrow 0$	$i \leftarrow 0$	
2			$ivl \leftarrow A$	$ivl \leftarrow A$
3		$(i \geq n)$?	$(ivl \geq n \times 4 + A)$?	$(ivl \geq n \times 4 + A)$?
4		goto *skip*	goto *skip*	goto *skip*
5	*body* :			
		print i	**print** i	**print** ???
6		$t \leftarrow 4 \times i$	$t \leftarrow 4 \times i$	
7		$*[A + t] \leftarrow \ldots$	$*[ivl] \leftarrow \ldots$	$*[ivl] \leftarrow \ldots$
8		$i \leftarrow i + 1$	$i \leftarrow i + 1$	
9			$ivl \leftarrow ivl + 4$	$ivl \leftarrow ivl + 4$
10		$(i < n)$?	$(ivl < n \times 4 + A)$?	$(ivl < n \times 4 + A)$?
11		goto *body*	goto *body*	goto *body*
12	*skip* :			

FIGURE 6: Intermediate code, showing a fragment insertion

new induction variable. In this case the fragment **print** i is translated into **print** $(ivl - A)/4$ as it passes through the strength reduction phase of the debugger. Figure 7 shows the successful fragment insertion.

The success of a particular fragment insertion performed by a phase of the debugger can also depend on the conjunction of the successes of a collection of fragment insertions performed by the subsequent debugger phase. A translation of that particular fragment may involve several fragment insertions, all of which must be successful for the translation as a whole to be successful.

For example, if a fragment is inserted within a region of code that is really two merged flow paths, path-determiner breakpoints must also be inserted before the start of the merged region. These breakpoints determine which flow path was actually taken, the path to the region of code that actually contained the insertion point, or the matching region of code that was merged to it. If either the true breakpoint or its path determiner breakpoints cannot be inserted, then the whole insert operation fails.

7 Implementation of the Debugger

In general, an object-oriented framework for the debugger seems helpful, such as Cargill's representation of processes and frames as objects. Portions of Cargill's design appear to be adaptable to the needs of our design. Some of the messages he describes would require extra arguments to account for the increased complexity caused by optimization. By changing the class hierarchy, the mappings can be decomposed into the individual mappings performed by the compiler.

A debugger built in accordance with our design would probably be part of an integrated programming environment. The large amount of information shared between the compiler and the debugger makes a programming environment a desirable setting. Some of the code implementing the compiler could be adapted for use by both the compiler and debugger. Also, with minor modifications, language tools in current

		Original Code	Strength Reduced	Dead Code Eliminated
1		$i \leftarrow 0$	$i \leftarrow 0$	
2			$iv1 \leftarrow A$	$iv1 \leftarrow A$
3		$(i \geq n)$?	$(iv1 \geq n \times 4 + A)$?	$(iv1 \geq n \times 4 + A)$?
4		goto *skip*	goto *skip*	goto *skip*
5	*body* :			
		print i	**print** $(iv1 - A)/4$	**print** $(iv1 - A)/4$
6		$t \leftarrow 4 \times i$	$t \leftarrow 4 \times i$	
7		$*[A + t] \leftarrow \ldots$	$*[iv1] \leftarrow \ldots$	$*[iv1] \leftarrow \ldots$
8		$i \leftarrow i + 1$	$i \leftarrow i + 1$	
9			$iv1 \leftarrow iv1 + 4$	$iv1 \leftarrow iv1 + 4$
10		$(i < n)$?	$(iv1 < n \times 4 + A)$?	$(iv1 < n \times 4 + A)$?
11		goto *body*	goto *body*	goto *body*
12	*skip* :			

FIGURE 7: Intermediate code, showing a fragment insertion

programming environments, such as source code browsers and editors, could be used by the debugger.

The debugging system for a compiler that performs optimizing transformations on its intermediate representation would comprise a single front end that maps from source code to intermediate code, a collection of transformation stages that map from intermediate to intermediate, and a final back end that maps from intermediate code to machine code. We discuss these phases in more detail below.

Front End

A particular front end for a debugger should be tailored to debug programs written in a specific source language. Ideally, the user interface allows the programmer to express as many debugging functions as feasible in the particular source language. The commands are parsed and compiled into the debugger's intermediate language, perhaps using parts of the compiler developed for the source language. The commands are then passed to subsequent stages of the debugger. The commands generated by the front end are either accepted or rejected by the back end, depending on whether or not the operation can be performed.

Transformation Stages

Each of the transformation stages of the debugger unravels the effects of a particular optimization. Requests from the preceding stage to create, insert, or remove fragments are translated into slightly different requests and sent to subsequent stages. A request to insert a single fragment can result in the insertion of many fragments in subsequent stages of the debugger. For instance, to place a breakpoint in a cross-jumped region, the original breakpoint must be supplemented with path-determiner breakpoints [Zel84]. References to memory in the original fragment are renamed to reflect the new locations of the desired values.

Back End

The back end maps the abstract machine presented between layers of the debugger onto the target machine. It can have a wide range of implementations, varying in efficiency, ease of implementation, and portability.

For fast prototyping, a simple interpreter for the debugger's intermediate language might be selected as a back end. This choice would yield a quick implementation, allowing the other parts of the debugger to be developed without delay. However, the resulting back end might be slow, adversely affecting the performance of the whole debugger.

If efficiency of the back end is a concern, the debugging operations sent to the back end, represented in an intermediate language for debugging, could be incrementally compiled, thereby reducing the amount of interpretation performed by the debugger. These compiled fragments could then be patched into the compiled code, reducing handshaking and context switches when performing the debugging operations implemented by these fragments.

In between these two extremes, is the possibility of a hybrid scheme that runs the unchanged code in its compiled form and interprets any changes. As an example, under UNIX, the back end could consist of two separate processes, the back end and the monitored process, with the monitored process controlled via calls to the operating system. We would expect that the efficiency of a debugger using this hybrid back end would be no better than a straightforward UNIX debugger having a complexity similar to our back end. Since the UNIX system calls employed are the bottleneck in these debuggers, efficiency would only be improved if the number of these calls were reduced, perhaps through more thorough analysis of the program.

7.1 Interaction with Program Edits

A sophisticated programming environment should support incremental compilation. In response to program edits, the system incorporates corresponding "edits" into the compiled version of the program. If the program is being debugged when an edit occurs, it may be desirable to edit the state of the monitored process, without restarting the program, if possible.

Although edits of the process text and state could be implemented using the *patch* and *unpatch* primitives, these debugging operations are inappropriate for supporting process edits. Edits of the process text differ from the patches applied to support debugging. Debugging patches are transitory, being applied to return control to the user or to monitor some condition. Edits of the process text are more permanent, reflecting a decision by the programmer to change some part of the program. To emphasize this difference, we introduce the editing operations *insert* and *delete*, which insert and delete a sequence of instructions at a point in the program.

It is possible that a *fragment* may have been *patched* within a region that is slated for deletion as part of an editing change. In this case, the correct action is for the debugger stage that patched the fragment to unpatch it before the code deletion occurs, and then later attempt to repatch it after the deletion has been completed. Similarly, a code insertion may alter the flow of information reaching a patched fragment. If the fragment was constructed using assumptions about the values that reach the patched fragment, the correct action is also to unpatch the fragment before inserting the new instructions, and then attempt to repatch it afterwards.

7.2 Impure Code

Some of the proposed intermediate instructions can be called meta-instructions, since they can alter the text of the program. That is, they represent, and are easily implemented as, self-modifying code. This implementation may be a problem when debugging programs on machines that were designed with the assumption that self-modifying code is rare or unnecessary. For example, such an assumption might be made in designing an instruction pipeline, in order to improve its speed. However, this restriction will be a problem for all interactive debuggers written for such architectures. The abstraction present in our design actually gives greater freedom to easily change the machine dependent part to suit the particular hardware.

Since we can specify debugging requests in an intermediate language, there are several possible implementations of requests. We can convert code modifications into data modifications, reducing the number of modifications to the instruction stream by modifying data instead. To accomplish this conversion, the back end allocates a new memory cell to hold a flag, and adds a branch, dependent on the flag's value, around the patched fragment. When an *unpatch* is performed on this fragment, the action of removing the fragment can be performed by changing the value of the flag.

7.3 Debugging at Lower Levels

A programmer using any source-level debugger will in some instances need to use a lower level of abstraction. For conventional debuggers, this lower level is the generated machine code. The desire to change to this level can occur when the user is unsure of the apparent behavior of generated code. This uncertainty can arise from bugs caused by errors in a compiler, defects in the hardware, the user's disbelief of the events actually happening, and a host of other causes.

In our model, each of the interfaces between stages of the debugger represents a level of abstraction that may be interesting to a programmer. By examining the program at various levels, an experienced user can understand the effects of optimization. When the user inspects a program object at one viewing level, the corresponding object or objects at a different level can be emphasized. By displaying these correspondences between objects from different levels of the program, the state of the transformed versions can be related to each other, and back to the original source program. For example, if the user selects a program location in the source, all the possible corresponding program locations in an intermediate language version of the program could be highlighted.

Also, optimizations that hinder debugging can sometimes be avoided by dropping to a lower level of abstraction. At lower levels of abstraction, the insertion of a fragment is more likely to succeed, because the translation of that insertion into machine code is more straightforward. For example, setting a breakpoint in an inlined procedure would entail patching fragments into all the copies of the procedure body replacing the call sites. The patch operation for one of these copies might fail, because of optimizations occurring only in that copy. By dropping to a level where the user can set breakpoints for individual call sites of a procedure, the one troublesome patch can be avoided.

8 Research

Previous Experience

As part of Rice University's $\boldsymbol{R}^n$ project [CCH+87] to build a programming environment for scientific software, we designed and built EXMON, a debugger for large, computationally intensive programs[CH87]. In the course of that effort, we encountered several problems in extending our design to provide more functionality. We chose a hybrid design for EXMON, allowing the monitored process to be a mixture of compiled and interpreted code. This mixture allowed trusted parts of the program to be executed quickly, but with minimal debugging support, and suspect portions of code to be interpreted, with greater debugging support. For our representation of interpreted subparts of the program, we chose the abstract syntax tree, which was the source representation used by $\boldsymbol{R}^n$.

Unfortunately, this representation proved to be too close to the source language, and too closely determined by the needs of the source editor. The addition of new source language dialects, the evolution of the source editor, and changes to other parts of the programming environment repeatedly changed the structure of these trees, and these changes in turn required corrections to the interpreter and other parts of the debugger.

Our experience with EXMON provided some of the impetus and insight for this design. The research we propose is an attempt to overcome some of the problems with conventional debuggers that we encountered. We hope to demonstrate that this new design for debuggers is feasible, effective for debugging optimized code, portable, and efficient.

Available Tools

As part of this research, an extended version of a compiler's intermediate language has already been defined along the guidelines in this proposal. We have chosen to extend ILOC, a fairly simple three-address code in use at Rice University. ILOC was designed as an intermediate language for optimizing compilers, and is reasonably suitable as a foundation for our debugger's intermediate representation. A compiler for ILOC exists, and already has several transformation phases, including value numbering, dead code elimination[Ken81], strength reduction[CK77], and partial redundancy elimination[MR79][DS88].

A fully interpretive back end for this extended version of ILOC exists. We anticipate changes to the intermediate language, so the efficiency of the interpreter is not yet a great concern. As the language becomes more stable, we will investigate efficiency issues in the back end. A hybrid back end is currently being designed by Mike Jones and Bob Hood. The hybrid back end will support a compiled program text patched with interpreted fragments. Using this hybrid back end, the efficiency of common debugging operations will be studied. A modified version of the $\boldsymbol{R}^n$ debugger will be used for the initial version of the user interface and front end of the debugger.

Research Plan

We will design parts of a debugger–compiler pair, trying to preserve opportunities for optimization by the compiler, rather than constrain optimizations to preserve debugging support. Thus, the emphasis of this research is on debugging optimized code, rather than optimizing debuggable code. Debugger phases

compensating for the transformations dead code elimination, strength reduction, value numbering, and partial redundancy elimination will be designed.

- Dead code elimination is an important inclusion because its effects are difficult to reverse. Hennessy in particular noted global dead code elimination as an optimization that impaired debugging support[Hen82].
- Strength reduction was included because the multiple translations easily obtainable from this transformation are an important aspect of the fragment translation. Strength reduction and dead code elimination together show promise as transformations whose mappings are separable, but which together produce a non-trivial transformation, in this case loop induction variable elimination. Also, we feel that the intermediate language used in the compiler may be an interesting medium for the private communication of transformation information between the strength reduction phases of the compiler and debugger.
- Value numbering was added to give the collection a realistic amount of complexity. Value numbering was chosen over similar data optimizations because it is currently used in the $\boldsymbol{R}^n$ compiler developed at Rice. The effects of value numbering on the debuggability of the generated code should be similar to that of any algorithm for folding constants and eliminating common subexpressions.
- Partial redundancy elimination was added to give the collection of transformations a realistic amount of complexity. This particular optimization was chosen because it is used in the $\boldsymbol{R}^n$ compiler used at Rice.

The merits and failings of these debugger phases, and their interactions with each other, will be studied. Stages corresponding to strength reduction and dead code elimination will be implemented, and their detailed interaction and efficiency will be studied. Debugger transformation stages for register allocation, cross jumping, loop unrolling, and procedure integration also show promise in compensating for these transformations.

A debugger phase that compensates for register allocation probably will not be studied directly as part of this research. A realistic treatment for register allocation would also involve all the other transformations found in the code generation phase of a compiler, such as instruction selection, instruction scheduling, and peephole optimizations. A debugger phase for the resulting mapping would be hard to design, implement, or study. If implemented, the performance of the debugger phase in terms of a particular back end transformation would be hard to evaluate, because the effects from all the different back end optimizations would be hard to separate and measure.

We conjecture that a straightforward composition of many simple mappings can be used to compensate for many compiler transformations. For example, simple cross-jumping appears to only produce simple mappings, while repeated cross-jumping produces more complicated mappings[Zel84]. The monolithic mapping developed for Navigator represents the composition of the individual simple merges that contributed to the repeatedly merged regions. We will compare the capabilities of a single monolithic mapping of the repeated cross-jumpings with those of a mapping produced by conceptually composing simpler mappings, each of which represents only simple merges without repeatedly merged regions.

The knowledge of when to inhibit a transformation may be difficult or inefficient to garner in a compiler that leaves all the mappings uncomposed. As an example of this effect, the composition of simple mappings that we propose may unfortunately yield a poorer handling of cross-jumping than that provided by a monolithic mapping. The level of debugging support should be similar to that described by Hennessy[Hen82], and

will be compared with it.

We expect that changes to the collection of transformations will be simpler if the mappings of the transformations performed by the compiler are left uncomposed. We will investigate one aspect of this by studying the effects of interchanging and repeating the various transformation phases of the compiler and debugger. It may be the case that each stage of the debugger will contain hidden assumptions about the nature of the stages following them, and that disturbing the ordering of the stages seriously impairs debugging support. The character and seriousness of the degradation in debugger support resulting from these changes will be studied.

For a debugger using the monolithic mapping, the interaction between a new transformation and each existing transformation must be considered. We feel that by leaving the mappings uncomposed, the interaction between existing transformations and an additional one is reduced. We will investigate this theory by examining the effects of adding another control flow optimizations, such as loop peeling[Lov77] to the two optimizations researched by Zellweger, and noting the impact on the overall design.

The ability to present different levels of abstraction, described earlier in section 7.3, will be studied. The adoption of this mechanism as a technique for following and understanding source to source transformations will also be attempted. Among the important problems remaining in developing this tool are mapping the cursor from one level to the next, displaying this information to the user in an understandable manner, filtering out extraneous intermediate language statements added during intermediate steps of the compiler's optimization process, and defining a small but useful set of debugging commands at these lower levels. We will study the benefits and problems resulting from allowing access to intermediate language representations of the program.

9 Conclusions

We have described a new design for debuggers that uses a structure analogous to that of modular compilers. The debugger is separated into stages corresponding to the parser, optimizer, and code generator of the compiler. Each of these stages of the debugger is responsible for unraveling the transformations performed by the corresponding stage of the compiler. The stages of the debugger communicate using an intermediate language derived by extending the compiler's intermediate language with primitives to support debugging.

This approach is designed to improve the cooperation between the compiler and the debugger, especially when debugging optimized code. Our framework should allow many debugger actions to be translated into intermediate code and perhaps machine code, resulting in more efficient execution of these actions. The scheme described should promote portability and modularity of the debugger. When the other language tools are modified to support a new source language, architecture, or transformation, our design should result in a high reuse of the debugger code.

References

[Ame87] American National Standard Institute. *X3J3 Draft FORTRAN Standard*, 1987.

[Car86] T. A. Cargill. Pi: A case study in object-oriented programming. In Norman Meyrowitz, editor, *OOPSLA '86 Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 350–360, 1986. Also appearing as SIGPLAN Notices Vol 21, Number 11.

[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.

[CH87] B. Chase and R. Hood. Selective interpretation as a technique for debugging computationally intensive programs. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques. June, 1987*, pages 113–124, June 1987.

[CK77] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11), November 1977.

[CMR88] Deborah S. Coutant, Sue Meloy, and Michelle Ruscetta. DOC: A practical approach to source-level debugging of globally optimized code. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 125–134, June 1988.

[DEC88] Digital Equipment Corporation, Maynard, Massachusetts. *VMS Debugger Manual, Version 5.0*, April 1988.

[DS88] Karl-Heinz Dreschler and Manfred P. Stadel. A solution to a problem with Morel and Renvoise's 'Global optimization by suppression of partial redundancies'. *ACM Transactions on Programming Languages and Systems*, 10(4), October 1988.

[Fri84] P. Fritzson. *Towards a Distributed Programming Environment Based on Incremental Compilation.* PhD thesis, University of Linkoping, 1984.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

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

[IBM87] International Business Machines Corporation. *VS FORTRAN Interactive Debug Guide and Reference*, 2nd edition, June 1987. Order number: SC26-4223-1.

[IBM88] International Business Machines Corporation. *OS PL/1 Programming: Using PLITEST*, 2nd edition, October 1988. Order number: SC26-4310-1.

[Ken81] Ken Kennedy. Program flow analysis: Theory and applications. In Steven S. Muchnick and Neil D. Jones, editors, *A Survey of Data Flow Analysis Techniques.* Prentice-Hall, 1981.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Lov77] David B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 10(3):121–145, March 1977.

[MMF81] R. Medina-Mora and P. H. Feiler. An incremental programming environment. *IEEE Transactions on Software Engineering*, SE-7(5):472–482, September 1981.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2), February 1979.

[PS85] L. Pollock and M. Soffa. Incromint – an incremental optimizer for machine independent transformations. In *Softfair II – A Second Conference on Software Development Tools, Techniques, and Alternatives*, pages 162–171. IEEE Computer Society Press, 1985.

[Sta88] Richard M. Stallman. *GDB, The GNU Debugger.* Free Software Foundation, Inc., 2nd edition, February 1988.

[UNI83] *UNIX Programmer's Manual, 4.2 Berkeley System Distribution.* Computer Science Division, University of California, Berkeley, CA, August 1983.

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