Debugging Optimized Code via Tailoring

Lori Pollock

University of Delaware
Newark, DE 19716
pollock@udel.edu
302 831-1953

Mary Bivens

Allegheny College
Meadville, PA 16335
mbivens@alleg.edu
814 332-2880

Mary Lou Soffa

University of Pittsburgh
Pittsburgh, PA 15260
soffa@pitt.edu
412 624-8425

ABSTRACT

Although programmers have come to expect compilers that generate high quality code through ambitious optimization, a lack of good interactive source level debuggers for optimized programs still remains. Techniques to debug optimized programs place restrictions on either the application of optimizations or the types of debugging capabilities provided. Even with the restrictions, the current debugging techniques often have to report the existence of noncurrent values. This paper presents a new approach to the problem of debugging optimized code. By employing techniques from incremental optimization analyses, optimizations are tailored to the type and context of debugging commands. Therefore, there are no a priori restrictions placed on either debugging commands or optimizations. In the presence of both ambitious, machine independent optimization and smart register allocation, we enable standard debugging commands such as breakpoint insertion, single stepping, and program state display and more advanced commands including the modification of variable values and program code with continued program execution. In response to a request specified during debugging, an optimization profile is used to determine if the expected response can be provided by the debugger. When an optimization inhibits transparently servicing the request, the profile is then used to recover the desired information or to tailor the code such that the debugging request can be honored transparently in a subsequent execution. Unnecessary limitations on optimization are avoided by basing the optimization on the conflicts with the user's current debugging requests rather than the entire repertoire of debugging commands.

1. Introduction

Modern compilers perform ambitious optimizing transformations to produce target code that is time and space efficient. While the meaning of the source program is necessarily preserved by an optimizing compiler, the execution behavior of the resulting optimized program is typically not what is **expected** from an inspection of the source program. Debugging techniques typically either limit the types of debugging commands that can be used when debugging optimized programs or place restrictions on the optimizations. Even with these restrictions, the response of a debugger in some cases is to report that a value is not available, and thus the user cannot adequately debug the optimized code. We present an approach in this paper that provides full debugging and optimization capabilities by using incremental analyses and tailoring code to eliminate only those particular optimizations in the context of the debugging requests that prevent actual debugging responses from being serviced in an expected manner.

A code optimizer may reorder, replace, or remove expression evaluations and stores while preserving the semantics of the source program. These transformations can cause variables displayed at different points during execution to differ from the values expected by the user at these points. These optimizations also inhibit the user's ability to insert breakpoints and single step through the executing code based on source statement steps. Modification of variable values or source program statements during execution can cause incorrect final results if the programmer's changes violate the correctness-preserving assumptions on which optimizations were performed. This is particularly troublesome when the optimizations have already affected the code at the current point of execution and are now invalidated by the programmer's action. Unraveling of optimizations is not sufficient in this situation without also reversing some of the execution.

Symbolic debugging can also be complicated by actions performed during target code generation. The most effective optimization performed during target code generation is register promotion where the goal is to keep values of variables in registers over the region of the program in which the value is live (i.e., may be used later in the program). When the programmer asks to display the current value of a variable at a breakpoint during execution, register promotion could cause the current value of the variable to be in a register and not in memory.

Ideally, we would like to have no restrictions placed on the debugging capabilities and the optimizations, and at the same time, respond to all debugging requests as expected, where expected responses during execution of an optimized version of a program are the same as the responses for an unoptimized version of the same program. However, this is not possible, given the interferences between the optimizations and debugging.

In this paper, we present a scheme to debug optimized programs using analyses developed for incremental optimization. The scheme involves tailoring the optimizations during debugging to the user's current debugging requests, taking into account both the request and its context within the program. As the user's debugging requests change throughout a debugging session, the optimizations that are allowed in the code also change, eliminating only those that interfere with responses in the expected manner to the user's debugging requests. Our debugging scheme handles both machine independent optimizations such as constant propagation and folding, common subexpression elimination, dead code elimination, and loop invariant code motion and target code optimizations including smart register allocation and code selection. In addition to the standard debugging commands such as breakpoint insertion, single stepping, and program state display, we support powerful debugging functions such as changing the value of a variable and inserting and deleting code. The scheme is applicable to control breaks when requested through explicit debugging commands submitted by users as well as control breaks caused by a run-time error or exception.

The techniques to implement this scheme rely on analyses that are used to incrementally incorporate source code changes into compiled optimized programs while exploiting the same opportunities for optimizations as an exhaustive approach [3, 18]. The incremental optimizing techniques include incremental data flow analysis, an incremental intermediate code optimizer and an incremental target code generator. Both the information saved by these systems and the incremental analyses that they perform have been exploited in our approach to symbolic debugging of optimized code.

In response to debugging requests submitted during execution, the information saved by incremental optimizing analyses is used to determine if the expected response can be ensured and if not, this information is used to tailor the code to ensure the expected response. In some cases, the code is tailored for the current execution by removing the interfering optimization, servicing the request and continuing with the current execution. When this is not possible, the user is notified of the situation and told that a new execution of the program will produce the expected response to the same debugging request. In this case, the code is tailored by removing the interfering optimization from the next execution. When the programmer knows the debugging requests before execution, the debugger accepts commands prior to an execution and tailors the optimized code using the incremental optimizing techniques to ensure that these *planned* debugging requests are serviced as expected during execution. The code tailoring approach allows the programmer to debug code as close to the final, production code as possible without inhibiting debugging capability. In our system, the user would be in control of whether or not to incur the overhead to service a request that is prevented by optimizations. Without the incremental optimization and code generation techniques, the tailoring of the optimized code to the current set of debugging commands would be impractical.

This paper extends previous work by the authors in two important ways [19]. First, the prior work only considered the use of incremental techniques to remove optimizations prior to a debugging session by having the user submit a set of planned debugging requests. Those optimizations that only interfered with the planned requests were eliminated from the optimized code. If an optimization interfered with an unplanned debugging request, then the user was informed of a noncurrent variable. Second, the previous work did not account for the effects of register allocation, and thus could report an incorrect value for a variable with a noncurrent value in memory.

The paper begins in Section 2 by surveying previous research on the problem of debugging optimized code. Sections 3 and 4 describe incremental techniques and the program database exploited in our approach. Our "tailored" model of debugging is described in Section 5 where an example is also given. Finally, conclusions are discussed in Section 6.

2. Previous Approaches to Debugging Optimized Code

Based on the premise that the programmer should be debugging the optimized code and be presented with the picture of what is actually happening at execution time, Brooks, Hansen, and Simmons [4] developed a visual debugger that includes highlighting and annotations to the source and assembly code. However, the programmer must use this visual feedback to understand how optimization has changed the code and to deduce the expected program behavior.

Another approach is to turn off optimization or use an interpreter during debugging and exploit optimizing compilation only after debugging is finished. However, additional bugs are inevitable, making it difficult to predict when debugging is "finished." Maintenance changes may also create a need for a symbolic debugger. In addition, the programmer must suffer the longer execution times of unoptimized code, or even worse, interpretation, during program development in order to obtain good debugging capabilities. Likewise, some large programs may not be able to execute correctly without optimization due to memory and time constraints. By accommodating the coexistence of interpreted and compiled code, Hood and Chase have developed a debugger that exploits the sophisticated debugging capabilities of an interpretive debugger in the module under development while providing fast execution through other modules in the program by executing their corresponding compiled, optimized code [12]. Although this approach should certainly improve the execution speed of large programs composed of relatively small modules, it will still suffer from slow execution when individual modules are large. More importantly, both an interpreter and compiler must be available and consistency maintained between them. Many times, bugs in an optimized program can not be reproduced in the unoptimized code or interpretation and vice versa due to timing differences.

Other approaches to symbolic debugging of optimized code have been described in the literature. Several debugging systems provide correct debugging information in the presence of optimization, but limit the application of optimization to small regions of the program thereby avoiding the contention between the transformations and correct symbolic debugging [9, 10, 22]. Other systems attempt to support more significant optimization by collecting information during optimizing compilation and using this information during debugging [8, 11, 21, 24]. Hennessy [11] augments the structures built during optimization to maintain information about intermediate code optimizations that were performed during compilation. His algorithms can detect non-current values of variables, and if possible, the correct values are recovered; otherwise the user is informed that the value is not current. Navigator, developed by Zellweger [24], is a source level debugger that gives the user expected behavior for setting breakpoints, determining the current execution location and reporting procedure traceback on code that includes only in-line procedure expansion and cross-jumping optimizations. Other techniques use tables constructed during code optimization and generation to provide the correct location of variables at any point in the program [8, 15]. Debugging requests that modify data and control flow information, such as insertion or deletion of statements and changes in the value of a variable, are not supported by this system.

A technique to detect and recover variable values that have been affected by instruction scheduling has been developed [1]. Instruction scheduling does not include the complete range of code modifications that optimizations encompass. Another research project addressed the detection of nonresident variables, that is, a variable who has no value due to register allocation [2]. Noncurrency is not addressed nor are the effects of optimizations other than register allocation. In both of these works, commands that change the value or code during debugging are not permitted.

Although all of these techniques allow some form of optimization with source level debugging, by necessity, all trade between restrictions in either debugging capability or compiler optimization. The supported optimizations in all of these systems are based on the whole repertoire of debugging commands offered by the debugger, without regard to the user's desired requests or the context of any of these requests. When optimization is restricted, either a select few optimizations are allowed throughout the whole program, or the application of optimizations is very localized.

3. Incremental Analysis

The approach to debugging optimized code via tailoring utilizes techniques in incremental code optimizing transformation, and incremental register allocation and code generation. These techniques rely on incremental data flow.

3.1. Incremental Code Transformation

Although modular programming with separate compilation aids in eliminating unnecessary compilation and reoptimization, code transformations such as loop unrolling and procedure inlining are often performed, increasing the size of the program modules. To avoid having to reoptimize the resultant large procedures from scratch, techniques for incrementally incorporating changes into globally optimized code can be applied.

In earlier work, we present algorithms for incrementally incorporating edits into globally optimized code [17]. In response to program modifications, changes in the conditions for safe machine-independent optimizing transformations are identified using an annotated intermediate representation of the program and incremental changes in the data flow information. The intermediate representation is incrementally updated to reflect the new optimized version of the program. Opportunities for newly validated optimizations are detected in the same manner as the identification of unsafe optimizations, and these optimizations are also incorporated into the intermediate representation. When an executable is demanded, high quality machine code can be generated quickly from this intermediate representation as well as the unoptimized equivalent code.

Changes in optimizing transformations can initiate changes in a procedure's data flow. A number of techniques, which vary in their approach and power, have been developed to incrementally update data flow information [5, 13, 14, 16, 20, 23]. As we are not concerned in this paper with the particular technique used for incremental data flow analysis, we refer the reader to the appropriate references above for more information on each approach.

3.2. Incremental Register Allocation and Code Generation

By altering the relationships between a variable's definitions and uses, program changes can also affect the program's register allocation. In particular, an edit can lengthen or shorten the time that a register is needed for a particular value or can modify the number of registers needed in a region of the program. This necessitates reallocation of registers. Reallocation of registers can cause changes to machine code, including the update of register numbers when allocations are changed and the addition or deletion of instructions that move data between memory and registers.

Bivens and Soffa [3] developed a technique for incremental register allocation in conjunction with an incremental target code generator, which also deals with the incremental regeneration of target code. This incremental technique supports both local register allocation within a basic block and global register allocation across basic block boundaries. In order to incrementally update register allocations, they maintain a summary of register usage representing existing allocations and "ideal" register allocation. The register summary is designed to be adaptable for use with various register allocation schemes.

Register allocation is based on the priority of the live ranges of the variables in the program, where a live range is defined to be the portion of the code over which a value for a variable is defined and used. Using the register summary, the register allocation and assignment are updated to reflect the changes in priorities of a live range, changes in the size of a live range, and changes in the allocation of a live range between global and local registers. Demands for registers are modeled by an interference graph in which nodes represent live ranges and edges connect nodes that may have demands for registers at the same point in the program. Register allocation is performed by coloring the interference graph [6, 7].

4. The Program Database

The information that we maintain for our debugger as a program database consists of an optimization profile, register summary, global data flow information, and mappings between the program representations. All of this information can be easily generated by an optimizing compiler.

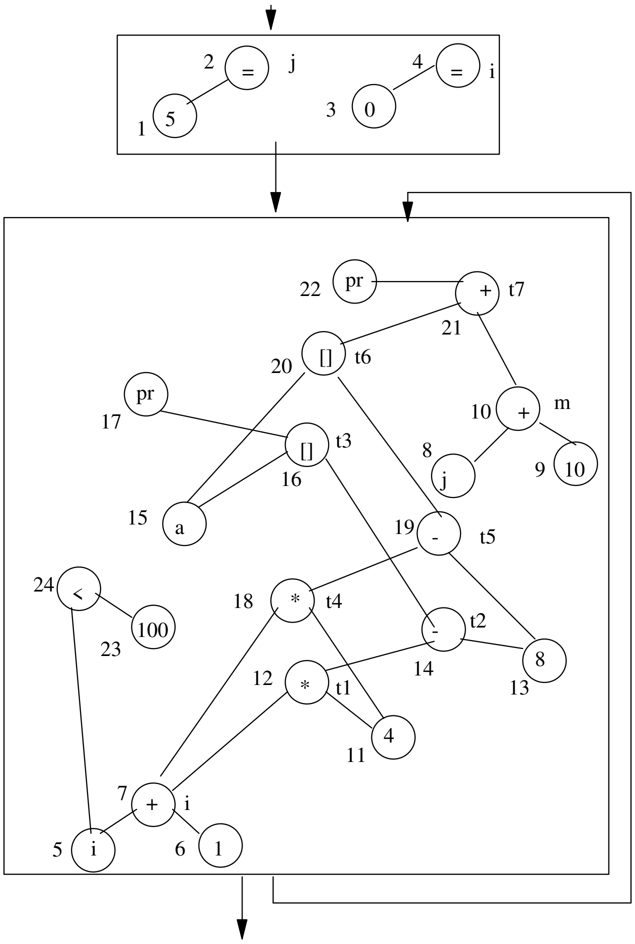
4.1. Optimization Profile

In order to determine whether a debugging request can be serviced transparently, and if not, attempt to report the source of inconsistencies to the user, information about compiler optimizations that have affected the current target code must be available to the debugger. This information is analogous to the *optimization profile* and *register summary* used in the incremental optimization and register allocation techniques for incrementally incorporating edits into optimized code. When an optimization is performed during incremental or batch compilation, information about the transformation, which is readily available at that time, is recorded in the optimization profile rather than destroyed. Similarly, when the incremental optimizer determines that an optimization is no longer valid after an edit, the record of the transformation is removed from the optimization profile. Since the order in which optimizations are performed is not important to either incremental optimization or symbolic debugging, a profile maintains no relative timestamp on optimizations.

Compiler optimization is typically performed on a control flow graph representation of the program. The data dependencies between statements in each sequential code segment (i.e., basic block) can be represented as a directed acyclic graph (i.e., dag). Figure 1 shows both the flow graph representation and the linearized intermediate code for a small source code segment. An optimization profile is constructed by annotating the dag nodes of this intermediate representation. The annotations enable the intermediate representation to simultaneously represent the unoptimized and optimized program code with a condensed history of the machine independent optimizations responsible for the current form of the optimized code [11, 18]. The optimized code can be obtained from this annotated intermediate representation by traversing the control flow graph with a postorder traversal of each dag, ignoring the annotations. The unoptimized code can be obtained by the same traversal, but using the annotations to regain the effect of the original code. We refer to the annotations in this intermediate representation as the optimization profile.

The optimization profile is dictated by the information required to maintain a complete snapshot of existing optimizations, adequate to determine when optimizations become unsafe due to edits and to determine when optimizations inhibit transparent symbolic debugging. Thus, the optimization profile depends on the optimizations being supported by the optimizing compiler. However, the information recorded for a transformation can be generalized. The profile must represent code that has been *eliminated, relocated,* or *replaced* by optimization.

In order to represent optimizations which remove, reorder, and replace statements, the variable labels on dag nodes (which correspond to simple identifier labels on traditional dag nodes, (see Figure 1), take the form of Figure 2. The **variable name** identifies the variable associated with the value computed at the node. Used to record eliminated variable stores, the **store status** reflects a NOSTORE value when the store to that variable has been eliminated by optimization and a STORE value when the store currently exists in the code at that point. The **original location** component consists of two subfields, the *node pointer* and the *label index,* which uniquely identify the original position of the store of the node's value to the variable.



Source		Target
...		...
j:=5;		j=5
i:=0;		i=0
repeat	100:	i=i+1
i:=i+1;		m=j+10
m:=j+10;		t1=i*4
print(a[i],a[i]+m);		t2=t1-8
until (i>=100);		t3=a[t2]
...		print(t3)
		t4=i*4
		t5=t4-8
		t6=a[t5]
		t7=t6+m
		print(t7)
		if i<100 goto 100
		...

Figure 1. Intermediate representation of a source program.

The *node pointer* points to the dag node corresponding to the original position of the store to the labeling variable prior to optimization. If the original store to variable *v* occurred at the unoptimized statement *s* mapped to node *n*, regardless of whether the store to *v* was moved by optimization, *v*'s *node pointer* points to node *n*. As in the traditional dag, a single dag node can represent several unoptimized intermediate code statements. Furthermore, the target node *n* of a variable's *node pointer* can represent several unoptimized statements (i.e., contain several variable labels). Using only *node pointer* information, it appears that the original store could have occurred at any of the unoptimized statements mapped to node *n*. Therefore, *label index* is included in the **original location** component to specify which unoptimized statement that maps to the target node of the variable's *node pointer* represents the variable's original store position.

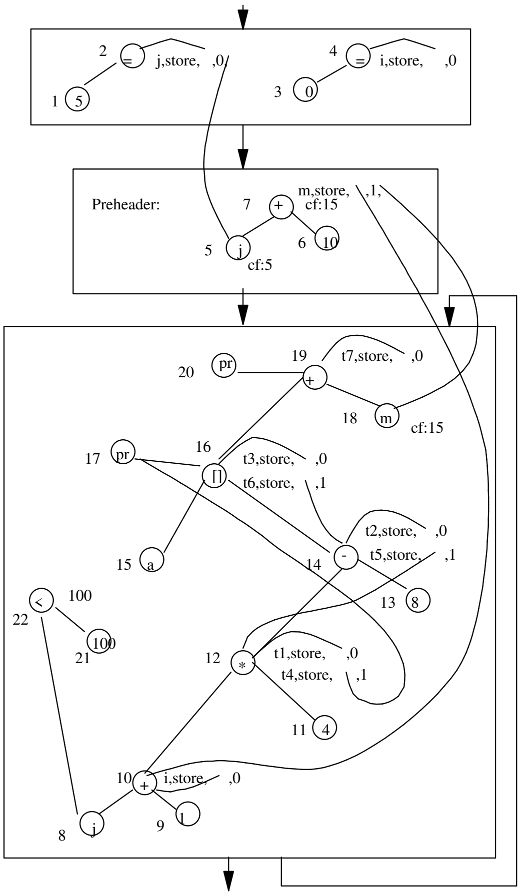
variable name (string)	store status (store/nostore)	original location		sub links (pointers)
		node pointer (pointer)	label index (integer)	

Figure 2. A dag node label.

The **sub links** component is either null or contains a set of pointers to all uses in *other* basic blocks to which the identifier (i.e., constant or variable) has been propagated. Figure 3 depicts an optimized version of the segment in Figure 1 and the corresponding optimization profile. We conservatively assume that temporaries and user variables may be used later in the program. Since no stores are eliminated by an optimization in this code segment, all variables labels have a STORE status. Constant folding can be identified at nodes 5 and 7 in the preheader where the constant value labels the nodes. The folded value 15 at node 7 was propagated to node 18 as indicated by the sublink from node 7 to node 18. Several common subexpression evaluations can be identified by the nodes 12, 14, and 16 which have more than one node label. Code motion has occurred in several places, as indicated by the node pointers of the original location fields of variable label *m* on node 7, variable label *t6* on node 16, variable label *t5* on node 14, and variable label *t4* on node 12.

For the debugger to determine whether a variable's value at a certain execution point is in a register, in memory or nonexistent, and to incrementally react to code changes during debugging, a register allocation summary consists of information about the actual and ideal register allocation, the demands for registers, and the points at which spill code have been introduced. Information about the ideal assignment is used as a goal when incrementally reallocating registers in response to code changes. The summary captures enough information to allocate both local and global registers, and is independent of the actual allocation heuristics used in each scheme. In particular, it is directed toward register allocation that uses some form of graph coloring. If simpler allocation schemes are used, then all of the information in the summary may not be needed [3].

The register allocation summary acts as the interface between name spans (definition of a variable and all its uses) and physical register usage. A name span that includes several blocks is denoted by a global register span which contains a virtual span for each block included in the name span. Each block



Source

```
...
j:= 5;
i:= 0;
repeat
  i:=i+1;
  m:=j+10;
  print(a[i],a[i]+m);
until (i>=100);
...
```

Unoptimized	Optimized
...	...
j=5	j=5
i=0	i=0
100: i=i+1	m=15
m=j+10	100: i=i+1
t1=i*4	t1=i*4
t2=t1-8	t4=t1
t3=a[t2]	t2=t1-8
print(t3)	t5=t2
t4=i*4	t3=a[t2]
t5=t4-8	t6=t3
t6=a[t5]	print(t3)
t7=t6+m	t7=t6+15
print(t7)	print(t7)
if i<100 goto 100	if i<100 goto 100
...	...

Figure 3. An optimization profile for machine independent optimization.

in a global span is marked by an E, U, P or T to indicate an entry block (contains a definition of the name), a use block (contains a use, but no definition), a pass-through block (contains no occurrences of the name, but the name is live at the end of the block) and a terminal block (contains a use of the name and the name is not live at the end of the block).

Each name in a basic block is associated with a *virtual register span* which is a span of intermediate code statements over which it is desirable to allocate a register to one or more names. The summary of register allocation inside a basic block is represented by a list of virtual register spans. A virtual register span includes the extent of the span (e.g., local or global), the allocation of the span (e.g., local or global), the physical register assigned to the span, and an indication of whether the span has been spilled.

The summary of global register allocation is a list of global spans, where each global span represents a live range of a variable. A global span consists of pointers to the virtual register span of the variable for each active block and a list of the pass-through blocks for the live range. For a global span, the virtual register spans associated with the active blocks are the local component of that global span. A

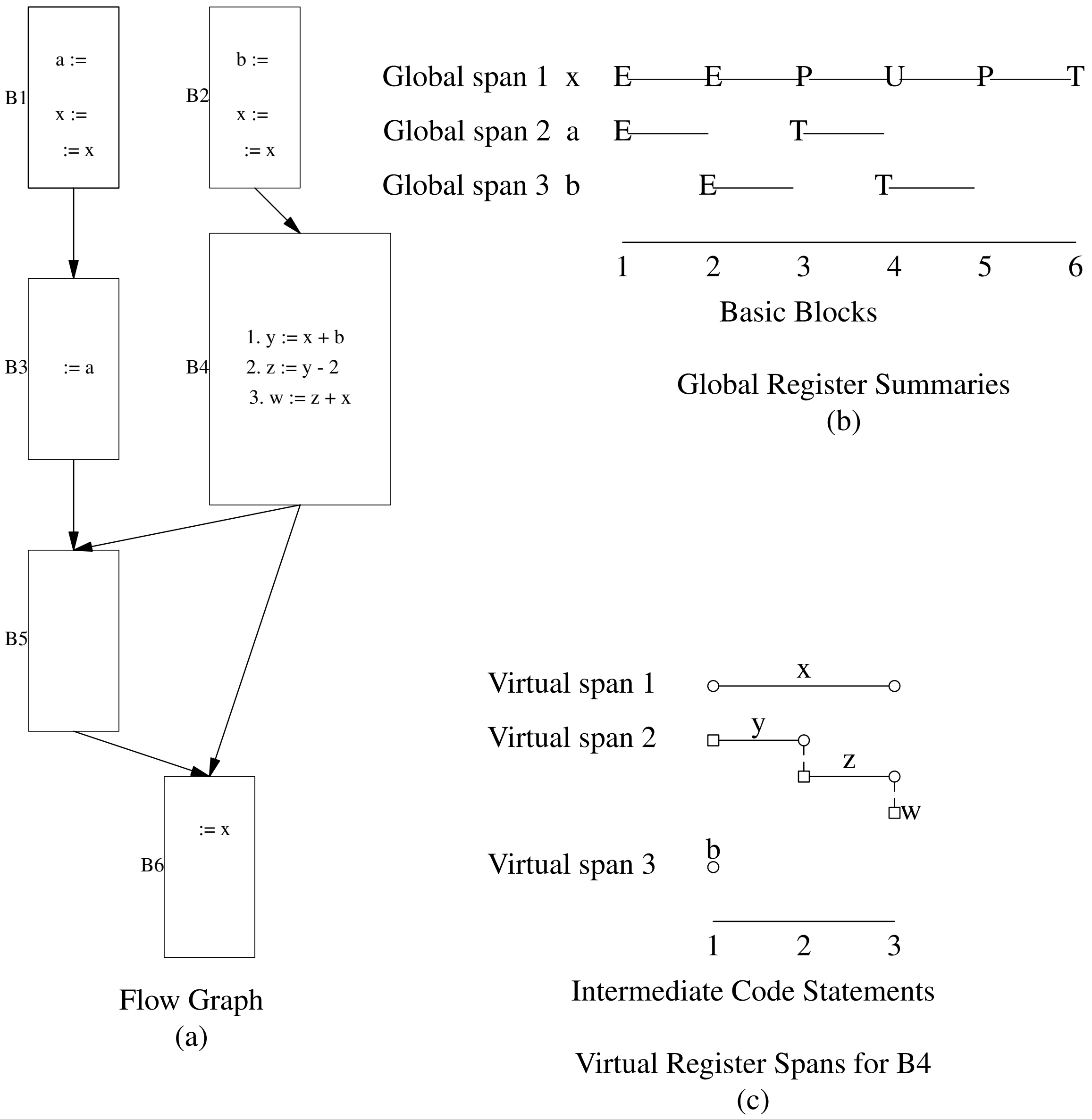


Figure 4. A register allocation summary.

global span includes the variable name, its register allocation, the physical register for the span (if globally allocated), and a list of virtual register spans for the entry blocks, use blocks and terminal blocks in the global span, and a list of pass-through blocks in the global span.

Figure 4 illustrates the summary of global and local virtual register spans for a program fragment. Part (a) shows a code segment in the form of the control flow graph. In part (b), the corresponding global register summaries indicate that the global span for *x* consists of entry blocks B1 and B2, use block B4, pass-through blocks B3 and B5 and terminal block B6. The variable *a* has global extent and its range consists of blocks B1 and B3. Global span 3, for variable *b*, consists of the blocks B2 and B4. It can be seen from part (b) that global spans 1 and 2 and spans 1 and 3 overlap. Thus, these three global spans could be satisfied with 2 global physical registers.

In part (c), the virtual register spans for basic block B4 are shown where the circles indicate variable uses and boxes indicate variable definitions. B4 is a use block for the global spans of *b* and *x*. Variable *b* has only a variable use at intermediate code statement 1 in B4, while *x* has a use in the first and last code statement and is live throughout the block B4. Virtual span 1 is the local component of global span 1 in B4, and virtual span 3 is the local component of global span 3. Virtual span 2 has local extent. If *x* and *b* are globally allocated, then virtual spans 1 and 3 will not be part of the local interference graph in B4; otherwise, they will be part of the graph and will compete for registers locally.

4.2. Global Data Flow Information

As optimizing compilers use data flow information to determine when it is safe to perform each optimizing transformation, the incremental optimization techniques used for debugging in our approach must also use this information. Data flow sets are maintained on entry and exit of each basic block, and incrementally updated for efficiency and to obtain the changes in data flow information which are used in detecting changes in the safety of optimizations in response to code and value changes. The kind of data flow information that is maintained depends on the optimizations that are supported by the optimizing compiler. Reaching definitions and reachable uses are sufficient for register allocation, but other information, such as available expressions or constant propagation information, may be needed for other optimizations.

4.3. Mappings between Program Representations

To set breakpoints and report error locations, the symbolic debugger must keep mappings between the source program and its executable code. To utilize the incremental optimization techniques for debugging, the mapping from source to target becomes a mapping from the abstract syntax tree to the annotated flow graph and a mapping from the annotated flow graph to the executable code. Both the incremental code generation techniques and the symbolic debugger need a two-way mapping between the annotated intermediate representation and the target code. The mapping from the intermediate representation to the target code can be maintained as part of the intermediate representation. A table maps the target code statements back to the annotated flow graph. The mappings can be incrementally updated by the incremental optimization techniques as code changes are processed.

5. The "Tailoring" Approach

The tailoring approach to debugging optimized code is presented in Figure 5. The debugger accepts debugging requests between program executions as well as during instrumented execution. The debugging requests submitted between executions are *planned* requests; these requests are guaranteed to be serviced transparently. Requests that are submitted during execution and involve a control break result in calling the *CheckLocation* routine. This routine uses the optimization profile to determine whether a unique location in the intermediate code can be identified with the source location of the control break. Other requests that are submitted during execution result in calling the *CheckTransparent* routine, which uses the optimization profile and register summary to determine whether the request can be serviced transparently. The *TransparentService* routine uses these information structures to give a transparent response to the programmer, which may include tailoring the code to remove an optimization. The *WantToTailor?* routine exhibits the choice that we give the user when it is found that a request submitted during execution can not be serviced transparently. If the requested information is not crucial to the debugging process, the user can continue with execution and with other debugging commands. However, if the user wants transparent behavior and we determine that it cannot be achieved, the user can submit requests before the next execution to ensure transparent behavior on the next execution. Execution is suspended, the optimization profile and register allocation summary are updated, the target code is tailored and execution is restarted.

Code is tailored in that the optimization that inhibits transparent debugging or depends on optimization that inhibits transparent debugging is not performed, but all other optimization remains. Thus, no unnecessary restrictions are put on the optimization in the executable code. During instrumented execution, the code that is not affected by debugging requests is still optimized and the debugging commands are serviced with the accuracy of unoptimized code. The conflicting effects of the debugging actions and the original optimizations will determine the closeness of the code being debugged to the final optimized code.

5.1. Fielding Requests Submitted During Execution

Ideally, a debugger should be able to provide an expected response to a request submitted at any time during execution regardless of how the executable code has been transformed by the optimizing compiler, where **expected response** means that the response during execution of an optimized version of a program is the same as the response would be for an unoptimized version of the same program. The debugger must first detect whether its usual actions in response to the request would provide a response consistent with the user's view; that is, whether any optimizations have created an inconsistency between the system's response and what the user expects. If it determines that it can not resolve the inconsistency, then the debugger must inform the user of the potential discrepancy, which can be eliminated in a subsequent execution.

```
Algorithm Debug.
 Begin
  Get next debugging request R;  {planned request}
  While R <> EXIT do
   If R = RUN
    then InstrumentedExecution;
    else update transparent request list;
   Get next debugging request R;
  endwhile
  Remove all effects of current transparent request list
   from optimization profile and register summary;
 End Debug.

Procedure InstrumentedExecution.
 Begin
  For each request R on transparent request list Do
   begin
    Interpret R as incremental edit;
    Incrementally update optimization profile and register summary for R;
   endfor;
  Tailor code for current transparent request list;
  Start instrumented program execution;
  While debugging requests are submitted Do
   begin
    Let the debugging request = R;
    If R is control break then
     begin
      unique = CheckLocation;
      If not unique
       then WantToTailor?;
       else TransparentService;
     endif;
     else  {not control break}
      begin
       ok = CheckTransparent
       If ok
        then TransparentService;
        else WantToTailor?;
      endif;
   endwhile;
 End InstrumentedExecution.

Procedure WantToTailor?.
 Begin
  action = AskUser;
  If (action = tailor) or (action = tailor_and_continue) of (action = tailor_and_skip)
   then update transparent request list;
  If (action = continue) or (action = tailor_and_continue)
   then give correct response with warning if appropriate; {not transparent behavior}
  If (action = skip) or (action = tailor_and_skip)
   then {ignore debugging request}
 End WantToTailor?.
```

Figure 5: The Debugging Algorithm.

Upon a debugging request at execution time, the mapping between the source and intermediate representation and the optimization profile is first used to determine whether any machine independent optimizations inhibit the correct response. If no optimization inhibits the request, then, depending on the debugging request, the debugging action is performed with accuracy, or the register summary is used to determine if register promotions prohibit the servicing of the request.

It is also possible that a request could be honored although it has been determined that optimization will affect correctness of the response. The debugger may be able to determine that the effect of the optimization can be easily reversed, allowing correct response. For example, if code is not generated for a statement A := B + C, the value of A can be recovered if the appropriate values of B and C are available. Another example is shown in the later subsection on the modification of a value.

5.2. Transparent Debugging With Maximal Allowable Optimization

The debugger interprets planned debugging requests as changes that temporarily destroy the correctness-preserving properties of some optimizations. The effect of the changes is removed upon deletion of the corresponding debugging request or by exiting the debugger. By interpreting debugging requests as temporary program edits, we can employ the incremental optimization algorithms to perform the analysis necessary to determine which optimizations inhibit transparent debugging and regenerate code that excludes those optimizations.

The destroyed optimizations are not actually removed from the optimization profile, but rather are marked to be ignored, which generates a list of changes to the intermediate code. Although the optimization in the debugging scheme is not permanently removed, its effects on data flow information must be removed to correctly enable the incorporation of other debugging commands. Marks indicate whether code should be ignored, optimizations should be ignored, or inserted code should be removed at the conclusion of the debugging session. When code is generated, this information on the nodes is used to determine if the code for the node should be traversed at all, traversed in an optimizing fashion or traversed in a non-optimizing fashion. This information must be incorporated into the target code, producing code that is tailored for the desired debugging command.

In response to a planned debugging request, the incremental optimization marking phase produces a list of intermediate code changes, insertions and deletions, that are incorporated into the machine code by incremental machine code generation. The intermediate code changes are incorporated as *temporary* modifications to the machine code and do not result in updates of the register summary or reallocation of registers. Instead, incremental code generation merely consists of the allocation of registers, if necessary, for the variables in the statement, the generation of code for the inserted statement, and perhaps, the addition of code to move variables at other points in the program. Thus, the incorporation of debugging requests as temporary changes to intermediate code does not involve the extensive analysis that is normally performed to update the register summary and reallocate and reassign registers.

In addition to leaving the register summary unchanged, the incremental machine code generator is designed to move deleted machine code to a save list, save the original mappings when changes are made, create temporary code, and provide easy restoration of code to its original form when the debugging session is over. When the user requests full debugging between two points in the program, in addition to temporary insertion and deletion of intermediate code statements, the incremental machine code generator will insert temporary code to store all definitions of user defined variables not presently stored in that region of the program.

5.3. An Example

Figure 6 shows an example segment of source code along with the source to intermediate code mapping (source statement lines and corresponding dag nodes), equivalent unoptimized intermediate code, optimized intermediate code, and tailored intermediate and target code after several debugging requests. Figure 7 depicts the intermediate program representation with the embedded optimization profile and the program's register summary. In our example, we assume that all temporaries are dead at their last use within a basic block, while all other variables except *j* are live at the end of basic block B3.

We start with the debugging request: "IF *m* > 10 BREAK after statement 103 and EXAMINE values of ALL variables". The debugger first determines that statement 103 is mapped to dag node 10, which contains only one variable label, indicating that the breakpoint is inserted at a uniquely identifiable location in the code. Using the optimization profile, the debugger finds that this use of variable *m* invalidates only the code motion optimization, and the optimization profile is marked to indicate that the optimization should be ignored when generating the tailored code. Thus, the tailored code moves the folded M=15 back to its original location, but does not undo any other machine independent optimizations, in particular, the constant folding of this expression, because it does not inhibit a transparent response to this debugging request.

After the incremental optimizer has tailored the intermediate code representation, the deletions and insertions of intermediate code statements are passed to the incremental machine code generator. Machine code to be temporarily deleted (marked with '-') is copied to a save list, and inserted code (marked with '+') is generated. Register summaries are also checked to determine whether stores will have to be inserted to ensure that current values of user variables are available at the breakpoint. In the example, temporary code is generated to store the value of *m* at statement 104. Displaying all other variables at this point is determined to require no further intermediate code optimization reversals. However, the target code optimization (register allocation) that elided machine code for statement 101 will be reinserted in order to insure that the correct value of *j* is displayed.

Line	Dag Node		Unoptimized Code		Optimized Code		Tailored Code		Target Code
101.	2		j = 5		j = 5		j = 5		+Move 5 j
102.	4		i = 0		i = 0		i = 0		Move 0 R1
103.	10	L:	i = i + 1		m = 15		m = 15 {Delete}		
104.	7		m = j + 10	L:	i = i + 1	L:	i = i + 1	3:	Add 1 R1
							m = 15 {Insert}		+Move 15 m
105.	12		t1 = i * m		t1 = i * 15		t1 = i * 15		Move R1 R2
									Mult 15 R2
106.	12		w = t1		w = t1		w = t1		Move R2 R3
107.	12		t2 = i * m		t2 = t1		t2 = t1		
108.	12		x = t2		x = t2		x = t2		Move R2 R4
109.	14		if i < 100		if i < 100		if i < 100		Move 100 R2
									LT R1 R2
			goto L		goto L		goto L		Jumpt R2 3

Figure 6: An example code segment with mapping for debugging.

Now consider a request to "CHANGE the VALUE of *w* to *"constant"* after statement 106 and continue execution". The incremental optimizer detects that *w* is not involved in an optimization and that the request can be passed to the machine code generator. Using the register summaries, the incremental machine code generator determines that the current value of *w* after statement 106 is in register R3. The new value specified by the user is placed in this register, and the user's request is serviced transparently.

It should be noted that these requests were chosen purposely to illustrate situations which inhibit optimizations. There are many scenarios in which powerful debugging commands inserted at various locations require no (or few) optimizations to be reversed.

5.4. Interpretation of Debugging Requests

To demonstrate more fully the interpretation of debugging requests by the debugger, we describe the interpretation of the more common debugging requests. This list is not to be construed as exhaustive; any debugging command can be included if it can be interpreted as a change that disables any optimization affecting the correct response to the command.

Display of Values. To examine the values of specific variables at specified program points, the optimization profile is consulted to determine if the variables are involved in an optimization that changes the order of storing the values with respect to these program points or eliminates the store of the values. That is, the examination of a value can be interpreted as insertion of a variable use. For example, a request to examine the value of *x* after statement 108 in the code of Figures 6 and 7 would result in the debugger using the optimization profile to see that *x* is being stored, and using the mapping into the target code and the register summary, the debugger would also determine that *x* is in register R4 at this point in the program. Thus, this request can be serviced by writing out the value in R4.

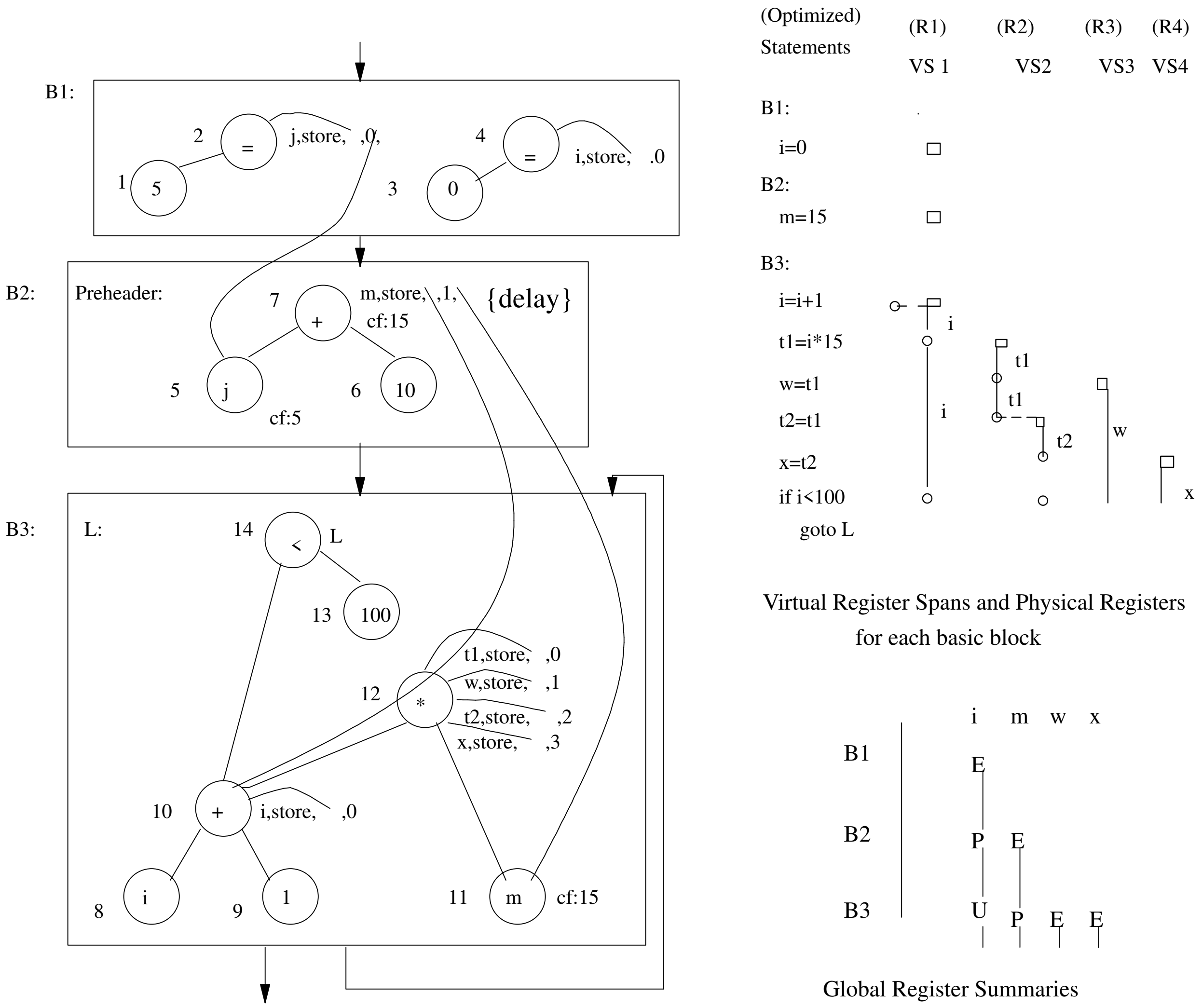


Figure 7: Information Structures after debugging requests.

Modification of Values. Determining if requests for modification of specific variable values at specified points can be honored by interpreting the request as inserting a variable definition and using the optimization and register profile. For example, a request to modify the value of *j* to 7 after statement 101 and continue execution could actually be honored effectively using the incremental analyses. The optimization profile (node 2) indicates that *j* is involved in constant folding. Without incremental analysis, the programmer would have to be notified that the variable *j* is involved in an optimization at that point, and that changing its value could produce incorrect results. However, the incremental optimization analyses would determine that the code involved in the constant folding has not been executed yet, and thus, the code can be tailored for this request. That is, the code for propagating the value of 7 could be generated rather than the current code that propagates the value of 5. For another example, when the programmer requests to change the value of *i* to 101 after statement 104, the optimization profile (node 4) indicates that the original location of the statement at 104 was after node 10 and that code for evaluating this node was already executed. The register summary indicates that *i* is stored in R1. The debugger can implement the request by loading 101 into R1.

Breakpoint Insertion. To insert a breakpoint, the debugger uses the mapping between source code and dag nodes in the annotated graph. Due to optimization, this mapping is many-to-one, and the breakpoint location may be mapped to a node that is shared by a number of statements. This is indicated in the annotated graph by multiple labels attached to the node. In Figure 7, the insertion of a breakpoint after source code statement i := 0 would map to node 4 and to target code instruction 1. However, the insertion of a breakpoint after statement 106 would map to node 12 which contains four labels. In this case, we disable the optimizations that cause the multiple labels on the node, but no other optimizations would be disabled.

Conditional Breakpoint. As demonstrated in the example section earlier, a conditional breakpoint is interpreted as a combination of breakpoint insertion and examination of values of variables appearing in the conditional expression. Thus, the interpretation consists of a temporary insertion of a variable use and and examination of possible problems with the exact location of the breakpoint. Any optimizations invalidated by either of the two actions would be identified by the incremental analysis and disabled in the code.

Single Stepping. To service the single stepping command, the user either details what variables are to be accessed (changed) at any point or assumes the default, which is that the user has access to all of the variables. In the first case, all optimizations involving the use (or definition) of these variables (including their promotion to registers) as well as any optimization that creates multiple variables on nodes would interfere with single stepping and be disabled. In the second case, all optimizations would be temporarily disabled. These actions combine the actions of breakpoint insertion, examination and modification of values.

Modification of Program Code. The modification of code can consist of both insertion of new code and deletion of source code already part of the original annotated graph. As is the case with all of the debugging commands, code inserted during debugging is considered temporary code and thus only easily detected optimizations are performed on this code. However, the inserted code could invalidate optimizations in existing code, so the debugger is called to analyze each inserted statement according to its variable definitions and uses. Similarly, the debugger uses the optimization and register profiles to analyze the effects of a temporary deletion of code during debugging. One example would be the user asking to "INSERT the statement $z = i * 15$" after statement 108. The incremental optimizer would determine that not only can this statement be inserted and execution continue from this point safely, but the statement that is being inserted can be easily optimized. The debugger will treat this request as a temporary statement insertion, map it to the node 12, and identify that the expression evaluation has already been done. Using the register summary, it will determine that the current value of the expression is in registers R2, R3, and R4. It can then insert a "MOVE R2, Z" at this point in the code, or just record that variable Z has its current value at this point in one of these registers.

6. Conclusions

We have presented a new approach to symbolic debugging of optimized code. By exploiting recent work in incremental compiler analysis, our technique eliminates unnecessary restrictions on compiler optimization while providing the programmer with a large repertoire of debugging commands, encompassing those offered by other systems supporting optimization. Traditional machine independent code optimizations are supported as well as register allocation.

If this debugger is part of an environment that includes an incremental optimizing compiler, then the information and analyses needed by the debugger would already be part of the incremental optimizing compiler. Thus, there would be very little extra cost for this debugger. If only an optimizing compiler is available, then the profile and summary information would have to be generated by this optimizing compiler. The optimizing compiler would require only minor modifications as the information needed by the debugger is already being computed by an optimizing compiler; all that would be necessary is that the required information be saved. The analysis to tailor the code would have to be added to the debugging system.

References

1. Ali-Reza Adl-Tabatabai and Thomas Gross, “Detection and recovery of endangered variables caused by instruction scheduling,” *ACM SIGPLAN ’93 Conf on Prog Lang Design and Implementation*, pp. 13-25, June 1993.

2. Ali-Reza Adl-Tabatabai and Thomas Gross, “Evicted variables and the interaction of global register allocation and symbolic debugging,” *Conf Rec of the 20th POPL*, pp. 371-383, January 1993.

3. M. P. Bivens and M. L. Soffa, “Incremental Register Allocation,” *Software-Practice and Experience*, vol. 20, no. 10, pp. 1015-1047, 1990.

4. Gary Brooks, Gilbert J. Hansen, and Steve Simmons, ‘‘A new approach to debugging optimized code,’’ *ACM SIGPLAN ’92 Conf on Prog Lang Design and Implementation*, pp. 1-11, June 1992.

5. M. Burke, ‘‘An interval-based approach to exhaustive and incremental interprocedural data-flow analysis,’’ *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 341-395, July 1990.

6. G.J. Chaitin, ‘‘Register Allocation and Spilling Via Graph Coloring,’’ *Proceedings of SIGPLAN ’82 Symposium on Compiler Construction*, pp. 98-105, 1982.

7. Fred C. Chow and John L. Hennessy, ‘‘The Priority-Based Coloring Approach to Register Allocation,’’ *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 4, pp. 501-536, 1990.

8. Deborah S. Coutant, Sue Meloy, and Michelle Ruscetta, ‘‘DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code,’’ *Proceedings of SIGPLAN ’88 Conference on Programming Language Design and Implementation*, pp. 125-134, 1988.

9. Peter H. Feiler and R. Medina-Mora, ‘‘An Incremental Programming Environment,’’ *5th International Conference on Software Engineering*, pp. 44-53, 1981.

10. Peter Fritzson, ‘‘A Systematic Approach to Advanced Debugging,’’ *Proceedings of SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pp. 130-139, 1983.

11. John Hennessy, ‘‘Symbolic Debugging of Optimized Code,’’ *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 323-344, July 1982.

12. R. Hood and B. Chase, ‘‘Selective interpretation as a technique for debugging computationally intensive programs,’’ *SIGPLAN ’87 Symposium on Interpreters and Interpretive Techniques*, pp. 113-124, June 1987.

13. James Keables, Katherine Roberson, and Anneliese von Mayrhauser, ‘‘Data flow analysis and its application to software maintenance,’’ *Proceedings of Conference on Software Maintenance*, pp. 335-347, October 1988.

14. T. Marlowe and B. Ryder, ‘‘An efficient hybrid algorithm for incremental data flow analysis,’’ *Conf Rec of the 17th Annual ACM Symp on Principles of Programming Languages*, January 1990.

15. P. P. Pineo and M. L. Soffa, ‘‘Debugging Parallelized Code Using Code Liberation Techniques,’’ *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 108-119, May 1991.

16. Lori Pollock and Mary Lou Soffa, ‘‘An incremental version of iterative data flow analysis,’’ *IEEE Transactions on Software Engineering*, vol. 15, no. 11, pp. 1537-1549, 1989.

17. L. Pollock and M. L. Soffa, ‘‘Incremental global reoptimization of programs,’’ *ACM Transactions on Programming Languages and Systems*, vol. 14, no. 2, pp. 173-200, April 1992.

18. L. L. Pollock and M. L. Soffa, ‘‘Incremental Compilation of Locally Optimized Code,’’ *Conference Record of the Twelfth Annual ACM Symposium on POPL*, pp. 152-164, January 1985.

19. L. L. Pollock and M. L. Soffa, ‘‘High-Level Debugging with the Aid of an Incremental Optimizer,’’ *Proceedings of the 21st Hawaii International Conference on System Sciences*, pp. 524-532, January 1988.

20. Barbara Ryder and Marvin Paull, ‘‘Incremental Data Flow Analysis,’’ *ACM Transactions on Programming Languages and Systems*, vol. 10, no. 1, pp. 1-50, January 1988.

21. Ron Tischler, Robin Schaufler, and Charlotte Payne, ‘‘Static Analysis of Programs as an Aid to Debugging,’’ *Proceedings of SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pp. 155-158, 1983.

22. H.S. Warren, Jr. and H.P. Schlaeppi, Design of the FDS interactive debugging system, RC7214, IBM T.J. Watson Center, Yorktown, NY, June 1978.

23. Frank Kenneth Zadeck, ‘‘Incremental Data Flow Analysis in a Structured Program Editor,’’ *Proceedings of SIGPLAN ’84 Symposium on Compiler Construction*, pp. 132-143, 1984.

24. Polle Zellweger, ‘‘An Interactive High-Level Debugger for Control-Flow Optimized Programs,’’ *Proceedings of SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pp. 159-171, 1983.