Source-Level Debugging of Optimized Programs Using Data Flow Analysis*

Lutz Berger, Roland Wismüller
Department of Computer Science
Munich University of Technology
Arcisstr. 21, D-W-8000 München 2
Tel.: +49-89-2105-8164
wismuell@informatik.tu-muenchen.de

December 4, 1992

Abstract

With the development of advanced processor and machine architectures that need optimizing compilers to be efficiently programmed in high level languages, the need for source level debuggers that can handle optimized programs is rising. The difficulty in debugging optimized code arises from the problem to establish mappings between source and object code locations and between source and object code variables. If we are not willing to restrict the optimizer's job, the debugger must analyze the effects of the optimizations done to establish these mappings. We will deduce some requirements this analysis should meet and present an algorithm that uses data flow analysis techniques to solve the data value problem for globally optimized code. We also give first experiences with an implementation.

Key words: debugging, compiler optimization, noncurrent variables, data flow analysis

1 Introduction

Source level debuggers have been proven to be important tools for program testing and for locating bugs in faulty programs. Current source level debuggers, however, rely heavily on the code not being optimized by the compiler. But modern computer architectures and processor designs demand for optimizing compilers to get the performance promised by the hardware design. Hardware features that need compiler support include caches, register files, pipelines, multiple functional units in superscalar and VLIW processors and also multiple processing units in parallel computers.

Although the earlier phases of program development often do not have a strong need for optimizers, in later stages, especially in prototype versions, it is desirable to optimize the code. In addition to run-time (or memory) savings, that may be quite large as in the case of automatic parallelization, there are some other reasons to use debuggers with optimized programs. First, as Copperman has shown [CM91], there may be bugs in a program that only have effects in the optimized version, even if the optimizer is correct. Furthermore, some bugs may appear not before the code is already in production use, at a point where it should be optimized.

For now, there are two main solutions to overcome this conflict. The first is to use two different versions of the program, an optimized one for real use and a non-optimized one for debugging. Besides the difficulties of maintaining two versions, this does not help when a bug does not show up in the

*This work is partly funded by the German Science Foundation, Contract: SFB 0342, TP A1

non-optimized program. Even if it does, there is still the problem to reproduce it. Another 'solution' is to use no debuggers at all. Myers from the National Security Agency states that one reason for them to use print statements for debugging is that debuggers don't work with optimized code [Mye91].

A debugger able to handle optimized code must provide solutions for two major problems [Zel84]: The *code location problem* is related to the mapping between lines or statements of the source program and instructions in the object code. If a programmer sets a breakpoint on a source line, the debugger must be able to decide where to stop the object program. If the program raises an exception, a reverse mapping is needed to report the location in terms of the source program. Once the program has been stopped (either by a breakpoint or an exception) the user typically wants to inspect the program's state, i.e. the program variables. To support this functionality, the debugger must be able to establish a relationship between the values of variables in the original source program and the optimized object code, i.e. it must be able to compute the values the source variables should have from the values of the object code variables. This is commonly referred to as the *data value problem*.

In the next section we will give examples for these problems and examine current solution techniques. Then we concentrate on solving the date value problem by data flow analysis. Section 3 defines requirements an algorithm solving the data value problem should meet, the difficulties arising from these requirements and solutions for them. Section 4 outlines an algorithm with the desired properties, the next section shows how the information needed is gathered by the compiler. Finally, section 6 concludes the work presented here.

2 Basic Problems and Current Solutions

As already stated, breakpoint based debuggers, that we will consider throughout this paper, must support two basic functions:

- The debugger must be able to stop the program being debugged at any point in the source code.
- Once the program has stopped, the debugger must be able to display the status of the program at the current point of execution. This status includes the procedure traceback and the values of all program variables.

Figure 1 shows an example program fragment and its corresponding object code in Sparc assembler. We will use this example to demonstrate the problems we get when trying to debug optimized code and to justify the requirements we will deduce. The code location problem arises as soon as the programmer tries to set a breakpoint. If he wants to break at line 4, where should the debugger halt the optimized program? Zellweger has proposed to use different mappings, the *semantical mapping*, where "[...] the corresponding object code is the object code that performs the operation specified by the program statement" and the *syntactical mapping* "[...] that preserves the position of a statement with respect to its neighboring statements" [Zel84].

In fig. 1 it is clear that the semantical mapping of line 4 is instruction 2. However, it is not easily seen what the syntactical mapping should be. There are several possibilities, e.g. instruction 10,11 or 12 (since they are in the same conditional branch), instruction 8 (since that is the instruction corresponding to the successor of line 4) or instruction 21 (immediately after the instruction corresponding to the predecessor line).

The appropriate mapping depends heavily on the programmer's intention in setting the breakpoint. If he wants to examine the old value of f or the values of g and h before they are used to compute f the semantical mapping should be used. However, if he wants to determine which $a[i]$ are positive, the program should be stopped at instructions 10, 11 or 12. When he is interested in the value of y, it may probably be the best to stop at instruction 21, if register i5 is positive.

Suppose the program has been stopped at instruction 11. When trying to inspect variables, a couple of difficulties arises:

```
 1 for (i=0;i<10;i++) {
 2   if (a[i] > 0) {
 3     y = y + a[i];
 4     f = g+h;
 5     w = a[i] * 2;
 6     z = z+w;
 7     x = a[i] + 1;
 8   }
 9   else {
10     w = a[i] * 2;
11     x = -a[i] + 1;
12     z = z-w;
13     y = y + a[i];
14     f = g+h;
15   }
16 }
```

```
 1      mov   0,%i3          i = 0
 2      add   %o0,%l6,%l6    f = g+h
 3      add   %fp,-48,%i2    i2 = &a[i]
 4                           loop:
 5 L1:  ld    [%i2],%i5        i5 = a[i]
 6      tst   %i5
 7      ble   L2
 8      sll   %i5,1,%i1        w = a[i] * 2
 9                             if (a[i] > 0) {
10      add   %i5,1,%i0          x = a[i] + 1
11      b     L3
12      add   %i4,%i1,%i4        z = z+w
13                             } else {
14 L2:  sub   %g0,%i5,%i0
15      inc   %i0                x = -a[i] + 1
16      sub   %i4,%i1,%i4        z = z-w
17                             }
18 L3:  inc   %i3              i++
19      cmp   %i3,10
20      add   %l7,%i5,%l7      y = y + a[i]
21      bl    L1
22      inc   4,%i2            i2 = &a[i]
23                             if (i<10) goto loop
```

Figure 1: Program fragment before and after optimization

- The value of x cannot be determined, since the proper register $i0$ already contains the value that should be assigned not before line 7. Hennessy [Hen82] calls x a *roll back variable* in this case.
- The situation for variable y is just reversed. y should have been already assigned according to the source code, but the corresponding store to register $l7$ is delayed until instruction 20. y is therefore called a *roll forward variable*.
- In case of variable f, the situation is more complicated. In the first iteration of the loop, f is a roll back variable, since its proper register has already been assigned in instruction 2. In all later iterations, the value of f can be determined by reading $l6$. In this case f is said to be *current*. However, if the debugger has no information about the iteration count of the loop, it must conservatively regard f as noncurrent.

In the last years several researchers have developed techniques to overcome the problems stated so far. These solution techniques can be roughly classified into four categories:

2.1 Efficient Source Code Instrumentation

Since the difficulties in debugging optimized programs primarily arise from the non-trivial mapping between source and object code, most problems silently disappear if the program is instrumented on the source level. The easiest way to do this is to insert print statements into the source code. When using this simple approach, the program must be recompiled each time the monitoring requirements change.

Gupta [Gup90] has proposed a slightly more advanced technique. He uses conditional monitoring statements that are put into the source program. The debugger then enables or disables conditions based on the user's debugging commands, thus avoiding recompilation, as long as the request can be processed with the current instrumentation.

Pollock and Soffa [PS88] take advantage of incremental compilation and optimization techniques to allow for a quick recompilation after a debug statement has been inserted or deleted from the source code. It is also possible to quickly generate an unoptimized version of the program.

Due to the necessary recompilation, both techniques do not allow the program to be continued but require it to be restarted after each change in the instrumentation.

2.2 Special Handling of Optimizations

One of the earliest implementations of a debugger handling at least some optimizations is the Navigator system developed by Zellweger [Zel83a, Zel83b, Zel84]. Navigator is able to set breakpoints and to display procedure tracebacks in the presence of inlining and cross-jumping. Cross-jumping is a storage optimization that merges identical paths in different branches of conditional statements, thus generating ambiguities when a breakpoint is set in such a region. Zellweger uses special data structures generated by the compiler that instruct the debugger in how to set breakpoints and how to display correct procedure tracebacks of inlined procedures. She also did some theoretical work, especially she introduced the terms *correct* or *truthful behaviour* and *transparent* or *expected behaviour*. If a request cannot be handled in the expected way, i.e. giving the same result as with an unoptimized program, a debugger providing only truthful behaviour must at least give the user enough information about the effect of optimizations to interpret the result correctly. In our example, a legal answer of the command 'display x' would be 'x = 123, but this value was set early during execution of line 7'.

Coutant, Meloy and Ruscetta developed a debugger named DOC [CMR88] that implements a mixture of correct and transparent behaviour. They use a special data structure, called *range records*, containing a program counter dependent mapping between source code variables and addresses, registers or constants. This structure must be updated in each optimization phase of the compiler. It enables the handling of register assignment and spill, register promotion, copy elimination, loop variable elimination and instruction scheduling within basic blocks.

The range record structure is also used in the commercial Convex debugger CXdb [Str91, BHS92]. In contrast to all other approaches, CXdb does not even attempt to hide the effects of optimization, but tries to visualize them. This has been done by introducing several levels of granularity for stepping and breakpointing (expressions, statements, blocks, loops and routines) and by highlighting the code elements that are currently executed in the source code listing. Though this is appropriate for single stepping, it seems of little help when the program stops at a breakpoint or due to an exception.

2.3 Inspection Points

This technique is based on the observation that the actual state of the program being debugged is only important at the points where a debugger can be invoked (*inspection points*). If these points can be reduced to only a few ones, the compiler can optimize the code between two inspection points without any constraints. At each inspection point the compiler must either leave the state unchanged or provide the debugger with a description on how to derive the unoptimized state from the optimized one.

Inspection points are useful for object oriented languages with safe primitives, where run-time errors can only occur at few well-defined points in the program. For procedural languages like C, where almost every statement may eventually raise an exception, optimization is essentially restricted to be local within statements. Otherwise, the additional information needed by the compiler grows immensely. Inspection points (or *interrupt points*) have been proposed independently by two research groups:

Hölzle, Chambers and Ungar have designed a compiler/debugger system for the SELF programming language [HCU92]. If the program stops at an interrupt point, which can only be a method prologue or the end of a loop body, the active methods are replaced with unoptimized versions while the stack frames are adapted properly to reflect that change. The SELF system uses a dynamic compilation technique that allows single methods to be recompiled during run-time. After this *dynamic deoptimization* debugging can be done as usual.

Zurawski and Johnson [ZJ91] made a very similar approach in their TS debugger. In addition, their implementation allows variables to be noncurrent at an inspection point, as long as the value can be reconstructed by evaluating a recovery expression provided by the compiler. In contrast to the SELF system the TS debugger only replaces methods by unoptimized versions, if the programmer wants to modify the state or uses single stepping.

2.4 Data Flow Analysis

The idea to use some kind of data flow analysis to detect noncurrent variables has first been published by Hennessy [Hen82]. He introduced the terms *current*, *roll back variable* and *roll forward variable* and gave some algorithms to compute these classes of variables in the exclusive presence of local optimizations. Hennessy also suggested to use *recovery* to recreate the values of noncurrent variables: The value can be determined by finding the variable's last definition[1] and evaluating the right hand side of this definition. Of course, recovery can only be done if all variables in the right hand side are available. Hennessy's paper also contains some considerations on global optimization.

More advanced studies in using data flow analysis to handle global optimization have been done by Copperman. To our knowledge, he was the first to give a formal definition of the term "current" [Cop92a]:

- A variable V is current at a breakpoint B along a path p, iff the store into V that reached B along p_o was generated from the definition of V that reaches B along p_u.
- V is current at B, iff V is current at B along each path to B.

Copperman refers to an assignment in the source code as definition, an assignment in the optimized code is called a store. p is defined as the pair (p_o, p_u), where p_o and p_u are the paths the optimized program takes or the source program would take, respectively, when both work on the same input. Based on this definition he has developed two algorithms to detect noncurrent variables in globally optimized code.

Since one of them [CM91] has proved to be not correct in some cases, we will give a short sketch of the second algorithm [Cop92b]. The idea is to compare the definitions of a variable V that should reach a breakpoint B against the stores of V that actually reach it. If all the reaching stores correspond to definitions that should reach along the same path and vice versa, then V is current. The algorithm is similar to the iterative solution of the reaching definitions problem. However, in order to reflect the condition that definition and store must reach along the same path, it computes for each basic block a set of pairs (d, s) of definitions that should reach and stores that actually reach the beginning of the block. V is stated to be current at a block entry, if for each pair (d, s) in the set of this block the store s has been generated from d.

Neither Hennessy nor Copperman seem to have implemented their methods to handle global optimized code. Nevertheless, the approach is promising, since it is mostly independent of special optimization methods or language features and does not put any restrictions on the optimizer.

In the following we will present our algorithm solving the data value problem for globally optimized code with data flow analysis. The requirements it is based on are much more general than in previous work and will be figured out in the next chapter.

[1] In case of local optimization, this definition is unique

3 Requirements on Solutions of the Data Value Problem

3.1 Independence of Optimization Details

Since we are interested in general solutions, the algorithm should be mostly independent of the optimizer's internals, like the chosen transformations and their order. The debugger should only be loosely coupled to the compiler and not depend on optimization specific data structures. This imposes, that the debugger must be able to at least conservatively handle all (or most of) the common code transformations.

We therefore chose to use flow graphs as the interface between compiler and debugger. Flow graphs have been proved to be suitable program representations for many kinds of data flow analysis and can be provided easily by the compiler, since most optimizers use flow graphs or comparable structures as intermediate representation.

In contrast to Copperman's approach, where a single flow graph represents both the unoptimized and the optimized program, we use separate graphs for source and object code and an explicit mapping between corresponding entities. As a consequence, we can also represent the effects of transformations that heavily change the basic block structure of a program, as cross-jumping, unswitching, unrolling, multiple version loops and loop transformations.

3.2 No Limitation to Live Variables

Several techniques in the literature [CMR88, CM91] can only handle live variables, i.e. they state variables as noncurrent outside their life range. Nevertheless the values of non-life variables are often of interest during debugging. If, for example, after an assignment $x = a + b$ the value of x is not what it should be, the programmer is interested in whether a or b did have a wrong value, even if a and b are dead after the assignment. A method based on tables that map source variables to object variables heavily burdens the compiler when information about non-live variables is needed, since all information about a variable is usually discarded after its last use.

However, with an appropriate program representation, the debugger can easily determine the proper object variable for some variable v at any point in the program by itself. It only has to calculate the set of reaching definitions of v. The variables written by the corresponding stores are those that correspond to v[2]. The information required by this analysis is needed in any way for currency determination.

3.3 Arbitrary Breakpoint Mapping

While in small examples it is usually trivial to find the syntactical mapping of a breakpoint, Zellweger [Zel84] has shown that control flow optimizations make the situation more complicated. In some cases a single breakpoint in the object code is not sufficient, in other cases a syntactical breakpoint location may not even exist.

For these reasons, and in order to separate the code location from the data value problem, we have designed an algorithm that can work with an arbitrary breakpoint mapping.

A further reason is the observation, that the exact location of a breakpoint in the object code is unimportant as long as the following condition holds:

> The breakpoint in the object program is reached, if and only if the breakpoint should be reached in the source program.

[2] If the object variable is not unique, v should be regarded as noncurrent

It is therefore possible to optimize the breakpoint mapping: We can choose the breakpoint location in the optimized program such that the above condition holds and as much variables as possible are current at the breakpoint. If the user is given the possibility to specify a set of variables of interest when defining the breakpoint, the debugger may ensure that a large subset of these variables are current. Logging or tracing may then be used to save the values of noncurrent and non-recoverable variables.

From the considerations above it follows that a unique syntactical breakpoint location does not exist. We therefore provide modified and extended definitions of the terms *syntactical* and *semantical mapping*:

- For a breakpoint b_s in the source program and a statement s_o in the optimized program let $P_o(b_s, s_o)$ be a predicate with $P_o(b_s, s_o)$ = True, iff it can be ensured that b_s has been reached or will be reached in the source program. A syntactic breakpoint location for b_s is any s_o with $P_o(b_s, s_o)$ = True.

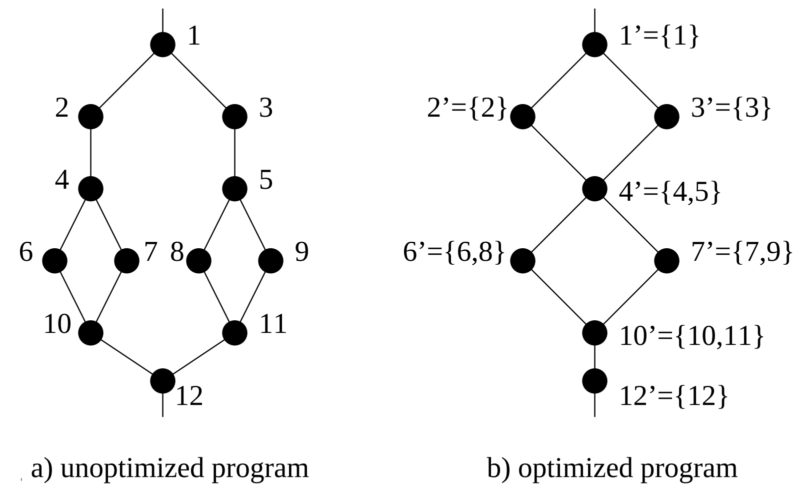

Figure 2: Cross-jumped program fragment

  The following example illustrates this definition: Assume a source program with a flow graph as in fig. 2a. After a cross-jumping optimization the flow graph may look like fig. 2b. Legal locations to stop the optimized programs when the user sets a breakpoint on statement 11 would be[3]:

  - instruction $3'$
  - instruction $4'$, if $3'$ has been previously executed
  - instruction $10'$, if $3'$ has been previously executed

  It would also be possible to place two breakpoints on instructions $6'$ and $7'$ with the condition 'instruction $3'$ has been executed'.

- A semantical breakpoint location b_o for a source breakpoint b_s with respect to a set of variables is a location where each variable in the set is current.

  In the example in fig. 1 the semantic breakpoint locations for line 4 with respect to the set f, g, h are instructions 1 and 2.

In order to make clear which breakpoint mapping is used, we will use a pair (b_s, b_o) to reference a breakpoint in the following. b_o is the instruction where the object program actually has been stopped.

[3]provided that the program has not yet been started

An algorithm that does not rely on a fixed breakpoint mapping can also be used when the program is halted due to an exception. The source location the debugger should display is the statement that generated this instruction. In general, there may be no syntactical mapping between this statement and the instruction that generated the exception.

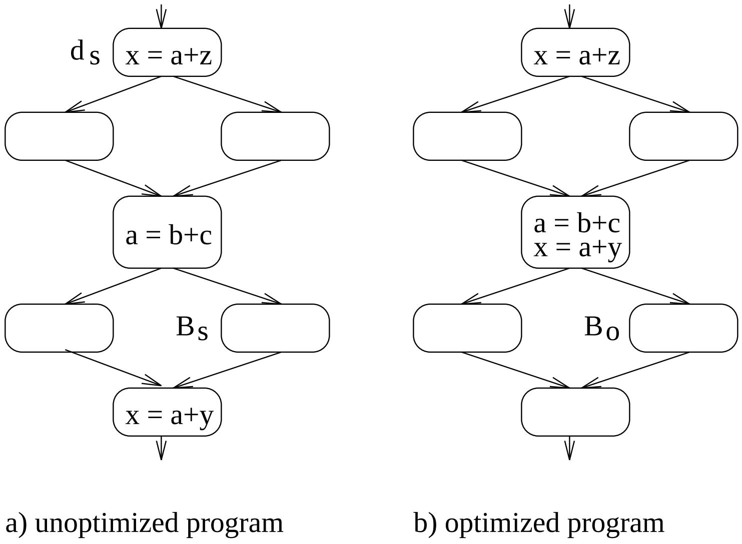

Figure 3: Scheduled program fragment

The ability to handle arbitrary pairs of source and object locations also allows for recovery, that is otherwise impossible. Fig. 3 illustrates this fact. In this code fragment, where for readability the optimized code is written in the same syntax as the source code, a percolation scheduler has moved the statement $x = a + y$. At the breakpoint (b_s, b_o) x is noncurrent. If we want to recover x by evaluating its last definition d_s, we must decide whether a and z are current at the point (d_s, b_o). So even we have started with a syntactical mapping, we now have to compute currentness for an arbitrary pair of code locations. In the example, a is noncurrent, so we cannot recover x directly. However, we may use a recursive recovery step to determine the value of a and thus also that of x.

3.4 Usage of Additional Run-Time Information

To get optimal results, the algorithm should be able to consider additional run-time information about which paths the program has taken in the past. Such information may be gathered by course grained traces or additional breakpoints invisible to the user. Since normally all possible paths have to be considered, restricting the number of paths may result in more variables being current or recoverable. In fig. 1, f is noncurrent at the breakpoint (4,11). If it is known from a trace, that the program has not been stopped in the first iteration of the loop, f can safely be stated as current.

In order to use run time information, the algorithm must be exact with respect to single definitions and paths. The definition of currency and the algorithm presented in chapter 2.4 do not have this property, as the following consideration will show. In the program in fig. 1 assume again the breakpoint (4,11). If it is known that the optimized program has taken a path ending with the instruction sequence (20,21,22,5,6,7,8,10) which corresponds to a path ending with statements (3,4,5,6,7,1,2,3) in the source program, the only reaching definition of variable y is statement 3, while the only reaching store is instruction 20. So according to the definition in chapter 2.4, variable y is current along the path the program has taken. However, the breakpoint is reached by definition 3 of the current loop iteration and by instruction 20 of the previous iteration.

In order to correctly handle these situations, the definition and the algorithm must distinguish between definitions in different loop iterations.

3.5 Handling of Pointers and Arrays

To be usable for practical programs, an algorithm determining currentness must not be restricted to simple variables, but must also be able to handle pointers and arrays correctly. Handling of arrays is important, since array accesses are the primary source for transformations enabling parallelization and vectorization.

3.6 Assurance of Currency by the Compiler

To avoid unnecessary analysis, the algorithm should be able to handle an assurance generated by the compiler, that a variable is current at some location in the code. Some compilers[4] handle pointer assignments, such as $*p = x + y$, as an implicit use of all variables the pointer can access. Therefore these variables must be current at the location given by the assignment statement and its associated instruction. Usage of such information may significantly speed up the analysis and may also lead to more precise results. Furthermore it allows for a combination between data flow analysis and the idea of inspection points: The compiler assures that all variables are current at at certain points, where then full debugging is possible. If the program stops between two inspection points, the currentness algorithm enables at least a restricted form of debugging.

4 An Algorithm to Determine Currency

In this chapter we present our algorithm to determine current variables. First we will define the program representation we use, then we will give a definition of current variables and develop an algorithm to detect them. Finally we show how pointers and arrays can be handled efficiently by this algorithm.

4.1 Program Representation

The program representation we choose are flow graphs, since the algorithm has been designed with intermediate code optimization in mind. The optimizer generates two flow graphs: the *source graph* $\mathcal{S}$ represents the original source program, the *object graph* $\mathcal{O}$ represents the optimized program. The nodes of these flow graphs are basic blocks that consist of a sequence of statements. Since the analysis only has to consider operations that change either variables or control flow (i.e. store, call and jump instructions), only those operations are included in the statement descriptions. Each statement is further characterized by the following fields:

- The statement type: we distinguish between assignments, indirect assignments, function or procedure calls and jumps.
- Assignments have an additional variable description that contains the name, address or register of the variable written. Since variables in the flow graphs always have simple types, an optional offset and the length of the written data word is used to handle assignments to structure or array components.
- For indirect assignments there is a set of definitions or stores, respectively, that may be killed by this assignment (MAYKILL). The set MUSTKILL of definitions or stores that are killed by a direct assignment and the set MUSTDEF of variables defined by an assignment are not explicitly stored, but can be determined from the variable descriptions.
- The graphs also contain the line number or code address of each statement.

[4] for instance the SUN Sparc compiler we used for our implementation

- In the source graph, the right hand side of an assignment may also be stored to enable recovery.

For we only handle intra-procedural optimizations, the representation for a whole program consists of two flow graphs for each procedure in the program. We will use the symbols B_s, S_s, E_s and B_o, S_o, E_o to reference the set of blocks, statements and control flow edges in the source or object graph, respectively. To model the effects of the code transformations performed by the optimizer, there are 3 additional relations:

- The primary code relation $\mathrm{CODE}_1 \subseteq (S_s \times S_o) \cup (E_s \times E_o)$ maps a source statement to its semantically equivalent object code instruction (*representative instruction* [CM91]). This means that a procedure call is mapped to the corresponding call instruction, a conditional branch to its corresponding (conditional) jump instruction, and an assignment to its proper store. In the latter case the relation holds only if both assignment and store always write the same values to corresponding variables. Relation CODE_1 also holds between corresponding flow edges in the source and object graphs. In this way also control flow transformations can be expressed.

- In contrast to the primary code relation, the secondary code relation $\mathrm{CODE}_2 \subseteq S_s \times S_o$ does not establish a mapping between source code and object code variables. Therefore CODE_2 also holds between an assignment d_s and a store d_o, if d_o writes to a variable that does not correspond to the one written by d_s, as long as both always write the same value.

- The last relation $\mathrm{CURR} \subseteq S_s \times S_s$ may be used by a compiler to assure that the variables possibly assigned by the statements in $\mathrm{CURR}(d_s)$ are current at the locations (d_s, d_o) with $d_o \in \mathrm{CODE}_1(d_s)$. For reasons explained in chapter 4.3 we require $\mathrm{CURR}(d_s)$ to be a subset of $\mathrm{MAYKILL}(d_s)$.

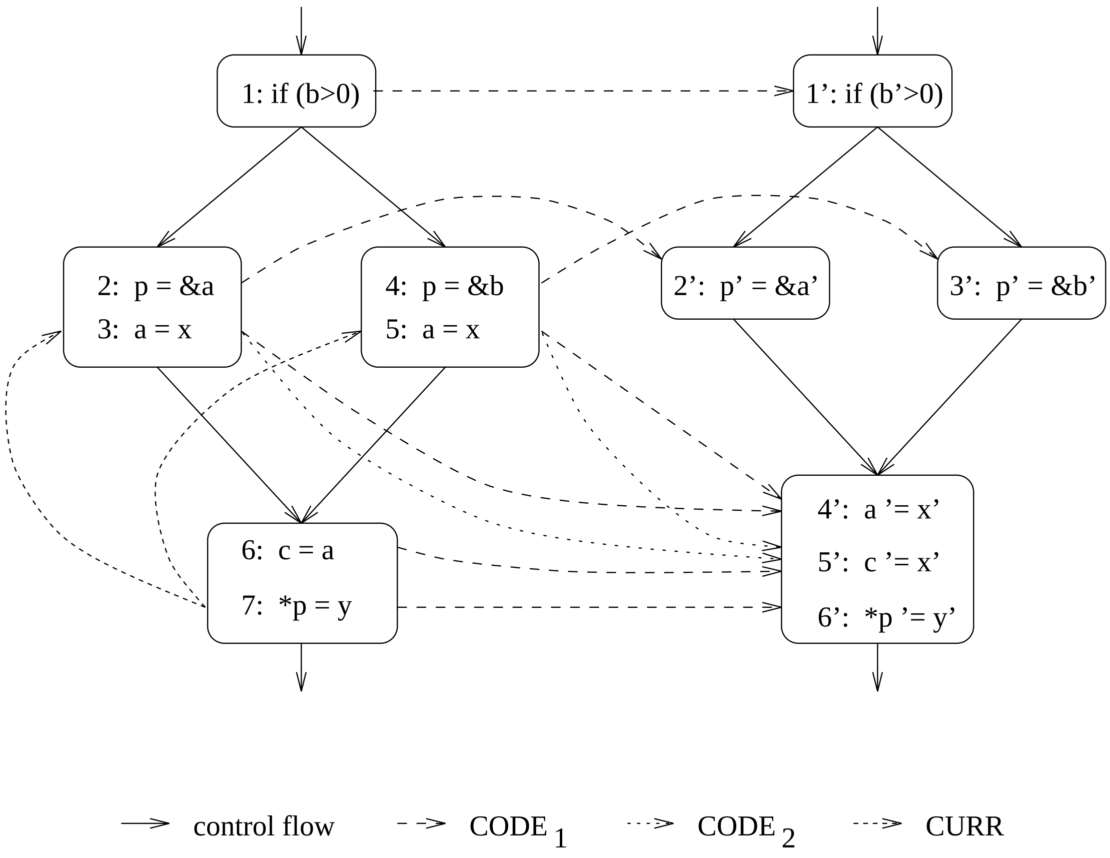

Figure 4: Relations between unoptimized and optimized flow graphs

Fig. 4 illustrates these relations. Relation CODE_2 holds between statements 3, 5 and instruction $5'$, since always the same value is written, namely that of x. However, instruction $5'$ writes that value to an object code variable that cannot be accessed by p', while the original variable a can be accessed by p. Therefore $\mathrm{CODE}_1(3,5')$ and $\mathrm{CODE}_1(5,5')$ do not hold. Including the relation CODE_2 improves the results of currency analysis, if assignments have been deleted. In the example, if p could not access a, instruction $4'$ could be deleted. However, the value of a can still be determined at the location $(7,6')$, since it is stored in c'.

In order to simplify the analysis, we assume that the flow graphs have single entry and exit nodes with a zero in-degree or out-degree, respectively. The entry node of each flow graph contains pseudo assignments for each variable used in a procedure. In this way there is at least one reaching definition for a variable along each path. The pseudo assignments to global and static variables have a corresponding pseudo store defining the variable's memory location. If the program is not optimized inter-procedurally, global and static variables must be current at procedure entry and therefore also have unique storage locations. Local (automatic) variables don't have a corresponding pseudo store instruction, since they are undefined at the beginning of a procedure.

Although the flow graph nodes are basic blocks, i.e. sequences of statements, for ease of explanation we subsequently assume a node being a single statement. If a node consists of more than one statement, additional local analysis has to be performed.

4.2 Algorithm

The main idea of the algorithm is similar to that stated in chapter 2.4: A variable v_s is current at a breakpoint (b_s, b_o), if in each execution of the program, where definition d_s of v_s should reach b_s, the corresponding store d_o reaches b_o. More formally, we define:

Definition 1: A path p in a flow graph is a sequence of edges $(<x_i, y_i>)_{i=1\ldots n}$, such that $y_i = x_{i+1}$ for $i = 1 \ldots n-1$.

Definition 2: An *execution* of a program is a pair (p_s, p_o), where p_o is the path the program did take in the object flow graph, while p_s is the path the program would take in the source flow graph, if the source code were executed with the same input data. Thus (p_s, p_o) characterizes the behaviour of both the unoptimized and the optimized program.

Definition 3: A *definition* $d_s \in \mathrm{RD}(b_s, v_s)$ is *current* at a location (b_s, b_o) with respect to an object code variable v_o, iff the following condition holds for each execution $p = (p_s, p_o)$, each instance d_s^α of d_s in p_s and each instance b_s^β of b_s in p_s:

If d_s^α writes to v_s and reaches b_s^β along p_s and b_o^δ is an instance of b_o in p_o with $b_o^\delta \simeq_p b_s^\beta$ then in p_o there is an instance d_o^γ of a store $d_o \in \mathit{code}(d_s)$ writing to v_o with $d_o^\gamma \simeq_p d_s^\alpha$ and d_o^γ reaches b_o^δ along p_o.

Definition 4: A *variable* v_s is *current* at (b_s, b_o), iff there is an object code variable v_o, such that each $d_s \in RD(b_s, v_s)$ is current at (b_s, b_o) with respect to v_o.

$\mathrm{RD}(b_s, v_s)$ is the set of *reaching definitions* of v_s at location b_s, CODE is defined as the union of CODE_1 and CODE_2. The term $d_o^\gamma \simeq_p d_s^\alpha$ denotes that instance d_o^γ semantically corresponds to instance d_s^α in execution p, i.e. in execution p both instances write the same values to corresponding variables. If d_o^γ has not moved out of a loop, $d_o^\gamma \simeq_p d_s^\alpha$ is equivalent to $\gamma = \alpha$.

The upper Greek indices are used to distinguish between instances of statements, i.e. executions of a statement in different iterations of the surrounding loops. Statement instances may be uniquely defined by the vector of iteration indices of the statement's surrounding loops [PW86]. This approach requires the program's loop structure to be known and well defined, i.e. the flow graphs must be *reducible* [Hec77]. Using index vectors and relations CODE_1, CODE_2 and CURR between statement *instances*, the above definition is applicable even in the presence of loop transformations, such as *loop interchanging, reversal* or *loop skewing*. However, since index vectors are not easily handled and our target compiler does not perform loop transformations, we decided to use the number of backward-edges on the path to a statement instance as its index. In this way, different instances of the same statement always get different indices.

In order to be able to use standard data flow analysis algorithms, we map a flow graph to a new graph, that is unrolled in a special way, such that all instances of nodes that have to be distinguished are mapped to different nodes in the new graph. For a flow graph $\mathcal{G} = (B, E)$, where B is the set of nodes (i.e. basic blocks) and E the set of flow edges, the unrolled graph is defined as $\mathcal{G}_m = (B', E')$ with $B' \subseteq B \times [-1, m+1]$ and $E' \subseteq B' \times B'$ such that $<(n_1, i), (n_2, j)> \in E'$, iff the following conditions hold:

- If $<n_1, n_2>$ is a forward edge in E, then $i = j$.
- If $<n_1, n_2>$ is a backward edge in E, then $i = j - 1$ or $i = j = -1$ or $i = j = m + 1$.

B' contains only those nodes (n, i) with

- Node (n, i) can be reached from a node $(s, -1)$ or $(s, 0)$, where s is a start node in $\mathcal{G}$.
- A node (e, m) or $(e, m + 1)$, where e is an exit node in $\mathcal{G}$, can be reached from (n, i).

The unrolled graph $\mathcal{G}_m$ contains exactly those paths from a start node of $\mathcal{G}$ to an exit node of $\mathcal{G}$ that include at least m backward edges. Fig. 5 shows how the unrolled graph looks like for $m = 1$ if the original flow graph contains two nested loops.

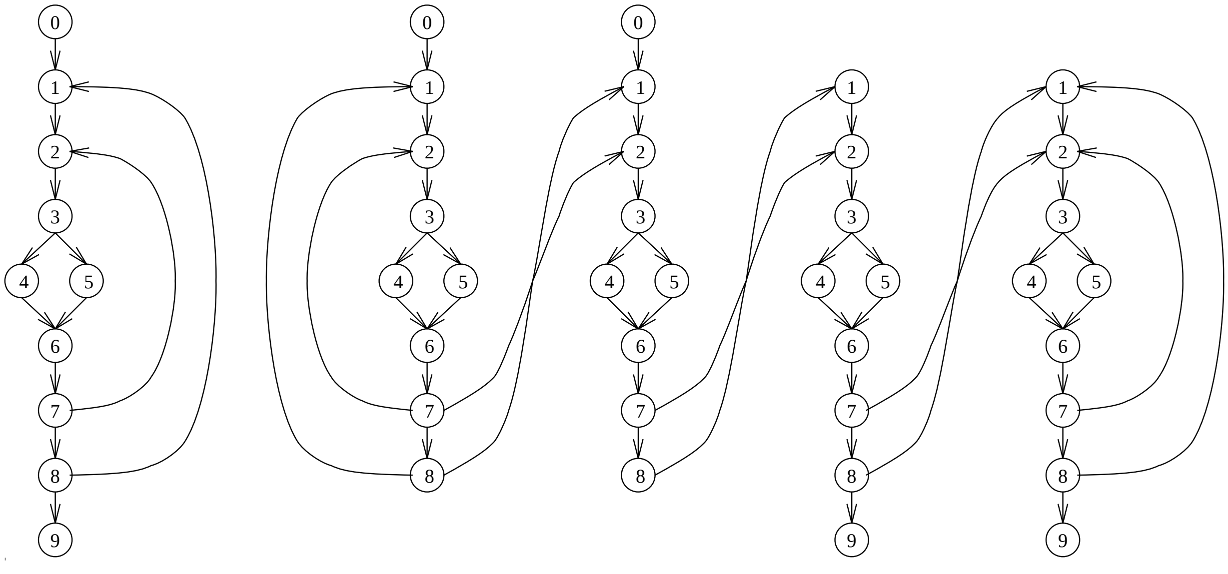

Figure 5: A flow graph $\mathcal{G}$ and its unrolled graph $\mathcal{G}_1$

With this definition, we can give a more algorithmic characterization of the term current:

Definition 3': A definition d_s is current at (b_s, b_o) with respect to an object variable v_o, iff for all $m \geq 0$ the following condition holds:

In the object graph $\mathcal{O}_m$, that has been unrolled m times, for each path p_o that is compatible with an execution $p = (p_s, p_o)$ where d_s reaches b_s along m backward edges, and for each (b_o, j) with $(b_o, j) \simeq_p (b_s, m)$ there is a store (d_o, i) with $d_o \in \text{CODE}(d_s)$ writing to v_o, $(d_o, i) \simeq_p (d_s, 0)$ and (d_o, i) reaches (b_o, j) along p_o.

The term $(a_o, i) \simeq_p (a_s, j)$ is equivalent to the following conditions:

- If there are backward edges $<x, y> \in \mathrm{CODE}(r_j)$ and $<x', y'> \in \mathrm{CODE}(r_{j+1})$ with $y \not> d_o \not> x'$, then $i = j$.
- If there is a backward edge $<x, y> \in \mathrm{CODE}(r_j)$ with $y > a_o$, then $i = \max\{k \mid (a_o, k) \text{ is in } p_o \text{ and } k \leq j\}$.
- If there is a backward edge $<x', y'> \in \mathrm{CODE}(r_{j+1})$ with $a_o > x'$, then $i = \min\{k \mid (a_o, k) \text{ is in } p_o \text{ and } k \geq j\}$.

where r_j is the j-th backward edge on path p_s[5] and $>$ is the partial order between flow graph nodes defined by $y > x$, iff there is a path from x to y containing only forward edges.

So the nodes $(n, 0)$ of $\mathcal{O}_m$ represent the instructions of the iteration where v_s is defined by b_s. The subgraph defined by nodes $(n, -1)$ models the execution before that point. Similar, nodes (n, m) represent instructions of the loop iteration containing the current instance of b_s, while the subgraph built by the nodes $(n, m+1)$ defines all subsequent iterations.

Disregarding indirect variable accesses for the moment we can restrict the unrolled object graph $\mathcal{O}_m$ by deleting edges, until only those paths in $\mathcal{O}_m$ are possible that are compatible with an execution, where d_s reaches b_s along m backward edges. Then the resulting data flow problem is very similar to the *available expressions* problem. Yet there are two difficulties: The range of m is unlimited and the relation $\simeq_p$ depends on the path in $\mathcal{O}_m$.

It can be shown that the range of m can be restricted to $[0, l_s]$ where l_s is the *loop connectedness* of $\mathcal{S}$, usually a number not greater than 2.

If we allow the algorithm to be conservative in some situations, the conditions $(d_o, i) \simeq_p (d_s, 0)$ and $(b_o, j) \simeq_p (b_s, m)$ can be replaced by path independent conditions $(d_o, i) \in \mathrm{DEF}$ and $(b_o, j) \in \mathrm{BRK}$, if the following constraints are met for all m and all $i, j \in [-1, m+1]$:

- $(d_o, i) \in \mathrm{DEF} \;\Rightarrow\; \forall p : (d_o, i) \simeq_p (d_s, 0)$
- $(b_o, j) \in \mathrm{BRK} \;\Leftarrow\; \exists p : (b_o, j) \simeq_p (b_s, m)$

So DEF is the set of instances of stores that surely correspond to d_s, while BRK is the set of instances of breakpoint locations that might correspond to b_s. One way to define DEF and BRK in $\mathcal{O}_m = (B'_o, E'_o)$ is:

$$
\begin{aligned}
\mathrm{DEF} = \; & \{(n, -1) \in B'_o \mid \text{ for all backward edges } <x, y> \text{ on paths from } (n, -1) \text{ to some } (n', 0)\text{: } n < y\} \\
\cup \; & \{(n, 0) \in B'_o\} \\
\cup \; & \{(n, i) \in B'_o \mid i \in [1, m+1] \wedge \text{ for all backward edges } <x, y> \text{ on paths from some } (n', 0) \text{ to } (n, i)\text{: } n > x\}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{BRK} = \; & \{(n, j) \in B'_o \mid j \in [-1, m-1] \wedge \text{ there is a path from } (n, j) \text{ to some } (n', m) \text{ that contains only backward edges } <x, y> \text{ with } n < y\} \\
\cup \; & \{(n, m) \in B'_o\} \\
\cup \; & \{(n, m+1) \in B'_o \mid \text{ there is a path from some } (n', m) \text{ to } (n, m+1) \text{ that contains only backward edges } <x, y> \text{ with } n > x)\}
\end{aligned}
$$

With these definitions, the following algorithm can be used to determine current variables. Due to the simplification above, this algorithm is conservative in the case of loop invariant code motion, i.e. it may return that a variable is noncurrent, though it is indeed current.

[5] Since we assume the compiler not performing loop transformations, for any execution (p_s, p_o) p_s and p_o contain the same sequence of backward edges

Input: Flow graphs $\mathcal{S}$ and $\mathcal{O}$, a source variable v_s and a location (b_s, b_o).

Output: The set $\mathrm{CURRENT}(b_o)$ of object variables, such that v_s is current at (b_s, b_o) with respect to each $v_o \in \mathrm{CURRENT}(b_o)$. If v_s is noncurrent, $\mathrm{CURRENT}(b_o) = \emptyset$.

Algorithm:

$$\mathrm{OVAR} = \bigcap_{d_s \in \mathrm{RD}(b_s, v_s)} \Big(\bigcup_{d_o \in \mathrm{CODE}(d_s)} \mathrm{VAR}(d_o) \Big)$$

For all nodes n in $\mathcal{O}$

$\mathrm{CURRENT}(n) = \mathrm{OVAR}$

For $m = 0$ to l_s

Determine the set $\mathrm{RD}_m(b_s, v_s)$ of all definitions d_s that reach b_s along m backward edges.

For each $d_s \in \mathrm{RD}_m(b_s, v_s)$:

$\mathrm{ODEF} = \{(d, j) \in \mathrm{DEF} \mid d_o \in \mathrm{CODE}(d_s)\}$

Unroll $\mathcal{O}$ m times and restrict $\mathcal{O}_m$ such that only paths corresponding to reaching paths from d_s to b_s are possible.

Solve the data flow equation for $\mathrm{AVAIL}(n, i)$ in $\mathcal{O}_m$:

$$\mathrm{AVAIL}(n, i) = \bigcap_{(p,j) \in \mathrm{PRED}(n,i)} \Big[\big[\mathrm{AVAIL}(p, j) - \mathrm{MAYKILL}(p)\big] \cup \big[\mathrm{VAR}\big(\mathrm{MUSTDEF}(p) \cap \mathrm{ODEF}\big) \cap \mathrm{OVAR}\big] \Big]$$

For all $(n, i) \in \mathrm{BRK}$:

$\mathrm{CURRENT}(n) = \mathrm{CURRENT}(n) \cap \mathrm{AVAIL}(n, i)$

The set of definitions $\mathrm{RD}_m(b_s, v_s)$ of v_s that reach b_s along m backward edges can be computed by a modified iterative algorithm for reaching definitions. If definitions of v_s are only taken into account in the first iteration of the algorithm, then the intermediate result $\mathrm{RD}(n, v_s)$ in the m-th iteration is the set of definitions that reach n along $m - 1$ backward edges.

Restriction of $\mathcal{O}_m$ can be done by deleting those edges $<(x, i), (y, j)>$ that correspond to an edge $<x, y>$ in $\mathcal{S}$ with $y \in \mathrm{RD}_j(b_s, v_s)$, but $x \notin \mathrm{RD}_i(b_s, v_s)$, i.e. $<x, y>$ cannot be part of a reaching path from d_s to b_s.

The data flow equation can be solved by a simple iterative data flow analysis algorithm, where some optimizations are possible due to the special structure of $\mathcal{O}_m$. VAR is a function mapping instructions to the object variables that may be written by them. OVAR is the set of all object variables that have to be considered in the analysis.

Instead of iterating m from 0 to l_s, it is also possible to iterate until $\mathrm{CURRENT}(n)$ is stable for all nodes n.

Returning to our first example we show how the algorithm computes currentness of f at the breakpoint $(4, 12)$. Fig. 6 shows source and object flow graphs for this example. As reaching definitions we get $\mathrm{RD}(4, f) = \{0, 4, 14\}$, OVAR is therefore set to $\{l6\}$. For $m = 0$, $\mathrm{RD}_0(4, f) = \{0\}$, and $\mathrm{ODEF} = \{(0, 0)\}$. The relevant part of the solution for $\mathrm{AVAIL}(n, i)$ is $\mathrm{AVAIL}(12, -1) = \mathrm{AVAIL}(12, 0) = \mathrm{AVAIL}(12, 1) = \emptyset$. Since $(12, 0) \in \mathrm{BRK}$, $\mathrm{CURRENT}(12)$ is set to $\emptyset$. For $m = 1$, $\mathrm{RD}_1(4, f) = \{4, 14\}$. For both $d_s = 4$ and $d_s = 14$ $\mathrm{ODEF} = \{(2, -1), (2, 0)\}$ and $\mathrm{AVAIL}(12, -1) = \mathrm{AVAIL}(12, 0) = \mathrm{AVAIL}(12, 1) = \mathrm{AVAIL}(12, 2) = \{l6\}$. This result does not change $\mathrm{CURRENT}(12)$ any more, so f is noncurrent at $(4, 12)$. If we knew that the program has not been stopped in the first iteration of the loop, then the pseudo assignment 0 cannot be a reaching definition, therefore $\mathrm{RD}_0(4, f) = \emptyset$. In this case the result is $\mathrm{CURRENT}(12) = \{l6\}$, i.e. f is current.

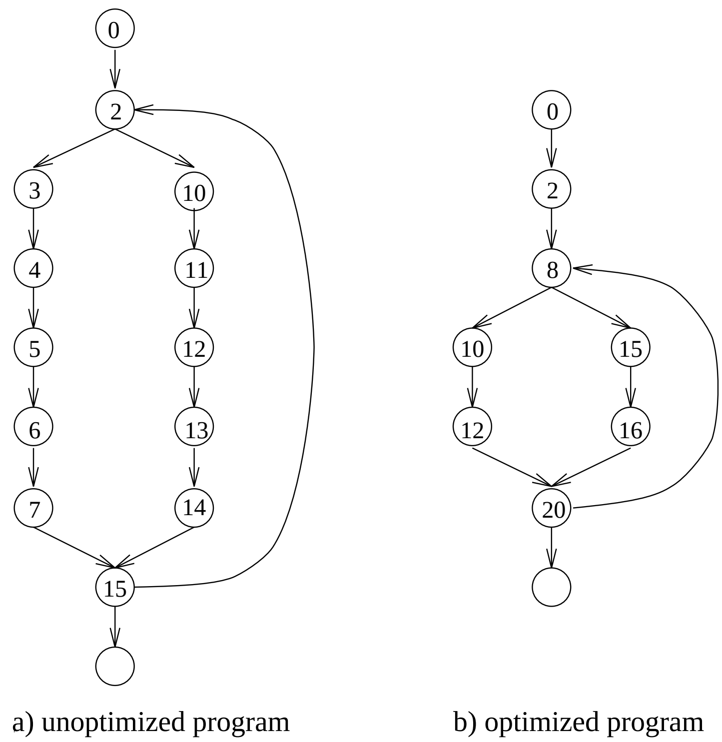

Figure 6: Flow graphs of the program fragment in fig. 1

4.3 Handling of Indirect Assignments

Our definitions of current variables also are applicable to programs containing indirect assignments to variables, i.e.. statements that may or may not write to a given variable. However, the condition that (d_o, i) reaches (b_o, j) along a path that is compatible with an execution $p = (p_s, p_o)$ where d_s reaches b_s not only restricts the paths in $\mathcal{O}_m$ to be examined by the algorithm, but also puts a restriction on the indirect stores on these paths. If, for example, path p_s contains an indirect assignment between d_s and b_s, then this assignment cannot have written to v_s. Therefore, the corresponding store on p_o also cannot write to the object variable that corresponds to v_s.

So we not only have to delete edges in $\mathcal{O}_m$, but also have to remove those instances of indirect stores that correspond to indirect assignments on a reaching path between d_s and b_s. In analogy, since we assume that d_s actually writes to v_s, we also must assume that the proper store actually writes to the object variable corresponding to v_s. Furthermore, if the variable v_s may have received its value by an indirect assignment, i.e. there is an indirect assignment $d_s \in \mathrm{RD}(b_s, v_s)$, we must only consider those object variables v_o that correspond to v_s. In the algorithm we have therefore to replace CODE by CODE_1 and to change the set ODEF, that now must only contain those v_o that correspond to v_s.

Although the algorithm can correctly handle indirect assignments with those modifications, there is still a problem: The set $\mathrm{RD}(b_s, v_s)$, and therefore also $\mathrm{RD}_m(b_s, v_s)$ may become very large. Since the complexity of the algorithm is linear in terms of the magnitude of $\mathrm{RD}(b_s, v_s)$, time for analysis would rapidly grow. Fortunately, since a compiler has to consider all executions of a program, where in some a variable v_s is actually written by an indirect assignment i_s, while in others v_s is not written by i_s, except for some weird cases v_s must be current at the locations (i_s, i_o) with $i_o \in \mathrm{CODE}_1(i_s)$. Some compilers enforce this by regarding an indirect assignment as an implicit use of all possible destination variables. The compiler can use the relation CURR to inform the debugger about this situation. Using this relation we can define the set of *independent reaching definitions* as solution of the following data flow equations:

$$\mathrm{ID}(n) \quad = \quad \bigcup_{p \in \mathrm{PRED}(n)} \Big[\Big(\mathrm{ID}(p) - (\mathrm{MUSTKILL}(p) \cup \mathrm{CURR}(p)) \Big) \cup \mathrm{CURDEF}(p) \Big]$$

$$\begin{aligned}\mathrm{CURDEF}(n) &= \{d \mid d \text{ is in Block } n \wedge \forall x \text{ in Block } n, x > d : d \notin (\mathrm{MUSTKILL}(y) \cup \mathrm{CURR}(y))\} \\ \mathrm{ID}(n, v) &= \{d \in \mathrm{ID}(n) \mid v \in \mathrm{VAR}(\mathrm{MAYKILL}(d))\}\end{aligned}$$

Loosely spoken, a statement d possibly writing to v is in $\mathrm{ID}(n, v)$ if there is a path from d to n, along which v may reach n and v is not assured to be current at an indirect assignment between d and n. Forcing CURR to be a subset of MAYKILL ensures that on each path there is exactly one statement $d \in \mathrm{ID}(n)$.

It can be shown that it suffices to execute the inner loop of our algorithm for each $d_s \in \mathrm{ID}_m(b_s, v_s)$, where ID_m is defined in analogy to RD_m. Since $\mathrm{ID}_m(b_s, v_s)$ may be a significantly smaller set than $\mathrm{RD}_m(b_s, v_s)$, assurance of currency by the compiler reduces the cost of analysis in the debugger.

5 Compiler Implementation

As a basis for the implementation on the compiler side, we used the SUN C-Compiler Version 1.0, whose source code has been kindly provided by SUN Microsystems. The compiler, when performing optimizations, translates a source program in four phases: The parser *ccom* translates the C-program text into an intermediate representation that is then restructured by the machine independent optimizer *iropt*. The optimizer passes the intermediate representation to the code generator *cg*, that generates assembly code. Finally the assembler *as* performs some machine dependent optimizations and produces an object file.

Since the main purpose of our implementation is to show that debugging of optimized code is viable, and in order to keep implementation effort within a reasonable range, we have selected a set of optimizations, for which debugging should be enabled. To avoid changing the assembler, its optimization facility has been explicitly turned off. Currently there is also no support for optimizations that need more complex mappings as *loop unrolling* or *procedure integration* (also called *inlining*). Techniques concerning these problems have been previously developed in [Zel84].

Generation and maintenance of the following information has been embedded in the compiler in order to support the algorithm presented in chapter 4:

- The traditional line table, mapping source lines to object code addresses.
- The traditional symbol table, containing type information of variables.
- A flow graph of each function in the source code.
- The corresponding flow graph of each function in the object code.
- A mapping between source code statements and object code instructions.
- Augmented unique names for symbols declared in anonymous blocks.
- Additional information about side effects of accessing arrays or variables through pointers.
- Recovery expressions for variables, parameters, function calls and branches.

Source Flow Graph

The source flow graph, that represents the structure of basic blocks in a function before the optimization phase, can be easily generated, since the sets of a block's predecessors and successors is already part of the intermediate code representation used by *iropt*.

Each basic block is associated with a set of relevant statements[6]. A statement is marked as relevant by *ccom*, if it may change the state[7] of the program. This class of statements includes direct and indirect definitions of variables, conditional branches and function calls. The source flow graph is issued by *iropt* immediately after computing the alias information.

Object Flow Graph

The object flow graph of a procedure represents the basic block structure after all optimizations have been applied to the code. All blocks in the object graph are labelled with the number of source blocks, from which they originated. If there is a block in the source graph, whose number does not occur in any object block's label indicating its origins, this source block has been eliminated. This can happen if *unreachable code elimination* has been applied to the code.

To realize this block mapping only small changes were necessary in the *block* structure of the intermediate representation. For each object block there is a new component that contains a list of the source blocks that are the origin of this block. To maintain a correct mapping, this list must be updated after each block-manipulating optimization.

Beside this mapping there is another mapping for single statements, which is discussed below. Since this mapping needs additional labels to be inserted into the assembly code, which is not easily possible within *iropt*, the object flow graph is written later by the code generator *cg* before assembly code generation starts.

Mapping Between Source Statements And Object Code Instructions

Since the compiler is split into three programs (*ccom*, *iropt* and *cg*), the mapping between source statements and machine instructions has also been divided into three parts:

1. A mapping between source line numbers and the statements that have been generated for this line before optimization.
2. The actual mapping between statements before optimization and their corresponding statements after optimization.
3. A mapping between statements after optimization and the addresses of their representative assembly instructions.

Each time the parser reads a new line of the program to translate, an intermediate instruction is generated to keep track of the current line being processed. Therefore all relevant statements of a source line are located between two such statements in the intermediate representation. So the first part of the mapping can be computed by scanning the intermediate representation prior to optimization.

In the traditional debugging information these statements are used to build the *line table*, a mapping between each source code line and the first assembly instruction generated from it. This table is still generated and is used as a breakpoint mapping, until more advanced mappings have been implemented.

The mapping between statements before and after optimization, respectively, is realized similar to the basic block mapping, i.e. the optimizer maintains a list of origins for each statement. If statements are moved, duplicated or deleted, the mapping is thus updated automatically without further actions. However, for *common subexpression elimination* and *copy propagation* the update of the mapping is more complex. As we have shown in chapter 4, the mapping actually is consists two mappings CODE_1 and CODE_2.

[6] Also called *triples*

[7] Either control flow or a user defined variable

In the case of copy propagation these mappings are built in the following way: A source definition of a variable a is mapped to its object counterpart, which is a store into the corresponding variable a', via CODE_1, while all copies $b = a$ that can be eliminated are mapped to this store via CODE_2. Taking into account the def-use chains, that are already built by *iropt*, for each copy statement $s_c : b = a$ that is eliminated, s_c is placed into the appropriate origin list of each definition of a reaching s_c.

Common subexpression elimination is handled in a similar way, if the right hand side of a statement consists entirely of the common subexpression. In this case the statement is mapped via CODE_2 to the store into the temporary variable holding the expression's value.

The third part of the statement mapping is implemented by mapping each relevant post optimization statement to a unique label. When assembly code is generated for a relevant statement, a **.stabs** directive is issued that contains the statement number in its text field and the label in its value field. Since a statement may produce more than one assembly instruction, a *representative instruction* is determined, that performs the relevant state change. The label is then placed in front of this instruction. Assembler and linker then translate the label into an absolute code address.

Additional Side Effect Information

When performing data flow analysis to determine current variables, the debugger must also take into account that a variable may be accessed through an alias, i.e. an indirect assignment. Therefore additional alias information is needed for each indirect store. This information is already being computed and stored in the intermediate representation, if *iropt* is invoked with the -O4 option (full optimization). For each statement there is a list of variables, that may be written by this statement. This list is included into both the source and object flow graphs.

The optimizer also computes whether two variables' address ranges overlap, i.e. a store to a variable may also change other ones. This information is passed to the debugger by including a list of overlapping variables in each variable description in a flow graph.

Recovery Expressions

If a variable's value is not current at an arbitrary point in the program, it may be recovered using the recovery expression that is generated for each direct or indirect definition by traversing the expression tree representing the definition's right hand side and translating it into postfix form. There are two cases where no recovery expression is provided. First, if a function call occurs in the right hand side of an assignment, recovery is impossible due to the side effects the function may have. Second, when the right hand side of a definition gets more complex, the chance for successful recovery decreases while the storage needed for the recovery expression increases. Therefore for complex right hand sides no recovery expression is generated.

Recovery information is also generated for function parameters and conditional branches. For each possible direction of a branch, the proper condition is stored as a recovery expression. The data flow analysis algorithm may use this information to determine control flow by evaluating these expressions.

Interface Between Compiler and Debugger

In order to be compatible with the traditional interface between compiler and debugger, all information is passed using **.stabs** pseudo assembler instructions. In this way the information is also correctly handled by an unmodified linker. The format of a **.stabs** instruction is

```
.stabs "name", type, undefined, description, value
```

To distinguish the new debugging information from the old one, four new **.stabs** types have been introduced, that are used in the following manner:

Type		Relocation	Used for
N_DEBUG	0x90	none	flow graphs
N_DBTEX	0x94	text	third part of statement mapping
N_DBDAT	0x96	data	initialized static variables
N_DBBSS	0x98	stack	uninitialized static variables

A brief example will give an impression how the debugging information is coded. Consider the following program fragment:

```
if (x > 20) p = &a
else        p = &b;
*p = 2;
```

The **.stabs** instruction for the statement $*p = 2$ will look like

```
.stabs "S 1 1 37 V26 C2 P V33 V27",0x90,0,0,0
```

The name field must be interpreted like this: 'S' identifies the entry as part of the source flow graph, the string '1 1 37' shows that this is an entry for an indirect assignment statement with the number 37, 'V26' represents the pointer variable used, 'C2' is the recovery expression (the constant 2). The variables following the 'P' are those that can be accessed by this indirect assignment (i.e. a and b).

6 Results and Future Work

As a criterion for the programming expenditure needed to support the proposed technique by a compiler, we give the number of the lines of code that have been added to the SUN compiler:

Most changes were necessary in the optimizer *iropt*, since here the largest part of the additional information is generated. In this module 1096 lines of code were added. In the compiler itself 329 lines have been added to the machine independent parts, the machine dependent part has been extended by 127 lines. More work has to be done in the code generator *cg* in order to make it work with *iropt*'s new information. Here 604 lines were added to the existing code. The whole compiler consists of 93705 lines of code, so less than 3 percent of the code had to be changed.

First measurements with the modified compiler have shown that compilation time increases by 30 to 50 percent, which is mainly due to the time needed to store the debugging information to disk. This information increases the size of the binary output file by a factor of about 10, while the traditional debugging information increases file size only by a factor of two. However, an optimized representation may largely reduce the storage requirements.

The algorithm presented in chapter 4 has also been implemented in a first version, that works with flow graphs and code mappings generated by hand. The time needed to detect currentness of a variable heavily depends on the number of reaching definitions. We have therefore done some statistics by evaluating the def-use chains of some representative programs, namely our modified C-compier, a part of the programming environment TOPSYS and a FORTRAN program for computational fluid dynamics. It turned out, that of over 200.000 uses of variables less than 9 percent have more than three reaching definitions. For these small numbers the response time is about one second on a SparcStation 2. However, the implementation is yet not well optimized and performs a certain amount of redundant computations.

We plan to perform more measurements with real programs, when the algorithm can work on the output of the modified compiler and has been integrated into the debugger DETOP, that is part of the parallel programming environment TOPSYS [BB91, BBB$^+$91]. We also hope to gain some experience then about the percentage of current and recoverable variables in optimized programs. Further work is also done in algorithms to determine syntactical and semantical breakpoint mappings.

References

[BB91] T. Bemmerl and A. Bode. An Integrated Environment for Programming Distributed Memory Multiprocessors. In A. Bode, editor, *Distributed Memory Computing, 2nd European Conference, EDMCC2*, volume 487 of *Lecture Notes in Computer Science*, pages 130–142, München, April 1991. Springer-Verlag.

[BBB+91] T. Bemmerl, A. Bode, P. Braun, O. Hansen, T. Treml, and R. Wismüller. The Design and Implementation of TOPSYS. SFB-Bericht 342/16/91 A, Technische Universität München, July 1991.

[BHS92] G. Brooks, G. J. Hansen, and S. Simmons. A New Approach to Debugging Optimized Code. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27(7) of *ACM SIGPLAN Notices*, pages 1–11, San Francisco, California, June 1992.

[CM91] M. Copperman and C. E. McDowell. Debugging Optimized Code Without Surprises. In *Proceedings of the Supercomputing Debugging Workshop 1991*, Albuquerque, New Mexico, November 1991.

[CMR88] D. S. Coutant, S. Meloy, and M. Ruscetta. DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23(7) of *ACM SIGPLAN Notices*, pages 125–134, June 1988.

[Cop92a] M. Copperman. Debugging Optimized Code Without Being Misled. Unveröffentlichtes Manuskript, March 1992.

[Cop92b] M. Copperman. Debugging Optimized Code Without Being Misled. Unveröffentlichtes Manuskript, July 1992.

[Gup90] R. Gupta. Debugging Code Reorganized by a Trace Scheduling Compiler. *Structured Programming*, 11(3):141–150, 1990.

[HCU92] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27(7) of *ACM SIGPLAN Notices*, pages 32–43, San Francisco, California, June 1992.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs*. The Computer Science Library. Elsevier North-Holland, New York, 1977.

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

[Mye91] T. Myers. Debugging at the National Security Agency: "'The State of the Use Message"'. In *Proceedings of the Supercomputing Debugging Workshop 1991*, Albuquerque, New Mexico, November 1991.

[PS88] L. L. Pollock and M. L. Soffa. High-Level Debugging with the Aid of an Incremental Optimizer. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, volume II, Software Track, pages 524–532, 1988.

[PW86] D. A. Padua and M. J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[Str91] L. V. Streepy. CXdb - A New View On Optimization. In *Proceedings of the Supercomputing Debugging Workshop 1991*, Albuquerque, New Mexico, November 1991.

[Zel83a] P. T. Zellweger. An Interactive High-Level Debugger for Control-Flow Optimized Programs. In *Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on High-Level Debugging*, volume 18(8) of *ACM SIGPLAN Notices*, pages 159–171, August 1983.

[Zel83b] P. T. Zellweger. An Interactive High-Level Debugger for Control-Flow Optimized Programs. Technischer Bericht CSL-83-1, Xerox Corporation, Palo Alto Research Center, Palo Alto, California, January 1983.

[Zel84] P. T. Zellweger. Interactive Source-Level Debugging of Optimized Programs. Technischer Bericht CSL-84-5, Xerox Corporation, Palo Alto Research Center, Palo Alto, California, May 1984.

[ZJ91] L. W. Zurawski and R. E. Johnson. Debugging Optimized Code with Expexted Behaviour. Unveröffentlichtes Manuskript, April 1991.