OPTVIEW: A New Approach for Examining Optimized Code

Caroline Tice
cmtice@cs.berkeley.edu
University of California, Berkeley

Susan L. Graham
graham@cs.berkeley.edu
University of California, Berkeley

Abstract

The task of mapping between source programs and machine code, once the code has been optimized and transformed by a compiler is often difficult. Yet there are many instances, such as debugging optimized code or attributing performance analysis data to source lines, when it is useful or necessary to understand at the source level what is occurring in the binary. The standard approach has been for tools to attempt to map directly from the optimized binary to the original source. Such mappings are often fragile, and sometimes inaccurate or misleading. We suggest an alternative approach. Rather than mapping directly between the original source and the binary, we create a modified version of the source program, still recognizable, but updated to reflect some of the effects of optimizations, thus facilitating mapping from the binary. We have implemented a tool, Optview, to demonstrate and test these ideas.

1 Introduction

It is the job of compilers to translate programs. A compiler takes a program written in one language, and outputs a semantically equivalent program written in a different language. Some compilers perform a simple, straight translation from one language to the other, while others, in addition to translating, perform various optimizing transformations, resulting in a more efficient final program. Although the optimized program exhibits the behavior expected of the original program, its machine instructions rarely correspond directly to the original program statements. This creates difficulties for debugging and analysis applications that have to reconcile these two different programs. Such applications often have useful and pertinent information about the optimized program which they attempt to convey to the programmer in terms of the original program. Unfortunately it is frequently the case that the transformed program is so far removed from the original that there is *no* transparent mapping between them, thus leading to confusing or inaccurate results.

Much work in debugging has been done to attempt to create exact mappings from the optimized machine instructions to original source statements ([1, 3, 6, 7, 11, 12]). *Transparent* debugging of optimized code, using such mappings, breaks under certain not uncommon conditions, at which point the debuggers fall back on the "truthful" behavior, telling the user the debugger cannot do or find what the user wants. This is less than ideal from the user's point of view.

We have been investigating the problem of understanding optimized code in terms of the original program, and we have concluded the wrong question is being asked. The question up to now has been how to exhibit a transparent mapping between the original and the transformed programs. A better question is how can we explain the correspondences between the transformed program and the original program. A fine-grained mapping directly back to the original source program may not be necessary. Rather it may suffice to derive a modified version of the original program, recognizable and understandable by someone familiar with the original program. This modified program would correspond more closely to the optimized binary, thus allowing for a more straightforward, robust, and correct mapping mechanism.

There are many tasks for which it would be useful to see such modified source code. For instance it would be of interest to programmers who want to understand how the optimizer transforms their programs. Performance data resulting from dynamic program analyses could be more accurately attributed using the modified source code. Compiler writers who are debugging the optimizer might find it useful to see the optimizations reflected at the source level. Programmers who are debugging optimized code may also find it useful to see the modified source code. Finally modified source code can be used as a teaching tool in computer classes studying compiler optimizations and program transformations.

To test our ideas we have built a tool, Optview, which generates such modified versions of original programs. Optview takes C programs as input. It works closely with the compiler, and outputs C programs modified to show the effects of various optimizations.

The rest of this paper is organized as follows. In Section 2 we give a more detailed explanation of what we mean by modified source code. Section 3 discusses some of the decisions involved in designing a tool to generate modified source code. Section 4 describes Optview in detail. Section 5 discusses some related work and in Section 6 we present our conclusions.

2 Explanation of Modified Source Code

Although the phrase "modified source code" could be applied to source code that has undergone *any* set of modifications, our concern is primarily with user-visible structural changes. The relevant modifications are obtained by applying a series of transformations to a program. These transformations can include reordering the code, rewriting expressions, replacing language constructs with different but equivalent constructs, duplicating code, or deleting code. Since the transformations that are applied to the original program to obtain the modified source code reflect the optimizations performed by the compiler, the modified source code is a partially optimized version of the source program.

To make the transformations visible, the original source language may be extended by pseudocode, and comments may be inserted into the modified source code for explanatory purposes. If pseudocode is used in the modified source, that code will not be compilable. Since its purpose is to convey information, serving as an intermediary between the original source code and the target code, non-compilability is not a problem. In our work on C, we have not found a need for pseudocode. The greatest difficulty in creating modified code that compiles has been to get the types and declarations correct.

A natural question at this point is why not use reverse engineering techniques to obtain an optimized source program directly from the optimized binary [2, 4, 9, 10]. The reason for not doing so is that the new program must be recognizable and understandable by someone familiar with the original program. Due to the loss of some types of control flow data, as well as the nature of certain optimizations such as software pipelining and loop optimizations, code generated by reverse engineering is likely to be unfamiliar and hard to comprehend. If the user cannot understand how the modified code relates to the original program, then the user will have difficulty in determining what meaning and significance to attach to it or how to use it. By starting from the original program and gradually transforming it we can retain all the parts of the original program that were not involved in the compiler optimization process, such as comments, declarations, or pre-compiler directives, as well as high level constructs such as structured control flow statements, which get lost in the compilation process. By keeping these features of the original program, the modified source code is more easily recognizable. Also, by starting from the original program, irrelevant information about the optimizations can be suppressed. Using reverse engineering, one has no such option.

Not all optimizations can be expressed at the source level, nor is it desirable to do so. Fine-grained instruction scheduling is an example of such an optimization. Another example is register allocation and spilling. If the modified source code were required to express all the effects of all optimizations, assuming that were possible, the final result would be very similar to the assembly code, thus defeating the purpose of using the source language. This raises two important questions: which optimizations should be considered when deriving the modified source code, and how can the effects of these optimizations be reflected in the source language. First we will address the choice of optimizations. Showing the effects of optimizations is discussed in Section 3.

When deciding which optimizations to reflect in the modified source code, one should keep in mind its purpose. The modified source code shows the user what is happening in the binary, but *at the source level.* The user wants to know which source statements are executing and when they are executing. The user needs an accurate idea of which variables exist at any given time and what values they contain. Users need not see low level machine specific optimizations. Therefore the optimizations of interest are those that visibly affect source code constructs, namely those that eliminate source code, move source statements, or change the form of source statements (e.g. altering expressions to use different constants, variables, or operators). One needs to be careful in considering optimizations that change the form of source statements. Some changes, such as replacing multiplication with register shifts, are probably not relevant, while others, such as using a different constant or variable in an expression, are. The optimizations that we have focused on initially in Optview are code motion, coarse-grained instruction scheduling, common subexpression elimination, partial redundancy elimination, copy propagation, and dead code elimination.

There are two basic requirements the modified source code must meet in order to be useful:

- It must be recognizable and understandable by someone familiar with the original program.
- It must correspond closely enough to the optimized binary to allow for a consistent and meaningful mapping between the two.

These two requirements are in direct conflict with each other. One of the difficult and interesting research issues is to find an appropriate balance between them. The more one modifies the original program to correspond to the binary, the less recognizable and understandable it is likely to become. One can imagine a continuum of transformed programs. At one end of the continuum is the original source code, and at the other end is a program obtained by reverse engineering the optimized binary. Our concept of modified source code falls somewhere between the two – the exact point on the continuum will vary depending on the intended use and audience for the modified program. If writing a tool for debugging optimizers, for example, one might focus more on close correspondence to the optimized binary, at the expense of recognizability. For indicating performance data, it might be more appropriate for the modified code to remain closer to the original source.

3 Generating Modified Source Code

During the optimization process, the source program is first translated to an internal form, and then portions of the internal representation are moved, duplicated, separated, eliminated, or altered. In order that the modified source code be recognizable to someone familiar with the original program, fragments of the original program are used as much as possible to construct the modified source code. The modified source code may contain original source statements that have been reordered, modified slightly, or split apart. It may also contain new statements inserted to make particular optimization effects explicit. It might contain fragments of pseudocode designed to explain important optimization effects which cannot be adequately described using constructs in the source language. In addition, the modified source code must indicate original source statements that have been eliminated.

3.1 Reordering source statements

An essential requirement for the modified source code is that the order in which statements occur accurately reflect the order in which the statements will be executed. This in turn requires knowledge about how the compiler rearranged the statements and some mechanism for identifying source statements in the optimized internal representation. In Optview we introduce *key instructions* for this purpose (see Section 4.1). The compiler tags key instructions in the binary for each source statement that was not eliminated. The order in which these key instructions occur in the binary then determines the order in which their corresponding source statements should occur in the modified source code.

```
(a) Original Code

  num = y;           /* num == y          */
  i = 3 * num - 17;  /* i   == 3 * y - 17 */
  num++;             /* num == y + 1      */

(b) After Reordering (Wrong)

  num = y;           /* num == y                */
  num++;             /* num == y + 1            */
  i = 3 * num - 17;  /* i   == 3 * (y + 1) - 17 */
                     /*     == 3 * y - 14       */

(c) Reordered & Updated (Correct)

  num = y;           /* num == y          */
  num++;             /* num == y + 1      */
  i = 3 * num - 20;  /* i   == 3 * y - 17 */
```

Figure 1: Reordering & Updating Code

3.2 Modifying source statements

To show the effects of certain optimizations, or to preserve the semantics of the program, some original source statements may need to be modified. Figure 1a shows a few lines of code from a source program. After these lines have completed executing, the variable i should contain the value of "3 * y - 17" and the variable num should contain the value of "y + 1". During the optimization process, the order of these calculations was changed, as shown in Figure 1b. To preserve the semantics of the original program it is necessary to change the constant from 17 to 20, to account for the fact that num is now incremented *before* the value of i is calculated. Figure 1c shows the correctly updated code.

3.3 Inserting new code

There are transformations such as common subexpression elimination for which new statements must be added to the source code to illustrate the optimization. For example, an assignment statement might be inserted at the point where the subexpression is evaluated, assigning the subexpression to a newly generated variable.

This new variable then can be substituted throughout the modified source wherever the original subexpression evaluation is eliminated by the compiler.

3.4 Eliminating source code

A standard feature in most optimizing compilers is elimination of dead code. The modified source code must make clear which source statements were eliminated by the compiler. There are many ways to do that. The particular representation chosen is immaterial so long as it conveys the information to the user.

3.5 Splitting apart statements

High-level, powerful source languages often contain single constructs that embody multiple pieces of functionality. The optimizer may scatter the functionally separate pieces of such a construct widely throughout the optimized binary. Since the task at hand requires that the modified source code accurately reflect the location and order in which these functional events occur in the binary, it becomes necessary for the modified source code to split apart these constructs. A simple example should clarify this point. Figure 2a shows a common C construct, the `for` statement. The `for` statement header contains three parts: the initialization statements, the loop test, and the increment statements. In order to allow the modified code enough flexibility to mirror the order of events in the binary, such a construct needs to be rewritten as simpler constructs, each embodying a single piece of functionality. Figure 2b shows the same `for` loop, rewritten as multiple C statements. The modification allows these various statements to be moved and reordered as necessary in the modified source code. When designing a tool to generate modified source code for a particular language, one needs to carefully consider which language constructs need to be broken down and split apart in this manner. Ideally whenever such a multifunctional construct needs to be rewritten, the source language will contain simpler constructs which can be used for this purpose. If that is not the case, pseudocode can be used to represent the pieces of functionality that need to be separated. Figure 2c shows the same `for` statement split apart and represented with pseudocode.

3.6 Creating pseudocode

Earlier we suggested augmenting the original source language with pseudocode when generating the modified source code. For the sake of simplicity and comprehensibility, the creation of pseudocode should be kept to a minimum. However there are situations in which it may be necessary. For example to show strength reduction in FORTRAN, one would need a mechanism for representing pointers. While some versions of FORTRAN support extensions that allow for the representation of pointers, others do not. An appropriate use of pseudocode would be to show pointer references in FORTRAN in this case. We do not advocate ad hoc creation of pseudocode. Rather when designing the tool to generate modified source code one should be able to identify and design the pseudocode that will be needed, by carefully considering the constructs of the source language and the optimization effects to be shown.

3.7 Presentation

The final issue to be considered when designing a tool to generate modified source code is the interface that will be used to present the modified source code to the user. How will the modified source code be presented so as to make its correspondence and relation to the original program clear? There are many possible solutions to this question. Probably a graphical interface, using some combination of windows, colors, icons, and layered effects may be best. As with all the other design decisions, the most appropriate choice will depend heavily on the intended task and audience.

4 Optview

Optview is the prototype tool we have implemented to demonstrate and test our ideas. It generates modified source code for optimized C programs. Optview is written to work with the Mongoose 7.2 C compiler, an aggressively optimizing commercial compiler developed by Silicon Graphics, Inc. (SGI).

The task we had in mind when we designed Optview was debugging optimized code, and the intended audience were applications programmers who may not know much about compiler optimizations. Hence the optimizations we focus on are those whose effects would be most visible to someone stepping through code or examining variables: code motion, code reordering (coarse-grained instruction scheduling), common subexpression elimination, copy propagation, partial redundancy elimination, dead code elimination and constant folding.

4.1 Key Instructions

Optview generates the modified source by reusing and modifying original source program fragments, based on the optimized target code. Determining the order in which the source statements will be executed becomes complicated after optimization, as there is no longer

```
for (current = list;
  current;
  current = current->next) {
   ...
}
(a) Original Statement
```

```
current = list;
while (current) {
 ...
 current = current->next;
}
(b) Using C Constructs
```

```
current <-- list;
loopwhile (current) do
 ...
 current <-- current->next;
od
(c) Using Pseudocode
```

Figure 2: Rewriting `for` statements.

a clear concept of individual source statements in the target code. They have been broken up, duplicated, recombined, and interleaved, making it difficult to state where one statement ends and another begins. In order to deal with this problem we introduce the notion of *key instructions*. By key instruction, we mean the single low level instruction that most closely matches the semantics associated with a given statement type. For example, the key instruction for an assignment statement is the one that stores the assignment value either to the variable's location in memory, or to a register (if the write to memory has been eliminated). The key instruction for a function call is the jump to the code for that function. Key functionality is not a completely new concept. It is closely related to the idea of semantic breakpoints [1, 6, 12]. Not all language statements have a single easily identifiable key instructions. Some control flow constructs have multiple key instructions.

Most types of language statements have an easily identifiable corresponding key instruction (for instance, the conditional branch used for a conditional statement). This is not true for assignment statements. Depending on the calculation involved in the assignment, there can be multiple store instructions associated with the source statement. Since the instructions do not reference variables by name it is often difficult to determine which one stores to the variable on the left hand side of the source assignment. To complicate matters further, the write to memory may have been eliminated, so the key "store" instruction might be an operator instruction whose destination register will contain the value of interest. For these reasons the only way to accurately determine the key instruction for assignment statements is to track the key assignment instructions from the front end of the compiler, all the way through the optimizations to the instruction generation phase. We modified the front end of the compiler to flag, for each assignment statement in the source program, the intermediate representation statement that stores the value. As the intermediate representation goes through various optimizations, transformations, and lowering stages, this flag is tracked and updated appropriately, and passed to the assembly code. The key instructions for the other statements are uniquely determined by the nature of the statement. At the point where the modified source code is generated, the order of these key instructions is used to determine the new order for the source statements. Using key instructions allows us to accurately reflect the effects of code motion and code reordering.

In languages other than C there may be language constructs, such as a vector assignment statement, that allow a single assignment statement to assign to multiple components on the left hand side of the statement. Such language constructs often can be split apart and rewritten as simpler constructs with statements assigning to single locations on the left hand side. Similarly, a C statement such as "a = b = c = 0;" should be rewritten as three separate assignment statements.

4.2 Rewriting Multi-functional Constructs

We identified four source language constructs in C which inherently embody multiple pieces of functionality that need to be split apart: `for` statements, conditional expressions, increment/decrement operators (`++`/`--`) embedded within other statements, and initialized declarations. All of these constructs allow for multiple assignments to variables within a single statement. Optview obtains information from the compiler about all such constructs contained in the original source program, and rewrites them in a simple manner. `For` statements are rewritten as `while` loops, with the loop initializations before the loop, and the loop increments inserted at the end of the loop body. Increment and decrement operators that are embedded within other statements are pulled out of the statements in question and written as separate statements, either before or after the containing statements, as appropriate. Initialization assignments are removed from variable declarations and inserted just prior to the next statement. Finally, assignment statements that have conditional expressions on the right hand side are rewritten as `if-then-else` statements. These changes permit much greater flexibility for rearranging and modifying these assignments as needed to reflect the optimizations.

The front end of the compiler writes a data file used by Optview to identify the source lines that contain the constructs that need to be rewritten. As the modified

```
Original Source Code       Modified Source Code

                           cse_var_1 = 2 * y;
                           cse_var_2 = cse_var_1 + 3;

c = 2 * y + 3;             c = cse_var_2;
a = 5 + 2 * y;             a = 5 + cse_var_1;
b = (4 + 2 * y - 1) / z;   b = cse_var_2 / z;
```

Figure 3: CSE in Optview

```
Original Source Code       Modified Source Code

if (...) {                 if (...) {
 a = x;                      a = x;
 y = a + b;                  cse_var_1 = a + b;
                             y = cse_var_1;
} else {                   } else {
  a = y;                     a = y;
                             cse_var_1 = a + b;
}                          }
z = a + b;                 z = cse_var_1;
```

Figure 4: PRE in Optview

source is generated, those lines are parsed to find and update the text that needs to be rewritten or moved. Our tool takes advantage of the fact that the input *must* be syntactically correct when it is called. This allows us to use a relatively simple parsing strategy.

4.3 Collecting Optimization Information

To accurately reflect the effects of optimizations such as common subexpression elimination (CSE), partial redundancy elimination (PRE), and copy propagation, Optview needs to know precisely what the compiler did. We modified the compiler to collect information about these optimizations and write it to a file which Optview later uses to reflect these optimizations in the modified source code. The information Optview requires includes the location where assignment statements are inserted; the left- and right-hand sides of these assignment statements; the location of source statements requiring substitutions; which assignment statement is relevant for each substitution; and which expressions were propagated to which statements. Optview must also determine whether the expressions involved in these optimizations came from the original source program, since only optimized source expressions are shown.

Once it has collected all the necessary information about the optimizations performed by the compiler, Optview modifies the source code to make the effects of CSE and PRE explicit, as shown in Figures 3 and 4. First it inserts a new assignment statement assigning the "common" expression to a new temporary variable. Next it replaces the relevant expression or subexpression in the appropriate statements with the new temporary variable. Although these new variables are not part of the original program, our expectation is that, if properly annotated, they will not be too confusing to the user.

One difficulty we encountered when implementing this part of Optview is that the "common" subexpressions generated by the compiler are in a canonicalized form, whereas the subexpression in the source statement where the substitution is to be performed is not. The solution we used is to pass the intermediate representation for the original source statement through a tool that outputs the text in the same canonical form used by the compiler.[1] Once both the source text and the compiler optimization data are in canonical form, the substitution becomes relatively simple.

Optview handles copy propagation (CPP) differently. CPP is an optimization that enables CSE and other optimizations. Thus the compiler applies CPP before applying many other optimizations. However the effects of CPP are often transitory, as a later optimization that was enabled by CPP may completely overwrite or eliminate the propagated expressions. Our solution to this problem has two parts. First we keep track (via the data gathered by the compiler) of every source statement to which an expression was propagated, as well as the exact propagated expression. After all of the other optimization effects have been reflected in the modified source code, Optview goes through and checks each resulting source line to which an expression was propagated. A comment is added to the end of each line stating that copy propagation occurred there. If the original variable still exists in the source statement, Optview replaces it with the propagated expression.

4.4 Summary of Compiler Modifications

We have modified the Mongoose C compiler to create two small data files for use by Optview. The front end of the compiler creates a data file in which it records general location and parse information for the source language constructs that will need to be rewritten. The back end of the compiler creates a data file in which it records information about the optimizations it performs. Currently the information in this file pertains to partial redundancy elimination, copy propagation and common subexpression elimination.

In addition to creating these data files we modified the compiler mechanism for keeping track of the original

[1] The canonicalized text is obtained from an existing tool that translates pieces of internal representation to C.

source code positions, making it more accurate and giving it a finer granularity; and we modified the compiler to flag key instructions for assignment statements in the intermediate representation. Finally, we modified the compiler so that immediately prior to outputting the assembly and/or binary code, it invokes Optview to generate the modified source code.

4.5 The Structure of Optview

Optview first reads in the original source file, and stores the text and some additional information for each source line in an array, one entry per source line. It labels each source line as being white space, a comment, a declaration, a pre-processor directive, or executable code. Next Optview rewrites `for` loop headers, conditional expressions, increment or decrement operators that are embedded within other statements, and initialized declarations, as explained in Section 4.2, using the data file created by the front end of the compiler.

Once these source statements have been rewritten, Optview uses the data file written by the back end to make PRE and CSE transformations explicit in the source code, as explained earlier. The next step is to determine the new order in which statements should appear. The new order is determined from the order of instructions in the target code as follows. For each source statement that has corresponding instructions in the target code, a key instruction is identified. If the source statement is an assignment statement, then this instruction is the key instruction that was flagged throughout the compiler. If the source statement is a function call, the key instruction is the jump instruction. Otherwise the last instruction for the source statement is the key instruction for that statement. Once every source statement has a key instruction associated with it, the order of these key instructions in the target code becomes the new order for the source statements in the modified source code. Any source line in the original program which contains executable code, and which does not have any associated instructions in the target code is determined to be dead code that was eliminated, and is indicated as such. After dead code elimination, Optview goes through the modified source code and annotates or updates statements at which copy propagation occurred (see Section 4.3 for details).

Recall that after reordering the source statements, some expressions must be modified to maintain semantic equivalence to the original program (see Figure 1). Optview uses a simple algorithm based on code motion to determine which expressions to update. Using an existing SGI tool, it translates the intermediate representation for those expressions back into source code. Since the intermediate code has already been optimized by the compiler, this generates correctly updated source.

Original Source Code	Modified Source Code

<pre>int main ( )
{
 int i, j, k;
 int num1, num2;
 	
 for(i = 0; i < num1; i++)
 {
   j = 15 * num2;
   num1--;
 }
 k = 15 * num2;
 i = 3 * k;
 num2++;
 	
 foo (j);
 foo (num1);
 foo (i);
 foo (k);
 	
 return 0;
}</pre>
	<pre>int main ( )
	{
	 int i, j, k;
	 int num1, num2;
	
	 i = 0;
	 cse_var_76 = (num2 * 15);
	 j = cse_var_76;
	 while (i < num1)
	 { i++;
	   num1--;
	 }
	 i = 45 * num2;
	 num2++; /* DEAD CODE */
	
	 foo (j);
	 foo (num1);
	 foo (i);
	
	 k = cse_var_76;
	 foo (k);
	
	 return 0;
	}</pre>

Figure 5: Optview Output in Simple GUI

4.6 The User Interface

We are exploring alternative graphical user interfaces for presenting to the user the modified source code generated by Optview. The choice of a user interface has great impact on the user's ability to understand the modified source code. The interface is critical for showing the relationship between the original program and the modified source code. Some possibilities we have considered include graying out dead code; drawing lines with arrows in the "margins" to indicate which lines have moved and from where; highlighting altered constants or expressions in a different color; allowing the user to undo and re-do certain optimization effects dynamically. Another aid to understanding the modified code would be to use a vertically split screen with the original code on one side and the modified code on other. Whenever a line is is highlighted or selected in one window, the corresponding line(s) in the other window are automatically highlighted.

The question of how to visualize optimized code has been considered in other work ([3, 5]). As yet there has not been an entirely satisfactory solution. Figure 5 shows an example of some C code, both the original source and the modified code that is generated by our tool, as it might look with a nice GUI.

5 Related Work

There are two previous attempts to convey the effects of optimizations explicitly to programmers. The designers of the Convex debugger for optimized code [3], created an extremely fine-grained, accurate mapping between the optimized binary and the original source. The debugger then used a combination of highlighting and code animation (in the original source) to show the user exactly what was happening in the binary. There are several problems with this approach. Since a lot of information is conveyed to the user via code animation during single stepping of the program, if the user sets a breakpoint and runs the program to that breakpoint it is not always clear which statements around the breakpoint have or have not executed, nor which values variables should have. Also, unless the user knows a lot about optimizations already, it can be more confusing than informative to see highlighting jump around all over the code. Cool describes the design of a system to make instruction scheduling apparent in a VLIW machine [5]. Cool focussed entirely on this single optimization, and never actually implemented his design.

Some of the work done by Optview is reminiscent of *term rewriting systems* (TRSs) and could possibly be done using such a framework [8]. In particular, rewriting multifunctional language constructs in simpler terms, as well as applying CSE, PRE, and copy propagation effects could all be done using such a system. The tracking of key instructions throughout the compiler also has a lot in common with this work, but unless the compiler was written with such a system in mind, modifying it to propagate key instructions in this manner would require too much work. TRSs would not be applicable to some transformations we do, such as reordering statements based on the location of key instructions in the machine code.

6 Conclusion

We have presented a new approach for displaying the effects of compiler optimizations at the source level, and have summarized some of its potential uses. We have described Optview, a tool that we have implemented to experiment with this approach, and have discussed in detail many of the issues, both at a conceptual level and in the design of Optview. We have shown that it is possible to create a tool that explicitly shows the effects of many common optimizations at the source level, for a highly aggressive optimizing compiler.

There is still work to be done with Optview. We intend to add information about high-level loop transformations and function call inlining to the modified source code. We also plan to enhance the user interface with many graphical interface techniques. Our initial investigations indicate that the modified source code is recognizable and understandable. We will continue to investigate the viability of this approach, and the uses of modified source code, especially for debugging.

References

[1] A. Adl-Tabatabai, "Source Level Debugging of Globally Optimized Code", Ph.D. Dissertation, Carnegie-Mellon University, May 1996.

[2] P. Breuer and J. Bowen, "Decompilation: The Enumeration of Types and Grammars", in *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 5, Sept. 1994.

[3] G. Brooks, G. Hansen, and S. Simmons, "A New Approach to Debugging Optimized Code", *Proceedings of the 1992 PLDI Conference*, 1992

[4] C. Cifuentes and K. Gough, "Decompilation of Binary Programs", *Technical Report FIT-TR-94-03*, Faculty of Information Technology, Queensland University of Technology, Australia, April 1994.

[5] L. Cool, "Debugging VLIW Code After Instruction Scheduling", M.S. Thesis, Oregon Graduate Institute of Science and Technology, July 1992

[6] M. Copperman, "Debugging Optimized Code without Being Misled", Ph.D. Dissertation, University of California, Santa Cruz, May 1994

[7] D. Coutant, S. Meloy, and M. Ruscetta, "DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code", In *Proceedings of the 1988 PLDI Conference*, 1988

[8] A. van Deursen, P. Klint, F. Tip, "Origin Tracking", *Journal of Symbolic Computation 15*, 1993

[9] D. Jackson and E. Rollins, "A New Model of Program Dependences for Reverse Engineering", in *Proceedings of the 1994 ACM SIGSOFT Conference*, December 1994.

[10] R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson, "Binary Translation", *Communications of the ACM*, Vol. 36, No. 2, February 1993.

[11] R. Wismueller, "Debugging of Globally Optimized Programs Using Data Flow Analysis", In *Proceedings of the 1994 PLDI Conference*, 1994.

[12] P. Zellweger, "High Level Debugging of Optimized Code", Ph.D. Dissertation, University of California, Berkeley, May 1984.