Contents

CMSC 731 (Spring 1998) Project Report

Suman Banerjee
Department of Computer Science
University of Maryland, College Park
{suman}@cs.umd.edu

June 16, 1998

Abstract

This project involved an implementation and understanding of various optimization techniques that can be built into a compiler to produce efficient code. Although, there is rarely a guarantee that the code so produced, will be the best possible, the general observation is that optimizations performed in a certain order might produce better code than others, as the results show.

1 Introduction

There are a large number of possible compiler optimizations that can be used to improve the efficiency of code. In this project, I implemented five basic optimizations, namely – local constant folding and common sub-expression elimination (using value numbering), unreachable code elimination, global constant propagation, loop invariant code motion and dead code elimination.

All these optimizations were tested on a set of six benchmark programs (5 fortran programs and 1 C program). These are standard benchmark programs which are computation intensive. They are as follows –

Program Name	Lines of Code	Subroutines (including main)
linpackd200.f	793	12
mgrid16.f	680	16
ora.f	453	2
swm256.f	487	9
tomcatv128.f	200	1
compress.c	1505	16

I used the SimpleSUIF compiler, developed at Stanford University, CA. The SimpleSUIF front-end outputs an intermediate representation of the code, which serves as the input to my optimizer. The output of the optimizer is used by the compiler back-end to generate the actual machine dependent code. The profiling of the optimized code was done using QPT2 (Quick Profiler and Tracer). I report the actual number of instructions executed by the optimized code as well as the number of SimpleSUIF instructions generated after optimization for each of the benchmarks.

2 Optimizations

2.1 Local optimizations

In each basic block, I perform value numbering to do constant folding and local common subexpression elimination in that block. The value of all variables that are detected to be constants are propagated locally for constant folding. Also, repeated use of the same expression is detected, and instead of performing the computation multiply, these values are stored in new pseudo-variables and are re-used.

2.2 Unreachable Code Elimination

This optimization is performed inherent to each of the other optimization passes. Initially, when basic blocks are computed, a directed search from the entry node detects all the reachable blocks. Any unreachable code is subsequently removed. Also, as a consequence of other optimizations, certain code may become unreachable. In the following example, constant folding would lead 2 instructions being unreachable.

```
Unoptimized

   ldc (s.32)     t1, 1
   btrue          t1, L1
   ldc (s.32)     t2, 4   /* Unreachable */
   cpy (s.32)     r3, t2  /* Unreachable */
   /* Some reachable code */
L1 :
   ldc (s.32)     t3, 2
   cpy (s.32)     r3, t3
```

```
Optimized

   jmp           L1
   /* Some reachable code */
L1 :
   ldc (s.32)     t3, 2
   cpy (s.32)     r3, t3
```

In my implementation, I would remove this unreachable code as and when they get generated. Hence, there is no separate unreachable code elimination phase of the optimizer.

2.3 Global Constant Propagation

To perform global constant propagation, I make use of an iterative data flow solver. I use the data flow solver to solve reaching definitions problem. The basic blocks are traversed in reverse PostOrder so that the solution converges quickly. The underlying assumption is that the control flow graph is not irreducible, or else the data flow solver might not even converge. If all the reaching definitions of a given exposed use have the same constant value, and there is no other path to this use (detected by using a variation of the dominance property), then the constant value is propagated to this use.

2.4 Loop-invariant Code Motion

In this case also, I make use of the solution to the reaching definitions problem using the generic data flow solver. First I detect whether an instruction is invariant for a given loop, and then for each invariant instruction I check if it is safe to be moved outside the loop. Loops might be nested in other loops. In such cases, I start by processing the innermost loop first and work outwards from it. In each case, the movable code is added to the loop pre-header, which gets executed only once per loop.

2.5 Dead Code Elimination

All instructions that define a certain variable, which is not subsequently used, is dead. Hence, such code can be eliminated. In general, programs written by compiler users will not have such code, but such occurances can happen due to other optimizations. The data flow solver is used to solve a backward data flow problem of liveness of variables. The solution is used to eliminate all code that is found to be dead.

3 Pitfalls encountered

Although, all the above optimizations described above are expected to be useful, there are certain cases where many of the optimizations would lead to incorrect code generation. Since in the project, we perform on alias analysis or interprocedural analysis, it was necessary to make a few conservative assumptions to begin with.

An interesting example of this is for global variables. Consider the following piece of C code fragment –

```
int a; /* global variable */

void main (void)
{
 a = 1;
 foo();
 return;
}

void foo (void)
{
 printf ("%d\n",a);
 return;
}
```

In this example, in the procedure main, without information about the use of the variable a, inside the procedure foo, it would be detected as unused. As a consequence, it might lead to the assignment of a being deleted in dead code elimination, which is clearly wrong. Hence, it was necessary to assume that any procedure call would lead to use of all global variables. However, in SimpleSUIF instructions, there is no way to differentiate between a global variable and a regular pointer variable. Both of them would be accessed using load and str operations, as pointer variables. Hence, to turn off optimizations for global variables would lead to turning off these optimizations for all pointers in general.

Another problem was of aliasing in C programs. The following C code fragment illustrates the point.

```
void main (void)
{
 int *pa;
 int x, y;

 pa = 0x234435;

 x = 1;
 *pa = 2;
 y = x + 1;
 printf ("%d\n",y);
 return;
}
```

In this example, it is possible the pa actually points to the address of the variable x. However, since the address of x is only determined at run time, at compile time we cannot realize that the statement (*pa = 2;) might actually be changing the value of x. Hence, if we use the value of x as 1 in constant folding to get the value of y, it might be incorrect.

However, the above kind of code is actually very rare, if not absent from programs, since the user is general will not be aware of the actual address of variables at run-time. But, for sake of correctness, it would lead to unsafe code, if such kind of issues are not taken care of. In my implementation I have such unsafe optimizations, having made the optimistic assumption that users will not write such kinds of code.

It may be noted, however, that such kind of code cannot be generated in Fortran code, and so this is not an issue for Fortran programs.

Another example of interest is for array variables. Consider the code fragment below –

```
void main (void)
{
 int A[10];
 int i;

 A[0] = 5;
 for (i = 1; i < 9; i++)
{
 A[i] = A[i-1];
   printf ("%d\n",A[i]);
```

```
}
 return;
}
```

It is clear that A[0] assignment cannot be deleted as a dead instruction, since it is used as A[i-1]. However, the address of A[i-1] will not be a constant, while the address of A[0] is. The address of A[i-1] would be computed as –

```
ldc  (a.32)     t9 = &A + 0
ldc  (s.32)     t10 = 4
mul  (s.32)     t11 = r1, t10
ldc  (s.32)     t12 = 4
sub  (s.32)     t13 = t11, t12
```

where, r1 denotes the variable i. To ensure correctness, I treat all uses of any variable, whose address is expressed as some function of the address of A, as an use of the entire array A. Hence, references of t9, t11 and t13 are treated as references to the array A. This prevents A[0] assignment from being detected as dead code.

4 Some Issues in SimpleSUIF

I found SimpleSUIF to be a good compiler tool in general. However, I felt that the type information it provided was insufficient. Particularly for mcpy and str operations, no type information was available for the underlying type on which the operation was being performed. Also, the simple_immed data structure does not contain good type information for the value it stores. It does not distinguish between signed and unsigned ints, floats and doubles. Also, there was a couple of discrepancies in the way address types were defined. There was no mechanism to extract the type of the underlying variable from the register storing its address. Under certain circumstances, address type pointers were not assigned the same values even when the underlying types were the same. The base type field matched in such cases, but as the SimpleSUIF manual mentions, it is not legal to conclude that two variables have the same type, unless the type pointers match and not just the base types.

However, most of its features were very useful in the implementation of the optimizer, particularly its use of temporary registers, using the same numbering space for temporary and pseudo registers, the instruction set and some others.

5 Experiments and Anlysis

In the optimizer that I have implemented there are four distinct phases – local optimization, global constant propagation, loop-invariant code motion and dead code elimination. The fifth optimization – unreachable code elimination is implemented locally with each of the other optimizations.

In general, doing dead code elimination the last would be a good idea, based on the assumption that since users do not generally write dead code, it would be wasteful doing that earlier. Other optimizing passes might lead to generation of some dead code.

On the given set of benchmark programs, I performed a sequence of experiments, in which I varied the order in which the optimizations were performed, and also the number of optimizations performed.

All optimization passes that I tested the code with, produced correct code, in as much that it gives the output as the un-optimized code.

Below, I show the results of the following optimizations (the name in parentheses represent the name used in the tables for each optimization order) –

1. No optimization (O)
2. Global Constant Propagation (C)
3. Local Optimization through value numbering (V)
4. Loop-invariant code motion (L)
5. Dead Code Elimination (D)

Results for all the sequence of optimizations that I tried is in the appendix, but below I present a few of the interesting optimization orders.

Result of each optimization tried individually (except dead code elimination, since dead code elimination by itself is not so interesting)

Program Opt pass	compress	linpackd200	mgrid16	ora	swm256	tomcatv128
O	6854362	1367275899	64453786	210919765	630993210	283342821
C	6866364	1367275885	64453738	210919765	630993210	283278805
V	6866360	1370006125	65442464	210919999	552456399	315376567
L	6647088	1368100545	61352343	210859030	470682915	242983103

Number of instructions executed

Program Opt pass Opt pass	compress	linpackd200	mgrid16	ora	swm256	tomcatv128
O	3155	3666	6317	1165	3149	1577
C	3154	3672	6320	1133	3149	1616
V	3116	3598	6179	1122	2706	1241
L	3156	3682	6737	1162	3333	1671

Number of SimpleSUIF instructions generated as compiler output

From these tables it can be seen that clearly loop-invariant code motion is a very powerful optimization technique. This manages to produce a good amount of optimization for most of the programs. Local optimization managed to improve performance of swm256 in particular. Global constant propagation did not enhance performance much, when used solely by itself. Since, most of the program life time would be spent inside loops (straight lines of code, would not run for very long, the number of SimpleSUIF instructions for the largest of the above programs is 6317), optimizing loops is thus a very good way to optimize programs.

To show that dead code elimination indeed is an useful postphase for most optimizations, let us compare data in the following table.

Program Opt seq	compress	linpackd200	mgrid16	ora	swm256	tomcatv128
O	6854362	1367275899	64453786	210919765	630993210	283342821
C V L	6671088	1368437982	62943845	214826464	438627683	292328128
C V L D	6635075	1368433136	62814466	211543264	438623908	292328079

Number of instructions executed

Program Opt seq	compress	linpackd200	mgrid16	ora	swm256	tomcatv128
O	3155	3666	6317	1165	3149	1577
C V L	3156	3766	6517	1148	2569	1265
C V L D	3127	3661	6346	1139	2552	1235

Number of SimpleSUIF instructions generated as compiler output

C V L : C followed by V followed by L

C V L D : C followed by V followed by L followed by D

Note, in all cases, the extra post phase of dead code elimination reduced the number of SimpleSUIF instructions and also reduced the number of instructions executed.

The next table compares the performance of these programs for some interesting orders of the optimization sequence.

Program Opt seq	compress	linpackd200	mgrid16	ora	swm256	tomcatv128
O	6854362	1367275899	64453786	210919765	630993210	283342821
L C V D	6635077	1368921953	62878418	214142464	429570147	241486291
C V L D	6635077	1368433136	62814466	211543264	438623908	292328079
V C L D	6635077	1368907585	62794430	214826443	415176361	292065637
C V L V D	6635075	1368434717	62755562	211543264	453356184	292333159
V C L V D	6635077	1368434727	62837286	211543243	435030173	292070444
C V D L D	6635077	1368433122	62861330	214142464	419199142	292328079

Number of instructions executed

Program Opt seq	compress	linpackd200	mgrid16	ora	swm256	tomcatv128
O	3155	3666	6317	1165	3149	1577
L C V D	3137	3689	6390	1139	2879	1500
C V L D	3137	3661	6346	1139	2552	1235
V C L D	3137	3690	6355	1138	2508	1292
C V L V D	3137	3661	6346	1139	2584	1235
V C L V D	3137	3664	6350	1138	2576	1264
C V D L D	3137	3659	6357	1139	2575	1235

Number of SimpleSUIF instructions generated as compiler output

From the above table it can be seen that among the optimization passes, performing loop-invariant code motion first does not provide the most optimized code. In most cases, (4 out of 6), the best optimization sequence is global constant propagation followed by local optimization followed by loop-invariant code motion and terminated by dead code elimination gives the best results.

I also tested multiple passes of some of the optimizations. An exhaustive set of results is listed in the appendix. But, it can be seen from the above table, that multiple passes of local optimizations and that of dead code elimination is pretty useful. In particular, if local optimization is done first, then another pass of local optimization after doing global constant propagation appears to be pretty helpful. Likewise, performing dead code elimination after doing global constant propagation and local optimizations appear to help in optimizing the code better.

It is interesting to note that in many cases that have less number of instructions actually executed (ie. faster code), the number of SimpleSUIF instructions generated in the output tended to be higher, probably reflecting that the code has more optimizing instructions inserted into it.

In general, only compress, linpackd200 and swm256 showed a noticeable better performance in the optimized versions. Some of the other programs in fact had a degraded performance. Most of the optimizations in all these programs were through loop-invariant code motion.

A reason for this low optimization in some of the programs can be attributed to the fact that some of the optimization scope in these programs were lost due to lack of alias analysis and interprocedural analysis. As explained before, these caused some opportunity of optimizations to be lost, due to the conservative assumptions being made. Also, register allocation and other such phases take place in the optimizer back-end. Some of the optimizations that I implemented would result in increasing the live ranges that might cause register allocation to generate more code than it would it in the un-optimized version. This would easily happen in both loop-invariant code motion as well as in local common sub-expression elimination.

6 Conclusions

In general, I can conclude that order of optimization passes play an important role in determining the amount of optimization. Although by running the optimizer on a set of six benchmark programs would give limited information, it looks to me that the best optimization may be got when the order of execution of the phases are – global constant propagation, local optimization (constant folding and local common sub-expression elimination), loop optimization and dead code elimination respectively. Adding multiple passes of dead code elimination and doing an extra pass of local optimization after global constant propagation and loop optimizations also are good techniques, though they increase compile-time overheads.

7 Appendix

For each program, the various combination of the optimization passes are listed below along with the results. The first number represents the number of SimpleSUIF instructions generated in the output code, and the second number is the actual number of instructions executed.

- O : Original
- C : Global constant propagation
- V : Local optimization (local common sub-expression elimination and local constant folding)
- L : Loop-invariant code motion
- D : Dead code elimination

swm256

Opt seq	Number of SimpleSUIF instrs	Instructions executed
O	3149	630993210
C	3149	630993210
V	2706	552456399
L	3333	470682915
V C D L	2539	428552403
C L V D	2914	444037211
V C L D	2508	415176361
C V L D	2552	438623908
C V L	2569	438627683
L C V D	2879	429570147
V C L V D	2576	435030173
C V L V D	2597	453356184
C V D L D	2501	419199142

ora

Opt seq	Number of SimpleSUIF instrs	Instructions executed
O	1165	210919765
C	1133	210919765
V	1122	210919999
L	1162	210859030
V C D L	1138	210859243
C L V D	1139	211543264
V C L D	1138	214826443
C V L D	1139	211543264
C V L	1148	214826464
L C V D	1139	214142464
V C L V D	1138	211543243
C V L V D	1139	211543264
C V D L D	1139	214142464

tomcatv128

Opt seq	Number of SimpleSUIF instrs	Instructions executed
O	1577	283342821
C	1616	283278805
V	1241	315376567
L	1671	242983103
V C L D	1292	292065637
C L V D	1501	242793123
V C L D	1292	292065637
C V L D	1235	292328079
C V L	1265	292328128
L C V D	1500	241486291
V C L V D	1264	292070444
C V L V D	1235	292333159
C V D L D	1235	292328079

linpackd200

Opt seq	Number of SimpleSUIF instrs	Instructions executed
O	3666	1367275899
C	3672	1367275885
V	3598	1370006125
L	3862	1368100545
V C D L	3690	1368907569
C L V D	3689	1368444353
V C L D	3690	1368907585
C V L D	3661	1368433136
C V L	3766	1368437982
L C V D	3689	1368921953
V C L V D	3664	1368434727
C V L V D	3661	1368434717
C V D L D	3659	1368433122

mgrid16

Opt seq	Number of SimpleSUIF instrs	Instructions executed
O	6317	64453786
C	6320	64453738
V	6179	65442464
L	6737	61352343
V C L D	6350	62771606
C L V D	6392	62997586
V C L D	6355	62794430
C V L D	6346	62914466
C V L	6517	62743845
L C V D	6390	62878418
V C L V D	6350	62837286
C V L V D	6346	62755562
C V D L D	6357	62861330

compress

Opt seq	Number of SimpleSUIF instrs	Instructions executed
O	3155	6854362
C	3154	6866364
V	3116	6866360
L	3156	6647088
V C D L	3137	6635077
C L V D	3137	6635077
V C L D	3137	6635077
C V L D	3137	6635077
C V L	3156	6671088
L C V D	3137	6635077
V C L V D	3137	6635077
C V L V D	3137	6623075
C V D L D	3137	6635077