IJCSN International Journal of Computer Science and Network, Volume 3, Issue 1, February 2014
ISSN (Online) : 2277-5420
www.IJCSN.org

Effectiveness of Inlining, Common Subexpression and Deadcode Elimination in Optimizing Compilers

[1] Jagdish Bhatta

[1] Central Department of Computer Science and Information Technology
Tribhuvan University, Nepal

Abstract - One of the most difficult and least understood problems in the design of the compilers is the generation of good object code. The most common criteria by which the goodness of a program is judged are its running time and size. Since the optimization is important during compilation, the way of obtaining optimal code must be done in appropriate way. Performing optimization is to decide when to optimize and what to optimize. Optimizing compilers are of great importance in resulting the better object programs. Due to the undecidable nature of those compilers, getting optimal optimization depth is of challenging issue. The development of a model for attaining optimization depth can be greater achievement towards the compiler optimization. As a part of which, major optimization passes, viz. inlining, common subexpression elimination and deadcode elimination, has been implemented, tested and analyzed for various instances of inputs to verify the depth. With inlining, we can improve run-time performance by replacing the text of a function call with the body of the called function itself. While using common subexpression elimination, it decreases the execution time of the program by avoiding multiple evaluations of the same expression with reduced number of computations. And at the same context, deadcode elimination aims to rectify the dead code instructions, whose removal reduces code size and eliminates unnecessary computations.

Keywords - *Compiler Optimization, Common Sub expression Elimination, Dead Code Elimination, Inlining*

1. Introduction

Generally, computer programs are formulated in a programming language and specify classes of computing processes. Computers, however, interpret sequences of particular instructions that are in low level, but not program texts. Therefore, the program text must be translated into a suitable instruction sequence before it can be processed by a computer. For this, some means of bridging the gap is required. This is where need of the *compiler* comes in [1, 2].

It can be possible to improve the performance of compiler through the proper use of optimization. Compiler optimization is generally implemented using a sequence of optimizing transformations, algorithms which take a program and transform it to produce a semantically equivalent output program that uses less resource. The multi-pass compilers contain optimization phase as a part of its passes. Most of the multi-pass compilers are optimizing compilers. Optimizing compilers are the standardized multi-pass compilers with major work focused with the optimization passes. The goal of the optimizing compilers is to enhance the code quality. Optimizing compilers focus on relatively shallow constant-factor performance improvements and do not typically improve the algorithmic complexity of a solution. Even though there exists lots of optimization techniques, redundancy elimination, data flow and tail call optimization are of great importance.

2. Optimizing Passes

2.1 Inlining

This optimization technique, also called open linkage, improves run-time performance by replacing the text of a function call with the body of the called function itself. Inlining is an important optimization for the programs that make use of the procedural abstraction [3, 4, 5].

In a procedure-based or modular programming language, a procedure is a series of instructions implementing a particular action. When a procedure is invoked, it initiates a construction of stack frame. A stack frame stores local variables and other call-return specific information, like return address. According to the authors view in [6], for small subroutines, the task of handling the arguments and constructing the stack frames can exceed the cost of the operations within the subroutine itself. With the popularity of modular programming techniques, the percentage of use of such small procedures has grown

dramatically. Hence, inlining eliminates the overhead of jumping to and returning from a subroutine i.e. the overhead for invocation is eliminated. The stack frames for the caller and callee are allocated together and the transfer of control is wiped out [6]. Thus this saves a considerable amount of execution time. An additional advantage of inlining is that it may uncover opportunities for further optimization of caller and the body of inlined procedure through the transformations like common sub-expression elimination, deadcode elimination, copy propagation.

The primary disadvantage of the inlining is that it increases code size, thus resulting slightly larger executable files. However, this increase in size is offset by the elimination of time-consuming procedure calls and procedure returns. In practice the control in increasing code size can be done by the selective application of inlining (e.g., to procedures that are only called from few places or to small procedures).

The example illustrated in following shows the effect of procedure inlining;

```
int D(int x, int y)
{
        if x>100 then
                ……..
        else
                z = g + y
        ……..
}

Main ()
{
        int  g, k;
        g = 10;
        Read(j);
        k = g + j;
        D(g, j);
}
```

Listing 1: Prior to Inlining

```
Main()
{
        int g, k, x, y;
        g = 10;

        Read(j);
        k = g + j;

        x = g
        y = j

        if x>100 then
                ……..

        else
                z = g+y
        ……..
}
```

Listing 2: After to Inlining

2.2 Common Sub expression Elimination

A computation in a program is redundant at a point if it has been computed previously and its result is guaranteed to be available at that point. If such computations can be identified, they can obviously be eliminated without affecting the behavior of the program.

When the compiler discovers two or more instances of a single expression separated by code that does not redefine any of the variables used in the expression, it can save the result of the first evaluation and replace the subsequent evaluations with a simple reference to the saved value. To locate opportunities for this optimization, known as global common sub expression elimination, the compiler must know which expressions are *available* at various points in the procedure. An expression *exp* is said to be *available* at the entry to a basic block if along every control-flow path from entry block to this block there is an evaluation of *exp* that is not subsequently killed by having one or more of its operands assigned a new value. Thus, Common Sub expression Elimination aims to decrease the execution time of the program by avoiding multiple evaluations of the same expression with reduced number of computations [7].

Optimizers frequently distinguish two kinds of CSE [1]:

- Local Common Sub expression Elimination works within a single basic block and is thus a simple optimization to implement.

- Global Common Sub expression Elimination works on an entire procedure, and relies on dataflow analysis which expressions are available at which points in a procedure.

The division is not essential because global common sub expression elimination catches all the common sub expressions that the local form does but the local form can often be done very cheaply. When determining common

sub expressions, it involves looking for expressions that are guaranteed to compute the same value, no matter how that value is computed. Common sub expressions are usually eliminated at the higher level but machine-level CSE can do more because all values are exposed at this level such as array indexing calculations [8].

Even though it may seem that common sub expression elimination is always valuable; because it appears to reduce the number of ALU operations performed; it is not always worthwhile. The simplest case where it is not advantageous is: if it causes a register to be occupied for a long time in order to hold an expression's value, and hence reduces the number of register registers for others use. This may further lead to expensive memory load store of the value because of insufficient registers available [1]. Following control graphs illustrate the common sub expression elimination.

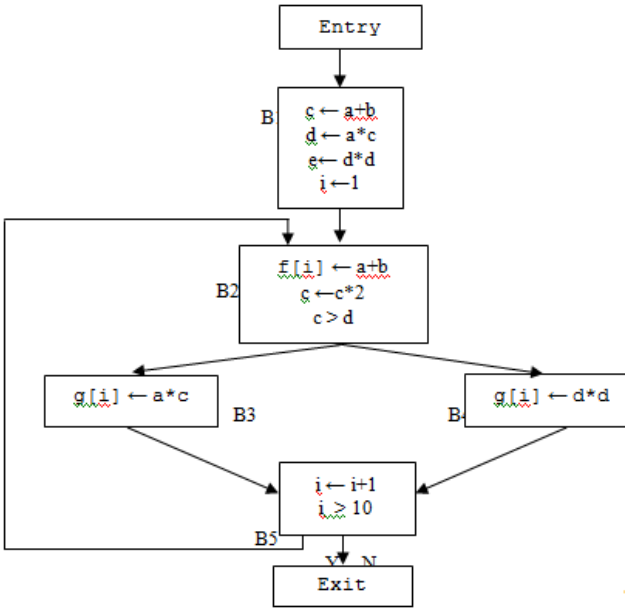

Figure 1: Example Flowgraph for CSE (Before Applying CSE)

2.3 Deadcode Elimination

A variable is dead if it is not used on any path from the location in the code where it is defined to the exit point of the routine. Dead code instruction refers to computations whose results are never used on any executable path leading from the instruction. The notion of "results not used" must be considered broadly [1, 9, 10, 11]. It may possible for a computation to generate exceptions or raise signals whose handling can affect the behavior of the rest of the program, then we cannot consider that computation to be dead. Code that is dead can be eliminated without affecting the behavior of the program as the code has no effect on program's output. Thus it reduces code size and eliminates unnecessary computations.

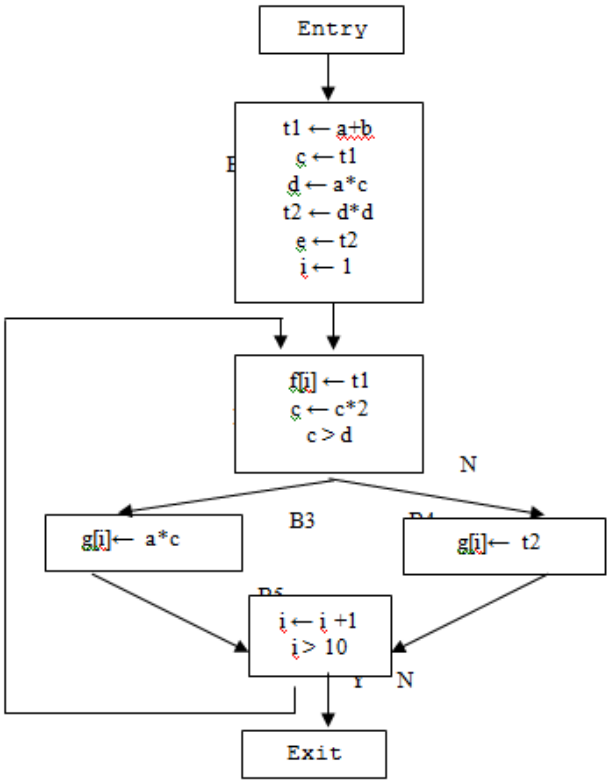

Figure 2: Example Flowgraph for CSE (After Applying CSE)

The deadcode elimination is much based upon the *live variable analysis*. We call a data member *m* live if there is an object *o* in the program that contains *m* such that the value of *m* in *o* may affect the program's observable behavior (i.e., output or return value). If there is no such object *o*, we call *m* dead. Link-time opportunities for dead-code elimination arise primarily as a result of unreachable-code elimination that transforms partially dead computations (computations whose results are used along some execution paths from a program point but not others) into fully dead ones

In the code block B1 of Figure 2, there occurs a computation that assigns the value of register t_2 to a variable e. After the e's definition block, in the path of the flowgraph, there is no use of the variable e. Hence the computation turns out to be a dead and can be removed without having any effect on the programs' flowgraph.

3. Implementation

3.1. Flow of optimizing passes

3.2.

The flow of the optimization passes, implemented so far using the SUIF platform [12, 13], along with the front and back-ends is as shown below:

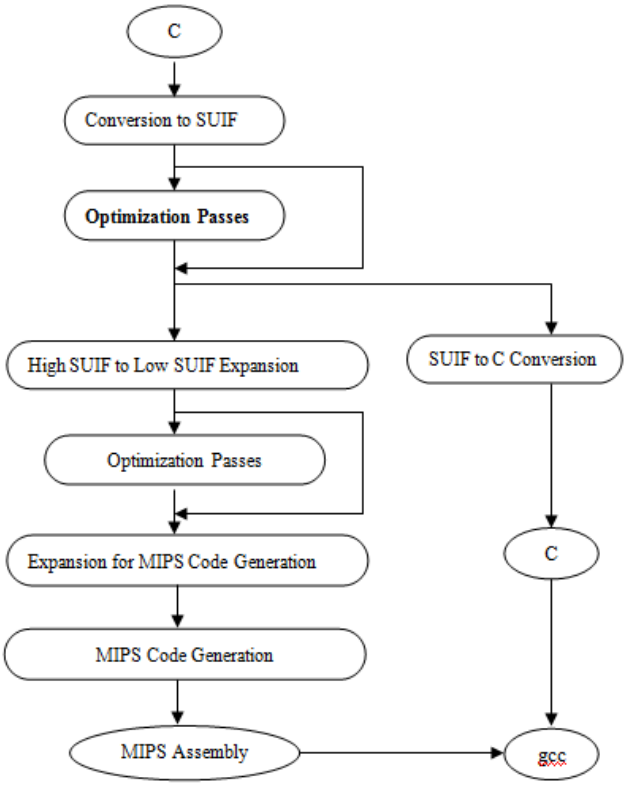

Figure 3: Sample flow of compilation

3.3. Integration of Passes in SUIF System

For each of the passes dissected above, the corresponding classes and their procedure body definition were constructed using the SUIF objects and data structures in the corresponding *header* and *cpp* files. The procedure body contains the do_procedure_definition (do_opt_unit) unit that defines the optimization task. Besides these, the two other files viz. *standard_main.cpp* and *passname_pass.cpp* were maintained. The prior is used to build the standalone program of the developed pass and the second one defines the SUIF pass built as the dynamically loadable library [14].

Here, as a part of the study, it includes implementation of the optimizing passes for inlining (procedure integration) and common sub expression elimination. The pass that is used for deadcode elimination comes with the SUIF compiler.

4. Analysis

As optimizations are to improve program efficiency somehow, the study has mainly focused on the runtime and compile time performance. The techniques implemented so far resulted better execution time. As a part of testing a number of input benchmark programs, having possibilities of above mentioned optimization passes, were taken. In order to get an idea of the cost of the implemented optimization passes, increase/decrease in compile time and run time for each of the benchmark programs were measured. The CPU clock cycles give the measurement at each level of optimization. To obtain reliable timing information, each operation was repeated multiple times and arithmetic mean of them was taken.

The following graphs summarize the outcomes of the experiments carried out during this study.

With many of optimizations, it may be possible that one optimizing pass may open the door for utilization of other subsequent optimizations so as to support further enhancement in the compiled code. Here, the inlining pass provides opportunity for further optimization in the code through deadcode elimination, common subexpression elimination, copy propagation. So all of the *Inlining,CSE, and DCE* optimization passes were enabled respectively in order to observe the performance result. An increase by 44% in runtime was obtained in case of applying all of the three passes together, hence resulting better runtime performance. Figure 4 illustrates the impact of all these three passes in runtime.

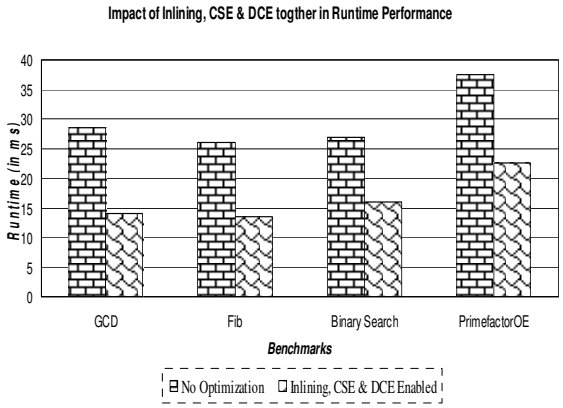

Figure 4: Impact of Inlining, CSE & DCE together in Runtime Performance

As discussed in the runtime analysis, better runtime was achieved when all of the three passes; *Inlining*, *CSE*, and *DCE* were enabled during the compilation. But at the

same context, the case was different in terms of compile time. Hereby, it was observed that there was a significant increase in the compile time by 772% of the time obtained during the compilation with no optimization. Thus calling all of those optimizations together seems to be costly in term of compile time. This result is illustrated in figure 5 below;

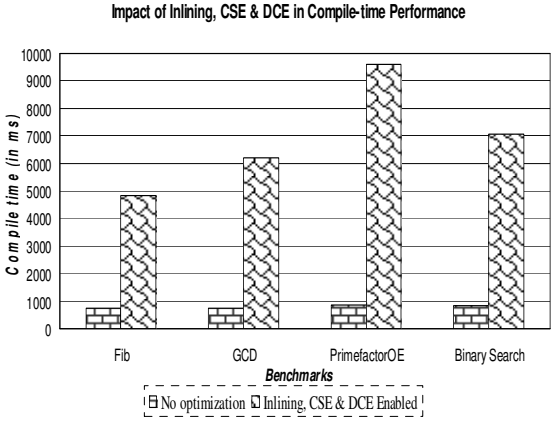

Figure 5: Impact of Inlining, CSE & DCE together in Compile-time Performance

5. Conclusion

The empirical analysis that has been done as a part of study shows change in runtime and compile-time in each of the cases. With this, it is clear of reducing runtime and increasing compile-time due to enabling of the developed passes during the compilation period. Notwithstanding, the ratio by which runtime decreases is less than the ratio of increase in compile time, as runtime is of major concern, the study has neglected the significant increase in compile-time, while measuring the overall performance. One aspect that caused to focus on runtime only, during consideration of overall performance, is that the runtimes get dynamically varied according as the input size. While the compile-time remains static whatever be the size of input. Hence, results came out at the better cost.

6. Recommendation and Future Work

In this paper, much of the efforts have been employed to develop the selected optimization passes and to perform the empirical analysis. Some of the developed passes are restricted to work in a limit; the *CSE* incorporates only local optimization issues that work in the basic blocks, so it can be enhanced to work globally through each program blocks. Similarly, restriction of *Inlining* to work with in a same file can be extended to inline programs from other files included.

References

[1] S.S. Muchnick, "Advanced Compiler Design Implementation", *Morgan Kaufmann Publishers*, 1997, pp. 219-703.

[2] T. Kistler, and M. Franz, "Continuous Program Optimization: A Case Study", *ACM Transactions on Programming Language and Systems*, Vol. 25, No. 4, July 2003, pp. 500-548.

[3] M. Schinz, M. Oderskey, "Tail Call Elimination on the Java Virtual Machine", *Published by Elsevier Science B.V.*, 2001.

[4] Y. Minamide, "Selective Tail Call Elimination", Institute of Information Sciences and Electronics University of Tsukuba and PRESTO, JST.

[5] O. Waddell., R. K. Dybvig, "Fast and Effective Procedure Inlining", Indiana University, Computer Science Department, Technical Report No. 484.

[6] H. G. Baker, "Inlining Semantics for Subroutines which are Recursive", *ACM Sigplan Notices 27*, 12(Dec 1992), pp. 39-46.

[7] O. Chitil, "Common Subexpression Elimination in a Lazy Functional Language", Lehrstuhl f˙ur Informatik II, Aachen University of Technology, Germany.

[8] J. W. Davidson, F. W. Christopher, "Eliminating Redundant Object Code", *ACM*, 1982.

[9] L. Song, Y. Futamura, R. Gluck, Z. Hu, "A Loop Optimization Technique Based on Quasi Invariance", 1999.

[10] C. K. Behera, P. Kumar, "An Improved Algorithm for Loop Dead Optimization", *ACM SIGPLAN,* Vol. 41(5), May 2006.

[11] K. Deibel, "On the Automatic Detection of Loop Invariants", February 25, 2002.

[12] R. Wilson *et al.*, "An Overview of the SUIF Compiler System", Computer Systems Lab, Stanford University, CA 94305.

[13] R. Wilson *et al.*, "The SUIF Compiler System: A Parallelizing and Optimizing Research Compiler", *Technical Report CSL-TR-94-620*, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, CA 94305-4055.

[14] M. D. Smith, G. Holloway, "An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization", Division of Engineering and Applied Sciences Harvard University, 2002

.

Jagdish Bhatta received the B.Sc. and M.Sc. degrees in Computer Science from Tribhuvan University, Nepal in 2004 and 2007, respectively. Since 2008 he is a full time faculty member at the Tribhuvan University. He has been involved in number of researches conducted at the department and also supervised graduate student's dissertation. He has published research papers in national and international journals. His research areas are Cryptography and Network Security, Compiler Optimization, Computational Geometry, Cloud Computing.