New trends and Challenges in Source Code Optimization

Anjan Kumar Sarma
National Institute of Electronics and IT,
Electronics Niketan, 6 CGO Complex,
Lodhi Road, New Delhi-110003 (India)

ABSTRACT

The front end of a compiler is generally responsible for creating an intermediate representation of the source program whereas the back end of the compiler constructs the desired target program from the intermediate representation and the information in the symbol table. Before the intermediate code is passed to the back end of the compiler, it is necessary to improve the intermediate code so that better target code will result. The code optimization phase in a compiler attempts to improve the target code without changing its output or without side-effects.

Today, most of the compiler research is done in the optimization phase. There are many classical techniques (e.g. Eliminating common sub-expressions, Dead-Code elimination, Constant Folding etc.) that have been used in code optimization. However, the increasing size and complexity of software products and the use of these products in embedded, web-based and mobile systems results in the demand for more optimized versions of the source code. This research paper discusses the challenges involved in code optimization for such systems and some recently developed techniques in code optimization.

Keywords

Optimization, Reverse Inlining, Cross Linking, Address-Code, Leaf Function

1. INTRODUCTION

1.1 What is Code Optimization

Code Optimization is the process of transforming a piece of source code to produce more efficient target code. Efficiency is measured both in terms of time and space. Optimization is generally implemented using a set of optimizing transformations, i.e., algorithms which take a piece of source code and transform it to produce a semantically equivalent output code that uses fewer resources. Most of the optimization techniques attempt to improve the target code by eliminating unnecessary instructions in the object code, or by replacing one sequence of instructions by another faster sequence of instructions.

1.2 Why Optimization is an important phase in Compiler Design

Optimization is one of the most important phases in a Compiler. Code optimization attempts to improve the source code so that better target code will result. Usually, a better target code is one that is better in terms of time and space. However, some other objectives may also be considered to measure the goodness of code, such as target code that consumes less power. In modern times, processor architectures are becoming more complex. With the introduction of multicore and embedded systems requiring a faster target code that consumes less space and power to execute. The code optimization phase in a compiler attempts to resolve these issues and produces better target code without changing the desired output.

1.3 Presence of the Optimization phase in the Compiler Architecture

Code optimization may either be performed on the intermediate representation of the source code or on the un-optimized version of the target machine code. If applied on the intermediate representation, the code optimization phase will reduce the size of the Abstract Syntax Tree or the Three Address Code instructions. Otherwise, if it is applied as part of final code generation, the code optimization phase attempts to choose which instructions to emit, how to allocate registers and when to spill, and so on.

2. OPTIMIZATION TECHNIQUES

There are many classical optimization techniques that have been used in code optimization since the last decade. Some of these techniques are applied to the basic blocks in the source code and others are applied to the whole function. As the result of recent researches, many new optimization techniques have been introduced. In this research paper, the emphasize will be on the new techniques of code optimization; however, a brief overview of the classical techniques have also been introduced.

2.1 Classical Optimization Techniques

The classical techniques for code optimization can be categorized as:

1. Local Optimization
2. Global Optimization
3. Inter-Procedural Optimization

2.1.1 Local Optimization

The code optimization phase in a compiler begins with partitioning the sequences of three-address instructions into basic blocks. These basic blocks become the nodes of a flow graph. Local optimization is performed within each basic block. We can often obtain a substantial improvement in the running time of code by performing local optimization within each basic block by itself. Since basic blocks have no control flow, these optimizations need little analysis.

Local optimization can be performed using the following techniques-

(i) Eliminating local common subexpressions,

(ii) Dead code Elimination

(iii) The use of algebraic identities-

  (a) The use of arithmetic identities

(b) Local reduction in strength, that is, replacing a more expensive operator by a cheaper one.

(c) Constant Folding

(iv) Reordering statements that do not depend on one another.

2.1.2 Global Optimization (Intra-Procedural Methods)

Global optimization techniques act on whole functions. In global optimization, improvement takes into account what happens across basic blocks.

Most global optimization techniques are based on data-flow analysis. The results of data-flow analysis all have the same form: for each instruction in the program, they specify some property that must hold every time that instruction is executed.

Some of the global optimization techniques are:

(a) Eliminating global common sub-expressions.

(b) Copy propagation.

(c) Dead-code elimination.

(d) Code Motion.

(e) To find induction variables in loops and optimize their computation.

(f) Constant folding.

2.1.3 Inter-Procedural Optimization:

Inter-procedural optimization (IPO) techniques are applied to programs that contain many frequently used functions. The technique is called inter-procedural because it analyses the entire program, whereas other optimizations such as local optimization or global optimization look at only a single function, or even a single block of code.

Inter-procedural optimization techniques can be categorized as-

(a) Address-token analysis

(b) Array-dimension padding

(c) Alias analysis

(d) Automatic array transposition

(e) Automatic memory pool formation

(f) Common block variable coalescing

(g) Common block splitting

(h) Constant propagation

(i) Dead code detection

(j) Formal parameter alignment analysis

The global code optimization technique is based on inter-procedural information only.

2.2 New Optimization Techniques

In general, code optimization is performed for reducing the compiled code size and execution time .However, reducing the code size has become low-priority due to steadily decreasing cost of memory. Still, it is favorable to spend effort on the code size reduction instead of wasting memory. This is obviously true for embedded systems, where memory usage is particularly crucial.

Some of the new optimization techniques used for code size reduction are discussed here

2.2.1 Reverse-Inlining (or Procedural Abstraction)

This code optimization technique attempts to reduce the size of the source code by replacing all common code patterns in a program with function calls. This technique is in fact opposite of the “Inlining of Code”, where the call to a function is replaced by a copy of the function body. In this technique, all common code patterns in a program are placed into compiler-generated functions. Every occurrence of these patterns in the program is replaced by a call to the corresponding function. This technique is particularly useful for optimizing programs in which more occurrences of a given pattern can be found.

Procedural abstraction involves the following steps:

1. The source code is partitioned into basic blocks.
2. A callgraph is created from these basic blocks. The following procedure is used to create the callgraph:

a. All the basic blocks except the one where program execution starts are considered unused.

b. The blocks which are actually entered by the control flow of the program are considered and marked as used.

c. The blocks which are not included in the completely generated call graph are removed.

3. A fingerprint is computed from each basic block. This fingerprint is used to find candidate blocks for outsourcing. The algorithm for creating the fingerprint is quite simple and it works as follows:A 64-bit value from each basic block is created by looking at the first 16 opcodes of the block, and each of these opcodes is assigned a 4-bit code in the fingerprint.
4. The compiler now checks all candidates for outsourcing based on the fingerprint, and if possible swaps them out into functions.

The average code size reduction using this approach is up to 30%. However, use of Object-oriented programming languages like C++, Java and C# further increases the opportunities for code compaction.

Example of Procedural abstraction

```
Load  r1,$5200     Load  r1,$5200     Load  r1,$5300
add r1,r2          add r1,r2          add r1,r2
rot r1,$2          rot r1,$2          rot r1,$2
mul r1,r1          mul r1,r1          mul r1,r1
                                      ret
```

(a)Original Code

```
call f    call f    load r1,$5300    f: Load  r1,$5200
                    add r1,r2           add r1,r2
                    rot r1,$2           rot r1,$2
                    mul r1,r1           mul r1,r1
                                        ret
```

(b) Procedural Abstraction Variant I

```
Load r1,$5200      Load r1,$5200       Load r1,$5300
call f             call f              call f

                                      f: add r1,r2
                                         rot r1,$2
                                         mul r1,r1
                                         ret
```

(c) Procedural Abstraction Variant II

2.2.2 *Cross-Linking Optimization:*

Cross-linking optimization technique is used for optimizing switch statements when multiple cases have the same tail code. These common tail codes can be factored out to reduce the code size. Cross-linking optimization can be applied both within a function and across functions.

Example

```
switch(a) {
case 1:         statement1;
                CodeSegment1;
                break;
case 2:         statement2;
                statement3;
                break;
case 3:         statement4;
                CodeSegment1;
                break;
case 4:         statement5;
                CodeSegment2;
                break;
case 5:         statement6;
                CodeSegment2;
                break;
default:        CodeSegment1;
                break;
}
/* break jumps to here */
```

Code: (a) Un-optimized Code

```
switch(a) {
case 1:         statement1;
break1;
case 2:         statement2;
statement3;
break;
case 3:         statement4;
break1;
case 4:         statement5;
break2;
case 5:         statement6;
break2;
default:        break1;
}
/* break1: */
CodeSegment1;
goto break;

/* break2: */
CodeSegment2;
goto break;
/* break: */
```

Code (b): Optimized Code

In the above example, the original code (code (a)) has two common code tails (CodeSegment1 and CodeSegment2). CodeSegment1 is included in case 1 and case 2 whereas CodeSegment2 has been included in case 4 and case 5. If these common code tails can be removed from the source code, there will be a substantial reduction in code size. It is done in the optimized version of the code by generating three exit labels: break1, break2 and break. The label break1 contains codesegment1, break2 contains codesegment2 and the label break is for normal exit. The labels break1 and break2 contains jump to the normal break.

2.2.3 *Address-Code Optimization:*

It is sometimes a good idea to rearrange the layout of data in memory to reduce or simplify address computations. As number of data processing instructions increases, the size of the address manipulation code also increases. The address code optimization techniques attempt to optimize the address manipulating instructions themselves by rearranging the layout of data in memory.

There are two variants of Address Code optimization: Simple Offset Assignment and General Offset Assignment. Both of these reduce address computation code by rearranging variables in memory.

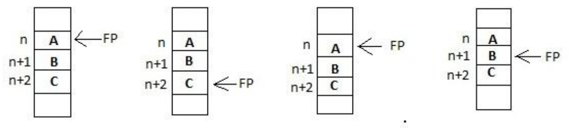

Fig: (a)

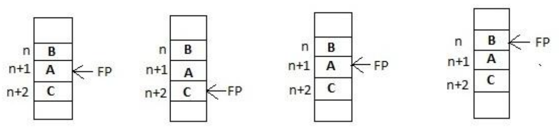

Fig: (b)

The above figure depicts the Simple Offset Assignment optimization in a DSP (Digital Signal Processing) specific compiler. The compiler allocates one of the Address Registers as a frame pointer (FP) which is used to address local variables on the stack. There are three local variables A, B, and C, on the stack. Suppose, the sequence S=(A,C,A,B) specifies the order in which these local variables will be accessed. Also, the auto increment range is suppose R=[-1,1].

In figure: (a), the variables are assigned to stack locations n, n+1 and n+2 in the order A-B-C and FP is initially pointing to variable A at address n. According to the sequence S, the next access goes to variable C located at address n+2. Thus, FP needs to be modified by the value +2. But, since the auto increment range is restricted to R=[+1,-1] and +2 does not fall in this range, the request cannot be served. The other two accesses of the variables B and C according to the sequence S require FP modifications by the values (-2) and (+1), of which only the last FP modification (+1) can be implemented by auto-increment. In order to implement the FP modifications +2 and -2, two extra instructions would be required. But, if the variables are reassigned to memory locations in the order B-A-C, these two extra instructions would not be required, since the access sequence now requires FP modifications by the values +1, -1 and -1. All of these values fall in the auto-increment range R and thus all FP modifications can be implemented by auto-increment.

It is therefore a good choice to rearrange the variable layout in memory to get better code. The goal of address code optimization techniques is to compute such good variable layouts.

2.2.4 Leaf Function Optimization:

A function that calls no other function is called a leaf function. These functions form the leaves of the call graph in the call graph representation. A leaf function is expected to run more efficiently if it does not make its own register window. Therefore the set of registers that would normally be reserved for making function calls and parameter passing can be used for general computation in a leaf function. In leaf-function optimization, non-leaf functions are transformed into leaf functions that can be represented as the leaves of a call graph.

There are several advantages of converting functions into leaf functions. Since leaf functions can be inlined easily, there is no need for function entry and exit code, which saves code size. Register constraints are removed which also saves code size. Besides, since the body of the inlined function becomes part of the parent function, it becomes easier to further optimize these functions as well.

Leaf functions can be derived from the original source code, but sometimes they are the results of procedural abstraction. Some non-leaf functions can even be treated as leaf functions.

For example, consider the factorial function

```
        int fact (int n, int acc)
        {
                if(n==0)
                return acc;
                else return fact(n-1, acc*n);

        }
```

This function has a recursive call at the end of the function. This recursive call can be transformed into a loop by implementing it as a jump to the beginning of the function. If implemented as a loop, the function behaves as a leaf function. However, in order to transform this non-leaf function into a leaf function, an accumulator variable need to be added in the function.

For the special treatment of leaf functions, some other conditions must also be met; for example, the registers must be used for their own variables and temporaries.

The GCC Compiler performs register numbering before it knows whether the function is suitable for converting to a leaf function. It therefore needs to remember the registers in order to output a leaf function. The GCC uses the following macros to accomplish this.

(i) Macro: LEAF REGISTERS

(ii) Macro: LEAF-REG-REMAP

2.2.5 Type-Conversion Optimization:

The type conversion optimization attempts to reduce the code size by reducing the size of the data itself.

Most of the compilers insert many implicit type conversions to support a variety of data types. For example, In C programming language arithmetic operators operate on integer-sized values (typically 32- bit) only. The compiler will always select the most efficient integer size if we declare an int, without specifying the size. Integers of smaller sizes (char, short int) will be converted to types to integers of the default size when doing calculations, and only the lower 8 or 16 bits of the result will be used. We can assume that the type conversion takes zero or one clock cycle. In a 64-bit system, there is only a minimal difference between the efficiency of 32-bit integers and 64-bit integers.

Consider the following C language statement:

```
char a,b,c;

………

c=a+b;

………
```

Before doing the addition, the C compiler will promote char (say signed 8-bit) variables a and b to integer types of the default size. The result of this addition is of integer type and it must be demoted to char type before being stored in the char variable c. If a and b are stored in memory, four instructions would be performed: two loads for loading a and b, one add for the addition and one store for storing back the result in c. However, if a and b are stored in registers, the compiler must generate code that sign extends a and b before doing the addition. The compiler will generate two instructions for each sign extension: a left-shift to move the char sign bit into the int sign bit, and then a right shift to restore the value and propagate the sign bit into the upper 24-bits of the register. However, if it is known in advance that the result of the addition will be converted to a char type then there is no need to sign extend a and b before addition. The addition can be performed with 8-bit precision. Thus a careful analysis before performing the addition can eliminate the sign extension operations.

The type conversion optimization techniques use basic principles type analysis to remove redundant type conversions that can produce useful reductions in code size.

2.2.6 Multiple Memory Access optimization

The instructions which load or store two or more registers simultaneously are called Multiple Memory Access (MMA) instructions. Many microprocessors use MMA instructions for reducing code size. MMA instructions include multiple loads and multiple stores where a set of registers are loaded from, or stored to, successive instruction words in memory. For example, the ARM7 has LDM and STM instructions for Load-multiple registers and Store-multiple registers. There is a significant compactness in code size by expressing a large set of registers with few instruction word bits. For example, in a ARM7 processor, the LDM instruction uses only sixteen bits to encode up to sixteen register loads (512 bits without the use of the LDM instruction).). Effective use of LDM and STM instructions in a ARM7 processor can save up to 480 bits opcode space.

2.2.7 Combined Code Motion and Register Allocation

This technique combines two traditionally antagonistic compiler phases: code motion and register allocation. Code motion aims to place instructions in less frequently executed basic blocks, while instruction scheduling within blocks or regions arranges instructions such that independent computations can be performed in parallel. The Register Allocation and Code Motion (RACM) algorithm aims to reduce register pressure firstly by moving code (Code Motion), secondly by Live-range splitting (Code-Cloning), and thirdly by spilling. This optimization is applied to the VSDG intermediate code, which greatly simplifies the task of code motion. Data dependencies are explicit within the graph, and so moving an operation node within the graph ensures that all relationships with dependent nodes are maintained. Also, it is trivial to compute the live range of variables (edges) within the graph; computing register requirements at any given point (called a cut) within the graph is a matter of enumerating all of the edges (live variables) that are intersected by that cut.

3. CHALLENGES IN CODE OPTIMIZATION

3.1 Code optimization for Parallel Processors

Code Optimization has a rich history that dates back half a century. The researchers on code optimization have contributed significantly to programmer productivity by reducing the size and execution time of the compiled code and by making code more portable. Often these innovations were accomplished by the introduction of new ideas such as interprocedural whole program analysis, pointer alias analysis, loop transformations, adaptive-profile directed optimizations, and dynamic compilation.

The current multi-core trend in the compiler industry is forcing a paradigm shift in compilers to address the challenge of ‘code optimization of parallel programs’. All computers-embedded, mainframe and high-end, are now being built from multi- core processors with little or no increase in clock speed per-core. This trend poses multiple challenges for compilers of future systems as the number of cores continues to grow, and the cores become more heterogeneous. In addition, compilers have to keep pace with a sudden increase of new parallel languages and libraries.

3.2 Code Optimization for Embedded processors

Computers are everywhere today. Beyond the desktop PC, embedded computers dominate our lives. Almost all electronic gadgets including digital alarm, digital radio and television, car fuel indicator, micro-wave ovens, remote-controlled devices etc. use embedded processors. Extensive growth of these embedded systems demands more efficient software to operate these systems. A crucial aspect in coding for these systems is that these systems use limited memory. Thus, code size reduction is particularly important for these systems. Continuous researches in the embedded systems results in the development of even smaller and smaller devices which use very little memory. Thus, code optimization for reducing size is a major challenge in case of the embedded processors. Also, since these systems run on battery power, developing code that consumes less power is another challenge for these systems.

4. CONCLUSION AND FUTURE SCOPE

This research paper tries to introduce the current trends in code optimization. Since, most of the compiler researchers are conducting their researches on code optimization which results in the development of newer techniques for code optimization on a daily basis, it is not possible to cover all the new techniques in this research paper. This paper includes a few of these techniques and the basic principles of these techniques. Since code optimization is a field of broad research, it is not possible to cover all aspects of code optimization in this paper.

The future scope of this research is to develop some new techniques based on these existing ones. This research paper will help the programmers to do smart coding by identifying the redundancies in their code and by applying these optimization techniques to their programs. This will also help the compiler designers to provide more optimization techniques in their compilers.

5. REFERENCES

[1] Alfred Aho, Ravi Sethi, Jeffrey D Ullman, “Compilers Principles, Techniques and Tools”, Pearson Education Asia

[2] O.G.Kakde, (2008), “Compiler Design”, Universities Press

[3] Carole Dulong, Rajiv Gupta, Robert Kennedy, Jens Knoop, Jim Pierce (editors), (2000) “Code Optimization – Trends, Challenges, and Perspectives” Dagstuhl-Seminar-Report; 286, 17.9.–22.9.2000 (00381)

[4] Caspar Gries, (2009), “New Trends in the Optimization of C-Code”,

[5] Kenneth Hoste Lieven Eeckhout,ELIS Department, Ghent University,” COLE: Compiler Optimization Level Exploration”

[6] Urban Boquist, “Code optimization Techniques for Lazy Functional Languages”, Thesis for the Degree of Doctor ,Goteborg University,

[7] Stan Yi- Huang Liao, “Code Generation and Optimization for Embedded Digital Signal Processors”, Massachusetts Institute of Technology

[8] Neil Edward Johnson, “Code Size Optimization for Embedded Processors”, Robinson College, Thesis for the Doctor of Philosophy at the University of Cambridge

[9] Neil Johnson and Alan Mycroft, “Using Multiple Memory Access Instructions for Reducing Code Size”, University of Cambridge

[10] Johnson, N., and Mycroft, A., (2003) “Combined Code Motion and Register Allocation using the Value State Dependence Graph.” In Proc. 12th International Conference on Compiler Construction (CC'03) (April 2003), vol. 2622 of LNCS (Springer-Verlag)

[11] Gergö Barany, “Integrated Code Motion and Register Allocation”, Thesis for the Degree of Doctor, Vienna University of Technology

[12] Qingfeng Zhuge, Bin Xiao, Edwin H.-M. Sha,“ Performance optimization of Multiple Memory Architectures for DSP”

[13] Josef Weidendorfer, “Analysis and Optimization of the Memory Access Behavior of Applications”

[14] Prajakta Gotarane, Sumedh Pundkar, (2015) “Smart Coding using New Code Optimization Techniques in Java to Reduce Runtime Overhead of Java Compiler”, International Journal of Computer Applications (0975 – 8887), Volume 125 – No.15, September 2015

[15] Keith D. Cooper 1 and L. Taylor Simpson, “Live Range Splitting in a Graph Coloring Register Allocator”, Rice University, Houston, Texas, USA,

[16] Preston Briggs, Keith D. Coope,Linda Torczon,, “Aggressive Live Range Splitting”, Houston University, Texas, USA

[17] Michael Burke, Linda Torczon, “Inter-procedural Optimization: Eliminating Unnecessary Recompilation”, IBM Research, Rice University