CS F363 Compiler Construction

Technical Report

Contemporary Advancements in Ever-Evolving Compiler Optimization

Gumma Varun
2017A7PS0165H

SNS Maneesh Sarma
2017A7PS0238H

Muzaffar Ahmed
2017A7PS0248H

Pavan Srihari Darbha
2017A7PS0011H

1 Introduction

With the onset of new high-level languages, computationally intensive programs and embedded systems, the demand for better, faster and seamless processing sets in. Latest CISC architectures come with VLIW (Very Large Instruction Word) which demand larger ISA (Instruction Set Architecture) and instruction size. But to maintain or achieve larger execution rates, the intermediate/machine code generated must be as minimal as possible while not compromising the semantics of the user's program. This in turn requires the back-end of the compiler to be as optimized and accurate as possible.

So, in this report, we try to analyse the state-of-the-art techniques in compiler optimization such as CCMRA, Leaf-Function Optimization, Type Conversion Optimization and MMA Optimization along with the traditional optimization techniques like dataflow analysis, dead code elimination and loop unrolling etc.

2 Methodology

2.1 Conventional Optimization Techniques

In this section, we will have a look at the traditional optimization techniques implemented in regular compilers.

2.1.1 Dataflow Analysis

Dataflow analysis deals with the flow of data in a Control Flow Graph (CFG). The user program after being converted to Three Address Code (TAC) is divided into blocks according the *leaders* in the code. Each block can be a Definition Point, Evaluation Point or Reference Point for a variable. A definition D is said to reach a point s in the CFG if and only if it is not *killed* (reassigned) in the path to s. This technique helps to know how far a variable is *available* so that it can substituted with its value along its *available path* during compile time itself to avoid unecessary evaluation overhead during runtime.

2.1.2 Constant Propagation

This optimization is a direct result of the Dataflow Analysis. In this case, the compiler replaces the occurences of a variable by its constant assigned value (if one exists) till wherever the variable reaches. This is another corollary of Dataflow Analysis. When an assignment operation is carried out between two variables (say $y = x$), we can use the value of y in all paths where x and y are both reachable, i.e. neither of them has been *killed*. This reduces the overhead of handling an extra variable. Copy Propagation leads to dead code; a block of code which is either redundant or never executed or is a non-contributor to program evaluation. These code pieces can be safely removed. This is helpful in reducing the number of instructions to be processed later.

Example of dead code: `x = 0; y = x; if(y) printf("here");`
It can be very clearly observed that y is assigned 0, and the `if` statement is never executed (dead code) and hence can be removed.

2.1.3 Constant Folding

This technique is an extension to Constant Propogation. When the values of variables are available at compile time, the values are substituted and results of any operations on those variables are computed and directly stored or used further. This reduces the overhead on computation during runtime.
Example of Constant Folding: `a = 3; b = 5; return ((a + b) * a * b);`
This whole snippet of code corresponding to a function can be boiled down to a single statement `(return 120;)` by precomputing the value of the expression to be returned with the available variables.

2.1.4 Loop Unrolling

The execution time of a program is highly dictated by the number of loops and size of each loop, and hence optimizing loops so that they have fewer iterations to perform gives us a significant advantage in terms of runtime. A typical `for` loop has an initialization statement, an increment statement and a boundary check statement. For a `for` loop running N times, the initialization occurs once, the boundary check statement is executed $N + 1$ times and the increment statement is executed N times. The overall *meta-loop* overhead for a for loop is $2N+2$ which is more than twice the number of times the body of the loop is executed! In loop unrolling, the loop statement is first identified in the TAC generated (as a cycle between the blocks) and then unfurled, in the sense that the loop block is sequentially written N times (supposing the loop is executed N times) in the source code and then executed, rather than using a loop. This eliminates the *meta-loop* overhead completely. This does drastically increase the source code size and is known as *space-time* tradeoff. This technique can only be used if we know the number of iterations of the loop (N) beforehand.
Example of a regular loop: `for(i = 0; i < 3; i++) printf("here");`
Equivalent unrolled loop: `printf("here"); printf("here"); printf("here");`

Other primitive optimization techniques include Redundancy Elimination, Strength Reduction and Algebraic Reduction, which are not discussed here.

2.2 Newer Optimization Techniques

Now, we attempt to discuss the newer optimization techniques discussed in the paper.

2.2.1 Reverse-Inlining

Inlining refers to the replacement of a function by its equivalent code similar to Macro substitution. This occurs during runtime, unlike the latter which occurs during the preprocessing phase itself. Inlining increase the size of source code, but reduces the over head of function calls related to stack frame creation, memory allotment and stack frame removal. In reverse-inlining (a.k.a procedural abstraction), the common and recurring code snippets are replaced by function calls. Using this method a reduction of upto 30% is observed in the source code size. This does increase the overhead of stack frames

management. Reverse-inlining can be very useful in embedded computers, where the instruction cache for the processor is limited.

2.2.2 Leaf Function Optimization

Leaf function calls refer to those function calls which do not call any other function directly. In this technique, non-leaf functions are converted to leaf functions as leaf calls can be inlined and register overhead for creating function calls and parameter passing can be used for computation in the function. Non-leaf functions can be converted to leaf functions by converting the function calls to loops. A function call can be converted to a loop by a adding jump statement to the start to the function, thus converting it to a leaf call. This conversion has a few more requirements; the registers must be used for their own variables and temporaries. Now that the function has been inlined, various further optimizations can be applied on the inlined method. We can see that this method proves to be highly useful when we have a very limited set of registers.

2.2.3 Type Conversion Optimization

Consider the following code snippet: `char a, b, c; c = a + b;`
In this case, we observe that our attempt is to add two characters (assuming char takes one byte) and store the result in another character. With addition being defined only for integer operands, (assuming integer takes four bytes) this program, when converted to machine code, has two load byte operations, two sign extend operations (to convert 1B to 4B), one ALU operation and one store byte operation. Here, the sign extension of the operands is an unecessary operation which can be avoided by analysing the output data type. If the output type is known to be char, the sign extension operation can be safely skipped and an 8-bit addition is performed without a conversion to 32 bits. Though this optimization saves hardly a clock cycle or two, its effect is significant when working with smaller registers (like 1-2 Bytes).

2.2.4 Multiple Memory Access Optimization

Loading or storing multiple registers or memory locations simultaneously is called MMA. To illustrate a simple example of MMA, let us consider c = a[2] + a[3] in which we load registers from two memory locations at the same time. By implementing MMA, we can load both registers using a single instruction parallelly, rather than using two separate load calls. This helps in reducing the source code size as well as the overhead of multiple discrete memory accesses. This technique proves to be useful with processors which have good support for parallelism.

3 Suggestions/Drawbacks

Some of the optimizations we have mentioned in this paper are processor specific such as MMA, which may not be possible to implement in regular compilers. We need not limit ourseleves to just fetching operands of current instruction in MMA, but can fetch operands of any neighbouring instructions, i.e. following 1-2 instructions, if they contain any simultaneous data. This obviously puts an overhead on the number of available registers as we will need extra *free* registers to be available during execution.

Another theoretical optimization can be the amalgamated use of loop unrolling and

reverse-inlining. The unrolled loop consists of multiple repeated code patterns which can be converted to function calls which can reduce the size of the source code. This does add the overhead of stack operations, but can be executed (stack frame creation, variable memory allotment and frame disposal) quicker than loops (with lesser meta-instructions) if they do not contain significant recursive calls.

4 Summary

This report tries to shed some light on some newer as well as standard optimization techniques being used in various compilers. Processors specific to a certain task (i.e. GPUs, TPUs) or with structural limitations (i.e. Embedded Systems) require special compilers which can optimize the code such that the task can be performed with minimum overhead. Also, the newer compilers will be able to handle larger instruction words with limited instruction cache and Data cache with these optimizations. Some of these optimizations, which are specific to machine architecture, can help compiler developers build better back-end and front-end for the compiler for packages like TensorFlow and Torch and for new upcoming languages.

5 References

1. Enyindah and Uko: *The New Trends in Compiler Analysis and Optimizations*
2. Y.N Srikant, IISc Banglore: *NPTEL Course on Compiler Design*
3. Anjan Kumar Sarma: *New trends and Challenges in Source Code Optimization*