Low-Level Software Optimization

Christoph Strößner

Sommerakademie in Leysin, August 2016

Abstract

Low-level software optimization focuses on improving the performance and energy consumption of a given program. During compile time the code can be modified to be more suitable in general and specifically for the desired system. The reduction of the number of unnecessary and redundant processor operations always leads to a more efficient program. Additionally the memory can be managed in order to obtain a better cache miss ratio. Also pipelining and dynamic-voltage-and-frequency-scaling profit from software optimization during both compile- and run- time.

Contents

1 Introduction

This paper gives an overview of possible techniques to improve the performance of a program. Usually several of these are used to modify the given code without changing its functionality. Without such optimization it would usually be rather slow and less energy efficient.
When executing a program the way it is written, it contains many unnecessary and redundant assignments. Especially loops and equations can often be simplified a lot. This can lead to a significantly faster execution of the given code. Also "nowadays" programmers do not care too much on the optimal usage of memory allocation, pipelining or dynamic-voltage-and-frequency-scaling. This way parallelism is only exploited partially.

2 Compiler Optimization

This section gives an overview on how a compiler modifies the program code when set to optimize the performance. The compiler can not improve the general program design nor the algorithms and data-structures, which are the most important factors to be considered. Yet manly low-level optimizations are possible.

2.1 Simplification of Equations

The first optimization method is presented using an example. Consider the calculation of an interpolation polynomial through three points. Mathematically this problem can be expressed as follows:

$$\text{given } x_1,\ x_2,\ x_3 \subset \mathbb{R},\ f : \mathbb{R} \to \mathbb{R},\ y_i = f(x_i) \text{ find } p(x) = a_2x^2 + a_1x + a_0 \text{ such that } p(x_i) = y_i$$

When solving this equation to calculate the variables a_1, a_2, a_0 naively three equations are obtained:

$$a_0 = \frac{-x_2^2x_3y_1 + x_2x_3^2y_1 + x_1^2x_3y_2 - x_1x_2^2y_3 + x_1x_2^2y_3 - x_1x_3^2y_2}{(x_1 - x_2)(x_1x_2 - x_1x_3 - x_2x_3 + x_3^2)}$$

$$a_1 = \frac{x_2^2y_1 - x_3^2y_1 - x_1^2y_2 + x_2^2y_2 + x_1^2y_3 - x_2^2y_3}{(x_1 - x_2)(x_1x_2 - x_1x_3 - x_2x_3 + x_3^2)}$$

$$a_2 = -\frac{-(x_3 - x_1)(y_1 - y_2) + (-x_1 + x_2)(y_1 - y_3)}{-(x_1^2 + x_2^2)(-x_1 + x_3) + (-x_1 + x_2)(-x_1^2 + x_3^2)}$$

This formula can be implemented directly. But many calculations would be done several times. It is possible to pull expressions occurring multiple times out of the formula and to calculate them in advance:

$$q_1 = x_1^2, q_2 = x_3^2, q_3 = x_2^2, q_4 = x_1 - x_3, q_5 = \frac{1}{q_4(x_1 - x_2)(x_2 - x_3)}$$

$$a_0 = q_5(-q_4x_1x_3y_2 + x_2(-q_2y_1 + q_1y_3) + q_3(x_3y_1 - x_1y_3))$$

$$a_1 = q_5(q_2(y_1 - y_2) + q_1(y_2 - y_3) + q_3(-y_1 + y_3))$$

$$a_2 = q_5(x_3(-y_1 + y_2) + x_2(y_1 - y_3) + x_1(-y_2 + y_3))$$

This technique is called *common subexpression elimination*. In this case it reduces the number of flops required from 116 to only 55 by introducing five more temporary variables. However only common subexpression elimination should be applied. Changing the equations in a different way may algebraically express the same, but it might change the numerical stability, which usually is not wanted. E.g. $a_2x^2 + a_1x + a_o$ and $x(a_2x + a_1) + a_0$ behave differently when executed on a computer.
Overall common subexpression elimination is quite useful and can improve the performance significantly by only increasing the number of code lines slightly. It usually is left to be automatically applied by the compiler as the modified equations can be significantly harder to read.

2.2 Loop Unrolling

An other universal technique to improve programs is *loop unrolling*. The idea is to remove loops of known length from the code and to replace them by listing all the calculations one after an other. This increases code size tremendously. However the execution is faster as the jump and compare orders can be skipped completely.
One example for this method is the calculation of matrix products. Let the matrices be symmetrical and of order 3. Then the code usually looks as follows:

for $i = 1 : 3$
for $j = 1 : 3$
$M_{i,j} = 0$
for $k = 1 : 3$
$M_{i,j} = M_{i,j} + A_{i,k}B_{k,j}$

Loop unrolling leads to nine lines of code and requires no loops at all:

$M_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} + A_{1,3}B_{3,1}$,
$\cdots$
$M_{3,3} = A_{3,1}B_{1,3} + A_{3,2}B_{2,3} + A_{3,3}B_{3,3}$

The new code contains exactly the same operations as the original code just without the loops. This means it is executed faster. Also it is more suitable to apply common subexpression elimination in this form.

2.3 Method Overview

There are several other optimization methods. A few of the most important ones are listed in this section.
Inlining: Instead of using a function call the function could be written directly within the code. This leads to faster execution as the function call requires quite some runtime.
Constant Calculation: Calculate all values that can be computed during compile time and use those directly instead of equations.
Loop Inversion: By using the if do while statement two jump statements can be saved overall. This is a minor improvement.
Factoring out invariants: By removing invariant lines out of loops redundant calculations can be avoided.
Remove Recursion: Recursion requires a lot of function call management. It is usually faster to

replace tail recurrent code by an iteration.
These are just a few methods. In practice many more methods are used and applied several times after each other.

2.4 Example: Technical Compiler Optimization

This section explains how the methods mentioned previously could be implemented. Therefore an intermediate language is used. It allows to do these optimizations more easily than high level or assembly code. Also it is kept on a machine independent level such that the final optimization for the specific machine can be quickly applied afterwards.
The language contains the following orders:

1. register adresses: $x, y, a, b, \cdots$
2. basic operations: $+, *, \cdots$
3. jumps to jump labels: $L:$
4. conditional jumps: $if\ A\ goto\ L$

A *basic block* is a series of orders not containing any conditional jumps of labels apart from the first and last order.
This definition allows the partition of the code in basic blocks. Each of these blocks can be optimized directly without caring about branching. Also a *control flow graph* can be generated. Its vertices are the basic blocks and the edges are given by possible jumps after the execution of the basic block. This graph can be used to do *branch prediction* - the estimation of jump targets in order manage the memory correctly such that variables required soon are automatically put in the best possible cache. This can be done by running the program a few times and analyzing the control flow within the graph in order to create a Markov-chain.
When analyzing singe basic blocks the easiest optimization is to remove unnecessary statements such as adding zero to a variable. By iterating through the basic blocks one can also easily eliminate unreachable blocks completely. Furthermore a useful form for further optimization methods can be obtained: *Single assignment form* assigns each register only once within a block. This means every statement $x = ...$ is the only statement containing x on the left side. Especially common subexpression elimination can be directly applied afterwards.

2.5 Memory Optimization

Memory Optimization is the device specific optimization of a program, such that the memory is used optimally. One of the strategies hereby is to use *prefetching*: The loading of variables to registers or low level caches before they are required. This allows to save a lot of time since the processor does not have to wait as long to get variables. This is usually achieved by using branch prediction as introduced before.
In order to optimize the memory usage it is necessary to use the advantages provided by hardware to full extend. Main memory usually allows for *burst access*: accessing a sequence of memory locations and *pages memory* the fast access to data on the same memory page. The cache can be optimized by keeping important variables accessible in low level caches and by pulling several values with a single cache. Also data should be stored in cache lines within access order to save energy. This is achieved in practice by counting access numbers and using placement heuristics.

2.6 Example: Compiler Optimization with GCC

The programmer usually leaves the low-level software optimization for the compiler. That way he can create more readable and easy to understand code, while still achieving decent optimization in the final program.
One of the compilers for C / C++ is GCC. GCC offers several optimization levels which can be selected within its settings. There is the default level $O0$ which barely optimizes and allows the programmer to debug his code. The levels $O1$ and $O2$ perform stable optimization in order to improve the code. When set to $O3$ some "experimental" methods will be applied. Those do not guarantee that numerical stability is preserved and should thus not be used, when exact calculations are critical. $Ofast$ takes this even further. Numerical Stability is ignored for even faster run time execution. A different approach is taken by the compiler setting Os instead of runtime the required memory of the program is minimized. This basically does the opposite of loop unrolling and common subexpression elimination.
When set to $O1$ the GCC compiler for example eliminates dead code, which is not reachable within a normal program flow. Further is moves invariants out of loops, tries to do branch prediction in order to prefetch the right variables and many more individual optimizations. When set to $O2$ these methods are extended in that the common subexpression elimination is no longer limited to individual functions but is applied globally on the code.

3 Conclusion

To put it into a nutshell many optimization methods exist which improve the performance of a program. Some of them are more significant than others and some are only usefully when optimizing for a certain goal (e.g. memory / runtime), whereas others depend on the underlying system and optimize to use the fully exploit the advantages of memory management, pipelining and dynamic-voltage-and-frequency-scaling. Overall the optimization (either done by hand or, as it is usually done, by the compiler) leads to a program running faster and more energy efficient.

4 Bibliography

W. Wolf, High-Performance Embedded Computing, 2007, Elsevier

D. Patterson, Computer Architecture A Quantitative Approach, 2011, Elsevier

S. Kaxiras , Computer Architecture Techniques for Power-Efficiency, 2008, Morgan Claypool Publishers

A. Dall'Osso, Computer algebra systems as mathematical optimizing compilers, 2006, Elsevier