

New Trends in the Optimization of C-Code

Caspar Gries

January 11, 2009

Abstract

Despite all of the recent progress concerning the tools and techniques of Software Development, which allow for shorter development cycles by automating common tasks and generally shifting the field of human engagement to more abstract levels, only a small fraction of all projects benefit from this advancement. A great part of Software Development still uses concepts that have been devised decades ago, namely the C programming language. The reasons for this are manifold. Firstly, there are billions of lines of legacy code, which can only be re-written and re-tested at great financial and personal expense. Secondly, the C programming language has up-to-now-unmatched qualities in many areas, including portability, speed and low-level hardware access. Lastly, the increased emergence of embedded systems specialized on all kinds of different tasks demands for a minimal, yet powerful programming language.

But while C itself has changed not much over time, the techniques of compiling it into efficient machine code have been subject to constant research since its very beginning. This paper summarizes some of the more recent trends and approaches.

1 Compiler Optimization Level Exploration (COLE)

Compilers these days have a large number of built-in optimization features, most of which can be enabled or disabled independently of all of the others. For example, the popular gcc compiler has about 60 separate optimization features. To facilitate the handling of such a great amount of options, statically preselected sets of features have been defined, called “optimization levels”. These sets determine the degree of optimization applied, as well as certain special optimization targets (e. g. size). Manually defining good optimization levels is a quite

tedious task, because of the huge number of possible combinations, and also because some features might influence each other.

[Hoste08] suggests another approach: Defining appropriate optimization levels by using a modified evolutionary algorithm. The modification, which is explained later on, is necessary because the present problem is a multiobjective optimization: Any given optimization level should provide an optimal compromise between compile speed, execution speed, program size et cetera. These sets are called “Pareto optimal”. More generally spoken, a given set of options is Pareto optimal if no further improvement can be made without a de-

terioration of another objective function. However, there exist multiple Pareto optimal sets, which represent points in a n -dimensional graph, whereas each dimension is one optimization target. If all of these sets are connected, the so-called Pareto frontier becomes visible. This frontier represents the maximum possible degree of optimization.

One of the most important aspects of the work is the algorithm used. It is a very straightforward evolutionary algorithm, but with some specific modifications. There is not only one population, but instead multiple “mating pools”, in which the populations evolve independent from each other. Single individuals may randomly migrate between mating pools. The fitness of an individual is determined by compiling a set of benchmarks with the optimization settings the individual represents. In case of equal fitness, the solution with less activated optimization compiler switches is preferred. During the execution of the algorithm, evolution then occurs by crossover, mutation, migration and selection, until no more improvement can be measured. After termination, every population has its own Pareto frontier. The overall Pareto frontiers is then computed by comparing all of the individual frontiers.

The results are quite surprising. The manually defined optimization levels of the gcc compiler are far from optimal: By using evolutionary algorithms, an improvement in execution speed of 3,1% can be achieved, at the same time reducing the compilation time by 37,6%. What is far more interesting, however, is the fact that within the 12 Pareto optimal optimization levels found by the algorithm, only 15 of gcc’s 60 compiler optimizations are used at least once. This means, that many of the optimization features gcc provides might be

more or less without any effect.

2 New Optimization Techniques

2.1 Code Size Reduction

Even though reducing the compiled code size has become a low-priority target due to the steadily decreasing cost of (bulk) memory, it is still favourable to spend a one-time effort on the code size instead of wasting memory on every machine the code runs on. This holds especially true for embedded systems, where the memory usage of the software has a direct influence on the unit price.

[Debray00] has refined an already existing optimization technique to save memory, called *squeeze*. It is basically the opposite of the so-called “inlining” of code, where the call to a function is replaced by a copy of its body. However, simply not inlining functions would be trivial. The idea is therefore to find similar sequences of instructions and replace them with a call to a newly created function. This process has several steps, which operate on the machine code generated by a compiler.

1. The code is divided into basic blocks. A basic block is a linear sequence of instructions which does not contain any instructions that directly manipulate the program counter (such as jumps, function calls, instructions that might throw an exception and so on.)
2. As an intermediate step, a callgraph is then created. It works as follows: All blocks are considered unused except the one where program execution starts. By following the paths of

execution, the blocks which are actually entered by the control flow are determined. When the callgraph is complete, all unused blocks are removed as a first measure of optimization.

3. A fingerprint is computed from each basic block, which is used to find candidate blocks for outsourcing. The algorithm which does the fingerprinting simply creates a 64-bit value from each basic block by looking at the first 16 opcodes of the block, assigning each a 4 Bit code in the fingerprint. Because there can be only 16 opcodes represented, a mapping ta-

ble with the 15 most frequently occurring opcodes plus one for “other” is previously calculated. Fingerprints are, however, only a rough criterion to find potentially matching blocks.

4. Based on the fingerprints, the core algorithm checks all candidates, swapping them out into functions if possible.

Step 4 is the most complicated of all. To increase the effectivity of squeeze, not only byte-identical blocks are considered, but also semantically similar ones. In the simplest case, instructions have to be reordered or registers renamed.

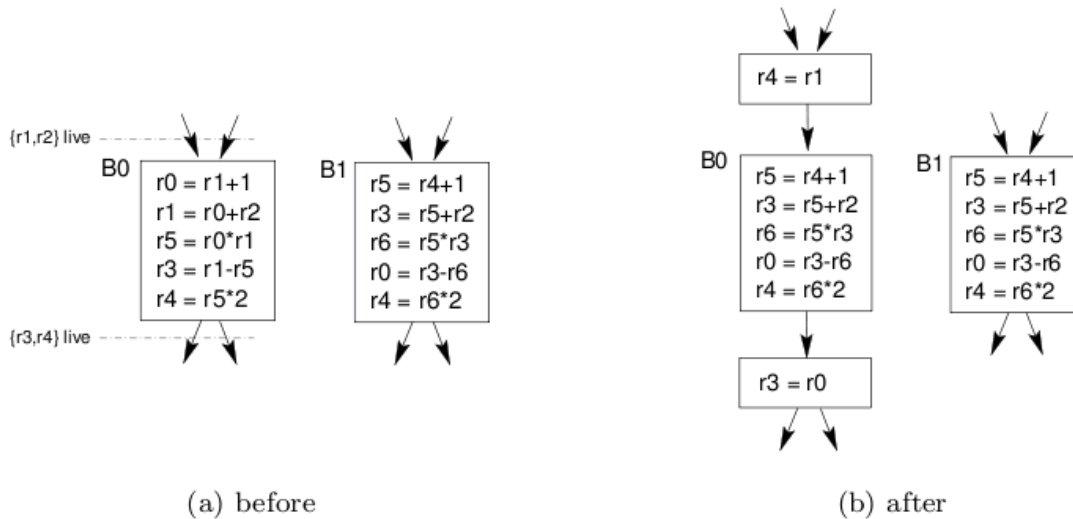


Figure 1: Register renaming within semantically similar blocks. (Source: [Debray00])

Besides simple reordering, squeeze’s algorithm is capable of many more things. For example, a code segment that is to be abstracted into a function might consist of more than one basic block, thusly containing jump and call instructions. In that case, it might be necessary to allocate addi-

tional space on the stack used for return addresses, without introducing errors so that the optimized code changes its behaviour in undesirable ways. Unfortunately, a thorough description of these procedures would go beyond the scope of this article. The average code size reduction using this ap-

proach amounts to 30%.

2.2 Using De-optimization to Re-optimize Code

This approach sounds contradictory at the first glance and needs further explanation. Every optimization which makes use of different techniques that are applied sequentially will encounter a problem known as phase ordering. In short, this means that the total effectivity of the optimization depends on the order in which the steps are applied. Each step spends resources like registers, which are then no longer available. This applies not only to the optimization done automatically by compilers, but also to hand-tuned assembly code, because programmers have an inherent “order” by which they try to optimize their code. What makes it even worse is the fact that there is no ideal order of the optimization phases for a complete program. Instead, the optimal order of optimization may depend on the respective algorithm and architecture. One way to handle the phase ordering problem is suggested by [Hines05]. Their solution, called VISTA (VPO Interactive System for Tuning Applications) is based on the optimization framework VPO (Very Portable Optimizer), which was created by [Benitez88]. The main improvement VISTA introduces are the interactive aspect, which makes it possible to manually engage with the optimization process and the core component which allows for execution and rollback of optimization steps. It uses, rather unsurprisingly, an evolutionary algorithm to determine the best order of optimization steps. VISTA makes use of a special intermediate source format, called RTL. A translator program is used to transform assembly code to RTL format. At the end

of the optimization progress, machine code for the target platform is generated.

The results show, that by applying re-optimization without prior de-optimization, the average code size as well as the instruction count go down by 2-3%. With de-optimization enabled, this result will improve even a bit more, while in no case the outcome was worse than with optimization alone.

2.3 Continuous Program Optimization

[Kistler03] addresses a yet unused potential for optimization of every type which is surprisingly often neglected, probably because there have been no appropriate techniques hitherto. This potential lies in the lack of optimization of software to its target platform. As usual, it has several different aspects:

Legacy Software While the release cycles of new hardware platforms today are steadily becoming shorter, users do not want to change the software they use with each hardware upgrade. This leads to a significant percentage of 32-Bit systems running 16-Bit software, and sometimes whole processors being emulated by software to be able to run old applications. Obviously, this backward compatibility comes with a price, which is mostly the emulation speed, or rather, the lack thereof.

Hardware diversity The number of major CPU platforms like x86, MIPS, etc. is limited. But most platforms have multiple manufacturers, and each of them produces a whole array of different CPU models. All processors that belong to one platform are guaranteed

to implement all of the architecture's core features, so that code compiled for a certain architecture will run on any of them. However, most processors possess some extra features, like opcodes which can carry out special operations faster than usually possible. Generic software now has two choices: Either it uses only the core features, surrendering the possible speed advantage, or it makes use of special code for different CPU models, increasing the code size thereby.

Modularization One of the most important factors of modern software design is the reusability of code. Much effort is invested into modularizing functionality to save money and time. By using dynamically linked libraries, the different components are integrated at runtime on the client system. This approach comes with a cost, however. At link time, it is too late to make any optimizations which could have been applied to statically linked code.

The source of all these problems is the fact that at compile time, there is not enough information available to produce better code. The compiler has therefore to be overly conservative about the optimizations it can use. An obvious solution is to defer code generation until the client specs are known. Several operating systems like the different BSD flavours and Gentoo Linux are doing this since the beginning of their existence.

[Kistler03] takes this one step further by introducing a run-time optimizer. This software profiles the target application constantly, collecting statistical data for future optimization. If enough data has been collected, the application is then hot-swapped

against a new, better version of it that has been compiled in the background.

Thusly, a given application can not only be adjusted optimally to the hardware, it can even be optimized such that the common usage patterns can be executed especially fast. This holds true even for changing usage patterns, because of the constant profiling.

In detail, the system consists of multiple components. First, there is the code-generating loader, which compiles the source code on the first start of an application. It does very little time intensive optimization, because the user has to wait until the loader has done its work. Next comes the already mentioned profiler. It records different parameters of the applications currently executed. Most important here is the number and type of variable accesses, and cache hits and misses within different functions, revealing information about the data locality of the code. The information gathered by the profiler is stored in a database, that the system manager uses to generate a list of functions where optimization would be worthwhile. Especially, re-optimization is considered if changes in the system usage characteristics are detected. These may occur if, for example, a shared library is used by another application which has a different usage profile. The system manager queries the optimization manager for a prioritized list of optimization candidates, which are passed to the optimizer in the order of expected improvement. When a new set of functions has been calculated, the replacer then substitutes the old code on the fly for the new. Therefore, even optimization techniques that take a long time to perform and yield rather little advantage can still be considered useful.

Like all approaches, the benefits are not

for free. Only if a program is run for a certain time, the runtime optimizer can improve it sufficiently, so that it is more ef-

ficient than a program which did not undergo that procedure. Fig. 2 exemplifies that effect.

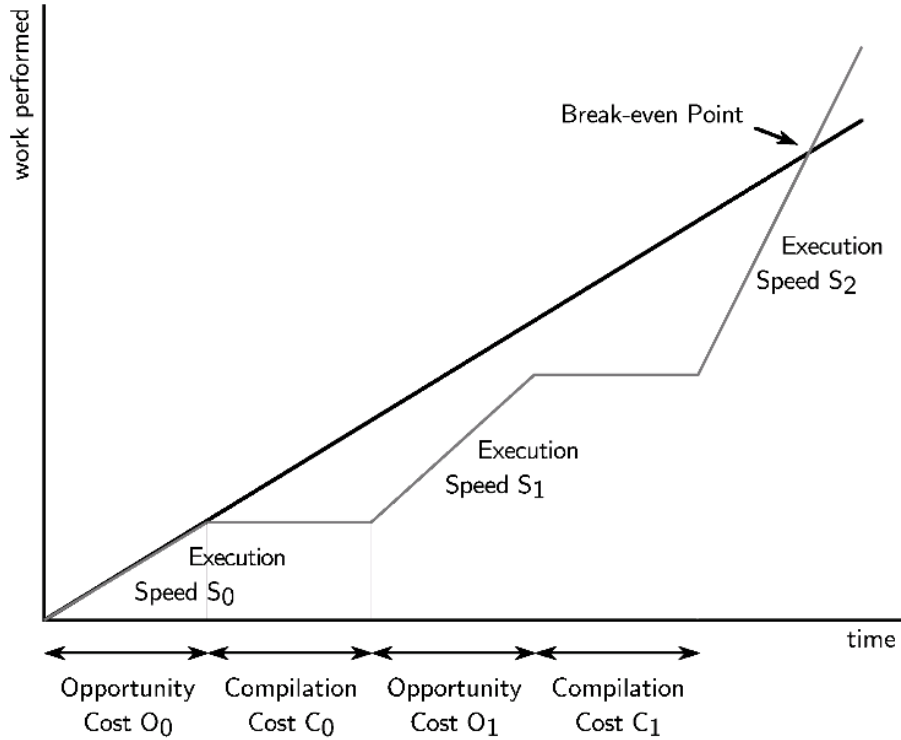


Figure 2: Comparison of a two programs, one of them runtime-optimized. The break-even point is only reached after a certain time. (Source: [Kistler03])

Results show, that the optimization potential is tremendous: Over 120% performance increase can be achieved in comparison to non-optimized programs.¹ Interestingly, even code within shared libraries used by applications with different demands can be improved in a way so that every single application performs better afterwards.

2.4 Optimizing for power consumption

With highly mobile embedded systems which run on batteries or accumulators, power consumption obviously becomes an important point. Mostly, it is seen as a problem that is to be solved with more efficient hardware. But this point of view neglects, that significant reductions in power consumption can be made with optimized

¹These results are valid for code written in the Oberon language. While the techniques presented here are certainly applicable to other languages such as C, the results may vary. For risks and side effects, consult your doctor or pharmacist.

software. Some hardware manufacturers like ARM have therefore created profiling tools which have knowledge about the amount of energy consumed by every single instruction. [Šimunić00] developed an extension for such a tool, which makes it possible to track the energy consumption not only of a whole program, but rather on function level. Applied to an embedded software for decoding mp3, [Šimunić00] were able to identify the most energy-consuming parts of it (software floating-point emulation), and reduce the power input by over two thirds in comparison with the original, unoptimized code.

2.5 Profit-driven optimization

Up to now, optimizations are mostly applied unconditionally, without consideration of their possible improvements. [Zhao06] presents a framework which makes use of a profitability engine, that is capable of estimating the benefit of a certain type of optimization at a given point. Results show, that this approach can be used to greatly effect optimization quality of most code.

3 Dynamic code generation for C

[Poletto99] introduce in their work an add-on for the C language, designed for dynamic code generation. It is called ‘C (pronounced *tick C*) and is a superset of ANSI C. One of its most important features is the tick operator, which marks a statement or a block for dynamic compilation. Fig. 3 shows a code excerpt using that operator. Additionally, there are a number of constructs to store references to dynamically generated code and variables, and to

enable the interactivity between static and dynamic code.

The semantic enhancements made by ‘C allow for optimized implementations of generic functions. As an example, take a function that copies memory from one address to another. As is widely known, such an operation can be done faster if the loop that copies the values is (partially) unrolled, so that there are less conditional jumps in the resulting machine code. Doing loop unrolling by hand is tedious and error-prone, and having the optimizer do it gives one less control over how it is done. Enter ‘C, which allows to write a function that generates at runtime a function that has exactly the amount of loop unrolling that is required for a special task. Another example would be a function which computes the binary logarithm of a 16-bit integer. Since there are only 16 different results, it is much faster to look up a pre-computed result than calculating it every time it is needed. The fastest way to accomplish this is by using a binary tree consisting of if/else-statements. Again, it is a very elegant solution to generate the code for this automatically, rather than producing such a tree by hand.

4 Conclusion

All approaches presented over the course of this work prove that code optimization – regardless of its goal – has lost nothing of its importance it had when resources like memory and cpu cycles were by far more scarce then they are today. On the contrary, today’s powerful hardware is used to automate optimization processes which were a significant part of the work a software developer had to do just a few years ago. Interestingly, the two fields where op-

```

void make_hello(void) {
    void (*f)() = compile('{ printf("hello, world\n"); }, void);
    (*f)();
}

```

Figure 3: ‘C function which generates and executes dynamic code that prints “hello, world” (Source: [Poletto99])

timization is most lucrative are embedded systems (as described in the abstract), and supercomputers. In both cases, the reason is that “every cycle counts”. As for the future, there is no reason to assume that the importance of optimization will decrease. Being able to save on hardware with each shipped unit at the expense of one-time

code analysis will most likely remain a good reason why there will be continued research on this field. While it is difficult to provide a lookout for the things to come, it can certainly be said that there are still revolutionary discoveries to be made in this relatively young scientific discipline.

References

- [Hoste08] K. Hoste et al.: *Cole: compiler optimization level exploration*, Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization, 2008
- [Debray00] S. Debray et al.: *Compiler techniques for code compaction*, ACM Transactions on Programming Languages and Systems (TOPLAS), 2000
- [Hines05] S. Hines et al.: *Using De-optimization to Re-optimize Code*, Proceedings of the 5th ACM international conference on Embedded software, 2005
- [Benitez88] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. *In Proceedings of the SIGPLAN '88 conference on Programming Language Design and Implementation*, pages 329–338. ACM Press, 1988.
- [Kistler03] T. Kistler and M. Franz: *Continuous Program Optimization: A Case Study*, ACM Transactions on Programming Languages and Systems (TOPLAS), 2003
- [Poletto99] M. Poletto et al.: *‘C and tcc: A Language and Compiler for Dynamic Code Generation*, ACM Transactions on Programming Languages and Systems (TOPLAS), 1999
- [Šimunić00] T. Šimunić et al.: *Source Code Optimization and Profiling of Energy Consumption in Embedded Systems*, International Symposium on Systems Synthesis, 2000
- [Zhao06] M. Zhao et al.: *An Approach Toward Profit-Driven Optimization*, ACM Transactions on Architecture and Code Optimization (TACO), 2006