Compiler optimizations: Where/how can I get a feel for what the payoff is for different optimizations?

Asked 16 years, 1 month ago Modified 15 years, 9 months ago

Viewed 682 times

5

In my independent study of various compiler books and web sites, I am learning about many different ways that a compiler can optimize the code that is being compiled, but I am having trouble figuring out how much of a benefit each optimization will tend to give.

How do most compiler writers go about deciding which optimizations to implement first? Or which optimizations are worth the effort or not worth the effort? I realize that this will vary between types of code and even individual programs, but I'm hoping that there is enough similarity between most programs to say, for instance, that one given technique will usually give you a better performance gain than another technique.

optimization compiler-construction

Share

Improve this question

Follow

asked Nov 4, 2008 at 0:44

Andru Luvisi

25.3k ● 6 ● 55 ● 66

8 Answers

Sorted by: Highest score (default)

13

I found when implementing textbook compiler optimizations that some of them tended to reverse the improvements made by other optimizations. This entailed a lot of work trying to find the right balance between them.

So there really isn't a good answer to your question. Everything is a tradeoff. Many optimizations work well on one type of code, but are pessimizations for other types. It's like designing a house - if you make the kitchen bigger, the pantry gets smaller.

The real work in building an optimizer is trying out the various combinations, benchmarking the results, and, like a master chef, picking the right mix of ingredients.

Share Improve this answer Follow

answered Nov 4, 2008 at 10:42

Walter Bright
4,307 ● 2 ● 25 ● 28

1 "Everything is a tradeoff" - we certainly don't see that statement enough on stackoverflow. +1 – Olof Forshell Feb 26, 2011 at 0:42

Tongue in cheek:

1. Hubris

4

2. Benchmarks
3. Embarrassment

More seriously, it depends on your compiler's architecture and goals. Here's one person's experience...

Go for the "big payoffs":

- native code generation
- register allocation
- instruction scheduling

Go for the remaining "low hanging fruit":

- strength reduction
- constant propagation
- copy propagation

Keep bennchmarking.

Look at the output; fix anything that looks stupid.

It is usually the case that combining optimizations, or even repeating optimization passes, is more effective than you might expect. The benefit is more than the sum of the parts.

You may find that introduction of one optimization may necessitate another. For example, SSA with Briggs-Chaitin register allocation really benefits from copy propagation.

Share Improve this answer Follow

answered Nov 4, 2008 at 1:02

Doug Currie
41.1k ● 1 ● 102 ● 120

3

Historically, there are "algorithmical" optimizations from which the code should benefit in most of the cases, like loop unrolling (and compiler writers should implement those "documented" and "tested" optimizations first).

Then there are types of optimizations that could benefit from the type of processor used (like using SIMD instructions on modern CPUs).

See Compiler Optimizations on Wikipedia for a reference.

Finally, various type of optimizations could be tested profiling the code or doing accurate timing of repeated executions.

Share Improve this answer Follow

answered Nov 4, 2008 at 0:54

friol
7,086 ● 4 ● 49 ● 82

I hadn't looked at the Wikipedia article, but it seems to have the same problem as most other optimization discussions. It tells me what an optimization is, and how it's done, but not how much good it would do me. – Andru Luvisi Nov 4, 2008 at 0:59

1 Yes. Profiling tells you how much good it will do you. And you won't know until you try. – mmr Nov 4, 2008 at 1:03

2

I'm not a compiler writer, but why not just incrementally optimize portions of your code, profiling all the while?

My optimization scheme usually goes:

1) make sure the program is working

2) find something to optimize

3) optimize it

4) compare the test results with what came out from 1; if they are different, then the optimization is actually a breaking change.

5) compare the timing difference

Incrementally, I'll get it faster.

I choose which portions to focus on by using a profiler. I'm not sure what extra information you'll garner by asking the compiler writers.

Share Improve this answer Follow

answered Nov 4, 2008 at 0:51

mmr
14.9k ● 29 ● 97 ● 148

2

This really depends on what you are compiling. There is was a reasonably good discussion about this on the LLVM mailing list recently, it is of course somewhat specific to the optimizers they have available. They use abbreviations for a lot of their optimization passes, if you not familiar with any of acronyms they are tossing around you can look at their passes page for documentation. Ultimately you can spend years reading academic papers on this subject.

Share Improve this answer Follow

answered Nov 4, 2008 at 1:03

Louis Gerbarg
43.4k ● 8 ● 83 ● 91

1

This is one of those topics where academic papers (ACM perhaps?) may be one of the better sources of up-to-date information. The best thing to do if you really want to know could be to create some code in unoptimized form and some in the form that the optimization would take (loops unrolled, etc) and actually figure out where the gains are likely to be using a compiler with optimizations turned off.

Share Improve this answer Follow

answered Nov 4, 2008 at 0:49

tloach
8,040 ● 1 ● 35 ● 44

1

It is worth noting that in many cases, compiler writers will NOT spend much time, if any, on ensuring that their libraries are optimized. Benchmarks tend to de-emphasize or even ignore library differences, presumably because you can just use different libraries. For example, the permutation algorithms in GCC are asymptotically* less efficient than they could be when trying to permute complex data. This relates to incorrectly making deep copies during calls to swap functions. This will likely be corrected in most compilers with the introduction of rvalue references (part of the C++0x standard). Rewriting the STL to be much faster is surprisingly easy.

*This assumes the size of the class being permuted is variable. E.g. permutting a vector of vectors of ints would slow down if the vectors of ints were larger.

0

One that can give big speedups but is rarely done is to insert memory prefetch instructions. The trick is to figure out what memory the program will be wanting far enough in advance, never ask for the wrong memory and never overflow the D-cache.