Why are compilers so stupid?

Asked 15 years, 11 months ago Modified 4 years, 4 months ago Viewed 10k times

41

I always wonder why compilers can't figure out simple things that are obvious to the human eye. They do lots of simple optimizations, but never something even a little bit complex. For example, this code takes about 6 seconds on my computer to print the value zero (using java 1.6):

```
int x = 0;
for (int i = 0; i < 100 * 1000 * 1000 * 1000; ++i) {
    x += x + x + x + x + x;
}
```

```
System.out.println(x);
```

It is totally obvious that x is never changed so no matter how often you add 0 to itself it stays zero. So the compiler could in theory replace this with System.out.println(0).

Or even better, this takes 23 seconds:

```
public int slow() {
   String s = "x";
   for (int i = 0; i < 100000; ++i) {
       s += "x";
   }
   return 10;
}
```

First the compiler could notice that I am actually creating a string s of 100000 "x" so it could automatically use s StringBuilder instead, or even better directly replace it with the resulting string as it is always the same. Second, It does not recognize that I do not actually use the string at all, so the whole loop could be discarded!

Why, after so much manpower is going into fast compilers, are they still so relatively dumb?

EDIT: Of course these are stupid examples that should never be used anywhere. But whenever I have to rewrite a beautiful and very readable code into something unreadable so that the compiler is happy and produces fast code, I wonder why compilers or some other automated tool can't do this work for me.

performance language-agnostic compiler-construction

Share

Improve this question

Follow

edited Aug 14, 2020 at 12:56

community wiki
3 revs, 2 users 74%
martinus

6 Plus `100 * 1000 * 1000 * 1000` is optimized to 100000000000 – Brad Gilbert Jan 3, 2009 at 6:05

83 Because the computers are built to obey, not to think. – Rook May 23, 2009 at 11:04

7 You're aware that the JVM performs bytecode optimisation, right? – Rob Jun 4, 2009 at 10:24

42 Compilers aren't meant to make optimizations that are "obvious to the human eye". That's what programmers are for. The compiler is meant to optimize the generated code beyond what a human can do without an unreasonable amount of effort. – Nick Lewis Jul 20, 2009 at 23:07

15 If you know how to do these kinds of compiler optimisations and be 100% confident that you won't contravene the intentions of the programmer that wrote the code, then I recommend you pursure a career as a compiler developer. If you don't know how to do this, then I think it's very unfair to "ask why are compilers (and their developers, by proxy), so stupid" – Dónal Feb 13, 2010 at 2:16

29 Answers

Sorted by: Highest score (default)

118

In my opinion, I don't believe it's the job of the compiler to fix what is, honestly, bad coding. You have, quite explicitly, told the compiler you want that first loop executed. It's the same as:

```
x = 0
sleep 6 // Let's assume this is defined somewhere.
print x
```

I wouldn't want the compiler removing my `sleep` statement just because it did nothing. You may argue that the sleep statement is an explicit request for a delay whereas your example is not. But then you will be allowing the compiler to make very high-level decisions about what your code should do, and I believe that to be a bad thing.

Code, and the compiler that processes it, are tools and you need to be a tool-smith if you want to use them effectively. How many 12" chainsaws will refuse to try cut down a 30" tree? How many drills will automatically switch to hammer mode if they detect a concrete wall?

None, I suspect, and this is because the cost of designing this into the product would be horrendous for a start. But, more importantly, you shouldn't be using drills or

chainsaws if you don't know what you're doing. For example: if you don't know what kickback is (a very easy way for a newbie to take off their arm), stay away from chainsaws until you do.

I'm all for allowing compilers to *suggest* improvements but I'd rather maintain the control myself. It should not be up to the compiler to decide unilaterally that a loop is unnecessary.

For example, I've done timing loops in embedded systems where the clock speed of the CPU is known exactly but no reliable timing device is available. In that case, you can calculate precisely how long a given loop will take and use that to control how often things happen. That wouldn't work if the compiler (or assembler in that case) decided my loop was useless and optimized it out of existence.

Having said that, let me leave you with an old story of a VAX FORTRAN compiler that was undergoing a benchmark for performance and it was found that it was *many* orders of magnitude faster than its nearest competitor.

It turns out the compiler noticed that the result of the benchmark loops weren't being used anywhere else and optimized the loops into oblivion.

edited Oct 4, 2014 at 2:21

community wiki
7 revs, 2 users 98%
paxdiablo

4 A 14" chainsaw is capable of cutting down a 22" tree, you just have to cut from both sides ;-) – dancavallaro Jan 2, 2009 at 5:38

14 -1 one for not distinguishing between standardised language definition and implementation defined run-time behaviour (== performance). Compilers are mostly allowed to emit whatever code they want to, as long as the *meaning* doesn't change. – David Schmitt Jan 2, 2009 at 9:14

2 @DavidS, having worked on a lot of embedded systems where timing loops are important, I'd prefer the compiler to suggest I can remove them for greater performance but not have that decision taken out of my hands. Ditto for "useless" memory accesses which may actually be memory-mapped I/O. – paxdiablo Jan 3, 2009 at 0:29

2 @DavidS (cont), but these are situations where *meaning* includes the speed of execution. That stuff is probably best done by using a lower optimization level, or declaring variables as volatile. – paxdiablo Jan 3, 2009 at 0:31

11 If you don't want optimisation, don't use optimisation flags. You clearly don't want optimisation if you are relying on loops that do nothing but are there to take up time. The optimisation compiler is *meant* to speed things up while keeping the functional aspects of the code the same. This is also what 99% of people want. For some reason you want compilers to be designed for your 1% use-case. – mxcl Oct 10, 2009 at 22:29

112

Oh, I don't know. Sometimes compilers are pretty smart. Consider the following C program:

```
#include <stdio.h>  /* printf() */

int factorial(int n) {
   return n == 0 ? 1 : n * factorial(n - 1);
}

int main() {
   int n = 10;

   printf("factorial(%d) = %d\n", n, factorial(n));

   return 0;
}
```

On my version of GCC (4.3.2 on Debian testing), when compiled with no optimizations, or `-O1`, it generates code for `factorial()` like you'd expect, using a recursive call to compute the value. But on `-O2`, it does something interesting: It compiles down to a tight loop:

```
    factorial:
.LFB13:
        testl   %edi, %edi
        movl    $1, %eax
        je  .L3
        .p2align 4,,10
        .p2align 3
.L4:
        imull   %edi, %eax
        subl    $1, %edi
        jne .L4
.L3:
        rep
        ret
```

Pretty impressive. The recursive call (not even tail-recursive) has been completely eliminated, so factorial now uses O(1) stack space instead of O(N). And although I have only very superficial knowledge of x86 assembly (actually AMD64 in this case, but I don't think any of the AMD64 extensions are being used above), I doubt that you could write a better version by hand. But what really blew my mind was the code that it generated on `-O3`. The implementation of factorial stayed the same. But `main()` changed:

```
    main:
.LFB14:
        subq    $8, %rsp
.LCFI0:
        movl    $3628800, %edx
        movl    $10, %esi
```

```
movl    $.LC0, %edi
xorl    %eax, %eax
call    printf
xorl    %eax, %eax
addq    $8, %rsp
ret
```

See the `movl $3628800, %edx` line? **gcc is pre-computing `factorial(10)` at compile-time.** It doesn't even call `factorial()`. Incredible. My hat is off to the GCC development team.

Of course, all the usual disclaimers apply, this is just a toy example, premature optimization is the root of all evil, etc, etc, but it illustrates that compilers are often smarter than you think. If you think you can do a better job by hand, you're almost certainly wrong.

(Adapted from a posting on my blog.)

Share

Improve this answer

Follow

edited Apr 21, 2019 at 17:53

community wiki
4 revs, 4 users 65%
Jason Creighton

7 That's kick-ass! Do you know if gcc does more pre-computation on function calls with constant values? What does gcc do when choosing an n which produces a factorial which is greater than a 64 bit value? – Scoregraphic May 23, 2009 at 11:39

3 I have no idea. It's all black magic to me. :-) Experimentation reveals that larger values will make my version of gcc not attempt pre-computation. If you're interested in playing around with this kind of thing, just compile a program with the -S flag. eg, if you do "gcc -O3 -S factorial.c", it will produce a file, factorial.s, which contains the generated assembly. There's probably some other flags to dump whatever gcc's intermediate representation is, although that's more specialized than straight assembly. – Jason C May 24, 2009 at 5:12

"movl $3628800" suggests me that gcc pre-calculates at least some functions at compile-time. --- When does gcc stop to pre-compute functions at compile-time? --- It is likely that gcc somehow estimates the size of the function's output, and this way, decides whether to pre-compute or not at compile-time. – Léo Léopold Hertz 준영 Jun 26, 2009 at 10:06

3 @Masi: you need to use the -S flag. so, eg, "gcc -O2 -S code.c" will produce a file named "code.s" containing the assembly. – Jason C Jun 26, 2009 at 14:33

2 its a simple static optimization(because the variable you do pass don't change), no big deal. Compilers do only cover the cases they are designed for, nothing else, they will never think/abstract/understand the algorithm – Quonux Jul 15, 2010 at 22:12

47

Speaking from a C/C++ point of view:

Your first example will be optimized by most compilers. If the java-compiler from Sun really executes this loop it's the compilers fault, but take my word that any post 1990

C, C++ or Fortran-compiler completely eliminates such a loop.

Your second example can't be optimized in most languages because memory allocation happens as a side-effect of concatenating the strings together. If a compiler would optimize the code the pattern of memory allocation would change, and this could lead to effects that the programmer tries to avoid. Memory fragmentation and related problems are issues that embedded programmers still face every day.

Overall I'm satisfied with the optimizations compilers can do these days.

Share

Improve this answer

Follow

edited Jan 2, 2009 at 1:56

community wiki
2 revs, 2 users 94%
Nils Pipenbrinck

4 Memory allocation/management happens to be one of those things Java programmers take for granted. – isekaijin Jan 2, 2009 at 3:32

I think most C++ compilers will eliminate the addition, but not the loop, nor the assignment. Assigning to the stack causes a side effect. Eliminating the side effect could potentially change the meaning of the program, and would therefore not be correct. – Scott Wisniewski Jan 2, 2009 at 10:08

2 Scott, I tried it with the latest GCC. It removes the entire calculation including the loop. All that remains is a printf call and the surrounding code that pushes a zero onto the stack. – Nils Pipenbrinck Jan 2, 2009 at 12:04

Ok. I had tried both GCC and VC++ with -O3 and /Ox, and they left the loop there. I don't think I was running the latest version of GCC though. I would think that you could make an argument against that being a safe optimization in C++. For Java, or Haskell, it would be safe, but not for C. – Scott Wisniewski Jan 2, 2009 at 21:11

3 Scott, You're right. I just checked VS2008 and it does not do the optimization. My GCC (4.3.1) however does the trick. The C-standard does btw not mention the stack at all, so it's valid to do any optimizations that change stack-allocation patterns. – Nils Pipenbrinck Jan 2, 2009 at 23:46

23

Compilers are designed to be **predictable**. This may make them look stupid from time to time, but that's OK. The compiler writer's goals are

- You should be able to look at your code and make reasonable predictions about its performance.
- Small changes in the code should not result in dramatic differences in performance.
- If a small change looks to the programmer like it should improve performance, it should at least not degrade performance (unless surprising things are happening in the hardware).

All these criteria militate against "magic" optimizations that apply only to corner cases.

Both of your examples have a **variable updated in a loop but not used elsewhere**. This case is actually quite difficult to pick up unless you are using some sort of framework that can combine dead-code elimination with other optimizations like copy propagation or constant propagation. To a *simple* dataflow optimizer the variable doesn't look dead. To understand why this problem is hard, see the paper by Lerner, Grove, and Chambers in POPL 2002, which uses this very example and explains why it is hard.

Share
Improve this answer
Follow

edited May 25, 2010 at 5:34

community wiki
2 revs
Norman Ramsey

1 Did you just make up the stuff about being predictable? – LtWorf Nov 23, 2015 at 10:19

16

The HotSpot JIT compiler will only optimize code that has been running for some time. By the time your code is hot, the loop has already been started and the JIT compiler has to wait until the next time the method is entered to look for ways to optimize away the loop. If you call the method several times, you might see better performance.

This is covered in the HotSpot FAQ, under the question "I write a simple loop to time a simple operation and it's slow. What am I doing wrong?".

Share
Improve this answer
Follow

edited Jan 2, 2009 at 1:40

community wiki
2 revs
Rune

1 +1 for reference to the Java HotSpot Technology. Those kind of optimizations are not the compilers job, but rather of the (virtual) machine, since only that knows the context and can verify the outcome. – Ilmbus Jan 3, 2009 at 19:15

I think they have on the fly stack replacement now, so you don't have to wait to the second call to get the JIT compile code running... – fortran Dec 17, 2009 at 12:48

15

Seriously? Why would anyone ever write real-world code like that? IMHO, the code, not the compiler is the "stupid" entity here. I for one am perfectly happy that compiler writers don't bother wasting their time trying to optimize something like that.

Edit/Clarification: I know the code in the question is meant as an example, but that just proves my point: you either have to be trying, or be fairly clueless to write

supremely inefficient code like that. It's not the compiler's job to hold our hand so we don't write horrible code. It is our responsibility as the people that write the code to know enough about our tools to write efficiently and clearly.

Share

Improve this answer

Follow

edited Jan 2, 2009 at 1:10

community wiki
2 revs
Daniel Schaffer

I want to write the readable code, and not code that makes the compiler happy. These are just stupid simple examples to show that the compiler is dumb, nothing I would use in a real program. – martinus Jan 2, 2009 at 1:07

"but that just proves my point": How does it prove your point? The question is a general one about why there are still things that are obvious to the human eye and not to the compiler, and you pick fault with the (obviously exaggerated for effect) examples instead. – ShreevatsaR Jan 2, 2009 at 1:30

The point is that if the examples weren't exaggerated for effect, they wouldn't exemplify the point of the question, which I went on to explain - you either write horrible code on purpose because you're trying to, or you do it by accident because you make a mistake or don't know what you're doing. – Daniel Schaffer Jan 2, 2009 at 1:34

1 Not that I agree, but he is saying that it proves his point because these are contrived examples and not snippets of "real" code. If someone could provide real, production code that was similarly (mis-)treated by the compiler, that would be a counterexample that undermines his argument. – Adam Bellaire Jan 2, 2009 at 1:34

^^ Neither of those is any fault of the compiler, and even if it could, I wouldn't want my compiler "fixing" things like that. – Daniel Schaffer Jan 2, 2009 at 1:35

13

Well, I can only speak of C++, because I'm a Java beginner totally. In C++, compilers are free to disregard any language requirements placed by the Standard, as long as the **observable** behavior is **as-if** the compiler actually emulated all the rules that are placed by the Standard. Observable behavior is defined as **any reads and writes to volatile data and calls to library functions**. Consider this:

```
extern int x; // defined elsewhere
for (int i = 0; i < 100 * 1000 * 1000 * 1000; ++i) {
    x += x + x + x + x + x;
}
return x;
```

The C++ compiler is allowed to optimize out that piece of code and just add the proper value to x that would result from that loop once, because the code behaves **as-if** the loop never happened, and no volatile data, nor library functions are involved that could cause side effects needed. Now consider volatile variables:

```
extern volatile int x; // defined elsewhere
for (int i = 0; i < 100 * 1000 * 1000 * 1000; ++i) {
    x += x + x + x + x + x;
}
return x;
```

The compiler is *not* allowed to do the same optimization anymore, because it can't prove that side effects caused by writing to `x` could not affect the observable behavior of the program. After all, x could be set to a memory cell watched by some hardware device that would trigger at every write.

Speaking of Java, I have tested your loop, and it happens that the GNU Java Compiler (`gcj`) takes in inordinate amount of time to finish your loop (it simply didn't finish and I killed it). I enabled optimization flags (-O2) and it happened it printed out `0` immediately:

```
[js@HOST2 java]$ gcj --main=Optimize -O2 Optimize.java
[js@HOST2 java]$ ./a.out
0
[js@HOST2 java]$
```

Maybe that observation could be helpful in this thread? Why does it happen to be so fast for gcj? Well, one reason surely is that gcj compiles into machine code, and so it has no possibility to optimize that code based on runtime behavior of the code. It takes all its strongness together and tries to optimize as much as it can at compile time. A virtual machine, however, can compile code Just in Time, as this output of java shows for this code:

```
class Optimize {
    private static int doIt() {
        int x = 0;
        for (int i = 0; i < 100 * 1000 * 1000 * 1000; ++i) {
            x += x + x + x + x + x;
        }
        return x;
    }
    public static void main(String[] args) {
        for(int i=0;i<5;i++) {
            doIt();
        }
    }
}
```

Output for `java -XX:+PrintCompilation Optimize`:

```
1       java.lang.String::hashCode (60 bytes)
1%      Optimize::doIt @ 4 (30 bytes)
```

```
2       Optimize::doIt (30 bytes)
```

As we see, it JIT compiles the doIt function 2 times. Based on the observation of the first execution, it compiles it a second time. But it happens to have the same size as bytecode two times, suggesting the loop is still in place.

As another programmer shows, execution time for certain dead loops even is increased for some cases for subsequently compiled code. He reported a bug which can be read here, and is as of 24. October 2008.

Share

Improve this answer

Follow

edited Nov 10, 2023 at 19:48

community wiki
5 revs, 2 users 96%
Johannes Schaub - litb

If you use GCJ you use the same code-generation backend as GCC, so you get more or less the same optimizations. You can also use GCJ to compile to .class files btw. No need to compile to native code. – Nils Pipenbrinck Jan 2, 2009 at 12:36

1 Nils, yeah i'm aware of that. i took the compile-to-machine code explicitly, to see how it behaves in that case (to see whether java possibly *forbids* those optimizations. apparently, it doesn't). – Johannes Schaub - litb Jan 2, 2009 at 17:46

Johannes - Genius as always ! – Saran-san Jan 27, 2014 at 3:10

9

On your first example, it's an optimization that only works if the value is zero. The extra `if` statement in the compiler needed to look for this one rarely-seen clause may just not be worth it (since it'll have to check for this on every single variable). Furthermore, what about this:

```
int x = 1;
int y = 1;
int z = x - y;
for (int i = 0; i < 100 * 1000 * 1000 * 1000; ++i) {
    z += z + z + z + z + z;
}
System.out.println(z);
```

This is still obviously the same thing, but now there's an extra case we have to code for in the compiler. There's just an infinite amount of ways that it can end up being zero that aren't worth coding in for, and I guess you could say that if you're going to have one of them you'd might as well have them all.

Some optimizations do take care of the second example you have posted, but I think I've seen it more in functional languages and not so much Java. The big thing that makes it hard in newer languages is monkey-patching. Now `+=` can have a side-

effect that means if we optimize it out, it's potentially wrong (e.g. adding functionality to `+=` that prints out the current value will mean a different program altogether).

But it comes down to the same thing all over again: there's just too many cases you'd have to look for to make sure no side effects are being performed that will potentially alter the final program's state.

It's just easier to take an extra moment and make sure what you're writing is what you really want the computer to do. :)

Share

Improve this answer

Follow

edited Oct 21, 2018 at 9:29

community wiki
3 revs, 2 users 95%
Chris Bunch

12 Compilers work in several passes, and the constant folding + constant propagation pass(es) will reduce your example to the one posted by the OP, so the code required to perform this optimization is the same for both cases. – Rune Jan 2, 2009 at 1:37

7

Compilers in general are very smart.

What you must consider is that they must account for every possibly exception or situation where optimizing or re-factoring code could cause unwanted side-effects.

Things like, threaded programs, pointer aliasing, dynamically linked code and side effects (system calls/memory alloc) etc. make formally prooving refactoring very difficult.

Even though your example is simple, there still may be difficult situations to consider.

As for your StringBuilder argument, that is NOT a compilers job to choose which data structures to use for you.

If you want more powerful optimisations move to a more strongly typed language like fortran or haskell, where the compilers are given much more information to work with.

Most courses teaching compilers/optimisation (even acedemically) give a sense of appreciation about how making gerneral formally prooven optimisatons rather than hacking specific cases is a very difficult problem.

Share

Improve this answer

Follow

answered Jan 2, 2009 at 1:27

community wiki
Akusete

6

I think you are underestimating how much work it is to make sure that one piece of code doesn't affect another piece of code. With just a small change to your examples x, i, and s could all point to the same memory. Once one of the variables is a pointer, it is much harder to tell what code might have side effects depending on is point to what.

Also, I think people who program compliers would rather spend time making optimizations that aren't as easy for humans to do.

Share

Improve this answer

Follow

answered Jan 2, 2009 at 1:13

community wiki
Chad DeShon

> Upvoted for the second part. It is much more fulfilling to make well-written code go faster by hitting some machine idiom than to optimize away badly written code. – Rune Jan 2, 2009 at 2:05

4

Because we're just not there yet. You could just as easily have asked, "why do I still need to write programs... why can't I just feed in the requirements document and have the computer write the application for me?"

Compiler writers spend time on the little things, because those are the types of things that application programmers tend to miss.

Also, they cannot assume too much (maybe your loop was some kind of ghetto time delay or something)?

Share

Improve this answer

Follow

edited Jan 2, 2009 at 1:19

community wiki
2 revs
Giovanni Galbo

2

It's an eternal arms race between compiler writers and programmers.

Non-contrived examples work great -- most compilers do indeed optimize away the obviously useless code.

Contrived examines will *always* stump the compiler. Proof, if any was needed, that any programmer is smarter than any program.

In the future, you'll need more contrived examples than the one's you've posted here.

Share

answered Jan 2, 2009 at 1:21

community wiki

Improve this answer S.Lott

Follow

2

As others have addressed the first part of your question adequately, I'll try to tackle the second part, i.e. "automatically uses StringBuilder instead".

There are several good reasons for not doing what you're suggesting, but the biggest factor in practice is likely that the optimizer runs long after the actual source code has been digested & forgotten about. Optimizers generally operate either on the generated byte code (or assembly, three address code, machine code, etc.), or on the abstract syntax trees that result from parsing the code. Optimizers generally know nothing of the runtime libraries (or any libraries at all), and instead operate at the instruction level (that is, low level control flow and register allocation).

Second, as libraries evolve (esp. in Java) much faster than languages, keeping up with them and knowing what deprecates what and what other library component might be better suited to the task would be a herculean task. Also likely an impossible one, as this proposed optimizer would have to precisely understand both your intent and the intent of each available library component, and somehow find a mapping between them.

Finally, as others have said (I think), the compiler/optimizer writer can reasonably assume that the programmer writing the input code is not brain-dead. It would be a waste of time to devote significant effort to asinine special cases like these when other, more general optimizations abound. Also, as others have also mentioned, seemingly brain-dead code can have an actual purpose (a spin lock, busy wait prior to a system-level yield, etc.), and the compiler has to honor what the programmer asks for (if it's syntactically and semantically valid).

Share answered Jan 2, 2009 at 3:18 community wiki Drew Hall

Improve this answer

Follow

gcc has built-in knowledge of the most common C functions, so that optimizations can be performed when they are used.It is true in general but normally for common types and functions (good) compilers have library knowledge. – LtWorf Nov 23, 2015 at 10:13

Did you compile to release code? I think a good compiler detects in your second example that the string is never used an removes the entire loop.

1

Share

Improve this answer

Follow

answered Jan 2, 2009 at 1:17

community wiki
Rauhotz

In C# release mode it isnt optimized out. In C++ it is (code in my post) – user34537 Oct 1, 2010 at 7:06

javac does not optimize – LtWorf Nov 23, 2015 at 10:03

Actually, Java should use string builder in your second example.

1

The basic problem with trying to optimize these examples away is that doing so would require *theorem proving*. Which means that the compiler would need to construct a mathematical proof of what you're code will actually do. And that's no small task at all. In fact, being able to prove that **all** code really does have an effect is equivalent to the halting problem.

Sure, you can come up with trivial examples, but the number of trivial examples is unlimited. You could always think of something else, so there is no way to catch them all.

Of course, it is possible for **some** code to be proven not to have any effect, as in your examples. What you would want to do is have the compiler optimize away every problem that can be proven unused in P time.

But anyway, that's a ton of work and it doesn't get you all that much. People spend a lot of time trying to figure out ways to prevent programs from having bugs in them, and type systems like those in Java and Scala are attempts to prevent bugs, but right now no one is using type systems to make statements about execution time, as far as I know.

You might want to look into Haskel, which I think has the most advanced theory proving stuff, although I'm not sure on that. I don't know it myself.

Share

Improve this answer

Follow

answered Jan 2, 2009 at 2:41

community wiki
Chad Okere

javac does no optimizations. Try to compile "1+1+1+1" and show the generated JVM assembly instructions – LtWorf Nov 23, 2015 at 10:01

1

Mostly what you're complaining about is 'why are Java compiler so stupid', since most other language compilers are much smarter.

The reason for the stupidity of Java compilers is historical. First, the original java implementations were interpreter based, and performance was consisdered unimportant. Second, many of the original java benchmarks were problematic to optimize. I recall one benchmark that looked a lot like your second example. Unfortunately, if the compiler optimized the loop away, the benchmark would get a divide by zero exception when it tried to divide a baseline number by the elapsed time to compute its performance score. So when writing a optimizing java compiler, you had to be very careful NOT to optimize some things, as people would then claim your compiler was broken.

Share

Improve this answer

Follow

answered Jan 2, 2009 at 3:52

community wiki
Chris Dodd

1

It's almost considered bad practice to optimize things like this when compiling down to JVM bytecode. Sun's `javac` does have some basic optimizations, as does `scalac`, `groovyc`, etc. In short, anything that's truely language-specific can get optimized within the compiler. However, things like this which are obviously so contrived as to be language agnostic will slip through simply out of policy.

The reason for this is it allows HotSpot to have a much more consistent view of the bytecode and its patterns. If the compilers start mucking about with edge cases, that reduces the VM's ability to optimize the general case which may not be apparent at compile time. Steve Yeggie likes to harp on about this: optimization is often *easier* when performed at runtime by a clever virtual machine. He even goes so far as to claim that HotSpot strips out javac's optimizations. While I don't know if this is true, it wouldn't surprise me.

To summarize: compilers targeting VMs have a very different set of criteria, particularly in the area of optimization and when it is appropriate. Don't go blaming the compiler writers for leaving the work to the far-more-capable JVM. As pointed out several times on this thread, modern compilers targeting the native architecture (like the `gcc` family) are *extremely* clever, producing obscenely fast code through some very smart optimizations.

Share

Improve this answer

Follow

answered Jan 2, 2009 at 4:58

community wiki
Daniel Spiewak

> Sun's javac does have some basic optimizations Does it? Can you name one? I have some experience with it and I never found it optimizing anything at all. Not even constant expressions (which is an extremely trivial optimization) – LtWorf Nov 23, 2015 at 10:10

1

I have never seen the point in dead code elimination in the first place. Why did the programmer write it?? If you're going to do something about dead code, declare it a compiler error! It almost certainly means the programmer made a mistake--and for the few cases it doesn't, a compiler directive to use a variable would be the proper answer. If I put dead code in a routine I want it executed--I'm probably planning to inspect the results in the debugger.

The case where the compiler could do some good is pulling out loop invariants. Sometimes clarity says to code the calculation in the loop and having the compiler pull such things out would be good.

Share

Improve this answer

Follow

answered Jan 2, 2009 at 5:59

community wiki
Loren Pechtel

2 Compilers can generate dead code all by themselves. It's useful to be able to pull it out. – David Thornley Jan 13, 2009 at 21:08

If the code is small enough it can fit in the CPU cache and be executed quickly. The larger it is, the more misses it generates. If it's really large it will even generate page misses. – LtWorf Nov 23, 2015 at 10:09

1

Compilers that can do strict-aliasing optimizations, will optimize first example out. See here.

Second example can't be optimized because the slowest part here is memory allocation/reallocation and operator+= is redefined into a function that does the memory stuff. Different implementations of strings use different allocation strategies.

I myself also would rather like to have malloc(100000) than thousand malloc(100) too when doing s += "s"; but right now that thing is out of scope of compilers and has to be optimized by people. This is what D language tries to solve by introducing pure functions.

As mentioned here in other answers, perl does second example in less than a second because it allocates more memory than requested just in case more memory will be needed later.

Share

Improve this answer

Follow

answered Jun 4, 2009 at 10:20

community wiki
HMage

1

In release mode VS 2010 C++ this doesnt take any time to run. However debug mode is another story.

```
#include <stdio.h>
int main()
{
    int x = 0;
    for (int i = 0; i < 100 * 1000 * 1000 * 1000; ++i) {
        x += x + x + x + x + x;
    }
    printf("%d", x);
}
```

Share

Improve this answer

Follow

answered Oct 1, 2010 at 4:33

community wiki
user34537

1

Absolute optimization is an undecidable problem, that means, there is no Turing machine (and, therefore, no computer program) that can yield the optimal version of ANY given program.

Some simple optimizations can be (and, in fact, are) done, but, in the examples you gave...

1. To detect that your first program always prints zero, the compiler would have to detect that x remains constant despite all the loop iterations. How can you *explain* (I know, it's not the best word, but I can't come up with another) that to a compiler?
2. How can the compiler know that the StringBuilder is the right tool for the job without ANY reference to it?

~~In a real-world application, if efficiency is critical in a part of your application, it must be written in a low-level language like C.~~ (Haha, seriously, I wrote this?)

Share

Improve this answer

edited Nov 23, 2015 at 13:53

community wiki

Follow

3 revs
Eduardo León

1 is really really easy to spot for a compiler. Good compilers will fix it. For the 2nd, yes the compiler needs to have some knowledge of the classes, but "String" is a basic type and I think it could be acceptable to have it. – LtWorf Nov 23, 2015 at 10:07

@LtWorf: For simple cases. In general, you are running against [en.wikipedia.org/wiki/Rice's_theorem](Rice's theorem). – isekaijin Nov 23, 2015 at 13:54

Check gcc, it's done there. I am aware that it can't be done for every function but it does make sense to do it for the common ones. – LtWorf Nov 23, 2015 at 14:31

1

A compiler's job is to optimize ***how*** *the code does something*, not ***what*** *the code does*.

When you write a program, you are telling the computer what to do. If a compiler changed your code to do something other than what you told it to, it wouldn't be a very good compiler! When you write `x += x + x + x + x + x`, you are explicitly telling the computer that you want it to set `x` to 6 times itself. The compiler may very well optimize *how* it does this (e.g. multiplying `x` by 6 instead of doing repeated addition), but regardless it will still calculate that value in some way.

> If you don't want something to be done, don't tell someone to do it.

Share

Improve this answer

Follow

answered Mar 17, 2020 at 15:41

community wiki
AlpaccaSnack

0

This is an example of procedural code v. functional code.

You have detailed a procedure for the compiler to follow, so the optimisations are going to be based around the procedure detailed and will minimise any side effects or not optimise where it will not be doing what you expect. This makes it easier to debug.

If you put in a functional description of what you want eg. SQL then you are giving the compiler a wide range of options to optimise.

Perhaps some type of code analysis would be able to find this type of issue or profiling at run-time, but then you will want to change the source to something more sensible.

Share

answered Jan 2, 2009 at 8:29

community wiki
pro

Improve this answer

Follow

0

Because compiler writers try add optimizations for things that matter (I hope) and that are measured in *Stone benchmarks (I fear).

There are zillions of other possible code fragments like yours, which do nothing and could be optimized with increasing effort on the compiler writer, but which are hardly ever encountered.

What I feel embarrassing is that even today most compilers generate code to check for the switchValue being greater than 255 for a dense or almost full switch on an unsigned character. That adds 2 instructions to most bytecode interpreter's inner loop.

Share

Improve this answer

Follow

edited Dec 17, 2009 at 11:54

community wiki
2 revs, 2 users 83%
blabla999

No, it's because javac doesn't optimize things. Gcc will detect and optimize it. – LtWorf Nov 23, 2015 at 10:05

0

I hate to bring this up on such an old question (how did I get here, anyway?), but I think part of this might be something of a holdout from the days of the Commodore 64.

In the early 1980s, everything ran on a fixed clock. There was no Turbo Boosting and code was always created for a specific system with a specific processor and specific memory, etc. In Commodore BASIC, the standard method for implementing `delay`s looked a lot like:

```
10 FOR X = 1 TO 1000
20 NEXT : REM 1-SECOND DELAY
```

(Actually, in practice, it more closely resembled `10FORX=1TO1000:NEXT`, but you know what I mean.)

If they were to optimize this, it would break everything—nothing would ever be timed. I don't know of any examples, but I'm sure there are lots of little things like this scattered through the history of compiled languages that prevented things from being optimized.

Admittedly, these non-optimizations *aren't* necessary today. There's probably, however, some unspoken rule among compiler developers not to optimize things like this. I wouldn't know.

Just be glad that your code is optimized somewhat, *unlike* code on the C64. Displaying a bitmap on the C64 could take up to 60 seconds with the most efficient BASIC loops; thus, most games, etc. were written in machine language. Writing games in machine language isn't fun.

Just my thoughts.

Share

Improve this answer

Follow

answered Nov 6, 2012 at 3:18

community wiki
Tortoise

> That is correct, but there are no function calls inside the 1st loop, so it is really obvious to see what the code does. – LtWorf Nov 23, 2015 at 10:04

0

Premise: I studied compilers at university.

The javac compiler is extremely stupid and performs absolutely no optimization because it relies on the java runtime to do them. The runtime will catch that thing and optimize it, but it will catch it only after the function is executed a few thousand times.

If you use a better compiler (like gcc) enabling optimizations, it will optimize your code, because it's quite an obvious optimization to do.

Share

Improve this answer

Follow

answered Nov 23, 2015 at 9:58

community wiki
LtWorf

-1

Compilers are as smart as we make them. I don't know too many programmers who would bother writing a compiler that would check for constructs such as the ones you used. Most concentrate on more typical ways to improve performance.

It is possible that someday we will have software, including compilers, that can actually learn and grow. When that day comes most, maybe all, programmers will be out of job.

Share

Improve this answer

Follow

answered Jan 13, 2009 at 20:24

community wiki
Jim C

i hope this will never happen, or it should happen if im no more alive :P – Quonux Jul 24, 2010 at 21:42

javac compiler does no optimization at all. It doesn't even replace constants so if you write "1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1" it generates code for adding "1" several times rather than just putting in the resulting constant. – LtWorf Nov 23, 2015 at 10:00

-1

The meaning of your two examples is pointless, useless and only made to fool the compiler.

The compiler is not capable (and should not be) to see the meaning of a method, a loop or a program. That is where you get into the picture. You create a method for a certain functionality/meaning, no matter how stupid it is. It's the same case for simple problems or extreme complex programs.

In your case the compiler might optimize it, because it "thinks" it should be optimized in another way but why stay there?

Extreme other situation. We have a smart compiler compiling Windows. Tons of code to compile. But if it's smart, it boils it down to 3 lines of code...

```
"starting windows"
"enjoy freecell/solitaire"
"shutting down windows"
```

The rest of the code is obsolete, because it's never used, touched, accessed. Do we really want that?

Share

Improve this answer

Follow

edited Dec 17, 2009 at 11:58

community wiki
2 revs, 2 users 76%
Mafti

1 Are you sure that gcc's compiler is not capable see the meaning of a method, a loop or a program? -- Jason's example suggests me that some pre-calculations is occurring at compile-time. – Léo Léopold Hertz 준영 Jun 26, 2009 at 10:21

Too bad we'll never see the Windows source code to test that out :). – S.S. Anne Apr 21, 2019 at 18:02

-2

It forces you (the programmer) to think about what you're writing. Forcing compilers to do your work for you doesn't help anyone: it makes the compilers much more complex (and slower!), and it makes you stupider and less attentive to your code.

Share

Improve this answer

Follow

answered Jan 2, 2009 at 1:09

community wiki
Sophie Alpert

2 Im not sure this is a good way to look at it, this way we might still be stuck with writing assembler code. – martinus Jan 2, 2009 at 1:22

Compilers having optimizations helps a lot, For example you can just write stuff in a loop and the compiler can use the SSE instructions if they are available, or can just compile the loop normally otherwise. To do this manually you'd need to write platform specific code for every CPU version. – LtWorf Sep 29, 2016 at 17:11