Why would it ever be possible for Java to be faster than C++?

Sometimes Java outperforms C++ in benchmarks. Of course, sometimes C++ outperforms.

See the following links:

- http://keithlea.com/javabench/
- http://blog.dhananjaynene.com/2008/07/performance-comparison-c-java-python-ruby-jython-jruby-groovy/
- http://blog.cfelde.com/2010/06/c-vs-java-performance/

But how is this even possible? It boggles my mind that interpreted bytecode could ever be faster than a compiled language.

Can someone please explain? Thanks!

java c++ performance

edited Sep 26 '11 at 7:03 Joachim Sauer 10.3k 3 46 44

asked Sep 26 '11 at 4:47 Deets McGeets 823 3 12 15

2 You can take a look at shootout.alioth.debian.org/u32/… to see the kind of problems that run faster on java/c++... See the pattern of problems, and not these specific problems... – c0da Sep 26 '11 at 5:48

2 See Why did java have the reputation of being slow? for a lot of details on this topic. – Péter Török Sep 26 '11 at 7:27

9 It is **against the law** (Section 10.101.04.2c) to produce a Java VM that performs faster than an executable binary produced with C++. – Mateen Ulhaq Sep 26 '11 at 23:48

1 @muntoo What on earth do you mean? Section 10.101.04.2c of what? – Highland Mark Sep 29 '11 at 18:51

2 @HighlandMark You are not a member of the circle. You are not to know of what goes inside the circle. The circle is absolute - the circle's laws superseed that of nature's. You cannot defy nor question the circle. **I am the circle and the circle is me.** – Mateen Ulhaq Sep 29 '11 at 23:56

15 Answers

First, most JVMs include a compiler, so "interpreted bytecode" is actually pretty rare (at least in benchmark code -- it's not quite as rare in real life, where your code is usually more than a few trivial loops that get repeated extremely often).

Second, a fair number of the benchmarks involved appear to be quite biased (whether by intent or incompetence, I can't really say). Just for example, years ago I looked at some of the source code linked from one of the links you posted. It had code like this:

```
init0 = (int*)calloc(max_x,sizeof(int));
init1 = (int*)calloc(max_x,sizeof(int));
init2 = (int*)calloc(max_x,sizeof(int));
for (x=0; x<max_x; x++) {
  init2[x] = 0;
  init1[x] = 0;
  init0[x] = 0;
}
```

Since `calloc` provides memory that's already zeroed, using the `for` loop to zero it again is obviously useless. This was followed (if memory serves) by filling the memory with other data anyway (and no dependence on it being zeroed), so all the zeroing was completely unnecessary anyway. Replacing the code above with a simple `malloc` (like any sane person would have used to start with) improved the speed of the C++ version enough to beat the Java version (by a fairly wide margin, if memory serves).

Consider (for another example) the `methcall` benchmark used in the blog entry in your last link. Despite the name (and how things might even look), the C++ version of this is *not* really measuring much about method call overhead at all. The part of the code that turns out to be critical is in the Toggle class:

```
class Toggle {
public:
    Toggle(bool start_state) : state(start_state) { }
    virtual ~Toggle() {  }
    bool value() {
        return(state);
```

```
    }
    virtual Toggle& activate() {
        state = !state;
        return(*this);
    }
    bool state;
};
```

The critical part turns out to be the `state = !state;`. Consider what happens when we change the code to encode the state as an `int` instead of a `bool`:

```
class Toggle {
    enum names{ bfalse = -1, btrue = 1};
    const static names values[2];
    int state;

public:
    Toggle(bool start_state) : state(values[start_state])
    { }
    virtual ~Toggle() {  }
    bool value() {  return state==btrue;    }

    virtual Toggle& activate() {
        state = -state;
        return(*this);
    }
};
```

This minor change improves the overall speed by about a **5:1 margin**. Even though the benchmark was *intended* to measure method call time, in reality most of what it was measuring was the time to convert between `int` and `bool`. I'd certainly agree that the inefficiency shown by the original is unfortunate -- but given how rarely it seems to arise in real code, and the ease with which it can be fixed when/if it does arise, I have a difficult time thinking of it as meaning much.

In case anybody decides to re-run the benchmarks involved, I should also add that there's an almost equally trivial modification to the Java version that produces (or at least at one time produced -- I haven't re-run the tests with a recent JVM to confirm they still do) a fairly substantial improvement in the Java version as well. The Java version has an NthToggle::activate() that looks like this:

```
public Toggle activate() {
this.counter += 1;
if (this.counter >= this.count_max) {
    this.state = !this.state;
    this.counter = 0;
}
return(this);
}
```

Changing this to call the base function instead of manipulating `this.state` directly gives quite a substantial speed improvement (though not enough to keep up with the modified C++ version).

So, what we end up with is a false assumption about interpreted byte codes vs. some of the worst benchmarks (I've) ever seen. Neither is giving a meaningful result.

My own experience is that with equally experienced programmers paying equal attention to optimizing, C++ will beat Java more often than not -- but (at least between these two), the language will rarely make as much difference as the programmers and design. The benchmarks being cited tell us more about the (in)competence/(dis)honesty of their authors than they do about the languages they purport to benchmark.

[Edit: As implied in one place above but never stated as directly as I probably should have, the results I'm quoting are those I got when I tested this ~5 years ago, using C++ and Java implementations that were current at that time. I haven't rerun the tests with current implementations. A glance, however, indicates that the code hasn't been fixed, so all that would have changed would be the compiler's ability to cover up the problems in the code.]

If we ignore the Java examples, however, it *is* actually possible for interpreted code to run faster than compiled code (though difficult and somewhat unusual).

The usual way this happens is that the code being interpreted is much more compact than the machine code, or it's running on a CPU that has a larger data cache than code cache.

In such a case, a small interpreter (e.g., the inner interpreter of a Forth implementation) may be able to fit entirely in the code cache, and the program it's interpreting fits entirely in the data cache. The cache is typically faster than main memory by a factor of at least 10, and often much more (a factor of 100 isn't particularly rare any more).

So, if the cache is faster than main memory by a factor of N, and it takes fewer than N machine code instructions to implement each byte code, the byte code should win (I'm simplifying, but I think the general idea should still be apparent).

edited Sep 17 '15 at 21:39 answered Sep 26 '11 at 5:53

Jerry Coffin


39k 4 69 143

25 +1, full ack. Especially "the language will rarely make as much difference as the programmers and design" - you'll often stumble about problems where you can optimize the algorithm, e.g. improve big-O which will give much more of a boost than the best compiler could. – schnaader Sep 26 '11 at 10:31

1 "In case anybody decides to re-run the benchmarks involved..." DON'T! Back in 2005 those old tasks were discarded and replaced by the tasks now shown in the benchmarks game. If anybody wants to re-run some programs then please re-run the current programs for the current tasks shown on the benchmarks game home page shootout.alioth.debian.org – igouy Sep 26 '11 at 16:15

13 +1 I don't like people coding C++ in either C or Java style and then stating Java to be superior. disclaimer: I don't call any language superior, but writing crappy C++ code in a style that might be perfectly suited to another language doesn't make both languages comparable. – Christian Rau Sep 26 '11 at 21:38

|

Hand rolled C/C++ *done by an expert with unlimited time* is going to be at least as fast or faster than Java. Ultimately, Java itself is written in C/C++ so you can of course do everything Java does if you are willing to put in enough engineering effort.

In practice however, Java often executes very fast for the following reasons:

- **JIT compilation** - although Java classes are stored as bytecode, this is (usually) compiled to native code by the JIT compiler as the program starts up. Once compiled, it is pure native code - so theoretically it can be expected to perform just as well as compiled C/C++ once the program has been running for long enough (i.e. after all the JIT compilation has been done)
- **Garbage collection** in Java is extremely fast and efficient - the Hotspot GC is probably the best all-round GC implementation in the world. It's the result of many man-years of expert effort by Sun and other companies. Pretty much any complex memory management system that you roll yourself in C/C++ will be worse. Of course you can write pretty fast/lightweight basic memory management schemes in C/C++, but they won't be nearly as versatile as a full GC system. Since most modern systems need complex memory management, Java therefore has a big advantage for real-world situations.
- **Better platform targetting** - by delaying compilation to application start-up (JIT compilation etc.) the Java compiler can take advantage of the fact that it knows the *exact* processor it is executing on. This can enable some very beneficial optimisations that you wouldn't be able to do in pre-compiled C/C++ code that needs to target a "lowest common denominator" processor instruction set.
- **Runtime statistics** - because JIT compilation is done at runtime, it can gather statistics while the program is executing which enable better optimisations (e.g. knowing the probability that a particular branch is taken). This can enable Java JIT compilers to produce better code than C/C++ compilers (which have to "guess" the most likely branch in advance, an assumption which may often be wrong).
- **Very good libraries** - the Java runtime contains a host of very well written libraries with good performance (especially for server-side applications). Often these are better than you could write yourself or obtain easily for C/C++.

At the same time C/C++ also have some advantages:

- **More time to do advanced optimisations** - C/C++ compilation is done once, and can therefore spend considerable time doing advanced optimisations if you configure it to do so. There's no theoretical reason why Java couldn't do the same, but in practice you want Java to JIT-compile code relatively quickly, so the JIT compiler tends to focus on "simpler" optimisations.
- **Instructions that aren't expressible in bytecode** - while Java bytecode is fully general purpose, there are still some things you can do at a low level that you can't do in bytecode (unchecked pointer arithmetic is a good example!). By (ab)using these kind of tricks you can get some performance advantages
- **Less "safety" contstraints** - Java does some extra work to ensure that programs are safe and reliable. Examples are bounds checks on arrays, certain concurrency guarantees, null pointer checks, type safety on casts etc. By avoiding these in C/C++ you can get some performance gains (although arguably this can be a bad idea!)

Overall:

- Java and C/C++ can achieve similar speeds
- C/C++ probably has the slight edge in extreme circumstances (it's not surprising that AAA game developers still prefer it, for example)
- In practice it will depend on how the different factors listed above balance out for your particular application.

edited Sep 28 '11 at 8:55

answered Sep 26 '11 at 6:40


mikera
17.7k 2 62 78

9 Ad "more time for optimisations in C++": That's *one* of the tweaks that the Oracle VM does when you chose the Server VM: It accepts a higher start-up cost in order to allow higher performance in the long run. The Client VM, however is tweaked for optimal startup time. So that distinction even exists *within* Java. – Joachim Sauer Sep 26 '11 at 6:55

7 -1: A C++ compiler can take much more time (hours, literally, for a large library) to crate a very optimized binary. Java JIT compiler cannot take so much time, even the "server" version. I seriously doubt that Java JIT compiler would be able of performing Whole Program Optimization the way the MS C++ compiler does. – quant_dev Sep 26 '11 at 12:56

19 @quant_dev: sure, but isn't that exactly what I said in my answer as a C++ advantage (more time to do advanced optimisation)? So why the -1? – mikera Sep 26 '11 at 13:37

13 Garbage collection is not a speed advantage for Java. It's only a speed advantage if you're a C++ programmer that does not know what you're doing. If all you're checking is how fast you can allocate, then yes, the garbage collector will win. Overall program performance, however, can still be better done by manually managing memory. – Billy ONeal Sep 26 '11 at 14:19

4 ... *But* with C++, you could always theoretically put a "JIT-like layer" that does similar branch optimizations at runtime, while maintaining the raw speed of a C++ program. (Theoretically. :( ) – Mateen Ulhaq Sep 26 '11 at 23:47

|

The Java runtime *isnt* interpreting bytecode. Rather, it uses whats called Just In Time Compilation. Basically, as the program is run, it takes bytecode and converts it into native code optimized for the particular CPU.

answered Sep 26 '11 at 4:58

GrandmasterB
34.2k 5 67 119

10 @Steve314: but the purely interpreting VMs *won't* be the ones that outperform C++, so they are not really relevant to this question. – Joachim Sauer Sep 26 '11 at 6:53

2 @starblue, well, it is somewhat possible with a static compilation - see the profile-guided optimisation. – SK-logic Sep 26 '11 at 10:08

|

All things being equal, you could say: *no, Java should never be faster*. You could always implement Java in C++ from scratch and thereby get at least as good performance. In practice, however:

- JIT *compiles* the code on the end-user's machine, allowing it to optimise for the exact CPU that they are running. While there's an overhead here for the compilation, it may well pay off for intensive apps. Often real life programs are not compiled for the CPU you are using.
- The Java compiler may well be better at automatically optimising things than a C++ compiler. Or it may not, but in the real world, things aren't always perfect.
- Performance behaviour can vary due to other factors, such as garbage collection. In C++, you typically call the destructor immediately when done with an object. In Java, you simply release the reference, delaying the actual destruction. This is another example of a difference which is neither here nor there, in terms of performance. Of course, you can argue that you could implement GC in C++ and be done with it, but the reality is that few people do / want to / can.

As an aside, this reminds me of the debate regarding C in the 80s / 90s. Everyone was wondering "can C ever be as fast as assembly?". Basically, the answer was: no on paper, but in reality the C compiler created more efficient code than 90% of the assembly programmers (well, once it matured a bit).

edited Sep 26 '11 at 13:57

answered Sep 26 '11 at 5:23

Daniel B
5,844 1 15 27

2 Regarding GC, it is not just that GC may delay the destruction of objects (which shouldn't matter in the longer term); the fact is that with the modern GCs, allocation/deallocation of short-lived objects is extremely cheap in Java compared to C++. – Péter Török Sep 26 '11 at 7:25

8 @PéterTörök But in C++, short-lived object are often put on the stack, which in turn is much faster than any GC-ed heap Java can use. – quant_dev Sep 26 '11 at 12:57

2 @DonalFellows What makes you think I have to worry about memory management in C++? Most of the time I don't. There are simple patterns you need to apply, which are different from Java, but that's it. – quant_dev Sep 26 '11 at 15:31

|

> But allocation is only half of memory management -- deallocation is the other half. It turns out that for most objects, the direct garbage collection cost is -- zero. This is because a copying collector does not need to visit or copy dead objects, only live ones. So objects that become garbage shortly after allocation contribute no workload to the collection cycle.

> ...
>
> JVMs are surprisingly good at figuring out things that we used to assume only the developer could know. By letting the JVM choose between stack allocation and heap allocation on a case-by-case basis, we can get the performance benefits of stack allocation without making the programmer agonize over whether to allocate on the stack or on the heap.

http://www.ibm.com/developerworks/java/library/j-jtp09275/index.html

answered Sep 26 '11 at 6:34

Landei
1,898 1 9 19

1 I like how the substance of this is : java is for noobs, trust the magic GC, it knows better. – Morg. Nov 21 '12 at 8:47

1 @Morg: Or you can read it that way: Java is for people who like to get things done instead of wasting their time with bit twiddling and manual memory management. – Landei Nov 21 '12 at 19:38

3 @Landei I think your comment would have way more credibility if any decent long-lasting widely used codebase had been written in Java. In my world, real OS's are written in C, postgreSQL is written in C, as are most important tools that would really be a pain to rewrite. Java was (and that's even the official version) to enable less skilled people to program in herds and yet reach tangible results. – Morg. Dec 1 '12 at 15:50

1 @Morg I find it very strange how you seem to focus just on OSs. This simply can't be a good measure, for several reasons. First, the requirements of OSs are crucially different from most other software, second you have the Panda thumb principle (who wants to rewrite a complete OS in another language, who wants to write his own OS if there are working and even free alternatives?) and third other software uses the features of the OS, so there is no need to ever write a disk driver, task manager, etc. If you can't provide some better arguments (not based entirely on OSs) you sound like a hater. – Landei Dec 4 '12 at 15:48

|

While a completely optimized Java program will seldom beat a completely optimized C++ program, differences in things like memory management can make a lot of algorithms idiomatically implemented in Java faster than the same algorithms idiomatically implemented in C++.

As @Jerry Coffin pointed out, there are a lot of cases where simple changes can make the code much faster -- but often it can take too much unclean tweaking in one language or the other for the performance improvement to be worthwhile. That's probably what you'd see in a *good* benchmark that shows Java doing better than C++.

Also, though usually not all that significant, there are some performance optimization that a JIT language like Java can do that C++ can't. The Java runtime can include improvements *after* the code has been compiled, which means that the JIT can potentially produce optimized code to take advantage of new (or at least different) CPU features. For this reason, a 10 year old Java binary might potentially outperform a 10 year old C++ binary.

Lastly, complete type safety in the bigger picture can, in very rare cases, offer extreme performance improvements. Singularity, an experimental OS written almost entirely in a C#-based language, has much faster interprocess communication and multitasking due to the fact that there's no need for hardware process boundaries or expensive context switches.

answered Sep 26 '11 at 10:05

Rei Miyasaka
4,176 1 26 32

Posted by Tim Holloway on JavaRanch:

> Here's a primitive example: Back when machines operated in mathematically-determined cycles, a branch instruction typically had 2 different timings. One for when the branch was taken, one for when the branch wasn't taken. Usually, the no-branch case was faster. Obviously, this meant that you could optimize logic based on the knowledge of which case was more common (subject to the constraint that what we "know" isn't always what's actually the case).
>
> JIT recompilation takes this one step further. It monitors the actual real-time usage, and flips the logic based on what actually is the most common case. And flip it back again if the workload shifts. Statically-compiled code can't do this. That's how Java can sometimes out-perform hand-tuned assembly/C/C++ code.

Source: http://www.coderanch.com/t/547458/Performance/java/Ahead-Time-vs-Just-time

answered Sep 26 '11 at 17:31

Thiago Negri
161 1

3 And once again, this is wrong/incomplete. Static compilers with profile-guided optimsation *can* recognise this. – Konrad Rudolph Sep 26 '11 at 21:12

2 Konrad, static compilers can flip the logic based on current workload? As I understand, static compilers generate code once and it stays the same forever. – Thiago Negri Sep 27 '11 at 15:50

1 Current workload, no. But *typical* workload. Profile-guided optimisation analyses how your program runs under typical load and optimises hot-spots accordingly, just like the HotSpot JIT does. – Konrad Rudolph Sep 27 '11 at 16:02

That is because the final step generating machine code happens transparently *inside* the JVM when running your Java program, instead of explicit when building your C++ proram.

You should consider the fact that modern JVM's spend quite a lot of time compiling the byte code on the fly to native machine code to make it as fast as possible. This allow the JVM to do all kinds of compiler tricks that can be even better by knowing the profiling data of the program being run.

Just such a thing as automatically inlining a getter, so that a JUMP-RETURN is not needed to just get a value, speeds up things.

However, the thing that really has allowed fast programs is better cleaning up afterwards. The garbage collection mechanism in Java is *faster* than the manual malloc-free in C. Many modern malloc-free implementations use a garbage collector underneath.

answered Sep 26 '11 at 6:10
user1249

1 "Many modern malloc-free implementations use a garbage collector underneath." Really? I'd like to know more; Do you have any references? – Sean McMillan Sep 28 '11 at 12:51

Short answer - it is not. Forget it, the topic is as old as fire or wheel. Java or .NET is not and will not be faster than C/C++. It's fast enough for most tasks where you don't need to think about optimization at all. Like forms and SQL processing, but that's where it ends.

For benchmarks, or small apps written by incompetent developers yes, the end result will be that Java/.NET is probably going to be close and maybe even faster.

In reality, simple things like allocating memory on stack, or simply using memzones will simply kill the Java/.NET on the spot.

Garbage collected world is using sort of memzone with all the accounting. Add memzone to C and C will be faster right there on the spot. Especially for those Java vs. C "high-performance code" benchmarks, that go like this:

```
for(...)
{
alloc_memory//Allocating heap in a loop is verrry good, in't it?
zero_memory//Extra zeroing, we really need it in our performance code
do_stuff//something like memory[i]++
realloc//This is lovely speedup
strlen//loop through all memory, because storing string length is soo getting old
free//Java will do that outside out timing loop, but oh well, we're comparing
apples to oranges here
}//loop 100000 times
```

Try to use stack based variables in C/C++ (or placement new), they translate into `sub esp, 0xff` , it's a single x86 instruction, beat that with Java - you can't...

Most of the time I see those benches where Java against C++ are compared it causes me to go like, wth? Wrong memory allocation strategies, self-growing containers without reserves, multiple new's. This is not even close to performance oriented C/C++ code.

Also a good read: https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf

answered Sep 26 '11 at 13:02

Coder
4,718 2 28 43

2 @SK-Logic: Wrong, with memzones or stack allocation there is NO memory allocation or deallocation AT ALL. You have a block of memory and you just write to it. Mark block as free with volatile variable like concurrency protection InterlockedExchange etc., and the next thread just dumps it's data to preallocated block without going to OS for memory at all, if it sees it's free. With stack it's even easier, with the only exception that you can't dump 50MB on the stack. And that object lifetime is only inside {}. – Coder Sep 26 '11 at 14:46

1 @SK-logic: For performance critical parts of the application, yes I do. Looping over null terminated byte-array is always faster than looping through tab terminated string via streaming libraries. Or searching via static hash tables vs. dynamic

name-value collections. You have to know when you should choose which. And GC is not going to save you from the fact that your data container is simply ineffective, it can amortize the costs, but it ends right there. And when you know you need to optimize one exact hotspot, it's a lot easier to do that in C/C++. – Coder Sep 26 '11 at 16:16

1 @SK-logic: Compilers are correctness first, performance second. Search engines, database engines, real-time trading systems, games are what I would consider performance critical. And most of those rely on flattish structures. And either way, compilers are mostly written in C/C++. With custom allocators I guess. Then again, I see no problems with using tree or list elements over rammap. You just use placement new. There is no much complexity in that. – Coder Sep 26 '11 at 16:38

2 @SK-logic: It's not much faster, every .NET/Java app I've seen, always turned out to be slower, and a real hog. Every rewrite of managed app into SANE C/C++ code resulted in cleaner and lighter app. Managed apps are always heavy. See VS2010 vs 2008. Same data structures, but VS2010 is a HOG. Correctly written C/C++ apps usually boot up in milliseconds, and don't get stuck on splash screens, while also consuming a lot less memory. The only downside is that you have to code with hardware in mind, and a lot of people don't know how it is nowadays. It's only benchmarks where managed have a chance. – Coder Sep 26 '11 at 17:12

1 your anecdotal evidence does not count. Proper benchmarks shows the real difference. It is especially strange that you're referring to the GUI applications, bound to the bulky and suboptimal GUI libraries. And, what is more important - in theory the performance limit is much higher for a properly implemented GC. – SK-logic Sep 26 '11 at 17:32

|

> See the following links ... But how is this even possible? It boggles my mind that interpreted bytecode could ever be faster than a compiled language.

1. Do those blog posts provide trustworthy evidence?
2. Do those blog posts provide definitive evidence?
3. Do those blog posts even provide evidence about "interpreted bytecode"?

Keith Lea tells you there are "obvious flaws" but does nothing about those "obvious flaws". **Back in 2005 those old tasks were discarded** and replaced by the tasks now shown in the benchmarks game.

Keith Lea tells you he "took the benchmark code for C++ and Java from the now outdated Great Computer Language Shootout and ran the tests" but actually he only shows measurements for 14 out of 25 of those outdated tests.

Keith Lea now tells you he wasn't trying to prove anything with the blog post seven years before, but back then he said "I was sick of hearing people say Java was slow, when I know it's pretty fast..." which suggests back then there was something he was trying to prove.

Christian Felde tells you "I didn't create the code, just re-ran the tests." as-if that absolved him from any responsibility for his decision to publicise measurements of the tasks and programs Keith Lea selected.

Do measurements of even 25 tiny tiny programs provide definitive evidence?

Those measurements are for programs run as "mixed mode" Java not interpreted Java - "Remember how HotSpot works." You can easily find out how well Java runs "interpreted bytecode", because you can force Java to **only** interpret bytecode - simply time some Java programs run with and without the -Xint option.

edited Sep 26 '11 at 18:03

answered Sep 26 '11 at 17:47

igouy

544 3 6

The reality is they are both just high level assemblers that do exactly what the programmer tells them to, exaclty how the programmer tells them to in the exact order the programmer tells them. The performance differences are so small as to be inconsequential to all practical purposes.

The language is not "slow", the programmer wrote a slow program. Very rarely will a program written the best way in one language outperfrom (to any practical purpose) a program doing the same thing using the best way of the alternate language, unless the author of the study is out to grind his particular axe.

Obviously if you are going to a rare edge case like hard realtime embedded systems, the language choice may make a difference, but how often is this the case? and of those cases, how often is the correct choice not blindly obvious.

answered Sep 26 '11 at 7:38

mattnz

19.6k 3 48 78

2 In theory, an "ideal" JITting VM *must* outperform statically compiled code, by adjusting its optimisations to the dynamically collected profiling information. In practice, JIT compilers are not that smart yet, but they're at least capable of producing a code of a similar quality as with their bigger and slower static peers. – SK-logic Sep 26 '11 at 10:15

JIT, GC and so on aside, C++ can be very, very easily made much more sluggish than Java. This will not show up in benchmarks, but same app written by Java developer and a C++ developer may be much faster in Java.

- Operator overloading. Every simple operator like "+" or "=" may call hundreds of lines of code doing safety checks, disk operations, logging, tracking and profiling. And they are so easy to use that once you overload the operators, you use them naturally and copiously without noticing how the usage stacks up.
- Templates. These don't affect speed as much as memory. Uncautious use of templates will lead to generating millions of lines of code (alternatives for the basic template) without you ever noticing them. But then binary loading times, memory usage, swap usage - all that acts against benchmarks too. And RAM usage goes through the roof.

As for advanced inheritance patterns, these are pretty much similar - C++ has some which Java does not and vice versa, but all of them introduce similar, significant overhead too. So no special C++ advantage in object-heavy programming.

One more caveat: GC can be faster or slower than managing allocations manually. If you allocate a lot of small objects, in GC environment usually a chunk of memory is allocated and pieces of it dispatched as needed for new objects. In managed - each object = separate allocation takes significant time. OTOH, if you malloc() lots of memory at once and then just assign pieces of it to your objects manually, or use few bigger instances of objects, you may come up much faster.

edited Sep 26 '11 at 7:23

answered Sep 26 '11 at 7:07

SF.
4,231 1 17 32

4 I disagree with both points. Whether you use operators or methods is irrelevant. You say that they will proliferate. Nonsense – not more than methods would; you either need to call them, or not. And templates result in no more code than hand-writing that particular code over again for multiple use. There may be more code than with runtime dispatch (virtual functions) but this, too, will be irrelevant: performance of instruction cache lines matters most in tight loops and here there will only *one* template instantiation be used, so there is no relevant memory pressure due to templates. – Konrad Rudolph Sep 26 '11 at 21:07

1 Trick question. I wouldn't second-guess this at all, I would measure, act *then*. I have *never* had a problem with that tactic. – Konrad Rudolph Sep 29 '11 at 8:09

1 @KonradRudolph: This is all true when it comes to clarity and ease of writing the code, making it bug-free and maintainable. The point about efficiency of the algorithm implementation still stands though: if you're about to write `obj.fetchFromDatabase("key")` three times within five lines of code for the same key, you'll think twice whether to fetch that value once and cache it in a local variable. If you write `obj->"key"` with `->` being overloaded to act as database fetch, you're far more prone to just let it pass because the cost of the operation isn't apparent. – SF. Sep 19 '15 at 20:42

|

It amuses me how pervasive this weird notion of "interpreted bytecode" is. Have you people ever heard of the JIT-compilation? Your argument can't be applied to Java.

But, leaving JVM aside, there are cases when a direct threaded code or even a trivial bytecode interpretation can easily outperform a heavily optimised native code. Explanation is quite simple: bytecode can be quite compact and will fit your tiny cache when a native code version of the same algorithm will end up having several cache misses for a single iteration.

answered Sep 26 '11 at 6:25

SK-logic
7,869 3 19 33

2 Umm, but the bytecode, in order to be run, has to be converted to native code -- you can't feed Java bytecode to the CPU. So the size argument is invalid. – quant_dev Sep 26 '11 at 12:58

|

Somehow Stack Exchange doesn't take my other stackpoints so ... no reply unfortunately...

However the second most voted answer here is full of misinformation in my humble opinion.

A hand-rolled App by an expert in C/C++ is ALWAYS going to be much faster than a Java application, period. There is no 'as fast as Java or Faster'. it's just faster, precisely because of the items you cite below:

JIT compilation : Do you really expect an automatic optimizer to have the smarts of an expert programmer and see the link between the intent and the code the CPU is really going to run ??? Furthermore, all the JIT you do is time lost compared to an already compiled program.

Garbage Collection is a tool that simply deallocates resources that a programmer would have forgotten to deallocate, in a more or less efficient fashion.

Evidently this can only be slower than what an expert (you picked the term) C programmer would do to handle his memory (and no there are no leaks in correctly written apps).

A performance optimized C application knows the CPU it's running on, it's been compiled on it, else that means you didn't quite take all the steps for performance does it ?

Runtime Statistics This is beyond my knowledge, but I do suspect that an expert in C has more than enough branch prediction knowledge to again outsmart automated optimization -

Very good libraries There are many not-very-optimized functions readily available through libraries in Java, and the same is true in any language, however the most optimized libraries are written in C, especially for calculation.

The JVM is a layer of abstraction, which implies both good things many of which are above, and also implies that the overall solution is slower by design.

Overall:

Java cannot ever reach the speed of C/C++ due to the way it works in a JVM with a lot of protection, features and tools.

C++ has a definite clear edge in optimized software, be it for computing or games, and it's commonplace to see C++ implementations win coding contests to the point that the best Java implementations can only be seen on second page.

In practice C++ is not a toy and will not let you get away with many mistakes that most modern languages can handle, however by being simpler and less safe, it's inherently faster.

And as a conclusion, I'd like to say that most people don't give two cents about this, that in the end optimization is a sport reserved only to a very few lucky developpers and that except in cases where performance really is a concern (i.E. where multiplying hardware by 10 will not help you - or represent a few millions at least), most managers will prefer an unoptimized app and a ton of hardware.

answered Sep 26 '11 at 16:15

Morg.
260 1 5

4 This answer is full of misconceptions about what modern optimisers can do. Hand-optimised code doesn't stand a chance against that. But then, C++ *also* has an optimising compiler. – Konrad Rudolph Sep 26 '11 at 16:54

I have seen at least two impressive mmo's done in Java, to say its not fast enough for games is a misnomer. Just because game designers favor C++ more over other languages says it is simply not just Java related, it just means said programmers have never really dabbled with any other programming languages/paradigms. Anything in any language as advanced as C/C++ or even Java can produce code that could technically meet or defeat the speed argument. All that well and said comes down to what programmers know, what teams work with most and most importantly why they use said tools. Since we are tackling the game development aspect of programming, then there must be more to the argument. Simply put it's all about money and time for a business dead set on using tools that meet QA and in the real world plays no weight on xx reasons for choosing C++ over Java or any other language. It is just a mass production decision. At the most basic level of computing algorithms all we are playing with are ones and zeros, the speed argument is one of the dumbest arguments ever applied to gaming. If you want speed gains that badly then drop programming languages entirely and work with assembly that is possibly the best advantage by far.

answered Jun 6 '15 at 11:41

Meh
1

2 This wall of text doesn't appear to add anything that hasn't already been stated in the other answers. Please edit your answer to be more readable, and please make certain that your answer addresses points not raised by the other answer. Otherwise, please consider deleting your answer as it only adds noise at this point. – GlenH7 Jun 6 '15 at 12:01

protected by gnat Jun 6 '15 at 20:50

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?