Java or C++: Practical Advice You Can Use

Sergey Kuksenko, Aleksey Shipilev, Charlie Hunt

Java or C++: Practical Advice You Can Use

Sergey Kuksenko, Aleksey Shipilev, Charlie Hunt

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Who are you?

- Charlie Hunt
  - Lead JVM Performance Engineer at Oracle
  - Lead author of Java Performance book (just published!)
- Sergey Kuksenko
  - JVM Performance Engineer at Oracle
- Aleksey Shipilev
  - JVM Performance Engineer at Oracle

Program Agenda

What to Expect

Making a Choice (Things to Remember)

Dynamic and Static Compilation Differences

Memory Management Differences

Concurrency Differences

Conclusion

What to Expect

- This is *neither* a session loaded with claims that Java is *always* better, faster, stronger, *nor* a session filled with C++ bashing!
  - Yes, you can go now :)
- We hope that at the end of this session, you can make an informed decision when asked to choose Java or C++ for your application.
- With this session, we would like to offer some practical advice to those who are faced with such a question.

Program Agenda

What to Expect

Making a Choice (Things to Remember)

Dynamic and Static Compilation Differences

Memory Management Differences

Concurrency Differences

Conclusion

Making a Choice

Where to Start

- Put on your architect “hat”
- Focus on the “...ilities” for the application
  - “...ilities” are non-functional requirements
  - Security, reliability, manageability, portability, scalability, performance, etc
  - Ask all stakeholders to rank them in order of importance
  - Do not forget about other “business” pressures
    - “time to market”, “TCO”, “ROI”, etc.
- We will focus more on scalability and performance

Making a Choice

Other considerations

- The “human factor”
- Suppose a Java application
  - A team of expert Java developers will achieve better results than a team of expert C++ developers on Java almost all the time.
- Suppose a C++ application
  - A team of expert C++ developers will achieve better results than a team of expert Java developers on C++ almost all the time.

Making a Choice

Java versus C++

- Trichotomy Principle
  - For an application under consideration, a given language attribute may be
    - + more attractive
    - – less attractive
    - ≈ it does not matter
- Freedom Principle
  - C++ offers developers “anything”, but language complexity comes with it
  - Java induces more “limits” on developers, but a less complex language

Making a Choice

Tied together with “business factors”

- Java generally offers a larger set of reusable components, i.e. 3rd party libraries (free and commercial)
  - Especially true for areas such as security and logging
  - Many reusable components are implementations of a Java standard
- C++ has reusable libraries (free and commercial)
  - But, there does not appear to be agreement on how to best implement them, lack of standardization
  - Reusability and security? Java is generally better.

Making a Choice

Portability

- Java claims: “Write once, run anywhere”
  - Reality: “Write once, run where JVM is available”
  - Java portability easier to realize if platform has a JVM
- C++ claims: “Write once, compile anywhere”
  - Reality: “Write according to C++ standard(s), compile it, if it fails to compile, refactor/port/re-write it, and compile again”
  - C++, same OS, different compiler… generally portable to more platforms

Making a Choice

Scalability

- Multicore platforms are common place now
  - Developing correct, fast, and scalable applications is... well, let's face it, it's not trivial!
- Much harder to write scalable and multi-threaded applications
- More on this later

Making a Choice

Performance

- Nearly all “Java vs C++” battles waged in performance
- But, there’s many aspects of performance (next slide)
- Beware of claims made in “Java vs C++” wars
  - Don’t trust performance myths, rumors and urban legends
  - Don’t trust cross-language benchmarks
- *Do* take time to understand and reason why differences exist
- Ask questions if suspicious

Making a Choice

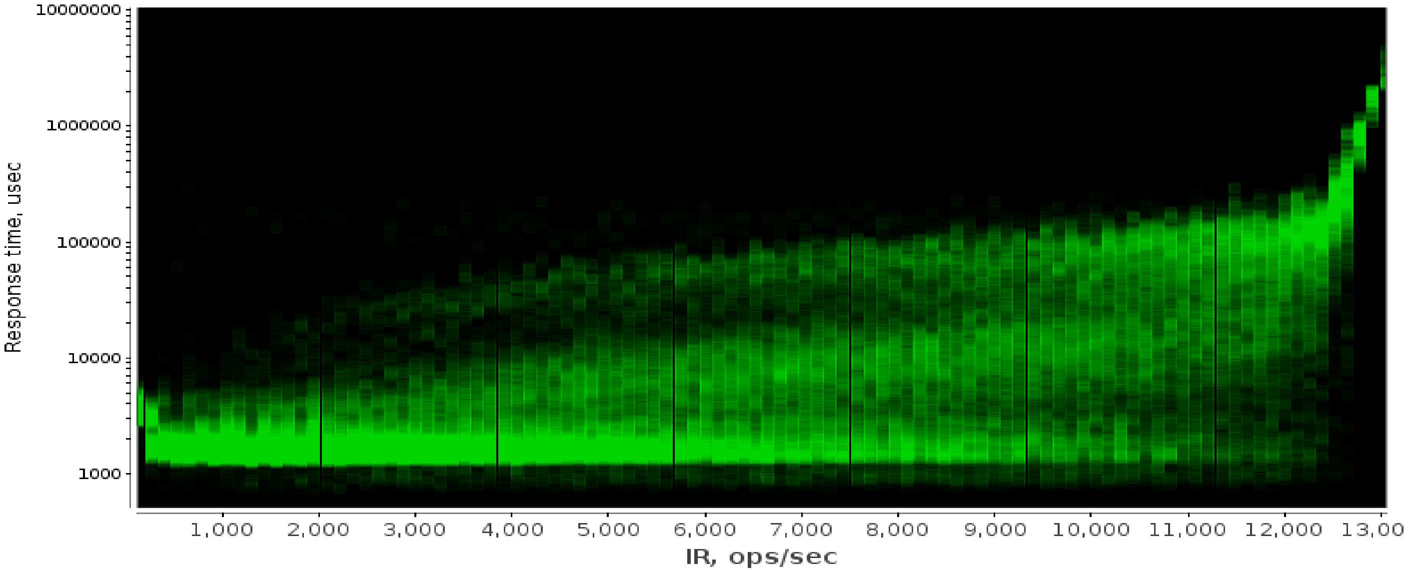
Performance Aspects

- **Throughput**, “How much work done per some period of time?”
- **Latency**, “How long it takes to respond to some stimulus?”
- **Footprint**, “How much memory does it consume?”
- **Startup time**, “How long does it take the application to initialize?”
- **Time to performance**, “How long does it take until the application hits peak performance?”
- **Predictability**, “How much jitter in all of the above?”

Making a Choice

Performance Aspects

- Throughput versus latency



Making a Choice

Common Java vs C++ mistakes

- Claim

  "Java has a big runtime (JIT, GC, classloaders) that is why Java will lose at all performance aspects"

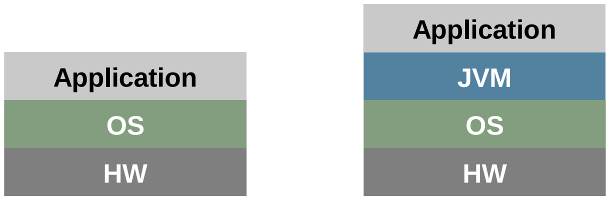
- Reality

  "Java has a big runtime (JIT, GC, classloaders) that is why Java may lose (or may win) at some performance aspects"

Making a Choice

Common Java vs C++ mistakes

C++ stack Java stack

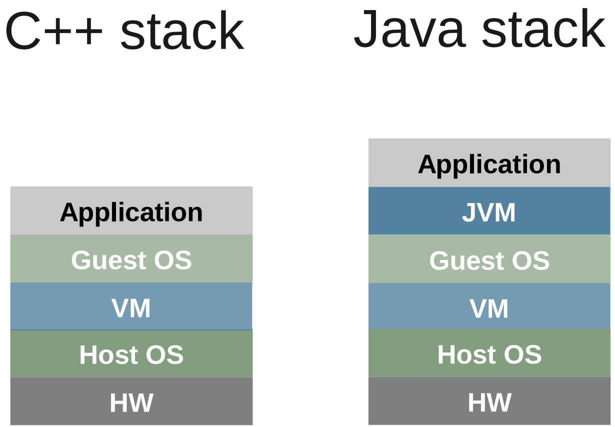


Does one extra layer affect performance?

- Nearly all performance degradation claims about JVM miss the fact hardware and operating system can also impact performance (and predictability)
- Real Story: C++ application migrated to more recent Linux kernel
  - Observed huge performance regression
  - Root cause: thread starvation due to change in default CPU scheduler

Making a Choice

Common Java vs C++ mistakes



In reality,
there are more layers!

- Could changes in host OS, VM, guest OS impact application performance?
- Note, update to a more recent JVM is subject to performance changes too
- Also realize a change to more recent hardware could also impact performance

Program Agenda

What to Expect

Making a Choice (Things to Remember)

Dynamic and Static Compilation Differences

Memory Management Differences

Concurrency Differences

Conclusion

Dynamic and Static Compilation Differences

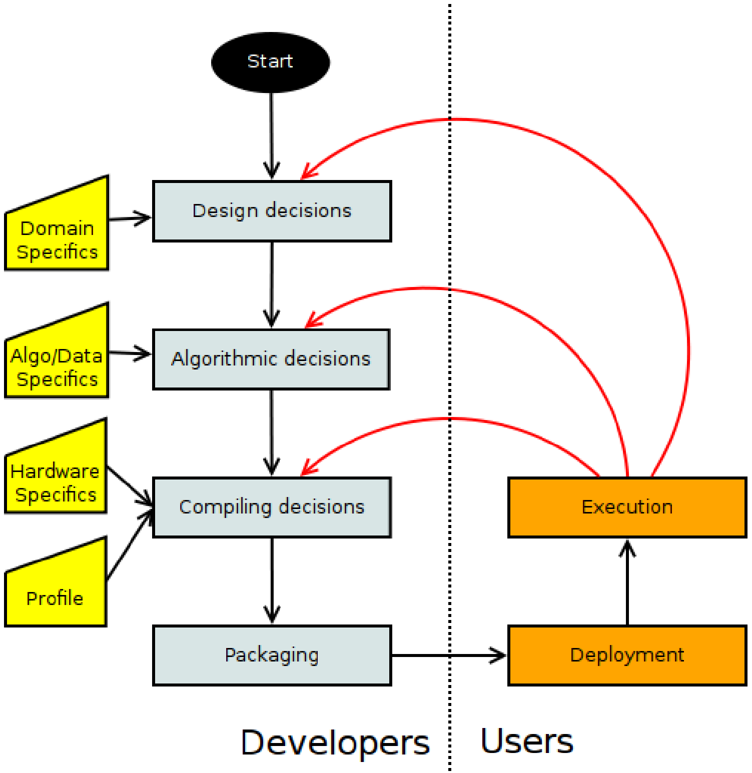
Bird-Eye Difference

- Modern C++ compilers are *static*
  - Source code → Native object code → Native executable
  - Most of compilation work happens before executing
  - “ahead-of-time” compilation
- Modern Java VMs use *dynamic* compilers
  - Source code → Bytecode → JITted code → Interpreter + JITted executable
  - Most of compilation work happens during executing
  - “just-in-time” (JIT) compilation

Dynamic and Static Compilation Differences

Static Compilation

- Static Compilation (C++ alike)
  - Has knowledge about all constructs in the program during compilation
  - Once compiled, there's no re-optimization unless the program is shut down, recompiled, and restarted
  - Execution of a code path usually takes the same execution time
  - Does not require code paths to be executed before compiling them



Dynamic and Static Compilation Differences

Static Compilation

- Most people argue that...

Static compilation has theoretically unlimited compilation time.

↓

Static compiler can do more sophisticated optimizations.

↓

Statically compiled code is always faster.

- Which consequence is really true?

Dynamic and Static Compilation Differences

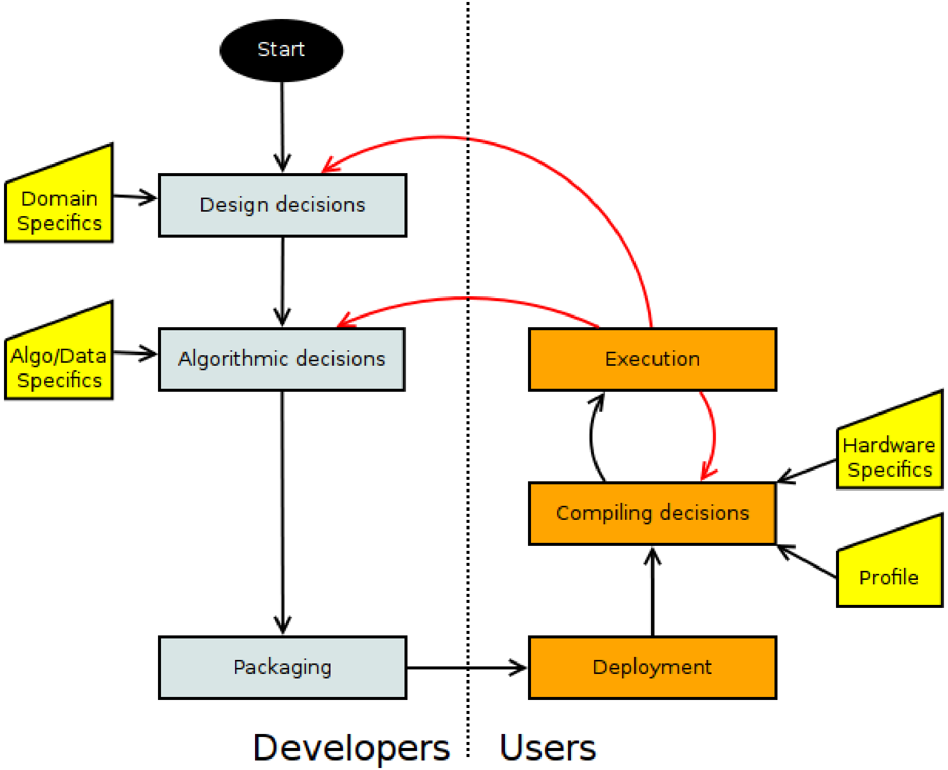
Dynamic Compilation

- Dynamic Compilation (JIT)
  - Has knowledge of the classes loaded and methods the program has executed
  - Makes optimization decisions based on code paths executed
    - Code generation depends on what is observed: classes that have been loaded, code paths executed, branches taken
  - May re-optimize if assumption was wrong, or alternative code paths taken
    - Instruction path length may change between invocations of methods as a result of de-optimization / re-compilation

Dynamic and Static Compilation Differences

Dynamic Compilation

- Can do non-conservative optimizations in dynamic
- Separates optimization from *product delivery* cycle
  - Update JVM, run the same application, realize improved performance!
  - Can be "tuned" to the target platform



Dynamic and Static Compilation Differences

Profile-guided optimization

- PGO = using profile for more efficient optimization
- Frequently advertised as unique Java feature
- ...but all modern C++ compilers also have it
  - Microsoft claims MSVC PGO gives +10% - +30% better performance
  - GCC claims +20% on average
  - Oracle Solaris Studio: +15% on more “integer” code, little gain on “floating”
- What kind of performance? (throughput? latency?)

Dynamic and Static Compilation Differences

Profile-guided optimization

- PGO in static compilers
  - Developers should care about it
  - Profiles should be collected before the final compilation and should cover ALL typical usage scenarios
  - Inconsistent profiled usage scenario may lead to performance degradation in the most common use case

Dynamic and Static Compilation Differences

Profile-guided optimization

- PGO in JVMs
  - Always have it, turned on by default
  - Developers (usually) not interested or concerned about it
  - Profile is always consistent to execution scenario

Dynamic and Static Compilation Differences

Inlining and devirtualization

- Inlining is the most profitable compiler optimization
  - Rather straightforward to implement
  - Huge benefits: expands the scope for other optimizations
- OOP needs polymorphism, that implies virtual calls
  - Prevents naïve inlining
  - Devirtualization is required
  - (This does not mean you should not write OOP code)

Dynamic and Static Compilation Differences

Inlining and devirtualization

- C++ inlining
  - “inline” keyword is just a hint, compiler itself decides what can be inlined
- C++ devirtualization
  - Manually controlled by developer
    - Developers should care at design stage, trading off extensibility for performance: should library method be *virtual,* or not? *final*, or not?
    - Often premature optimization
  - Can be done by compiler
    - Requires whole program analysis
    - May be broken by low-level code

Dynamic and Static Compilation Differences

Inlining and devirtualization

- JVM devirtualization
  - Developers shouldn't care
  - Analyze hierarchy of currently loaded classes
  - Efficiently devirtualize **all** monomorphic calls
  - Able to devirtualize polymorphic calls

Dynamic and Static Compilation Differences

Inlining and devirtualization

- JVM may inline dynamic methods!
  - Reflection calls
  - Runtime-synthesized methods
    - JSR 292
- Can you do the same in C++?

Dynamic and Static Compilation Differences

Inlining and devirtualization

- JVM may inline dynamic methods!
  - Reflection calls
  - Runtime-synthesized methods
    - JSR 292
- Can you do the same in C++?
  - You can, if you have a compiler, and not afraid to patch object code.
  - (Assuming you will not resort to Lua, etc.)

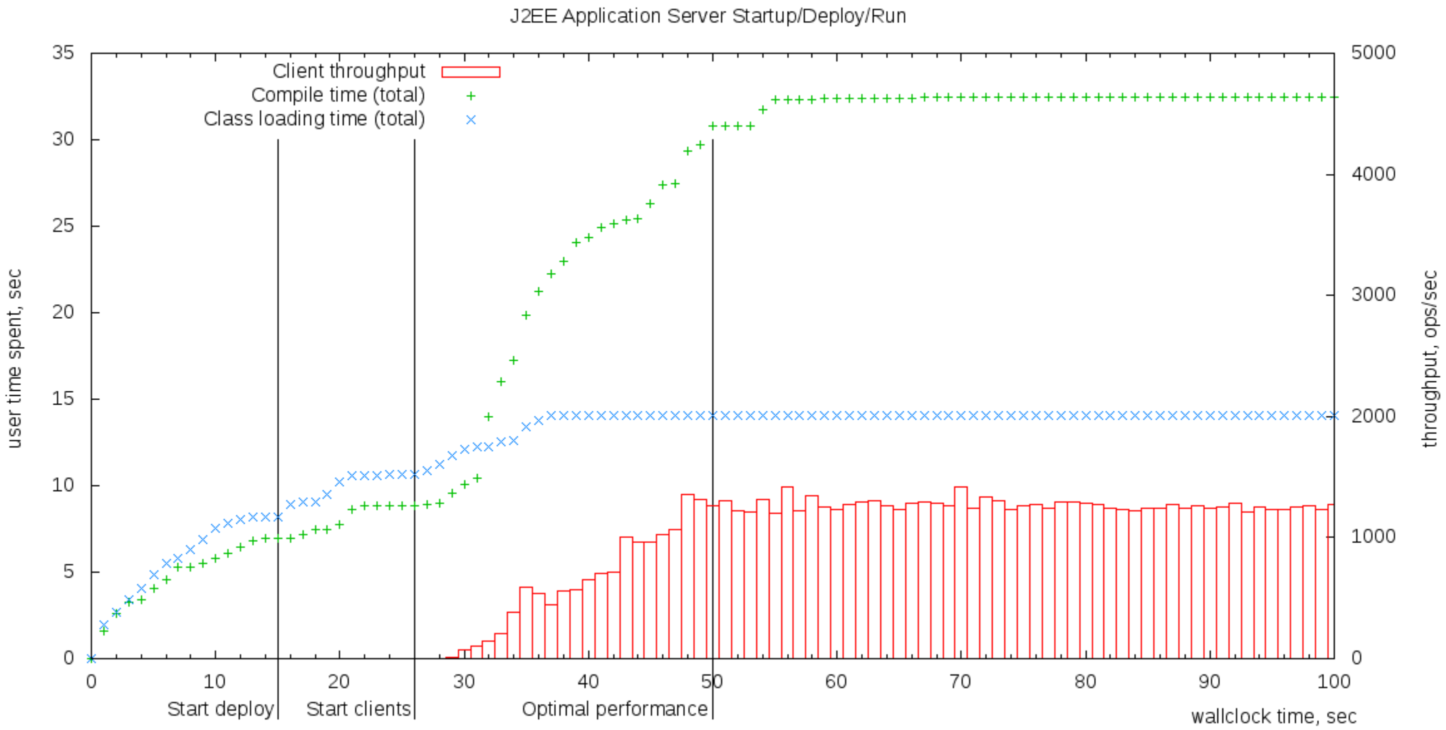
Dynamic and Static Compilation Differences

Dynamic compilation overhead

- Is dynamic compilation overhead essential?
  - The longer your application runs, the less the overhead
- Trading off compilation time, not application time
  - Steal some cycles very early in execution
  - Done automagically and transparently to application
- Most of “perceived” overhead is compiler waiting for more data
  - ...thus running semi-optimal code for time being

Dynamic and Static Compilation Differences

Dynamic compilation overhead



Dynamic and Static Compilation Differences

General Advice

- Look at what's needed for the application
  - Rapid startup? Must it run at peak performance immediately?
  - Is predictable throughput required, (varies little run-to-run)?
  - What about pause time requirements?
  - Are there branches in the hot code path that are considered “exceptional” branches?
  - Are there implementations of interfaces or virtual methods which are expected to never, or rarely, be taken?
  - Time to market? (programmer productivity and memory management, maintenance)

Program Agenda

What to Expect

Making a Choice (Things to Remember)

Dynamic and Static Compilation Differences

Memory Management Differences

Concurrency Differences

Conclusion

Memory Management Differences

- The largest battlefield in Java vs C++ wars
- Java:
  - Automatic dynamic memory management (garbage collection)
- C++:
  - Stack-based memory management (RAII)
  - Explicit dynamic memory management (new/delete, malloc/free)

Memory Management Differences

C++ RAII

- RAII – Resource Acquisition Is Initialization
  - Scope-based resource allocation/deallocation
  - Very good for heavy resources (file handles, etc)
    - Java analogue: try/finally, AutoCloseable (Java7)

- Is RAII good for memory?

Memory Management Differences

C++ RAII

- Stack-based memory allocation advantages:
  - Allocation/deallocation has marginal cost
  - Very good data locality
  - Very good thread locality
  - Allocation is cache-friendly

Memory Management Differences

C++ RAII

- Stack-based memory allocation caveats:
  - Buffer overrun is a top security bug for many years:
    - “CWE/SANS TOP 25 Most Dangerous Software Errors”, 2011
    - http://www.sans.org/top25-software-errors/
  - Without qualitative development, leads to more data copying than heap-based allocation (negates performance)
  - May require larger stack sizes
    - Running with lots of threads is hard

Memory Management Differences

Explicit memory management (C++)

- Common myths:
  - New/delete (malloc/free) have zero performance cost
  - Memory footprint is minimal (the same as required to the application)
  - No pauses
  - Manual memory management is always scalable
- Did you forget about memory allocator?
- What memory allocator is used in your application?
  - default (which one?), ptmalloc, dlmalloc, hoard, jemalloc, tcmalloc, custom, etc...

Memory Management Differences

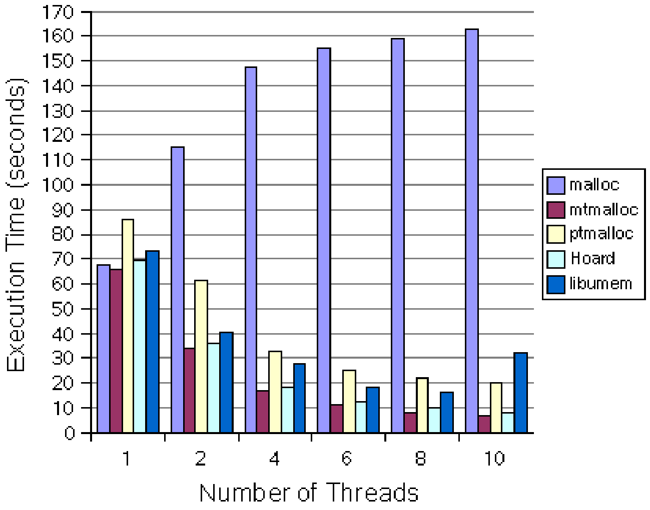
Explicit memory management (C++)

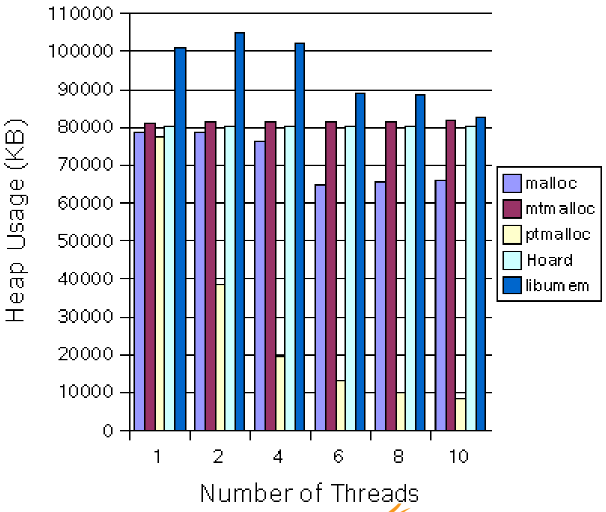
- Many articles challenge these mythes
  - David Detlefs , Al Dosser , Benjamin Zorn, "Memory allocation costs in large C and C++ programs", Software—Practice & Experience, v.24 n.6, p.527-542, June 1994.
  - Analyzed 11 applications and 4 allocators:
    - Program execution time varies by **20%**
    - Max memory footprint varies by **25%**

Memory Management Differences

Explicit memory management (C++)

- Joseph Attardi, Neelakanth Nadgir, "A Comparison of Memory Allocators in Multiprocessors", June 2003
  - http://developers.sun.com/solaris/articles/multiproc/multiproc.html





Memory Management Differences

Explicit memory management (C++)

- “How to avoid pauses in their entirety?”
- If you google it, most of the answers are in form:

Allocate all required memory
at application initialization

Memory Management Differences

Explicit memory management (C++)

- Issues:
  - Heap fragmentation
  - Scalability:
    - Heap contention
    - False sharing
  - Some concurrent lock-free algorithms may be implemented only with GC
  - NUMA
- All these issues are solvable (at what cost?)

Memory Management Differences

Garbage Collection

- Easy to use
- May give better performance due to:
  - Really cheap and thread local allocation (by default in HotSpot and other JVMs)
  - When the application satisfies the generational hypothesis
  - Improve data locality
  - -XX:+UseNUMA
  - -XX:+CompressedOops
  - -XX:+LargePages (just turn on)

Memory Management Differences

Garbage Collection

- Matthew Hertz , Emery D. Berger,
  "Quantifying the performance of garbage collection vs. explicit memory management"
  - "GC is almost always faster as explicit memory management, but requires larger amount of RAM"
  - GC may require as much as 3x more RAM to operate efficiently

Memory Management Differences

Garbage Collection

- Works well for throughput
  - just set up enough heap size and turn on ParallelOld collector.
- Can be challenging for predictable low latency
  - Tune CMS or use G1 (don't forget – may sacrifice some throughput)
- Can be difficult for guaranteed / soft real time latency
  - JRockit Deterministic Garbage Collection
- Poor choice when minimal / very small memory footprint is required
  - Paging is killer for GC

Program Agenda

What to Expect

Making a Choice (Things to Remember)

Dynamic and Static Compilation Differences

Memory Management Differences

Concurrency Differences

Conclusion

Concurrency Differences

Java

- Threads
  - Supported ever since
- Locks
  - Supported ever since
  - java.util.concurrent.locks.* since Java 5, circa 2004
- Memory Model
  - In place ever since
  - Correct since Java 5, circa 2004

Concurrency Differences

C++

- Threads
  - Before 2011, only libraries: Pthreads, Threads, Boost, Intel TBB
  - C++11: built-in support
- Locks
  - Before 2011, only libraries: Mutex, Boost locks, futex, etc
  - C++11: built-in mutexes
- Memory Model
  - Only in C++11: built-in memory model

Concurrency Differences

C++

- C++ is so flexible and powerful, that...
  - It's hard to write concurrent applications
  - It's hard to write *portable* concurrent applications
  - It's hard to write *efficient and* portable concurrent applications
- “Threads Cannot Be Implemented as a Library”, Hans-J. Boehm
  - No rules for memory ordering, i.e. no memory model
  - Compilers bail out on concurrency optimizations
  - (Primary motivation for built-in thread support in C++11)

Concurrency Differences

Java “unique features”

- Versatile concurrent lock-free primitives (j.u.c.*)
- Many j.u.c.* primitives support fairness
  - Otherwise impossible to fight starvation
- Thread pools
- Fork/Join framework
- Biased Locking
  - Impossible to do in unmanaged environments, e.g. C++

Program Agenda

What to Expect

Making a Choice (Things to Remember)

Dynamic and Static Compilation Differences

Memory Management Differences

Concurrency Differences

Conclusion

Conclusion

Java or C++?

- It depends :-) … on what is most important to the application and its stakeholders
- Which is best for your application can be determined by prioritizing the "ilities" and other "business factors"
  - Ask all application stakeholders to prioritize
    - May likely have different priorities
    - Get agreement on which are most important

Conclusion

Java or C++?

- Evaluate and document the advantages and challenges of using each technology
  - Start with the highest priority first
  - Take into consideration various aspects, especially performance
  - Don't forget to include and evaluate "business factors”
  - Also consider that technologies evolve and what existing challenges may be addressed for a technology during the application's lifetime
  - Evaluation may require some quantification of the magnitude of the difference between the two technologies, i.e. peak throughput, predictability, memory footprint, time to market

Conclusion

Java or C++?

- The Decision?
  - Prioritization and evaluation of advantages & challenges will guide you to an informed decision as to the best choice for the application under consideration.

JavaOne™

MOVING JAVA FORWARD

ORACLE®

JavaOne™