

The Tale of Java Performance

Oswaldo Pinali Doederlein, Visionnaire S/A, Brasil

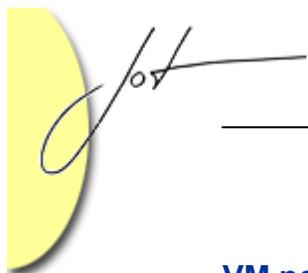
Abstract

The Java platform introduced Virtual Machines, JIT Compilers and Garbage Collectors to the masses and to mainstream software development. Demand and competition drove impressive improvements in the performance of Java implementations, and while the state of the art can be learned from JVM research papers and product benchmarks, we offer a “Making Of” exposing the challenges, tensions and strategies behind this history, extrapolating to similar platforms such as Microsoft .NET’s CLR.

1 BRIEF HISTORY OF JAVA IMPLEMENTATIONS

Java, like most new languages, suffered from immature implementations and very weak performance in the early days. The unprecedented success of early Java, though, should teach us the first important lesson about performance – it’s not always critical. The Web was expanding like the Big Bang, the world demanded features like secure, portable, Internet-friendly code. Even at that time, (then-) interpreted languages like Visual Basic or PERL were highly popular. Moreover, implementations always improve and Moore’s Law is always doing its magic. When I first used Java (1.0-beta) my desktop PC was a Pentium 166MHz with 64Mb RAM. Right now I’m working on a 1,6GHz Pentium-IV with 640Mb: coincidentally, an exact order of magnitude better in both speed and space (if we’re not too picky about hardware performance factors).

Java was not born to handle tiny web page embellishments forever, so it had to evolve to support everything from smart card applets to scalable enterprise applications. Moore’s Law barely compensates the distance from the “Nervous Text” applet to any current J2EE server. More than size, change in applicability imposes more and more efficiency constraints. Java debuted as a “glue” language, and as a front-end language; in these cases, most time is spend waiting user input or invoking external (fast) code, like GUI toolkits or network stacks. As soon as the language becomes successful, it must be intrinsically fast, not only look fast in easy scenarios (like servers that don’t care for large loading time or runtime footprint). Successful languages are always forced into domains their creators didn’t dream of (and didn’t design for), and the initial trade-offs are either removed or replaced by new ones as implementers strive to make developers happy.



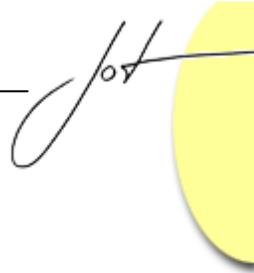
VM performance, By-the-Book

Portable code and Virtual Machines exist since the sixties, so by 1996 the field was already mature with superb previous art in interpreters, Just In Time compilers, Garbage Collectors, and more general items such as threading or multiplatform frameworks. Java introduced at least one significant performance challenge, security; but in retrospect, the costs of security only appeared important due to the overall bad performance of early implementations. Right now, only in the J2ME space the security features cause concern and drive new optimizations like pre-verification. Therefore, most of the exciting history of JVM optimisation resembles a Hollywood-style remake of an old movie by a famous director with lots of money: the result is a blockbuster loved by the public, even if some self-defined elite prefers the original black-and-white production.

Not all is remaking, though. The big news in Java history, of course, is becoming a mainstream platform. Even acknowledging that many successful real-world applications were built with precursor technologies, there is no comparison with Java, remarkably in the economics – very relevant if we consider that top performance costs huge investments from commercial implementers (even though most JVM performance tricks are rooted on academic research). Thanks to all previous work, the formula for the first wave of JVM improvements (the “JDK 1.1.x generation”) could be summarized as: implement the techniques that worked before for the other guys.

- **High Performance Interpretation.** The classloader can perform a few easy optimisations as bytecode-to-bytecode transformations, like devirtualizing calls to `final` methods. An optimal interpreter is typically generated at loading time with the best Assembly code for the machine.
- **Direct References.** The Classic VMs implemented references as indirect handles, making GC very simple but code slower. This was fixed in Sun’s EVM (aka “Solaris Production”), IBM JDK 1.1.6 and finally, Sun’s Java2 releases.
- **Just-In-Time Compilation and Generational GC.** The first generation of JIT compilers could do local optimisations and offered enough speed for applets. Late in the JDK1.1 cycle, the Sun EVM and the IBM JDK introduced stronger JITs and also the first decent GCs for Java. These were times of fast research and poor stability, so most JITs were disabled by default.
- **Library Optimisations.** Sun proceeded with many general enhancements during the 1.1 series, while Microsoft improved low-level libraries like the AWT to extract better performance in the Windows OS.

This set of improvements delivered a Java that was usable for many desktop applications and even some servers – mostly I/O-bound apps, like two-tier Servlet-based apps that are a thin bridge between a relational database and a web browser – provided that one didn’t need to serve more than a handful of simultaneous users.



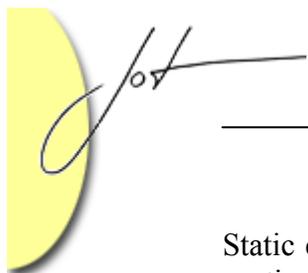
Static Compilers

People wanted to build large applications with Java long before Sun delivered the required performance. Sun initially focused on the Solaris SPARC platform with the EVM, but if the plan was luring hordes of Windows users into Solaris, it certainly didn't work. The high-profile, next-generation HotSpot project aimed high and would only bear fruit after JDK1.2, which created a big market opportunity for others. If the VMs couldn't show competitive performance, the obvious solution was "generating native code" with conventional compilers that could employ expensive global optimizations just like any other language. The main strategy was fixing the inadequacies of the existing VMs:

1. **Stressing global optimizations** that benefit from closed-world analysis, whereby all code used by the application is fixed and known to the compiler.
2. **Creating high-performance runtime support** in areas where the JDK was deficient, like threads, networking and memory management / GC.

The business plan was good, but the market was hard: TowerJ (a pioneer and leading product) went broke; SuperCede / JOVE was just killed by Instantiations; IBM's HPCJ is long retired (except for AS/400), and NaturalBridge's BulletTrain technology should belong to somebody else by the time you read this. The only commercial vendor left is Excelsior with JET, while the Free Software GCJ silently improves. Some difficulties made the success of static Java compilers harder than in other languages:

- **Compliance.** Platform-specific deployment is something unholy in Java; the certification rules enforce items like bytecode compatibility and fully dynamic classloading. The JDK sources could not be reused by non-licensed/certified products, so these companies had an option between clean-room rewriting of the libraries, or an uncomfortable dependency on the JRE (the license does not allow partial redistribution, and full compatibility with its native libraries is a pain).
- **Funding.** Products were created by small companies that couldn't compete with giants; JVMs improved faster and eventually regained leadership in most areas. IBM was the only big player to make a static compiler, but their commitment to J2EE probably killed HPCJ as much as the performance of its own JIT. See [Gu00] for a history of IBM JDK's evolution.
- **Strategy.** Static compilers excelled in the server side, but that was a bad long-term plan. The introduction of J2EE spelled bad news by stressing dynamic behaviour. Products tried to counter this with either (a) shared libraries, which requires dropping the closed-world optimizations that are a core sales pitch; or (b) a hybrid model, mixing native code with bytecode supported by an interpreter or JIT compiler – an apparently good idea that didn't suffice to save TowerJ.
- **Recession + commoditization (free high-performance JVMs).** The result is a small market, and some vendors practiced heavy pricing when they held a large performance advantage, which probably helped to kill them in the long run. But it's possible that some vendors couldn't afford to amortize R&D costs much slower, not to mention giving products away.



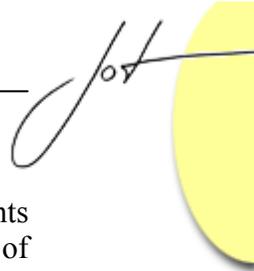
Static compilers still have advantages: Native code is the best obfuscation you can find; runtime footprint is smaller as it doesn't carry a sophisticated compiler; robustness tends to be superior, as compilers often have bugs themselves and static compiled code can be more thoroughly tested against these; deployment is often easier; loading is faster; and code performance is better for applications that are not affected by the weaknesses but do benefit from their strengths – roughly, non-dynamic apps that benefit from closed-world optimisations, and small apps for which loading time is important.

There is a place for static Java compilers in the future, but ironically, just like the JVMs, their new sweet spot may be very different from the original plans. JET is now making strides to better support desktop GUIs (with Swing and SWT), and GCJ (plus Classpath and GTK) will likely become a good alternative for Linux-centric GUI applications and console tools. Additionally, there's a trend of blurring the line between static and JIT compilation. JET, just like TowerJ before it, can mix static and JIT-compiled code to support dynamic loading, and conventional VMs with JIT compiler may cache generated code, a similar technique differing mostly in the deployment format and code management. Microsoft's CLR uses JIT caching for the .NET platform, and in the Java domain, IBM developed this model in [Serrano00] and now JET offers it too.

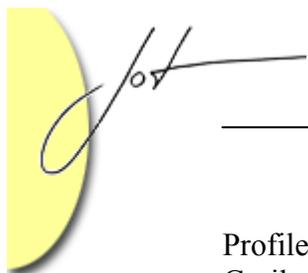
It's worth noting that other languages enjoyed portable code decades before Java, but "portability" usually meant "many CPUs and OS'es", not "many vendors and releases". I wonder how successful could Smalltalk have been if VisualAge, VisualWorks, Dolphin and others could agree on standard bytecode and comprehensive frameworks. Looking at Java's WORA, the network effect created by vendor independence certainly contributed more to its success than support for any non-Wintel architecture. It's enlightening to see that users of more pure OO languages like Smalltalk, and more dynamic languages like LISP or Haskell, never have philosophical issues against static compilers, monolithic deployment or native interfaces. The strategy of WORA was important to bring Java to the point where it is today, but it's likely that from now on the pragmatic deviations will only raise – the recent acceptance (by the community) of IBM's SWT, a non-Pure replacement for AWT and Swing, is another important sign of this tendency.

Magic VMs

The advents of Java2 (JDK1.2.0+), Sun HotSpot and IBM JDK, raised Java to previously undreamed-of performance, and has caught many hackers by surprise. Even these days, it's not very hard to find programmers that regard impossible for a language like Java to approach "native languages" in speed. In the other extreme, some Java advocates claim JVMs already beat C for anything, or will do soon. These are typically enthusiasts writing weak benchmarks and having little knowledge of the challenges still faced by Java. Between these extremes is the current status of Java, and we get there with a second batch of improvements, now bringing some innovations to the realm of VM-based OO systems:



- **State of the Art GC, Threading and I/O.** JVMs improved on enhancements introduced by static compilers. Current JVMs offer mature implementations of everything from basic generational collectors to SMP-hot parallel and concurrent collectors [Flood01]. For threading, fast monitors, multiple threading models (thin, native, N-M mappings) and thread-local heaps. Only for I/O, Sun chose a novel approach with JDK1.3's New I/O libraries, providing asynchronous I/O and faster native interfaces (the static compilers could not afford to create new APIs, so their approach for I/O was making the thread-per-connection cheaper).
- **Profile-driven optimisers.** The hardest problem for JIT compilers is the need to work fast. Profile-driven VMs implement mixed-mode execution [Agesen00], initially running bytecode (or a fast-JITted code) that's instrumented for profiling. Only critical methods ("hot spots") are compiled with full optimisation. The JIT can use expensive optimizations for these because only a fraction of all code is made of hot spots, but their optimisation delivers virtually the same performance as full optimisation. This is a "10%/90%" rule that, much like the Generational Hypothesis is suspicious when first heard but proves valid surprisingly often.
- **Speculative Optimisers.** Java shares with other high-level OOPs the costs of polymorphism and mandatory safety checks. Speculative optimisers find opportunities that depend on an optimistic (but not proven) assumption, e.g. "this polymorphic method is never overridden", and optimise anyway. If the optimiser makes a bad bet, this is trapped either by a precondition inserted in the compiled code, or by the classloader. This forces the program to one of three solutions:
 - Fall back to a "slow version" of the same code (a more traditional technique known as *versioning*, often used for loop optimisations); this requires a check before the optimised code, e.g. ensuring that a receiver is an instance of the predicted class (or a subclass that does not override the invoked method).
 - Perform *patching*, overwriting critical instructions to remove the optimisation. This often requires stubs to make patching easier; for example, a speculative devirtualization could be implemented as a call to a "trampoline" that jumps to one specific method, so the deoptimization is implemented by overwriting this jump with the address of another stub that does dynamic dispatch.
 - Perform *deoptimization*, dropping to interpreted/unoptimized code (at least until new optimisation can be done). This approach produces better code than others when all goes well, because the compiled method is simpler (without any checks, stubs or slow versions); inlining is more efficient.
- **State of the Art, Conventional Optimisations.** The former enhancements allowed JIT compilers to catch up with the harder optimizations. In the case of HotSpot, Sun apparently prioritised the next-generation optimisations over more conventional ones, so many of the easier optimisations were missing! In the first releases, HotSpot excelled in "heavily OO code" due to aggressive optimisation of method calls; in "low level code" scenarios though (integer and FP arithmetic, array manipulation or loops), HotSpot lagged until recently behind even the ancient Symantec JIT included in Sun's own JDK1.1.8.



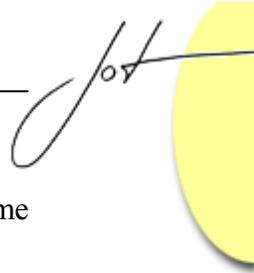
Profile-based and Speculative JITs like HotSpot and IBM JDK are often seen as the Holy Grail of Java performance. [Hölzle94] is the root of dynamic optimisation. We should then analyse the disadvantages. For profiling, the only problem is the need to run the code in “slow mode” until hotspots can be identified; this produces a relatively long warm-up period (loading time is short, but additional wait is required until cruise speed). Footprint is small, so this can be used by J2ME VMs like Sun’s recent CLDC HotSpot.

Deoptimization is very expensive, so the technique depends a lot on profiling and some global analysis. It’s also very difficult to implement: if Thread A is halfway through an $m^{fast}()$ when Thread B does something that invalidates it (typically, classloading), as soon as any side effect can propagate across threads, A’s activation of $m^{fast}()$ cannot proceed safely. The JVM is forced to freeze all application threads; find these activations in their call stacks; and update stack frames so the execution continues in the equivalent point of $m^{slow}()$. This On-Stack Replacement operation is difficult because $m^{slow}()$ and $m^{fast}()$ may have different control and data structure, so the compiler emits metadata describing this mapping at each call site. This metadata occupies memory, so OSR dependant optimisations cannot be employed too generously; one of the space-saving distinctions between the Server and Client editions of HotSpot is that Client lacks OSR.

In addition to deoptimization, HotSpot supports the inverse transition: interpreted activations can be patched to continue in the middle of the generated native code, so when $m^{slow}()$ is optimised, execution jumps immediately to $m^{fast}()$ and the benefit doesn’t need to wait for the next entry to the method. This is only relevant for very big, monolithic methods, present mostly in badly designed code and microbenchmarks.

Aggressive devirtualization and inlining [Detlefs99], as implemented by HotSpot, is both seductive and debatable. Some benchmarks show huge improvement; best-case scenario is when (a) the speed of a critical loop depends on inlining of a polymorphic method, and (b) multiple overrides of this method block more conservative techniques, and (c) the critical code always calls the same override so speculative inlining will work. The sceptics will say that the overhead of deoptimization is bigger than the benefits in the general case, mostly because any well-written code will avoid polymorphism inside critical loops (the first optimisation lesson one learns with OO programming). On the other hand, an increasing rate of poorly-optimised source code is produced by increased time pressure and use of reusable components tuned for flexibility (e.g., if your critical loop uses collections like `java.util.List`, it’s hard to avoid polymorphic calls).

The impressive evolution of Java created a tendency of justifying any performance limitation as: “Sun just didn’t optimise this yet”. The priority given to next-generation tricks in HotSpot, in detriment of conventional optimisations, created a perception that any missing optimisations are only missing due to not being critical to Sun’s strategy (e.g., J2EE apps), but can be added anytime if needed. If they did the hardest stuff like speculative inlining, they can obviously do all simpler optimisations, right? This thinking seems to be validated as only the latest Sun releases show good performance for many low-level benchmarks, because only in the 1.4.x series Sun added critical optimisations



for loop unrolling, array checks elimination, and FP. The optimist expects Java to become fast in everything without major spec changes.

Moore's Law helps Java as more time-consuming optimisations can be supported by JIT compilers every year, without increasing loading time. The IBM JDK implemented most low-level optimisations first and its performance has been stable for some time, so it seems that we're pretty close to the upper limits imposed by Java's current specs. Most further improvements should bring relatively small improvements, or big improvements for a relatively small niche that depend a lot on a very specific peephole optimisation.

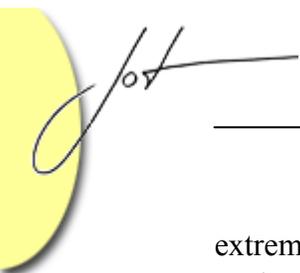
Server versus Client

Advanced compilers always carry a cost in footprint or compilation time; this is true for JITs as well as in traditional ahead-of-time compilers. Advanced runtime elements like GC will also perform some tradeoffs. The first releases of Sun HotSpot delivered much better performance, provided that one could put up with the extra memory usage and loading time. The IBM JDK is another speed champion, but has an even bigger footprint. Add this to the new, heavyweight Swing libraries, and desktop Java virtually died while the Java industry shifted gears towards the server-side. Sun reacted by splitting HotSpot in two editions. The Client VM (default option) omits the most expensive optimisations and favours flat profile code (compiles more methods, less aggressively); the Server VM [Paleczny01] (“-server”) does all optimisations and favors “spiky” profiles. The latest releases of the IBM JDK adopted a similar strategy; a new switch (“-Xquickstart”) tunes the VM for better startup and responsiveness. IBM implemented this just in time for its own first significant Java desktop app, Eclipse.

Lightweight Java

All discussion so far is specific to J2SE VMs, where fast CPUs and large RAM support sophisticated runtimes and speed/space tradeoffs. Life is tougher below J2SE. Sun defines subsets for different domains and capacities; this started with PersonalJava and JavaCard, but Sun later opted for a more structured specification. The result is J2ME, now the main architecture for light Java. J2ME [Grehan02] has a growing number of “configurations” (base VM, language and APIs for a class of hardware), “profiles” (high-level, horizontal APIs like GUI), and “optional packages” (vertical, or less fundamental APIs, like wireless networking). Each combination of configuration + profile defines a stable platform (optional packages are installable on demand), so devices can support a quasi best-fit configuration, avoiding the proliferation of vendor-specific platforms that plague embeddable OSes and is a real problem with installable or mobile applications.

The higher-level specs, like PersonalJava, are generous subsets of J2SE, which means enough functionality that the hardware must be in the high-end of the sub-PC realm, and the same basic techniques can be used. A PersonalJava VM may not afford a state-of-the-art JIT compiler, but it will certainly afford a conventional JIT. In the micro



extreme, JavaCard (a tiny platform for smart cards) preserves little beyond the language syntax: no floating point or 64-bit types; no multithreading, classloading or (optionally) GC – objects should be preallocated, and limited support for the otherwise-heretic manual memory management is provided. Execution is of course interpreted.

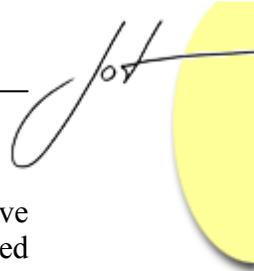
It's safe to state that the reason for the success of these platforms was Sun's pragmatic decision to not enforce WORA beyond physical barriers; the standards cut not only on system libraries, but also in core language and architectural features. There are no revolutions for performance; Sun adopted the usual strategies in embedded systems:

- **Cut as much features as possible.** Some APIs are completely dropped, others are simplified (e.g., convenient method overrides that only provide default values for some parameters are removed, leaving only the base method).
- **Allow fine-grained tuning.** Embedded JVMs allow device designers to fine-tune internal VM settings and options, like memory quotas for JIT compilation.
- **Provide specialised libraries for performance-critical tasks.** For example, J2ME offers multimedia and game-oriented APIs that are cut to fit the needs and resources of small devices. These APIs offer good performance and footprint as their implementation will usually be a thin wrapper over system APIs.
- **Precompute things in development time.** Java bytecode needs to be verified at loading time, but the standard verifier is relatively big and slow for the lesser devices. J2ME apps use a pre-verification tool that augments the bytecode with annotations, so the VM can use a much smaller and faster verifier. (This also benefits J2SE, so JDK1.5 will add the feature to standard JVMs [JSR202].) Similar techniques can help JIT compiler optimisations [Azevedo99], so the harder phase of optimisation can be precomputed too; for instance, the pre-optimiser detects that some array access is always valid (a potentially expensive analysis), then it annotates the bytecode with metadata describing this so the JIT compiler uses this information to generate better code very easily.

The latter optimisation is tricky in Java due to its security architecture: annotations (for security, optimisation or any other purpose) cannot be trusted more than bytecodes are, so the annotations themselves must be validated and this should require less effort than the expensive analysis that they encode. The most innovative aspect of the pre-verification (and possibly, pre-optimisation in future JVMs) lies in the easily-validated annotations.

The first embedded JVMs (like Sun KVM) were interpreted, but JIT compilers followed, and in addition to the design optimisations above, vendors added a host of implementation optimisations:

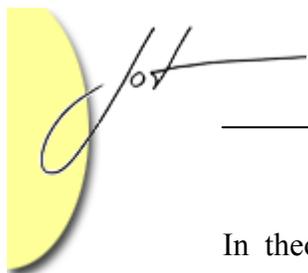
- **Bytecode optimisation.** Post-processor can use static analysis to reduce bytecode size with simple optimisations (like removing unused methods) and replacement of symbolic data with shorter identifiers (e.g., `myClassFieldWithBigName` becomes `m0`). These optimisations are available for J2SE programs, typically implemented by obfuscators, but rarely used as the performance benefit is not worth the trouble with good JITs. For J2ME though, it's a standard practice.
- **Static compilation.** Native code may be deployed to the device.



- **JIT compilers.** In their J2ME incarnations, JIT compilers have to drop expensive optimisations and be very well-behaved concerning resource usage. Profile-based optimisation of hot-spot methods is implemented by some products.
- **Relaxed threading.** Some VMs implement cooperative threading, which saves footprint if the underlying OS does not provide preemptive threads.
- **Cooperative resource management.** The JIT compiler typically shares memory with the application; i.e., temporary compilation data is allocated from the Java heap. Both JIT and GC try to work when the application is idle or using little resources, and ideally both can be preempted by application threads.
- **Careful heap configuration.** Under Java's dynamic classloading, bytecode, native code and JIT metadata may need to be produced (or garbage-collected) at any time. VMs like Sun's CLDC HotSpot implement homogeneous storage: a single heap for application objects, code and VM overheads; this reduces slack and simplifies all memory management (the catch: all these elements, including native code, must be relocatable to allow compaction). Other VMs offer tools to precompute the space required by each area, adding this data to deployment files, but this is less flexible, just like precompiled native code.
- **Mangling the JVM and OS.** Some products (like SavaJe, Symbian and JBed) blur the line between the operating system and JVM. OS architecture and services are tailored to the needs of the JVM, and the JVM implementation benefits from direct, low-level access to system services. These products realise the vision of a "Java OS" in environments where this concept makes sense.

Small-footprint platforms present challenges to Java implementations, but there are good tradeoffs and in many cases, the need for maximum speed is restricted by the application model. In consumer-oriented devices like PDAs and cell phones, applications are usually bound to user input, networking and other OS services, so the application code (written in Java or not) is acceptable with modest performance. This is, not coincidentally, the same rationale that helped Java's early releases – typical J2ME apps are not very different (in size, complexity and reliance on external resources) than the old browser applets.

Another important relief comes with comparison to native applications: these are not very fast either, due to the physical limitations of the hardware. Any optimisation that trades space for speed is restricted to all languages; you can't have it on your J2ME JIT compiler but you can't have it on your embedded C compiler either, as in both cases, the resulting code bloat is not acceptable. Examples of such optimisations are: inlining; intrinsic functions; code specialisation (like in C++ template expansion); loop unrolling, and memory alignment. The net effect is that native applications are not such a high reference score to compare against. Finally, whatever the language, developers of applications for resource-constrained systems are long used to programming tradeoffs – sacrifices of abstraction and clarity are more acceptable. For example, public variables and non-polymorphic (`final`) methods, while not recommended in the J2SE domain and heretic in large-scale J2EE systems, are perfectly acceptable in the J2ME code. When some critical code really needs a space/speed trade-off, programmers typically do it manually, e.g. by unrolling loops in the source, instead of relying on the compiler for this.



In theory, this shows an advantage of profile-driven JITs: these could apply space-expensive optimisations, as the bloat only affects a small number of hotspot methods.

RealTime Java

RealTime is the Last Frontier. [JSR1] (the effort that started the Java Community Process) delivered a ~300pp specification after more than three years of work, and only now the first compliant implementations are surfacing, so it's early to state the success of Java with real-time applications, in particular hard-RT.

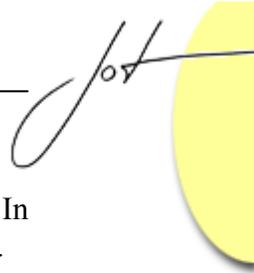
RealTime-ness is more a matter of predictability and control than raw speed. Java's major sources of unpredictability are GC, JIT compilation and concurrency, while the threading and synchronization do not suffice the needs of many RT apps. The RTJ specification fixes these problems with big updates to the JVM architecture.

The object heap can be arranged by the application into a hierarchy of areas with different management policies (standard garbage-collected Heap, plus Immortal and Scoped areas that need no GC). The application creates and "enters" areas, so the new statement will automatically allocate from the current area (this looks like C++ allocators, with additional facilities and rules). RealTime Threads support detailed scheduling options, and the most critical threads may run with higher priority than the GC (the trade-off is no access to garbage-collectable objects). There are other extensions for timers, synchronization, signal handling and asynchronous events.

This specification is orthogonal to Java editions (can be supported by J2SE or J2ME VMs), but RT systems are often embedded systems with tight hardware resources, so the full set of JSR1 features may be a significant footprint for the tiniest platforms. Another challenge is (again) security: JSR1 mandates that references between objects do not violate certain rules, so the use of memory regions cannot lead to memory corruption. For example, an object stored in Scoped memory cannot refer to objects from inner scopes, and Heap or Immortal objects cannot refer to Scoped objects. Without such restrictions, the single-shot destruction of Scoped areas might produce dangling references. All reference assignments must be checked and throw an exception if these rules are violated.

Many solutions developed for embedded Java could be applied to RealTime Java, even if the platform used by the RT application is not low-powered and could afford a full J2SE implementation plus JSR1: even if hardware resources support expensive on-demand optimisations, the application's RT restrictions are easier to satisfy if operations of unbounded cost can be offloaded to development time. For example, escape analysis could remove some runtime checks for reference assignments; this is an expensive optimisation so we could do it in development time and employ verifiable annotations.

The final trick is releasing the safety belt. In the embedded / RT domains (especially the latter), error recovery is often very limited (there's no big difference between a NullPointerException and a reboot; the latter is often better because it's sure to restore system sanity and it's typically very fast in embedded HW). Developers may choose to



spend extra time on testing, and then disable all runtime checks for production code. In any event, it's not worse than using languages like C where safety is not even an option.

2 PERFORMANCE TENSIONS AND CHALLENGES

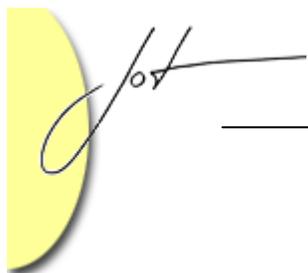
The impressive advance of JVM performance should not be interpreted as a statement that all remaining issues can be fixed with better implementations. There are difficult problems with no perfect solution, and the tradeoffs made by existing high-performance JVMs are evidence of the common sense that Brook's "free lunch" is not available here either. It's useful to assess how far we can go with better implementations of the same design, and which problems need fixes in the specification (language, bytecode or JVM architecture) if we consider critical to have maximum performance for everything.

The economics of language efficiency

Java is a general-purpose platform, so vendors and users wish to apply Java to every kind of software. Fitting very different domains is not impossible, but typically requires a large flexibility. The most successful language in terms of general-purposeness are C and C++, but these languages come with high costs for the developer. Low-level language features and libraries provide "cheap" efficiency for many programs, but the cost is the harder task of creating robust, evolvable, large-scale applications. Hybrid combination of low-level and high-level features, like in C++, cannot approach the benefits of pure high-level languages. Low-level languages can do anything because in truth, it's the programmer who does many difficult things by dealing with unsafe or complex features.

This is not necessarily evil. If all these features make C++ (say) 5X harder than Java, it's still a better choice when a 5X increase in development & maintenance cost is a good trade-off for C++'s advantages. The software market is usually split into three divisions:

- **Corporate.** Most developers write business applications deployed to a single company or a handful at best. Development is a large share of the system's total cost of ownership, because it is not diluted through many deployments. Even modest productivity enhancements translate to significant gains in TCO.
- **Shrink-wrapped.** A successful product sells thousands of copies. Unless the market is very competitive and the margins too tight (e.g., text editors), or the product is a mammoth requiring armies of developers (e.g., RDB servers), chances are that a 5X decrease in development costs will be no good if the resulting code is 30% slower and this makes you lose 5% of your customers.
- **Embedded.** These applications put all previous categories to shame in raw volumes. Unitary costs are critical, but the cost model is similar to shrink-wrapped, typically diluted by large deployment counts. Another significant factor is that code size and complexity is restricted by the small capacity of most platforms: embedded systems rarely suffer from featuritis. Developers often use



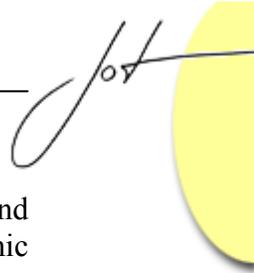
low-level languages, even down to Assembly, as abstraction and reuse are less seductive when the complexity to manage is small. On the other hand, robustness is often a much higher concern (crashing a huge e-commerce server is bad, but crashing a pacemaker or brake controller is simply not an option).

Java is attempting to please all these scenarios, but it's obviously more suited to the first, so it's no surprise that Java's greatest success lies in the "enterprise". J2ME is becoming successful in the embedded scenario, but this happens because the J2ME specification makes the necessary tradeoffs and because this market is changing – many so-called "micro" devices, like last generation cell phones and PDAs, offer more horse-power than last decade's top-of-line PCs. In the higher-end consumer devices, full J2SE support is soon becoming the standard, demanding custom APIs and programming only for device-specific functionality. The lower-end J2ME specs should move to "invisible" devices (your electric razor etc.) which have no flashy GUIs, so consumers cannot be convinced to pay for the additional hertz, bytes, pixels, sound channels etc.

Costs and Limitations

Java is a high-level, object-oriented, garbage-collected language, etc., and there is a cost behind most of these nice characteristics. This cost can often be removed automatically by advanced compilers and runtimes, but most often, the required optimisations impose tradeoffs that don't exist in low-level languages like C. On the other hand, the advanced runtimes often offer bonus advantages that beat most low-level code. Two examples:

- **GC vs. Manual Memory:** Sophisticated Garbage Collectors have bigger memory footprint due to more complex heap organisation and semi-spaces for copying. Dealing with pauses and unpredictability produces additional overhead. On the other hand, compaction allows trivial (stack-like) allocation, and some collectors (esp. the "Train") tend to keep related objects together, improving performance of some applications that depends heavily on cache efficiency. A competent C programmer would optimise the memory layout of data structures manually and achieve similar benefits, but with a big cost in development effort.
- **JIT vs. Static compilers:** JITs start in disadvantage as they must consume fewer resources; on the other hand, JITs benefit from system-specific compilation. Code generation can be tuned to the system's configuration, as the generated code will never be deployed elsewhere. For example, the latest x86 VMs from Sun and IBM will automatically exploit Intel's SIMD instructions (MMX/SSE/SSE2) on machines that support these (Pentium-IV and better will support all). This single factor doubles the score of FP-intensive benchmarks like JGF. Using the same VMs and bytecode, in lesser CPUs, the JIT will use the slower x87 instructions. In comparison, static compiled apps are typically limited to the least common denominator. Compilers often allow optimising for a CPU level (e.g., Pentium-IV) while keeping compatibility with lesser chips (e.g. Pentium); this supports architectural optimisations like instruction scheduling but not those depending on new instructions, and the optimised code may hurt performance in the inferior



chips. Applications can compile the same code for multiple configurations and select the best code dynamically (by calling different procedures or dynamic libraries); in practice, only a handful of applications go through the trouble.

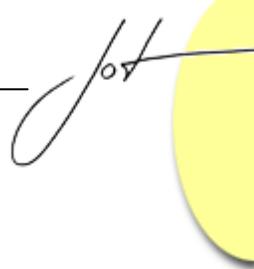
These comparisons are difficult and the most definitive and honest conclusion of most benchmarking studies, like [Doederlein], is the politically correct “test your own code”. Nevertheless, we’ll look at some aspects of Java’s design that so far, imply in undisputed performance disadvantages, meaning that the only possible fixes depend on specification changes. These issues are very important because Sun is very conservative in the advancement of the Java platform: the language [Gosling00] and VM [Lindholm96] specifications are mostly cast in stone and received only minor fixes and enhancements since the earliest releases (the intense growth of Java’s complexity and performance has been mostly dependant on new APIs and better implementations of the same specs).

- **Typesystem.** Java offers a limited set of primitive types compared to most languages of its family: there are no unsigned integrals, no structured types other than classes (structs or enums), no type aliasing (typedefs), and no references for valuetypes. On the plus side, the typesystem and syntax are kept simple. The bad news is that developers are often forced into awkward programming styles that have some cost (look no further than classes emulating enumerations, or the CORBA mapping’s infamous “Holder” classes mapping “out” parameters). JDK 1.5 [JSR176] will add enums and library-based unsigned arithmetic.
- **Lightweight Objects.** This includes several related items: user-defined valuetypes [Bäumer98], headerless objects (structs); by-value containment of object fields and array items; by-value object parameters and returns; `equals()` mapped to ‘==’. The idea is to avoid the one-size-fits-all object layout; this can be done without compromising object pureness (see Eiffel’s “expanded” classes) but not without adding complexity to the language and VM. The benefits are obvious for some niches, e.g. in numerical computing, we really need to encode some user-defined types like Complex as efficiently as possible. But even conventional apps could benefit from better implementation of core libraries (e.g., Java2D & Swing could use LWOs for performance-critical geometry and painting objects). Faster and simpler native interface is another boon to general applications: Java code often cannot produce data structures with the exact layout required by native libraries, so this interface needs wrapper native code that serves only to convert between graphs of Java objects (like a Rectangle containing two references for Point) and low-level C data structures (like the straight layout [x1,y1,x2,y2] produced by headerless, by-value Rectangle and Point structs).
- **Multidimensional Arrays.** Java offers only one-dimension arrays; arrays of arrays emulate multidimensional arrays. This severely impacts the performance of numerical applications that depend on multidimensional arrays, unless some very ugly coding style is adopted, e.g. a `Matrix3x3` class with nine `float` fields could emulate a `float[3][3]` with better performance. IBM developed a library-based solution that would fix this and implement additional FORTRAN-class trickery [JSR83] but this effort doesn’t seem to be moving. See also

[Moreira99] for an account of IBM's research to make Java a top platform for numerical computing; unfortunately, much of these improvements are still far from being accepted into the standard Java language and VM specs.

- **VM and Language Specifications.** The Java Memory Model (JMM) is too stringent in an attempt to prevent concurrency errors, and makes some important code reordering optimisations unsafe for multiprocessors [Pugh99]. The revisions that will be adopted by JDK1.5 should relax excessive constraints (properly synchronized code not affected). In the JLS there are other anti-optimiser: in `for (int i = 0; i <= array.length; ++i) array[i] = 1` which contains an `ArrayBounds` error (at `array.length`), the exception must be thrown exactly where the error happens, preserving any previous side effects. This makes array bounds elimination harder than in other languages: in nontrivial cases, compilers can use versioning, taking the fast track (exception-free) if all indexing can be checked at the loop prologue; otherwise, it goes the slow track (fully checked). This creates bigger methods, which impacts optimizations. [Ishizaki02] seems to present the best solution, but it demands a strong optimising compiler and hardware support, both beyond the specs of small platforms.
- **Insufficient control.** Current JVMs offers many start-up tuning switches, but the offer of programmable, runtime options is too scarce. The `System.gc()` call is comical: the only runtime control you have over GC, and it's most often ignored! Examples of additional control that would still be compatible with Java's safety:
 - *Optimisation hints.* In addition to machine-generated annotations, programmer could add hints that are not easy to discover automatically, like "this object is never shared with other threads". This could be specified by javadoc tags and encoded as bytecode annotations. These optimisations are often implemented as language keywords (like C++'s `inline` or `const`), but this complicates the syntax, typesystem, and linkage... with javadoc tags and annotations, the source is kept simple, and the bytecode is kept simple and compatible with VMs that don't implement some or all of the annotation-driven optimisations.
 - *Memory Allocation.* In the CLR, unsafe code can explicitly pin objects against relocation; in RTJ, objects can be explicitly allocated in different kinds of heap; in JDK1.4's `DirectBuffers`, Java code can allocate raw data from the non-Java heap. These are safe techniques that can boost performance in some important cases. It would be nice if we could further explore these ideas, within the limits of safety. For example, explicitly declaring at allocation time that an object is long-lived or heavily used by native code, could help the memory manager to apply tricks that reduce GC and JNI costs.

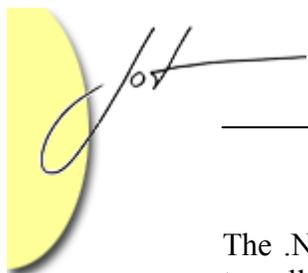
These are probably the top items; many performance-hungry Java developers will easily add a host of new items from their personal soapboxes. The most interesting point is that many critical improvements (like optimisation hints or library-based multidimensional arrays) can be added with small impact. Even if in some cases the result is less elegant code than an ideal solution (like language-based multidimensional arrays), the tradeoff is very good, especially considering Java's culture of considering most syntax sugar evil.



Microsoft .NET

This story would not be complete without a look at Microsoft.NET. Many environments share aspects of Java, but .NET can be directly compared to Java in design and relevance. The .NET VM (Common Language Runtime), and its primary language C#, are often described in terms of Java: “proprietary Java” for some people, “Java done right” for others, the matching is clear. Microsoft benefited from the Java experience, so they could improve in some areas [Gruntz02]. It’s tempting to analyse the differences; where one architecture is superior; if the superiority is due to design or implementation; and if the competitor could (and should) catch up. We could identify some core performance items:

- **Richer typesystem.** Includes a more complete set of primitives, non-OO structured types, and user-defined valuetypes (lightweight objects).
- **Full support for unsafe code.** The CLR implements many features not covered by .NET’s language-neutral Common Type System, like raw pointers, manual memory allocation and unchecked operations. These features are a quick-and-dirty path to maximum performance, subject to abuse by weak programmers but a better choice than JNI when managed code cannot do the task.
- **OS Integration.** The CLR benefits from tight integration to Windows. Many framework classes can be implemented as thin layers over native APIs; one could port the same libraries to other OSes (like the open-source Mono project is doing), but the Windows implementation will usually be simpler and more efficient. The interfaces to non-CLR code and COM benefit too from Windows-centric design.
- **JIT Compilation.** The JIT compiles all methods at first call. Loading time is potentially much worse than JVMs, so the CLR compensates with two strategies: the JIT disables very expensive optimisations, and large applications or libraries can use an install-time code generator (NGEN) to minimise JIT compilation. NGEN calls the same compiler as the JIT, but ahead-of-time, and enabling all optimisations, and stores code in the GAC (Global Assembly Cache). The CLR uses Microsoft’s mature compiler back-end, but it doesn’t seem to add much else.
- **Memory Management.** The CLR offers a three-generation collector that can run in stop-the-world or concurrent mode. The vast number of GC algorithms and tuning options found in JVMs is not available; this may change as .NET evolves, but the CLR makes easy for developers to use manual (unsafe) allocation, and the support for valuetypes result in smaller heap demands even for safe code.
- **Language Support.** Central to .NET is the support for many languages. This feature is not perfect: mainstream OOPLs like C# are better supported than others, but it’s better than the JVM. This creates performance opportunities, especially with managed languages producing executable code that’s trusted to run in tight integration with system services and middlewares. If a single VM supports all code, the IPC barriers can be greatly reduced. On the other hand, J2EE products have made considerable success in implementing everything you need in Java and convincing developers to write everything in Java, so typical Java deployments already enjoy the advantages of safe, monolithic integration.



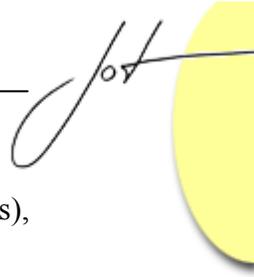
The .NET platform adds some performance-friendly designs (although it's still difficult to collect significant data about their effectiveness versus JVMs for real-world apps). A lot of attention is drawn to the typesystem and unsafe code, which stand out in .NET. Heated discussions are common, because these items trade performance for OO pureness or safety. Some of these features are silently finding their way to Java (JDK1.4's DirectBuffers add restricted unsafe memory access).

System integration helps performance in several ways: lightweight frameworks; privileged OS access; fast native interface; modifying the OS to benefit the VM. This raises more safety issues, as Microsoft tends to move critical features to lower layers of Windows. The competition will be fairer in sub-PC devices, or OSes produced by Java backers, where Java can also be implemented in privileged conditions. System-integrated JVMs incur the same risks as the CLR, but Java applications are less likely to include any unsafe code so absence of critical JVM bugs is enough to keep system sanity.

In some cases, .NET is in advantage as the richer architecture imposes fewer implementation problems, so .NET may compete with a simpler VM. The best example is memory management: GC for all objects is a challenge, but with .NET's typesystem more data can be stack-allocated trivially, and many remaining heap objects carry less header and indirection overheads. If less garbage is produced, the system may use a straightforward GC and heap organisation, reducing speed and space costs. On the other hand, because of Java's additional challenges, JVMs reached a sophistication that pays in the higher end – it's not likely that the CLR's current memory manager can scale to multi-Gb SMP servers as well as Java (papers like [Detlefs02] are an indication of Java's state of the art). It's good to have super advanced VMs, but not good to depend on them.

In the “mixed blessing” category we find the bytecode and execution strategy. MSIL, the .NET bytecode, is dynamic-typed (e.g. a single `add` opcode instead of `iadd` for ints etc., like Java), while good for the typesystem and generics, makes a trivial interpreter less efficient (although this can be fixed with loading-time type propagation and bytecode rewriting). Microsoft uses this as a marketing point (“.NET never interprets any code”) as developers equate interpretation to bad performance. But interpretation has proven useful for Java, enabling low-footprint mixed-mode execution and very compact, yet reasonably fast, VMs for constrained devices (bytecode is typically more compact than native code, even without space-costing optimisations). The .NET Compact Framework competes with J2ME only in the higher-end devices. The GAC creates some management problems (the cache lives inside the Windows folder, growing with each installation), and it doesn't enable closed-world optimisations while impeding aggressive dynamic optimisations.

Only time will tell if .NET beats or is beaten by Java with its different designs. Some apparently obvious improvements (like caching JIT) are very seductive at first sight, but may fail to deliver significant advantage; and less flashy improvements (like a better memory model) could prove to be more important but easier to adopt by Java. There is also the possibility that future optimisations will make most low-level tradeoffs obsolete. For example, [Bacon02] presents a groundbreaking technique to allow Java objects to use



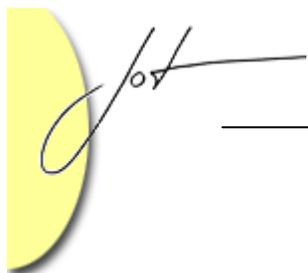
a single-word header (current JVMs, as well as the CLR, need at least two-word headers), which reduces the need for impureness in the object model.

The VM model

Whatever the kind (Java, .NET or others), Virtual Machines (aka “Managed Runtime Environments”) should dominate the next generation of applications. The final question is whether VM-based execution has intrinsic performance advantages or disadvantages over traditional processes, and if future improvements could remove the disadvantages.

It’s useful to remember that the current status quo – processes as we know them in standard operating systems – were not born with computing. Ancient operating systems (not to mention even older systems without an OS), did not offer memory protection, so all programs shared a single address space and other OS resources (like file descriptors), and nothing prevented buggy or ill-behaved programs from stepping over other’s toes. Right now everybody takes for granted the task isolation provided by OSES like Unix or Windows (typically using the term “Virtual Machine”), even though the overhead of this isolation is significant. Only in low-end embedded systems, it’s acceptable to go without memory protection in return to extra savings in speed and space. The software-based VMs like the JVM and CLR propose a higher-level of isolation.

- **Sharing.** Both Java and .NET applications demand more memory than equivalent native applications. This is partially caused by the more complex runtime and heap organisation, but typically, most overhead comes from absence of sharing. Native VM code is typically shared, but it’s harder to share bytecode, VM metadata and JIT-generated native code. [Czajkowski02] implements sharing for Sun HotSpot, a feature that might start surfacing in JDK1.5. Sharing can be combined to caching for better loading time: the CLR’s Global Assembly Cache organises code into relatively coarse-grained packages (“Assemblies”) that are stored in the filesystem. Any sharing solution has a trade-off in compatibility with dynamic optimisations: for instance, if a devirtualization depends on absence of (loaded) overrides of a polymorphic method, the code cannot be shared with other applications that may contain such overrides, so the JIT must either use a different devirtualization strategy or not share this specific compiled method.
- **Isolation.** The next step in sharing is having a single VM process hosting multiple applications (of any kind, not only the neatly-packaged and well-behaved J2EE Enterprise Applications). Java presents difficulties like the single event queue of AWT and system methods with global-VM effect like `System.exit()`; these issues are being fixed with the “Application Isolation” API [JSR121]. Full VM sharing is still difficult due to tuning and robustness concerns. Modern Java VMs offer a rich variety of knobs to select or fine-tune critical components like the JIT and GC, but these settings take effect at the VM scope. The same applies to management and global policies (e.g., enabling of assertions or remote VM monitoring). It’s likely that we will have multiple solutions – multiple JVM processes with shared memory for code; single VMs with JDK1.5’s Isolates, and

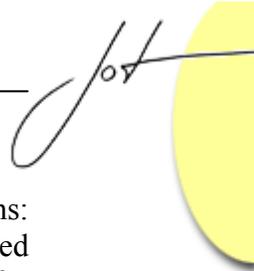


- application server instances – for different scenarios. The Isolates look good for applications executed by the Java PlugIn / WebStart, which typically are lightweight programs and don't require custom VM options.
- **Privileged Execution.** A type-safe language does not need memory protection: if all applications are pure Java, we could run these apps in a single address space without any risk because the constraints in each VM's typesystem and object model would not allow producing rogue pointers to access other VM's memory. One sad effect of .NET's liberal support for unmanaged code and system calls is that a larger number of .NET apps will be "non-pure" and Microsoft would have trouble to exercise their ownership of Windows to realise the academic dream of process isolation through type safety (The SPIN OS, using Modula3, is a proof of concept for this idea [Sirer96]). Application servers that outlaw unmanaged code (like J2EE servers) are less sexy, but a good compromise. Low-end hardware is another ideal scenario: we can already use Java on devices that have no memory protection, which is a robustness advantage against native applications written for the same platforms (and this safety is absolutely mandatory when connectivity allows free installation of software titles, or mobile code).
 - **Platform-specific Functionality.** Binary deployment to multiple platforms is a selling point of most VMs (even Microsoft's), but portable code creates more trouble than needing JITs. Full portability is only possible with portable APIs, so we quickly lose access to system-specific behaviours. Java's challenges with multiplatform APIs are well known, but even the .NET class frameworks don't cover the full Win32 APIs. How does a C# programmer write to a serial port? Answer: non-portable Win32 or COM calls. RS232 is pretty old stuff, but how about the Win32 Fiber Thread API?... Catching up with full OS APIs is neither possible, nor desirable in many cases (e.g. kernel APIs for drivers). The trick is supporting enough functionality that 99% of user-mode applications can be kept pure. The real problem is when an important feature is supported by multiple platforms, but very differently. Look no further than the event model of GUI toolkits. The only portable solution is creating a very heavyweight API that does things its own way, like Swing. Emulating or replacing the native features is possible (and easier with collaboration of the platform owner, like in MacOSX), but the footprint remains an issue as the implementation cannot be a relatively simple mapping from the framework to the platform. Notice that this problem is shared by any multiplatform solution; it's not specific to VM-based systems.

Managed, portable applications will never match native applications in all aspects, but it doesn't matter. Satisfying 90% of all applications is an outstanding success, if we remember that low-level programming environments cannot please everybody either.

Evolution and Compatibility

A critical analysis of the Java language, VM and frameworks will easily find many cases of "design rot", where bad decisions from the past make the current system more limited,

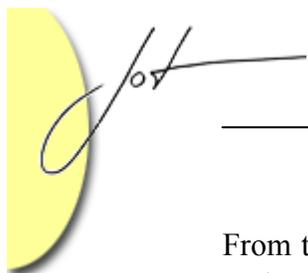


ugly or confusing than necessary. In many cases Sun “patched” Java with new solutions: Java carries obsolete APIs for GUI, collections, security, I/O, dates and more; all replaced by newer designs, but always keeping backwards compatibility, so developers still suffer with the old crust. Let’s look at this issue from the performance perspective.

A common request is “dropping all deprecated APIs”, but this is irrelevant as the obsolete methods and classes amount to a very small fraction of the current platform and the savings in footprint and clutter would be minimal. More significant improvements would require incompatible changes in existing APIs. For example, all methods that return `Vector` need incompatible changes to return Java2’s `List` or `Collection`, as overloaded methods cannot differ by return type only. The problem here is Java’s lack of versioning support: the system should be able to carry multiple versions of the same packages, not allowing these to be mixed (each application can access either the old or the new API, not both). Sun implemented versioning features for application code (in the Java Web Start), but this is very necessary in the Java core. Dropping old APIs may benefit performance by dropping less efficient APIs in places where they are used only for compatibility, and where new APIs allow more efficient application code (e.g. if `StringTokenizer` would take `CharSequence` instead of `String`, callers that have string data in a `StringBuffer` or `CharBuffer` wouldn’t need `toString()`).

The APIs evolve quickly, and newer APIs are generally better designed and more extensible (with strong reliance on interfaces) so they can evolve more smoothly even without versioning. On the other hand, everything else (language, bytecode, VM) moves very slowly, and Sun tends to adopt very conservative solutions. Everybody’s favourite example is inner classes, added to JDK1.1 to fix the event model without true first-class methods / closures (which would need VM and bytecode changes). A pragmatic thinking justifies inner classes: they avoid the performance problems of full closures but support their critical features. The problem is not the trade-off (`C#` delegates’ are bigger) but conservative implementation: by requiring a new class even for a simple one-line event handler or comparator, inner classes have a disproportionate cost in bytecode size and other issues. JDK1.5 will define a dense binary format for faster downloading [JSR200], reducing some of these costs. This is a good example of a new (implementation) patch fixing an old (design) patch. Simultaneously, JDK1.5 will (after very long wait) introduce generic programming, but again, a very conservative model that needs zero changes to existing VMs and code. Considering all the research that went on more advanced models of generic Java, including support for primitive types and Just-In-Time specialisation [Agesen97] (like Microsoft’s design for generic `C#`), the major reason for GJ’s limitations are: not wanting to “fix what’s not broken” (like JITs and core APIs), and not scaring users and licensees with big language and VM changes.

Good initial design, versioning, and some strategy (e.g. releasing new APIs as optional packages until they mature) help to evolve a platform, but at least once in a blue moon, the only solution is a major break with the past. Sun did it at least twice – Java2 (imposing major changes in implementations and APIs) and J2ME (abandoning WORA’s one-size-fits-all fundamentalism with platform-specific API, VM and language specs).



From the performance side, when Java looks bad in any domain compared to competing systems, Java advocates (like me) are quick to point research papers / implementations that solve virtually all Java problems, which means that the problems have solutions and Java implementers have the skills to implement these solutions.

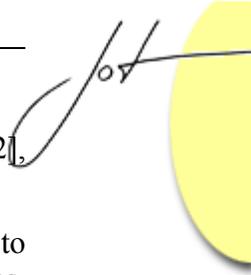
The problem is getting all these improvements included in the Java platform. Sun, its major licensees like IBM, and now all JCP members, face increased tension between improving Java and maintaining a hard-to-earn reputation of stability, utterly important at least in the J2EE space, and a huge competitive advantage. Java is mature enough to replace very mature technology like C/C++, and Java's first direct contender .NET faces an uphill battle for the hearts that value a stable platform where an application can be expected to run well today, and to run at all ten years later.

Virtual Machines have a big advantage for software evolution: a newer platform can emulate its older versions just like it emulates a processor when running portable code. In theory, we could have a Java3 release some time in the future, fixing all known problems of Java with a clean solution, dropping half-backed fixes along with obsolete features. Just like an OS that moves from a 32-bit architecture to 64-bit, the runtime could support legacy code with several tricks – loading a different VM; using versioning so legacy code sees different versions of some APIs; or translating bytecodes at loading time. The latter option is very important: the job is much easier than for native code, because bytecode is (by design) easy to inspect, translate and instrument. Next-generation OSes/CPUs typically demand hardware support to run old programs with acceptable performance (see the debate between Intel's pure IA64 Itanium and AMD's 32-bit-compatible Opteron, that finally pressed Intel to announce better support for 32-bit application support).

3 CONCLUSIONS

High-level programming languages (like OOPs) and execution environments (like VMs) offer many advantages to software developers, but these advantages usually embed trade-offs in size or speed. Decades of research and development in compilers, GCs and related technologies, try to eliminate these tradeoffs. In practice nothing is free, and trade-offs often only mutate – just like in physics, where matter and energy can be transformed into each other but never created or destroyed within a closed system. If we stretch this metaphor, the “closed system” must include compilation and linking.

In traditional environments with ahead-of-time compilation, programmers perceive a “free lunch” in compiler optimisations: the biggest trade-off for better application code is increased resource usage by the compiler. Developers' workstations typically need at least double the memory, disk and CPU than end users', so they don't die of boredom during full builds or profiler sessions. The good news is that once finished, the program doesn't carry heavyweight compilers to end user machines. It is theoretically possible to remove all overhead from high-level constructs with advanced compilers. This benefits



even runtime facilities; for example, some optimisations ([Gayy98], [Mikheev02], [Whaley99]) reduce heap allocations, removing part of the GC overheads at runtime.

VMs keep the compiler inside the deployed “closed system”. This makes harder to really eliminate overheads, from the end user’s perspective. Even in high-end systems, the remaining overheads are sometimes overkill: for example, background daemons like `cvs` have a near-zero memory footprint, and small tools like `grep` have near-zero loading time; in both cases a Java version (with current J2SE VMs) is not competitive.

VMs compensate enabling some unique opportunities, like dynamic optimisation and code management, which more traditional environments can sometimes approach but never emulate completely. We must accept that each model offers some absolute advantages and disadvantages over the other. The “absolute” disadvantages can be reduced with better or more specialised implementations, but never removed completely.

The Java VM has come a long way approaching maximum performance, first with better implementations, and more recently, with more specialised implementations. The very competitive market, with multiple implementers including licensees and clean-room, advanced the state of the art in a pace that no single company could do alone; competition played a major role in the “better implementations” stage but that’s certainly not enough. As evidenced by the evolution of J2ME, specialisation is very important in scenarios where Java’s usual trade-offs are not acceptable. Part of the solution here is making the VM work more like a traditional environment, removing as much hard work as possible from the application’s closed system: either moving work to development time (pre-verification, pre-optimisation, or install-time code generation) or even better, removing work completely (e.g., fixing the overweight memory model so optimised code uses less resources, or adding language features that reduce allocation in the heap so the garbage collector is triggered less often).

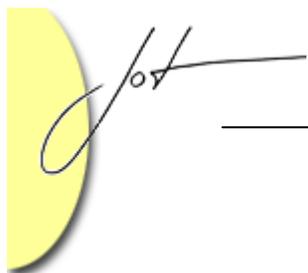
A few issues depend on fixes or enhancements in the current Java specs; now the trade-off is not against bytes or cycles, it’s a matter of politics and strategy: major VM or language changes impact the investment of all Sun licensees, and may force complexity and compatibility issues into developers. The evolution of Java standards is historically much faster in the APIs: licensees redistribute most libraries without change, and developers face a smoother learning curve for features encapsulated by class libraries.

REFERENCES

- [Agesen00] Ole Agesen, David Detlefs: “Mixed-mode Bytecode Execution”, *SMLI TR-2000-87*, June 2000.
- [Agesen97] Ole Agesen, Stephen Freund, John Mitchell: “Adding Type Parameterization to the Java Language”, *OOPSLA ’97*, October 1997.

- [Azevedo99] Anna Azevedo, Alex Nicolau, Joe Hummel: “Java annotation-aware Just-in-Time (AJIT) Compilation System”, *JavaGrande’ 99*, June 1999.
- [Bacon02] David Bacon, Stephen Fink, David Grove: “Space- and Time-Efficient Implementation of the Java Object Model”, *ECOOP ’02*, June 2002.
- [Bäumer98] Dirk Bäumer, Dirk Riehle, Wolf Siberski, Carola Lilienthal, Daniel Megert, Karl-Heinz Sylla, Heinz Züllighoven: “Values in Object Systems”, *Ubilab Technical Report 98.10.1*, 1998.
- [Czajkowski02] Grzegorz Czajkowski, Laurent Dayn`es, Nathaniel Nystrom: “Code Sharing among Virtual Machines”, *ECOOP ’02*, June 2002.
- [Detlefs99] David Detlefs, Ole Agesen: “Inlining of Virtual Methods”, *ECOOP ’99*, June 1999.
- [Detlefs02] David Detlefs, Ross Knippel, William Clinger, Matthias Jacob: “Concurrent Remembered Set Refinement in Generational Garbage Collection”, *USENIX JVM’02*, April 2002.
- [Doederlein] Osvaldo Doederlein, “The Java Performance Report”, JavaLobby site (<http://www.javalobby.org/members/jpr/>).
- [Flood01] Christine Flood, David Detlefs, Nir Shavit, Xiaolan Zhang: “Parallel Garbage Collection for Shared Memory Multiprocessors”, *USENIX JVM’02*, April 2001.
- [Gayy98] David Gayy, Bjarne Steensgaard: “Stack Allocating Objects in Java”, *Microsoft Technical Report*, November 1998.
- [Gosling00] Gosling, J., Joy, B., Steele, G., Bracha, G.: “The Java Language Specification”, 2nd ed., 2000.
- [Grehan02] Rick Grehan: “Deliver Big Functionality on Small Devices”, *Enabling the Wireless Enterprise*, July 25 2002.
- [Gruntz02] Dominik Gruntz: “C# and Java: The Smart Distinctions”, *Journal of Object Technology*, vol. 1, no. 5, November-December 2002.
- [Gu00] W. Gu, N. Burns, M. Collins, W. Wong: “The evolution of a high-performing Java virtual machine”, *IBM Systems Journal*, v39, n°1, 2000.
- [Hölzle94] Urs Hölzle: “Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming”, *PhD Thesis*, August 94.
- [Ishizaki02] Kazuaki Ishizaki, Tatsushi Inagaki, Hideaki Komatsu, Toshio Nakatani: “Eliminating Exception Constraints of Java Programs for IA-64”. *PACT ’02*, September 2002.

- [JSR1] JSR-1: “Real-Time Specification for Java”. *Java Community Process* (<http://www.jcp.org/en/jsr/detail?id=1>).
- [JSR83] JSR-83: “Multiarray Package”. *Java Community Process* (<http://www.jcp.org/en/jsr/detail?id=83>).
- [JSR121] JSR-121: “Application Isolation API Specification”. *Java Community Process* (<http://www.jcp.org/en/jsr/detail?id=121>).
- [JSR176] JSR-176: “J2SE 1.5 (Tiger) Release Contents”. *Java Community Process* (<http://www.jcp.org/en/jsr/detail?id=176>).
- [JSR200] JSR-200: “Network Transfer Format for Java Archives”. *Java Community Process* (<http://www.jcp.org/en/jsr/detail?id=200>).
- [JSR201] JSR-201: “Extending the Java Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import”. *Java Community Process* (<http://www.jcp.org/en/jsr/detail?id=201>).
- [JSR202] JSR-202: “Java Class File Specification Update”. *Java Community Process* (<http://www.jcp.org/en/jsr/detail?id=202>).
- [Lindholm96] T. Lindholm, F. Yellin: “The Java Virtual Machine Specification”, 1996.
- [Mikheev02] Vitaly Mikheev, Stanislav Fedoseev: “Compiler-Cooperative Memory Management in Java”. *IWSP'02*, 2002.
- [Moreira99] Jose Moreira, Samuel P. Midkiff, Manish Gupta: “From Flop to Megaflops: Java for Technical Computing”, *ACM Transactions on Programming Languages Systems*, 1999.
- [Paleczny01] Michael Paleczny, Christopher Vick, Cliff Click: “The Java HotSpot Server Compiler”, *JVM '01*, April 2001.
- [Pugh99] William Pugh: “Fixing the Java Memory Model”, *JavaGrande'99*, June 1999.
- [Serrano00] Mauricio Serrano, Rajesh Bordawekar, Sam Midkiff, Manish Gupta: “Quicksilver: A Quasi-Static Compiler for Java”, *OOPSLA'00*, October 2000.
- [Sirer96] Emin Sirer, Stefan Savage, Przemyslaw Pardyak: “Writing an Operating System with Modula-3”, *WCSS'96*, February 1996.
- [Whaley99] John Whaley, Martin Rinard: “Compositional Pointer and Escape Analysis for Java Programs”, *OOPSLA'99*, 1999.



About the author

Oswaldo Pinali Doederlein is Technology Architect to Visionnaire S/A, specialized in Java application development, object oriented technology and distributed systems. He can be reached at osvaldo@visionnaire.com.br.