Build Your Java Apps COMDISCO www.comdisco.com

# HotSpot: A new breed of virtual machine

## Sun's next-generation dynamic compiler generates bytecodes that *scream*

**Summary**
Sun's HotSpot technology promises to deliver interpreted bytecodes that run faster than a compiled program! Is such a thing possible? Apparently. HotSpot combines a Java virtual machine (JVM) and a compiler in a unique new way that up until now has existed only in universities. The result is a powerful new technology that threatens to blow the doors off language performance as we know it. Find out about the inner workings of Sun's dynamic compiler, and learn which kinds of applications are -- and are not -- more effective with HotSpot than with the fastest JIT. **Plus:** Primers compare HotSpot technology to its predecessors -- standard VMs, compilers, and just-in-time compilers. *(4,000 words)*

**By Eric Armstrong**

### Starting line: Warming up the engines

Lord knows, interpreters are slow. Pre-compiling Java source code into portable bytecodes saves the time needed for translating syntax, but interpreting bytecodes in the Java virtual machine (JVM) is still exceedingly slow compared to the native code produced by a compiler. Thus, Java performance is generally deemed acceptable for small applets but not for any sizable application.

Mail this article to a friend

A just-in-time compiler (JIT) runs many times faster than an interpreter. It makes a drastic, noticeable difference when you're running even an interactive

GOVERNMENT EXHIBIT 1342

application. Although a JIT falls short of compiled-code speeds, it greatly extends Java's applicability. But a reasonably-performing application is still compute-limited. As long as the app mostly interacts with the user or does I/O, it's fine. But if it starts doing a lot of graphic processing or extensive computing, performance rapidly drops to unacceptable levels.

Sun's new HotSpot technology, due to ship this summer (with a developer release due sooner), is a *dynamic* compiler -- a compiler built into the virtual machine that promises to run as fast *or faster than* compiled code in the majority of applications. HotSpot promises to extend Java's applicability into a wide variety of areas, from servers to mainstream desktop applications -- without sacrificing portability. Sun's Java roadmap indicates a HotSpot JVM developer's release was scheduled to be available early in the first quarter of this year; a deployable HotSpot JVM implementation is scheduled to ship this summer.

This article explores the inner workings of Sun's dynamic compiler, HotSpot, and details the kinds of applications in which HotSpot is -- and is not -- more effective than the fastest JIT. (For more information on compilers and interpreters, see the sidebar How compilers and interpreters work. To find out how a JIT improves performance, see the sidebar How a JIT works.)

## They're off!: Burning rubber with dynamic compilation

HotSpot is a dynamic compiler. It combines the best features of a JIT compiler and an interpreter, combining the two in a single package. That two-in-one combination provides a number of important benefits for code performance, as you will shortly see.

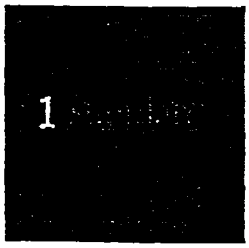
For all of its apparent newness, the concept of dynamic compilation is based on research done over the past 10 years at Stanford University and the University of California, Santa Barbara (UCSB). So the technology is actually fairly stable, but it's HotSpot that has propelled dynamic compilation into the limelight. And, in addition to dynamic compilation, there are two other major time-sinks that HotSpot addresses: garbage collection and synchronization.

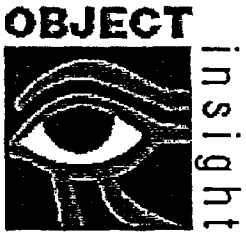
### The cost of computing: garbage collection and synchronization

Unfortunately, in the majority of applications, optimizing Java code is not all there is to improving performance. Some of the same features that help Java developers increase productivity and write more versatile, bug-free applications also consume mass quantities of runtime compute cycles. The major cycle-stealing features are garbage collection and threads. As the pie chart below shows, together these two features take up about 40 percent of the average application's run time.

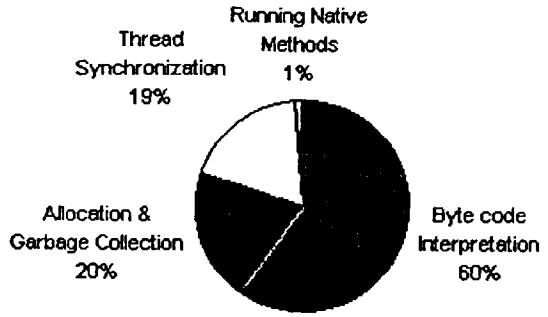
Java's built-in garbage collection eases the programming burden tremendously. It all but eliminates those "memory leaks" that used to freeze up a system when a program kept allocating memory without releasing it, until there was no longer any room for the system to do anything. But garbage collection takes time. Optimizing the application's bytecodes with a JIT has no effect at all on the virtual machine's garbage collector. It still has to spend the time to do its thing. In fact, some applications allocate and release memory so frequently that the time spent in garbage collection dwarfs the time spent in the application itself. In such cases, optimized application code is of little value.

Advertisements






Advertising Information




**This pie chart shows average garbage collection, synchronization, and run times for typical applications. It is based on a slide from Sun Microsystems, but with numbers rounded to make the trends stand out. Note that some apps can spend virtually *all* of their time in garbage collection and/or synchronization, while others may spend virtually no time on such tasks. These are average figures.**

The other major drain on system resources is thread synchronization. Java's threads (concurrently executing code segments) provide a lot of flexibility. For example, threads allow buffering input to improve performance, and let developers write a server app that services multiple clients simultaneously. But synchronization takes time, a surprising amount of time -- in fact, nearly 20 percent of a normal application's CPU time. Furthermore, that 20 percent is time that can't be optimized by compiling the application's code.

In addition to dynamic compilation, HotSpot uses an improved method of garbage collection called *generational* garbage collection. This technique has seen a lot of academic work, but is now finding a major application in the Java language. The idea behind generational garbage collection is that memory tends to be allocated, and then discarded, in chronological groups.

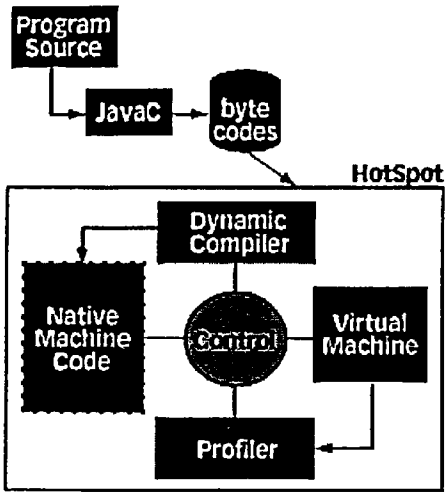
Dividing the code into different "generations" of allocated storage has two important benefits. This technique allows the garbage collector to work incrementally. Instead of traversing all of memory at one time -- which can bring the application to a screeching halt -- the generational collector can traverse smaller groups of allocations. So it runs frequently, doing a little a time, and therefore has a much less noticeable effect on performance. This is especially important for interactive applications, where users prefer predictable performance to erratic behavior. In addition, the process streamlines the garbage collection process so that it occurs more quickly.

HotSpot also improves the synchronization process. In many cases, what used to be multiple lines of synchronization code has been reduced to a single instruction. So using multiple threads in an application has virtually no penalty.

Together, improved garbage collection and synchronization can make a drastic difference in an application's runtime -- even without optimizing the application code. But this article is about how dynamic compilation works, so lets get on with that.

**How the dynamic compiler works**

HotSpot includes both a dynamic compiler and a virtual machine to interpret bytecodes, as shown in the following figure.




**The dynamic compiler includes both a compiler and an interpreter. The bytecodes produced by the Java compiler (JavaC) are first interpreted in the virtual machine. As they run, the profiler keeps track of performance information, and selects a method for compilation. Compiled methods are stored in a cache of native machine code. When a method is invoked, the native machine-code version is used, if it exists. Otherwise, the bytecodes are reinterpreted. The "control" function shown in the diagram is likely to be an indirect jump through a memory location that points to either the native code or the interpreter. (Details have not been disclosed, but such an implementation is likely.)**

When bytecodes are first loaded, they are run through the interpreter. The profiler keeps a record of runtimes for each method. When a method is found to be taking a lot of time, HotSpot compiles and optimizes it. Every future call to that method uses the native machine instructions produced by the compiler.

As with a JIT, the results of the compilation are not kept between runs. Because the bytecodes are more compact, this saves loading time as well as storage space. It also retains portability, and allows the optimization to reflect the way the program is currently being used. Currently, no attempt is made to save information that might help future runs become efficient more quickly. Although this approach may be a future possibility, Sun has no such plans at this time.

**Profiling and compiling heuristics**
The most fascinating aspect of the whole system (to anyone interested in the subject of machine reasoning, at least) is how the profiler decides which methods to optimize. At present, the system uses a very minimal heuristic that does not involve any sort of artificial intelligence. Although the exact details are proprietary, it's reasonable to suppose that the system predicts the time it will take to optimize a given method. Once the time spent in that method reaches a predefined threshold set at, say, 70 percent of the predicted optimization time, it becomes a candidate for optimization. (The exact threshold is undoubtedly the result of a considerable amount of research.)

Fortunately, say the folks at Sun, it's unnecessary for such a system to be perfect. As long as it's right most of the time, it can make an astonishing difference in how a program runs.

And, who knows? There is always the possibility that some day more advanced artificial intelligence heuristics can be used. There are no firm plans to pursue this possibility -- the company expects to have its hands full refining its current implementation over the next year or two -- but it's an interesting possibility to consider.

**The advantages of dynamic compilation**
The beauty of integrating a compiler with an interpreter is that it can do things a normal static compiler simply can't do. In particular, it can perform *optimistic compilation* and take advantage of *runtime information* to do robust *inlining*.

***Optimistic compilation***
Because the interpreter is always available to run the bytecodes, HotSpot's dynamic compiler can assume that exceptions and other hard-to-optimize cases don't happen. If they do, HotSpot can revert to interpreting the bytecodes. For a static compiler, optimizing in a way that handles all the unusual cases is a very difficult and time-consuming process. By ignoring those cases, HotSpot can rapidly produce highly optimized code for the code that is most likely to run. That produces a lot of bang for the buck -- significant performance improvement for a small investment in optimization time.

***Runtime information***
The second major advantage of dynamic compilation is the ability to take into account information that is known only at runtime. Again, the details are proprietary. But it's not hard to imagine, for example, that if a method is invoked in a loop that has an upper index value of 10,000, it's an immediate candidate for optimization. If, on the other hand, the upper loop index value is 1, the optimizer will know to ignore that method and let it be interpreted. Because a static compiler has no way of knowing for sure what the values might be, it can't make such judgements.

***Inlining***
One of the important optimizations HotSpot performs is *inlining* frequently-called methods. That means that instead of being invoked with the overhead of a method call, the method's code is copied into the calling routine and executed as though it had been coded there. The smaller the method, the more sense inlining makes. A one-line method might spend more than twice as much time entering and exiting the routine as it does executing.

Inlining provides a tremendous boost in performance, but it has a size penalty. If every method in a program were copied to every place it was called from, the program would balloon to five or ten times its original size. But HotSpot optimizes only the *critical* sections of a program. The 80/20 rule says that 80 percent of a program's runtime is spent in 20 percent of the code. By inlining as much as possible of that 20 percent, HotSpot can achieve a significant boost in performance with an acceptable size penalty. Here, runtime information makes a big difference, because HotSpot knows where to find that critical 20 percent.

As with all good things, of course, there are tradeoffs. If a method is very large, the time spent entering and exiting it is only a small fraction of its execution time. In such a case, the space required to duplicate the method may not compare favorably to the time saved by inlining. So HotSpot undoubtedly has an upper limit on how large an inlined method can be. (The exact details, however, are proprietary.)

Finally, because HotSpot does optimistic compilation, it can inline only the normally-executing methods, instead of every possible method. So it can ignore difficult cases that rarely occur instead of working hard to resolve them. That way, you get the maximum performance payoff for the minimum optimization effort.

**When dynamic compilation succeeds -- and when it doesn't**
In the large majority of cases, HotSpot provides an extreme performance advantage over JIT technology, not to mention normal JVM interpretation. But, as with any performance-enhancing technology, the HotSpot dynamic compiler fails to achieve optimum performance in distinct cases. For example, the profiler might decide to optimize a piece of code just as it finished executing for the last time. In an extreme case, the optimizer could conceivably run around optimizing every method just as it finished executing for the last time, which obviously wouldn't work out at all. Fortunately, such cases are rare.

HotSpot would perform badly in a 100 percent compute-bound application with little or no garbage collection, no thread synchronization, and very small amounts of initialization code. An example of such an application might be one that multiplied the first 1,000 prime numbers together and displayed the result. For such a program, a good JIT would provide superior performance because the JIT doesn't wait before optimizing -- it optimizes everything preemptively.

And, finally, HotSpot technology falls short in an artificial benchmark like the CaffeineMark. Because such benchmarks do not emulate real-world programs, it's possible to write a compiler that detects the nonsense cases (such as do-nothing loops) and optimizes them away. Such a compiler can produce

extremely fast benchmark times that are not reflected in production programs. A compiler that doesn't optimize these weird cases may look bad by comparison, even if it's better in the real world. (Sun recently demonstrated this point by designing a compiler that specifically "optimized" these benchmarks down to a near-nothing program that ran in basically zero time. Although their attempt backfired in the public opinion polls, it made a strong statement that such benchmarks may not make very reliable measures of performance.)

As with any performance-enhancing technology, the bottom-line question is: How well does it work for *your* application? In the vast majority of normal applications, where garbage collection and thread synchronization occur and where real code repeats, the HotSpot approach provides a dramatic performance boost.

**How HotSpot compares to native code**
In this article, we've been comparing dynamic compilation to other ways of processing Java code. But we've always compared one form of Java to another. The skeptics in the audience are crying out, "Yeah, but how well does it do compared to *my* favorite compiler?" The answer is that it compares very favorably today, and it has an even more promising future. Sun has its sights set on achieving performance levels that rival that of an optimized C program. Given time, Sun is confident it can come close to that target.

At the moment, good JIT compilers meet or exceed the performance of static C++ compilers for object-oriented programs, as described in Performance tests show Java as fast as C++. And HotSpot is faster than a JIT. That's still not up to the performance of a compiled C program, but it's getting there.

> **Note:** C++ can be used like Java to create an object-oriented program, or it can be used like C to create very fast, non-object-oriented code. Today, Java compares very favorably to C++ for an object-oriented program. But C++ can be optimized for performance with C-style coding in ways that Java doesn't allow. This can make the code harder to maintain, but very fast.

## Finish line? The race has just begun!

HotSpot is the first working version of a dynamic compiler for the Java language. Correct operation has been verified. The goal now is progressive refinement and enhancement, until absolutely optimal performance is achieved. Sun believes that in the next several years it may even be able to approach the speeds of an optimized C program!

The outlook is exciting. The prospect of platform-independent code that outperforms even native optimizers holds out the possibility that, over the next few years, the Java language will become a realistic candidate for virtually every type of application. It's not just for applets anymore! ■

Also this month in JavaWorld Go

**Resources**

- For further information on HotSpot and architecting your applications for maximum performance, see http://java.sun.com/javaone/sessions/slides/TT06/title.htm
- See the results of the *JavaWorld* poll "What do you think of Sun's possible decision to sell its HotSpot high-speed compiler separately (instead of including it in the next version of the JDK)?" http://nigeria.wpi.com/cgi-bin/gwpoll/gwpoll/past.html
- For further information on generational garbage collection, see:

  - http://www.dcs.gla.ac.uk/~sansom/1993_gen-gc-for-haskell.html
  - http://csgrad.cs.vt.edu/~groener/courses/briefing.shtml
- For further information on thread synchronization, see:
  - http://www.javaworld.com/jw-07-1997/jw-07-hood.html
  - http://www.math.utah.edu/java/native/implementing/threadsync.html
  - http://www.javasoft.com/docs/white/langenv/Threaded.doc2.html
- To find how JITs stack up against C++, see Carmine Mangione's article, *Performance tests show Java as fast as C++*.
  http://www.javaworld.com/jw-02-1998/jw-02-jperf.html
- For further information on the Symantec JIT that is now included with the Windows version of the Sun JDK, see
  http://www.symantec.com/jit/jit_readme.html
- For further information on Sun's JIT for Solaris, see
  http://www.sun.com/solaris/jit/
- For Sun's Java roadmap, including the release schedule for HotSpot, see
  http://www.sun.com/smi/Press/sunflash/9712/sunflash.971210.5.html
- For A *PC Week* news article discussing HotSpot and Sun's plans to sell it as a commercial product, see
  http://www8.zdnet.com/pcweek/news/1215/15java.html
- For the transcript of a JavaOne '97 conference presentation titled "High Performance Java: Programming Tips, Techniques and Choices" by Peter Kessler and David Griswold, see
  http://sunsite.compapp.dcu.ie/IJUG/javaone/transcripts/perform.html

---

**About the author**
Eric Armstrong has been programming and writing professionally since before there were personal computers. He has firsthand knowledge of HotSpot based in part on his current work as a contractor at JavaSoft. His production experience includes artificial intelligence (AI) programs, system libraries, real-time programs, and business applications in a variety of languages. He is a regular contributor to *JavaWorld,* and the author of *The JBuilder Bible*, to be published by IDG Books this spring. Reach Eric at eric.armstrong@javaworld.com.



©1996-98 Web Publishing Inc./IDG—Click for Trademarks & Notices

If you have problems with this magazine, contact webmaster@javaworld.com
URL: http://www.javaworld.com/javaworld/jw-03-1998/jw-03-hotspot.html
Last modified: Wednesday, September 02, 1998

---

# How compilers and interpreters work

This section gives a historical overview of language compilers, interpreters, and bytecode interpreters.

## Machine code and assembly language

At bottom, computers process only a very limited instruction set known as *machine code*. Machine-code instructions are basically a series of numbers that have a particular meaning to the computer. Once upon a time, programs were written into the computer's memory by flipping toggles on the front panel to produce the appropriate series of numbers (instructions) for the computer to follow.

A major leap forward occurred when the computer was used to translate *assembly language*. An assembly language is exactly like machine code, except that it uses alphabetic mnemonics like "LDA" to load a value into the computer's internal accumulator, instead of a numeric instruction. There's still a one-to-one correspondence between assembly language and machine-code instructions, but now the

computer does the work of *assembling* the program into machine code.

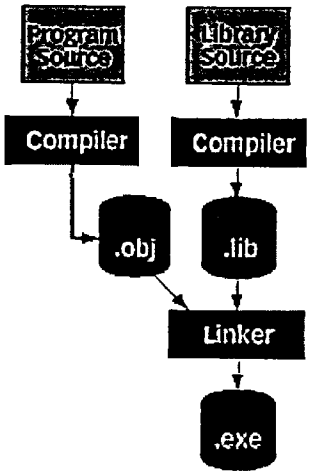
The figure below shows the result of a hypothetical translation where the machine code for LDA is 1378, accumulator (or register) "A" is number 00, and the variable "N" is at memory address 1000. The example is only loosely related to anything that might exist in reality, but it shows how assembly language instructions are related to machine code.

**Assembly language statements are translated one-for-one into numeric machine code instructions.**

## How a static compiler works

The next major leap occurred when higher-level languages like C and Pascal were invented. In these languages, one instruction can translate into many lines of machine code. The translation process for these languages is known as *compilation*. The goal is the same: to produce machine code the computer can understand out of a language that is readable enough for humans to work with. But now humans are working with a higher-level language, and machines are doing more work to accomplish the translation.

The results of multiple compilations are then *linked* together to produce an executable program. At a minimum, you generally link your program with a previously compiled system library to produce an executable program, as shown in Figure N-N. (Libraries also can be linked when the program runs. Such libraries are known as dynamically linked libraries, or DLLs.)




**The compiler translates higher-level statements into multiple instructions. The linker ties together the results of multiple compilations, including libraries of previously compiled routines, to create an executable program.**

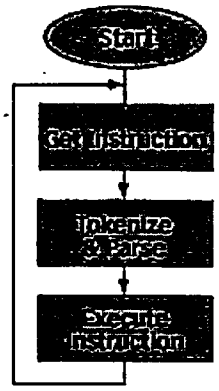
There are basically four steps in the compilation process: *tokenizing*, *parsing*, *code generation*, and *optimization*. In the *tokenizing* step, individual items called *tokens* are recognized. For example, in the statement: `if (myVal == 123) {`, the individual tokens are `if`, `(`, `myVal`, `==`, `123`, `)`, and `{`.

In the *parsing* step, the individual tokens are combined into meaningful statements. If the statement doesn't make any sense, the compiler generates an error message. But if the compiler recognizes the statement, it can go on to the next step, *code generation*.

In the code-generation step, the appropriate machine code gets produced. (**Note:** Frequently, assembly language is produced, which is then assembled to produce the machine code. But it amounts to the same thing.) Once all the instructions have been generated, the compiler can begin the *optimization* process. For example, one instruction might store a value in an internal register at a memory location. The next instruction might read that very same value for use in another computation. The optimizer can spot the inefficiency and simply reuse the value in the internal register, eliminating the extra read.

## How a language interpreter works

Although it produces very efficient programs, a compiler takes a while. And the better it optimizes, the longer it takes. This is fine for production programs, but it was a real problem for people who were learning to program. They might have had to compile 30 or 40 times to make a small program run. With long compile times, learning tended to be more frustrating than fun.



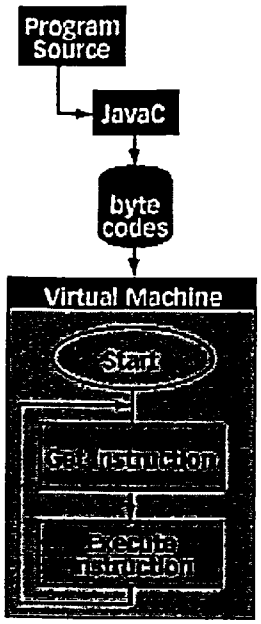

**The language interpreter is basically a big loop. It gets an instruction, parses it, and executes it. Then it gets the next instruction, and continues that process until it runs out of instructions or until it processes an instruction telling it to stop.**

Languages like Basic solved the program by getting the computer to act as an *interpreter*. Instead of compiling a complete program into machine code and then getting out of the way, the interpreter sits there in a loop, looking for instructions. Each time it gets a new instruction, it tokenizes the instruction, parses it, and then executes the machine code associated with that instruction right on the spot.

Command shells like DOS and CSH are interpreters that recognize a limited number of commands. The interpreter's loop accepts a command, processes it, and then prompts you for another command. The typical Basic language command interpreter operates the same way.

One source of inefficiency you can see right off is that there is no way to optimize the resulting code. Since each instruction is executed as it is seen, you might very well store a value in one instruction, only to retrieve it in the next. In addition, such a pure interpreter tokenizes and parses every line that it sees. That isn't too bad when you're processing a line at a time (like a command shell) or a very small program, but it slows things down tremendously when you are processing a very large program.

## How a byte-code interpreter works




**A byte-code interpreter works on instructions that have been translated in a previous step. It, too, runs in a loop, but it doesn't have to do any of the tokenizing or parsing that a language interpreter performs.**

Twenty years ago in San Diego, the University of California created UCSD Pascal, which popularized the system that Java uses today. This system makes interpretation more efficient by using a *byte-code interpreter*. The following figure shows how a byte-code interpreter works.

With this system, a program is first translated into a series of bytecodes. The program that does the translation for Java is `javac` -- the Java "compiler". The bytecodes are numeric machine codes for a computer that doesn't really exist -- a *virtual machine*. An interpreter that understands those bytecodes then runs on the native machine. It runs in a loop, executing each bytecode it sees on the native computer. The Java virtual machine (JVM) is an example of such an interpreter.

Using a byte-code interpreter means that the tokenizing and parsing occur in the translation stage. So a byte-code interpreter is faster than a language interpreter. The resulting programs are more compact than a fully compiled program. Perhaps more importantly, byte-code interpreters are portable. They can be run on any computer that has a virtual machine capable of properly interpreting the translated byte codes.

However, there is still a translation step as the bytecodes are interpreted. The interpreter identifies the equivalent series of machine instructions, and then executes them. The overhead involved in any single translation is trivial, but overhead accumulates for every instruction that executes. In a large program, the overhead becomes significant compared to simply executing a series of fully-compiled instructions. In addition, there is once again very little that can be done in the way of optimization. So the good news is that you have compact, machine-independent code. The bad news is, it's slow.

Back to story

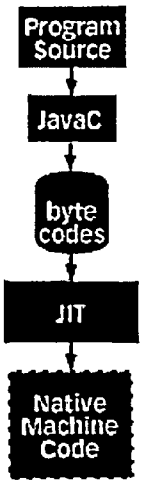
---

# How a just-in-time (JIT) compiler works

A just-in-time compiler, or JIT, makes major improvements in the runtime performance of bytecodes by compiling them into native machine code before executing them. This section gives a brief overview of how a JIT works. For more information on compilers and interpreters, see How compilers and interpreters work.

## What makes a JIT tick

A JIT translates a series of bytecodes into machine instructions the first time it sees them, and then executes the machine instructions instead of interpreting the bytecodes. This process is shown in the diagram below.




**A just-in-time compiler (JIT) translates bytecodes into machine instructions as they are read in, and performs a degree of optimization. The resulting program is then run. Parts of the program that don't execute aren't compiled, so a JIT doesn't waste time optimizing code that never runs.**

The machine instructions aren't saved anywhere except in memory. (That's why the diagram shows them in a dashed box.) The next time the program is run, the bytecodes are translated into machine code once again. The result is that the bytecodes are still portable, and they typically run much faster than they would in a normal interpreter -- as much as 50 times faster, in fact. Typically, that's fast enough for most I/O-intensive and interactive applications.

## When a just-in-time compiler succeeds -- and when it doesn't

Because a JIT translates a series of instructions, rather than executing one instruction at a time, it can do some optimization. For example, a JIT can keep from reloading a value into an internal register when the value is already present. For that reason, and because it is executing machine code, a JIT does really well with computational programs that execute the same series of instructions many times.

On the other hand, a JIT does not produce much of an advantage for I/O-intensive programs or code that does a lot of garbage collection -- the application code takes up such a small amount of time that any optimizations are virtually unnoticeable. Initialization code also poses a problem for a JIT. Because a JIT optimizes everything it sees, a lot of time may be spent uselessly optimizing initialization code and other "one-time" methods. Finally, a JIT can't afford to be too good at optimizing -- because some of its time inevitably must be wasted, the "better" the JIT, the more time it will spend on useless optimization.

Back to story