

A White Paper on Java and Performance:

Is it possible to use Java to design high-performance software applications?

December 2004

Executive Summary

Java has a reputation for slow performance. Performance is viewed as the Achilles' heel of an otherwise fine language.

In fact, there's no truth to the assertion that Java is slow. Java has evolved to be a great solution for developing ultra-fast software applications. Java is now being used as a general-purpose language for implementing all types of programming challenges, including performance-intensive applications.

This white paper discusses Dieselpoint's experience in developing a high-performance search engine entirely in Java. It gives a little history, presents some benchmarks, and explains some of the theory on why Java is now faster. *It shows that it's possible to write Java code that outperforms code written in other programming languages, such as C++, if you know what you are doing.*

For software developers who are writing high-performance Java applications, this article presents coding methods and design tips to improve performance. Many of these techniques are garnered from our many years of Java coding experience, developing fast search technology that performs, in most cases, faster than competitor's software written in C++.

What is Dieselpoint Search?

Dieselpoint Search is the leading Search and Navigation™ solution for documents, databases, and XML.

Dieselpoint has developed industry-leading advances in search technology, combining advanced full-text and linguistic techniques with high-performance data navigation to yield intuitive user interfaces not possible with other approaches. Typical applications include large-scale product catalogs, website search, document repositories, intranet search, and OEM software.

Choosing Java

Dieselpoint chose Java for primarily for its cross-platform capabilities. Enterprise applications need to be able to run on a wide variety of hardware, from PCs to servers to mainframes. Dieselpoint Search is currently in production use on Intel, Sun, and IBM hardware running Windows, Linux, Solaris, IBM iSeries and AS400 OSes, and Mac OS X. Customers have been able to implement Dieselpoint Search on legacy hardware, and then upgrade to newer hardware without losing their software investment.

Another happy consequence of choosing Java was that Dieselpoint was able to develop an engine which is fully standards-compliant and can rely on a large body of third-party tools and components.

In the early days, though, there was a downside: Java's reputation for slow execution. And back in 1999 when Dieselpoint started development, the problem was real.

The most visible manifestation of the slow performance was on the client side. Applets were far less responsive than native Windows applications. Interfaces drew slowly, and apps hesitated before they responded to mouse clicks.

On the server side, there was also a problem, because Java was a quasi-interpreted language. Java translated source code to byte code, which was in turn interpreted at run time. The arrangement led to performance considerable slower than native compiled code.

Speed Increases

Recent versions of Java, however, have overcome the performance problem, and now run standard benchmarks as fast as or faster than C++. A number of independent studies have been performed (some of which are cited under References below), and the results are generally consistent: Java version 1.4.2 equals or beats C++ on the majority of benchmarks.

Today, Java is no longer interpreted; it's a fully compiled language like C++. A Just-In-Time (JIT) compiler loads Java classes when they're first referenced and compiles them to native code. This native code is no less efficient than C++.

And in some cases, it is more efficient C++, because Java can take advantage of dynamic recompilation.

Java loads code on an as-needed basis. This means that the JIT compiler can take advantage of knowledge about the runtime environment that could not be known if the code were compiled in advance. The compiler can know, for example, that a particular section of code is called frequently and can be inlined. C++, on the other hand, is statically compiled, and can't dynamically adjust at runtime.

The following table shows an excerpt from a Dieselpoint Search log file. The system was indexing a large number of small records from a SQL database. The second column shows the number of milliseconds elapsed between log entries:

8:02:04 PM	2404 Items indexed: 1000
8:02:05 PM	200 Items indexed: 2000
8:02:05 PM	240 Items indexed: 3000
8:02:05 PM	301 Items indexed: 4000
8:02:05 PM	170 Items indexed: 5000
8:02:05 PM	190 Items indexed: 6000
8:02:06 PM	200 Items indexed: 7000
8:02:06 PM	181 Items indexed: 8000

8:02:06 PM	90 Items indexed: 9000
8:02:06 PM	100 Items indexed: 10000
8:02:06 PM	200 Items indexed: 11000
8:02:06 PM	80 Items indexed: 12000
8:02:06 PM	71 Items indexed: 13000
8:02:06 PM	60 Items indexed: 14000
8:02:07 PM	80 Items indexed: 15000
8:02:07 PM	60 Items indexed: 16000
8:02:07 PM	70 Items indexed: 17000
8:02:07 PM	50 Items indexed: 18000
8:02:07 PM	80 Items indexed: 19000
8:02:07 PM	60 Items indexed: 20000
8:02:07 PM	60 Items indexed: 21000

The time to index 1,000 records drops from 200-300 milliseconds down to only 60 milliseconds over the course of several thousand iterations. Each iteration gives the compiler a few more statistics about how the code is being called and what kind of optimizations are acceptable, and the compiler optimizes the code as it goes along. This result repeats itself over multiple runs, indicating the effect is not due to disk caching.

Another reason that Java has caught up to C++ is adaptive garbage collection. Java allocates memory dynamically and only cleans up discarded memory on a periodic basis. Modern generational garbage collectors can optimize this process using knowledge about current memory usage, particularly knowledge about whether an object is short- or long-lived. Most C++ compilers, by contrast, use a static algorithm which allocates objects on the heap and usually tries to fit new objects into the spaces left by objects that have been deleted. This process is slow. A good discussion can be found in the “A Bit About Garbage Collection” reference below.

Optimal garbage collection is particularly important for server-side applications where large numbers of users can generate a lot of garbage quickly.

Designing a Fast Java-Based Search Engine

Designing a search engine is a tricky process. Even when the core language is fast, there are still quite a few cases where sub-optimal coding techniques can reduce performance. Search engines are very sensitive to performance issues, more so than many other applications. In particular, search engines must:

- Process large volumes of data, much of which must reside on disk because of memory constraints;
- Operate in a highly multi-threaded environment, with locking and resource contention issues;
- Manage large memory buffers, which can lead to garbage collection issues; and
- Deal with string manipulation

The standard programming language benchmarks don't generally address these issues. These benchmarks are more focused on the execution speed of very small programs. They measure integer performance, looping, floating point calculations, method calls, and the like. While not unimportant, these functions have less impact on the speed of a search engine than the search engine's ability to move and process large blocks of memory and to optimize disk access.

Ultimately, good coding is the main driver of good performance. Dieselpoint Search's architecture is the result four years of research and development and numerous optimization cycles. For us, the architectural features that yielded the largest performance benefits were the choices we made for algorithms and internal data structures. The main trick was to minimize the amount of work that had to be done to execute a query.

There are a number of generic strategies, though, that can be applied across similar server-side applications. If you are a Java programmer, building any speed-intensive application, we outlined some important tricks that we've used. You can incorporate these techniques into your own solutions to enhance performance.

Tips and Tricks for Writing Fast Java Applications

The first and most important strategy is to **optimize disk access**. In our case, if the collection being searched is small, then it can fit entirely in memory and the problem is minimized. But this is not the case for a great many typical search engine applications.

Accessing data on disk is several orders of magnitude slower than accessing it in memory: a single seek can take an average of 4.5 milliseconds for server drives and 9 milliseconds for desktop drives. A query that requires access to data that is widely dispersed across the index can require a number of disk seeks, and is a sure performance killer. The problem is magnified in a multi-user environment because seeks across a single disk must happen serially, not in parallel.

The solution to the problem is to cache frequently-accessed data in memory and, as much as possible, see to it that different data elements that are likely to be accessed during the same query are physically located near one another. By encouraging locality of reference within a disk file the odds are increased that a single seek can pick up multiple elements in one sweep.

The second strategy is to **use compression** in the data files. Compression can slow indexing a small amount, but has less effect during a search because decompression can be very fast. By making data files smaller, the system increases the percentage of the files that are cached in memory, and can reduce the number of seeks. The best compression methods are ones that are suited to the particular data structure at hand; some compression methods will allow you to process parts of queries without any decompression at all.

The third strategy is to **avoid synchronization**. Multi-user apps require access to shared resources, and contention for those resources by different threads has to be managed. One quick way to slow performance to a crawl, however, is to synchronize access to common

objects. Multiple threads that must each lock and release a common resource will run sequentially, not in parallel. There are a number of classes in Java that use internal synchronization unnecessarily, and the use of these classes should be avoided. In a number of cases we have written our own simple unsynchronized replacements for common Java utility classes.

The final strategy is to **reuse memory buffers**. Java garbage collection is good, but it's not magic. Allocating and discarding memory is still costly. The cost of allocating large numbers of small objects is not as great as it once was, but it should still be avoided. A good heap profiler can detect the problem and help design around it. It's also a good idea to be particularly careful about allocating large buffers, because the amount of time it takes to create a buffer is roughly proportional to its size.

By applying these rules and good data design principles Dieselpoint has managed to develop a search engine that outperforms the majority of search engines coded in C++.

References

Nine Language Performance Round-up: Benchmarking Math & File I/O
http://www.osnews.com/story.php?news_id=5602

The Java Faster than C++ Benchmark
<http://www.kano.net/javabench/>

Java vs C++ "Shootout" Revisited (updated version of study above)
<http://sys-con.com/story/?storyid=45250&DE=1>

Performance of Java versus C++
<http://www.idiom.com/~zilla/Computer/javaCbenchmark.html>

A Bit About Garbage Collection, from Bruce Eckel's Thinking in Java
<http://www.javacoffeebreak.com/articles/thinkinginjava/abitaboutgarbagecollection.html>