Performance Limitations of the Java Core Libraries

Allan Heydon Marc Najork

Compaq Computer Corporation
Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301, USA
{heydon,najork}@pa.dec.com

Abstract

Unlike applets, traditional systems programs written in Java place significant demands on the Java runtime and core libraries, and their performance is often critically important. This paper describes our experiences using Java to build such a systems program, namely, a high-performance web crawler. We found that our runtime, which includes a just-in-time compiler that compiles Java bytecodes to native machine code, performed well. However, we encountered several performance problems with the Java core libraries, including excessive synchronization, excessive allocation, and other performance problems. The paper describes the most serious pitfalls and how we programmed around them. In total, these workarounds more than doubled the speed of our crawler.

1 Introduction

This paper describes our experiences using Java to build Mercator, a high-performance web crawler [11]. Web crawlers have many of the characteristics of classical systems programs: they run for days or weeks at a time, use large amounts of memory, have significant I/O requirements, must perform well, and need to be fault-tolerant. These requirements place very different burdens on the programming libraries and runtime environment than Java applets do.

When we started this project, we were not sure if our Java runtime environment or the Java core libraries were indeed suitable for such a systems program. In the process of implementing and optimizing Mercator, we were pleased to find that the "just-in-time" (JIT) compiler we used was quite efficient. In the latest version of the crawler, only about one third of the CPU cycles are spent executing compiled bytecode; the remaining cycles are spent in the Java runtime or executing native methods. As a matter of fact, half of Mercator's cycles are spent in the Unix kernel.

However, we also discovered that in many places, the Java core libraries traded off performance for ease of use. While this tradeoff is quite sensible for interactive applications such as applets, it is problematic for classical systems programs. The focus of this paper is on the performance problems we discovered in the Java core libraries, and the methods we used to overcome them.

The remainder of the paper is structured as follows. Section 2 describes the tools we used to develop and profile Mercator. Section 3 describes its main characteristics, focusing on its system requirements. The bulk of the paper, Sections 4-6, enumerates the performance problems we discovered in the Java core libraries, describes how we programmed around them where possible, and quantifies the performance improvements that resulted. Finally, Section 7 offers our conclusions.

2 Our Development Tools

The performance of any Java program depends on the compiler that produced its bytecodes, and on the Java runtime under which it is run. A variety of tools are available for debugging the performance of Java programs, such as Intel's *VTune* [17], Intuitive Systems' *OptimizeIt* [16], KL Group's *JProbe* [13], and IBM's *Jinsight* [12]. In this section, we describe the development tools we use and their features for measuring various performance properties of Java programs.

We use a standard Java compiler, namely, the Symantec Visual Cafe compiler. All of our coding, compiling, and small-scale testing is done on Wintel PCs.

Our production runs and large-scale tests are done on multi-processor Alpha machines running Compaq's Tru64 Unix operating system. For all our runs on this platform, we use a prototype in-house Java runtime called *srcjava* [9], which includes a JIT compiler that generates straightforward Alpha machine code from Java bytecode with little optimization.[1] As of September 14, 1999, this runtime holds the SPECjvm98 benchmark record by a substantial margin. One of srcjava's strengths is that its implementation of Java's synchronization primitives (i.e., locks and condition variables) are both time- and space-efficient. In the absence of lock contention, acquiring and releasing a lock takes only 23 machine cycles combined.

There are currently two versions of srcjava, one using an unmodified copy of Sun's JDK 1.1.4 class libraries, and the other using an unmodified copy of Sun's JDK 1.2.2 class libraries. Some of the inefficiencies we have discovered in JDK 1.1.4 have been eliminated in JDK 1.2.2; we mention those cases explicitly.

Srcjava includes extensive performance debugging support. It has command-line switches for measuring and reporting several aspects of a program's run-time performance, including lock acquisition and contention, garbage collection, heap allocation, and heap usage:

- **Lock acquisition and contention.** Srcjava can collect statistics about a program's lock acquisitions. It uses stack traces to identify acquisition sites. As a result, the same *synchronized* statement may produce multiple entries. For each acquisition site, srcjava prints the number of lock acquisitions performed at that site. The synchronization data is printed at the end of the run. Srcjava can also collect and report information about lock contention, including the elapsed time spent waiting on contested locks.
- **Garbage collection.** One cost of excessive allocation is increased garbage collection (GC). Srcjava can print the elapsed time spent on each collection, as well as the total GC time.
- **Heap allocation.** Srcjava can print the number and total size of all allocations made over the life of a program. For each allocation site (again identified by its stack trace), the output lists the type of the allocated object, the number and total size of allocations at that site (both as an absolute number of bytes and as a percentage of all allocations), and the site's stack trace. The allocation profile is printed at the end of the run.
- **Heap usage.** Srcjava provides two mechanisms for reporting on the contents of the heap. First, it can print a type-based profile of the heap before and after each GC. This profile is rather flat, but it can still be quite useful. The post-GC profile gives a profile of the live heap. Array types are divided into different groups, based on their sizes (in bytes). Second, srcjava can produce a more structured breakdown of the live heap based on reachability paths.

The other tool we frequently use for performance debugging is DCPI, the Digital Continuous Profiling Infrastructure [2, 8]. DCPI is a freely-available profiling tool that runs on Alpha systems. It uses counters built into the Alpha processor to profile code running in both user space and the kernel. DCPI's profiling overhead is quite low, typically 2%-6%.

We use DCPI to get CPU cycle breakdowns of Mercator. By virtue of DCPI's universal sampling, we can simultaneously profile the srcjava runtime, compiled bytecode, and native methods, including kernel routines. In addition to its profiling core, DCPI also includes tools for printing CPU cycle breakdowns at various granularities, including whole images, functions, and individual instructions.

[1] srcjava has recently been publicly released as the Compaq Fast VM v1.2.1.

3 The Mercator Web Crawler

Mercator is a high-performance web crawler written in Java that can be configured to perform a variety of tasks. For example, we have used it to collect statistics about the web, to perform random web walks [10], and to save pages from the web. Mercator is a production quality artifact, and will be included in version 3 of the AltaVista Search Intranet product [1].

We have used Mercator to download sizable portions of the web. For example, during a crawl in May 1999, Mercator requested 77.4 million web documents over the course of 8 days, achieving an average download rate of 112 documents/sec and 1,682 KB/sec. This crawl was performed on a Digital Ultimate Workstation with two 533 MHz Alpha processors, 2 GB of RAM, 118 GB of local disk, and a 100 Mbit/sec FDDI connection to the Internet.

Mercator is designed to be extensible and scalable. By extensible, we mean that Mercator is designed in a modular way, with the expectation that new functionality will be added by third parties. We use Java's dynamic class loading facilities to load user-supplied modules at run-time. By scalable, we mean that it should be able to crawl a web of indefinite size while using a bounded amount of memory. The bulk of Mercator's data structures are stored on disk; in production crawls, they often amount to tens of gigabytes.

Mercator is highly multi-threaded, typically using 500 threads to fetch and process web pages in parallel. Java's support for multi-threading allows us to use synchronous socket operations, which significantly simplifies the code, and allows our crawling process to utilize all the processors of a multiprocessor machine. By comparison, other crawlers documented in the literature [5, 7] are single-threaded and achieve concurrent downloads via asynchronous I/O, which is somewhat awkward to program.

Mercator's I/O requirements are also indicative of a classical systems program. It uses TCP sockets to download documents (with peak download rates of hundreds per second), UDP sockets to perform domain name service (DNS) resolutions, and both sequential and random-access files to access its large disk-based data structures.

For portability, Mercator is written in 100% pure Java. We run it on both Wintel PCs and Alpha Unix workstations. It is a medium-sized Java program; its Java source files (including test classes and comments) total 24,700 lines.

When crawling the world wide web, we found that the responsiveness of web servers fluctuates greatly over time, making it hard for us to measure Mercator's end-to-end performance in a repeatable fashion. We therefore wrote a program that, from Mercator's point of view, acts like an HTTP proxy, but synthesizes web pages instead of actually fetching them. We used this "web synthesizer" to measure the end-to-end performance of the initial and optimized versions of Mercator. Overall, the optimizations described in the following three sections more than tripled Mercator's download rate. One important caveat to this remark is that these optimizations were made in parallel with functional enhancements to Mercator. However, if anything, these functional enhancements have increased Mercator's computational burden, so we believe the before-and-after performance comparisons we draw in this paper are conservative.

4 Excessive Synchronization

Writing correct multi-threaded programs is far from trivial; it requires training and discipline [4]. Most programmers have not been exposed to programming with threads, and therefore lack the needed experience. Presumably for this reason, the designers of the Java core libraries decided to minimize the likelihood of race conditions by making many of the core classes thread-safe. This was done by declaring the classes' public methods to be *synchronized*. By doing so, they sacrificed performance for ease of use, since many of the lock acquisitions that result from this approach are unnecessary.

In practice, this tradeoff can lead to large performance penalties. Using DCPI, we determined that Mercator initially spent about 20% of its cycles on synchronization-related operations. Using srcjava's synchronization profiling support, we found that Mercator initially acquired an average of 2,315 locks per downloaded page. After we applied the optimizations described below, the fraction of cycles spent on synchronization fell to 1.5%, and the average number of locks acquired per downloaded page fell to 61.

4.1 StringBuffers

The over-synchronization problem is exacerbated if the class in question is a low-level class with many clients. The best example is the class *java.lang.StringBuffer*, all of whose public methods are synchronized. The *javac* compiler uses *StringBuffers* to implement the string concatenation operator +. For example, the expression *"foo"+"bar"* is translated to:

new StringBuffer().append("foo").append("bar").toString()

Evaluating this expression results in 3 lock acquisitions (one for each of the method calls), even though there is no potential for race conditions, since the *StringBuffer* is not accessible to any thread.

To make matters worse, *StringBuffers* are used by many other low-level Java classes. For example, up to JDK 1.1.5, the methods *Integer.toString* and *Long.toString* use *StringBuffers* to convert numbers into strings. In so doing, they perform $n + 2$ lock acquisitions, where n is the length of the resulting string. These methods in turn are called by overloaded versions of the *StringBuffer.append* method. For example, the expression *"THX"+1138* uses *Integer.toString* to convert 1138 into a string, resulting in 6 lock acquisitions, and then performs 3 more acquisitions to compute the final result.

The previous example is by no means artificial. For example, *InetAddress.getHostAddress*, which converts an IP address to a string, is implemented in exactly this way. As a result, converting the address 172.18.229.100 into a string requires 27 lock acquisitions, all of which are unnecessary.

Concatenating strings and converting numbers and IP addresses to strings are common operations in Mercator, so *StringBuffers* showed up prominently in srcjava's synchronization profile. To avoid the unnecessary lock acquisitions introduced by *StringBuffer*, we implemented a *Formatter* class that combines *StringBuffer* functionality with facilities for formatting various types (subsuming for instance *Integer.toString*). The use of *Formatters* significantly reduces Mercator's lock acquisition rate; it also avoids unnecessary heap allocations, as described below in Section 5.

Sun could easily address these issues by providing an unsynchronized variant of *StringBuffer* in addition to the synchronized version, and by changing the implementation of the string concatenation operator + (in the *javac* compiler) to use this unsynchronized variant. Moreover, this unsynchronized *StringBuffer* variant could safely be used in many places in the core libraries, such as in *java.util.Hashtable.toString*, where the *StringBuffer* is accessible from only one thread, thereby making synchronization unnecessary.

As of JDK 1.1.6, the implementations of the *Integer.toString* and *Long.toString* methods have been changed to use an array of characters rather than a *StringBuffer*. Note that although these new implementations cause no synchronization and require fewer function calls, they do perform an extra character array allocation and copy compared to the original versions.

4.2 I/O Streams

The *java.io* package provides a stream abstraction for performing byte I/O, and a reader/writer abstraction for performing Unicode character I/O. Both abstractions are designed to be composable; for example, the standard way to read a sequence of integers from a file is to compose a *FileInputStream* (to access the file) with a *BufferedInputStream* (to avoid excessive kernel calls), which in turn is composed with a *DataInputStream* (to convert bytes to integers).

The designers of the *java.io* package decided to make some (but not all) I/O classes synchronized. In particular, the *read* methods of *BufferedInputStream* and the *write* methods of *BufferedOutputStream* are synchronized. This design leads to fairly fine-grained locking, particularly when small amounts of data are read or written at a time. An alternative design would be to leave the methods unsynchronized, and to instead require clients to ensure single-threaded access. Of course, since the latter approach places the burden of lock acquisition on the client, it requires more discipline. However, the benefit of the latter approach is that many lock acquisitions can be avoided, since the client can protect an entire I/O transaction with a single lock acquisition.

Unfortunately, many methods of the *DataOutputStream* class write data to the underlying stream a byte at a time. For example, the method *writeLong* performs 8 write operations to its underlying stream. In the typical case where the underlying stream is a *BufferedOutputStream*, a single call to *writeLong* results in 8 lock acquisitions. Similarly, writing the ASCII representation of an n-character string results in n lock acquisitions. The *DataInputStream* class has exactly the same problems.

The *PrintStream* class is the standard way to write the string representation of a variety of types to an underlying stream.[2] As opposed to *DataOutputStreams*, *PrintStreams* write data to their underlying streams in larger chunks. For example, calling *PrintStream.print* causes a single *write* operation on the underlying stream. Unfortunately, the *PrintStream* implementation performs locking of its own. Using the *print* method to print a string causes six lock acquisitions within the *PrintStream* implementation (and using *println* causes 13 acquisitions), plus any synchronization performed by the underlying stream.

The extra synchronization introduced by the core stream classes is not just an academic problem. Like many systems programs, Mercator writes extensive logs. For example, it writes a line to a log for every web page that it downloads. Using srcjava's synchronization profile, we discovered that writing one line to the log caused 67 lock acquisitions!

To work around the stream problems described above, we wrote a *BufferedDataOutputStream* class that combines features of *PrintStream*, *DataOutputStream*, and *BufferedOutputStream*. Our *Buffered-DataOutputStream* class is completely unsynchronized, placing the synchronization burden on its clients. Using this class, writing a line to our download log requires only a single lock acquisition. We also wrote an analogous *BufferedDataInputStream* class that provides similar benefits.

Sun could address these performance problems by providing an unsynchronized variant of the *BufferedInputStream* and *BufferedOutputStream* classes, and by improving the implementations of the *DataInputStream* and *DataOutputStream* classes to perform fewer calls on the underlying stream. Although it is not documented in its API, *PrintStream*'s methods are thread-safe, so removing synchronization altogether is likely to break existing clients that have relied on this undocumented feature. However, only one lock acquisition is required to make *print* and *println* thread-safe.

4.3 Host name resolution

Any program that contacts sites on the internet whose identities are provided in the form of symbolic host names must resolve those names into IP addresses. This process is known as *host name resolution*, and it is supported by the *domain name service* (DNS). DNS is a globally distributed service in which name servers refer requests to more authoritative name servers until an answer is found. Therefore, a single DNS request may take seconds or even tens of seconds to complete, since it may require many round-trips across the globe.

All common computing platforms provide standard routines for performing host name resolution. In Java, these are the *getByName* and *getAllByName* methods of the *InetAddress* class.

Mercator makes extremely heavy use of these methods. To avoid multiple downloads of the same document, a web crawler must maintain a set of discovered URLs. When Mercator extracts a URL from a web page, the URL is converted to a canonical form and then tested against the set. As part of the canonicalization process, the host name is resolved. Hence, Mercator performs a host name resolution for every link in every page it downloads. These requests are made in parallel by its 500 crawling threads.

Our initial measurements showed that host name resolution in Mercator was a severe bottleneck: it accounted for 91% of each thread's elapsed time. Upon studying the implementation of *InetAddress.getByName*, we discovered that it tries to avoid issuing costly DNS requests by caching the results of previous resolutions. However, we also found that the cache is protected by a single lock, which is held for the entire duration of the resolution. Hence, if one thread misses in the cache and therefore must contact a name server, any other thread attempting to resolve a host name will block until the first thread's request completes, even if the second thread's request could have been served out of the cache.

[2] The Java core libraries appear somewhat schizophrenic with regards to *PrintStreams*: although officially deprecated, they are still used for the standard output and error streams.

Our first attempt at rectifying this problem was to keep our own cache. The lock protecting our cache is held only while accessing or updating the cache, but *not* while calling *InetAddress.getByName* in the event of a cache miss. As a result, host names that are contained in our cache can be resolved immediately, even while calls to *InetAddress.getByName* are in progress. Unfortunately, such calls are still serialized by the lock in *InetAddress*. This optimization reduced the elapsed time spent by each thread on DNS to 70%. Despite this improvement, host name resolution remained a bottleneck, since the rate at which Mercator discovers new hosts outpaces the rate at which host names can be sequentially resolved.

To overcome this problem, we implemented a DNS resolver in Java. Our resolver allows host name resolutions to be performed in parallel. It does not use *InetAddress.getByName*, but instead uses *DatagramSockets* to issue DNS requests directly to a local name server. Using our own resolver reduced the elapsed time spent by each thread on host name resolution to 14%.[3] More importantly, this change significantly increased the crawler's download rate.

Starting with JDK 1.2.2, Sun corrected the coarse-grained locking problem in *InetAddress.getByName*: a lock is held only while interacting with the DNS cache, but not while performing a name server transaction. However, it is worth noting that such transactions are done using the native method *InetAddressImpl.lookupAllHostAddr*, which most Unix JVMs implement using the standard *gethostbyname(3)* function. Unfortunately, on many Unix systems (namely, those that use the BIND library [3]), *gethostbyname* is synchronized by a single lock, and therefore single-threaded.

5 Excessive Heap Allocation

In Java, there are two costs associated with heap allocation. Aside from the direct cost of the *new* operation, there is also the cost of performing garbage collections (GC). The GC cost can be substantial, especially if the program has a high allocation rate. Moreover, most current Java runtimes (including srcjava) perform single-threaded, non-concurrent garbage collection. As a result, the garbage collection penalty is increased on multi-processor machines, since all but one of the processors are idle during collections.

Although we cannot easily measure the allocation cost, srcjava can report the elapsed time spent on collections. Initially, Mercator allocated 19.4KB on the heap per downloaded document, spending an average of 257 elapsed microseconds per downloaded document on garbage collection. After making the optimizations described below, the allocation rate was reduced to 3.6KB per document, and the elapsed garbage collection time was reduced to 61 microseconds per document. Our findings agree with those of Klemm, who found that Java programs often spend a lot of time performing allocations and object-to-object copy operations, and who developed a set of guidelines for avoiding such operations [14].

One problem with the Java core libraries with respect to allocation is that many objects cannot be reused. In some cases, it is possible to implement alternatives that allow object reuse. However, there are other cases where this is not possible, primarily because important native methods have been declared private, and are therefore inaccessible to any such alternative implementation. Given our philosophy of writing 100% pure Java code, we ruled out the option of implementing our own publicly accessible native methods.

FileInputStream is one example of a class whose instances are not reusable. A *FileInputStream* object is associated with a file when it is constructed, but there is no mechanism for reopening an existing *FileInputStream* on a different file. This means that the only way to access a different file is to allocate a new *FileInputStream* object. An alternative version of *FileInputStream* that avoids this problem cannot be written because the native method for opening a file is private to the *FileInputStream* class. The same remarks hold for *FileOutputStreams* and *RandomAccessFiles*.

Similarly, a *Socket* object is associated with a network connection at the time of its creation; it cannot be reopened on a different connection. So, the only way to establish a new network connection is to allocate a new *Socket* object. Moreover, each *Socket* allocation causes the allocation of 13 other

[3] Currently, when a worker thread processes a page, it extracts URLs from the page and canonicalizes them in sequence. We could further reduce the elapsed time spent by each thread on host name resolution by performing these canonicalizations in parallel.

objects (consuming a total of 304 bytes per *Socket* on our runtime). Again, it is impossible to write an alternative socket implementation because the native methods for establishing network connections are all private to *PlainSocketImpl*. Obviously, a web crawler like Mercator establishes hundreds of network connections per second. Due to the design of the *java.net* package, these connections cause thousands of unavoidable allocations per second.

Like *FileInputStream* and *Socket* objects, instances of *BufferedInputStream* are not reusable. The *BufferedInputStream* constructor associates the stream with an underlying stream, and there is no mechanism for reassociating it with a different underlying stream later. So, accessing a new network connection via *BufferedInputStreams* requires the allocation of both a *Socket* and a *BufferedInputStream*, which in turn causes the allocation of a (potentially large) internal buffer.

However, unlike *FileInputStreams* and *Sockets*, it is possible to write an alternative version of *BufferedInputStream* that promotes object reuse through an *open* method, partly because *BufferedInputStream* has no native methods. We have implemented two such alternatives. One is the *BufferedDataInputStream* class mentioned in Section 4.2; the other is a class called *RewindInputStream* that is used to read web pages from network connections and buffer them. At the beginning of a crawl, we allocate one *RewindInputStream* object per crawling thread. These stream objects (and their large underlying buffers) are then reused throughout the life of the crawl.

Sun could foster object reuse by extending the stream and socket classes with methods for reopening them. In the *FileInputStream*, *FileOutputStream*, and *RandomAccessFile* classes, such methods already exist, but are not declared public.

In the examples described so far, the design of some classes in the core libraries prevented reuse of their instances. However, there are other classes, such as *StringBuffer*, whose instances are easily reusable, but whose points of use are widely scattered throughout the program, making it hard to identify an instance that can be reused. The same is true of *Formatter*, our *StringBuffer* replacement described in Section 4.1. To promote reuse of *Formatter* objects (and their internal buffers), we maintain a pool of unused *Formatters*. To avoid unnecessary synchronization when checking *Formatters* out of and back into the pool, we divide the pool into per-thread sub-pools. Once Mercator reaches its steady state, no new Formatters are allocated.

6 Other Pitfalls

In addition to the synchronization and allocation overheads described above, we have also discovered several other unrelated pitfalls in the Java core libraries.

- The *java.io.RandomAccessFile* class is unbuffered, and no buffered variant is supplied. This means that every single read or write operation on a *RandomAccessFile* incurs the cost of a kernel call. To work around this problem, we wrote our own *BufferedRandomAccessFile* class. Our implementation is a subtype of *RandomAccessFile*, but it uses a buffering scheme that combines the designs of Modula-3's reader and writer abstractions [6]. Sun could provide a buffered variant of *RandomAccessFile* similar to ours as part of the *java.io* package.
- Constructing a new instance of *java.io.PrintStream* is quite expensive: constructing a *PrintStream* on our production runtime takes 750 times longer than constructing a *DataOutputStream*. The cost of *PrintStream* construction is relevant because it is quite natural to use *PrintStreams* to write strings to sockets. Since a new *SocketOutputStream* is created for each TCP connection, and since *PrintStreams* cannot be reopened on different streams, a new *PrintStream* must be allocated for each TCP connection.

  After some investigation, we discovered that the *PrintStream* constructor allocates a *BufferedWriter* and a *OutputStreamWriter*, which in turn allocate an 8K character array and an 8K byte array, respectively. 70% of the time required to construct a *PrintStream* can be attributed to these two array allocations. The vast majority of the remaining time is spent by a call to the *sun.io.CharToByteConverter.getDefault* method, which creates a converter from Unicode characters to bytes.

We overcame this problem by allocating one instance of our *BufferedDataOutputStream* class (see Section 4.2) per thread for the life of the crawl. *BufferedDataOutputStreams* have an *open* method, which allows them to be reassociated with a different underlying stream. This method allows each thread to reuse the same *BufferedDataOutputStream* for all sockets it creates.

We were alerted to this problem by the poor performance of our synthetic proxy, which was constructing a new *PrintStream* on which to write the response to each newly received HTTP request. After replacing the *PrintStreams* in our synthetic proxy with *BufferedDataOutputStreams*, the rate at which the proxy could synthesize documents increased by a factor of 2.75.

- There are cases in the core libraries where two operations have similar semantics, but drastically different performance. For example, the *java.lang.String* class has six constructors for creating a string from a byte array. Four of them use byte-to-character converters to convert bytes to Unicode characters, while the other two take a byte as an argument and use it as the higher-order byte of each Unicode character. Despite the similarities of these constructors, the variants that use converters are 10 to 40 times slower, depending on the Java runtime. In hindsight, it is obvious why the constructors that perform conversions are more expensive; however, we think this cost should be advertised in the class documentation.

  In Mercator, converting byte arrays to strings is a common operation, since web pages are downloaded as byte streams, but in the process of extracting links from HTML pages, parts of the byte stream are converted to strings. When we inadvertently introduced uses of the expensive constructor into the link extraction code, Mercator's crawling rate dropped by a factor of 3.

- The *java.net* package provides networking facilities, including an implementation of the HTTP protocol. Our very first version of Mercator made use of this HTTP implementation. However, we soon discovered that it contains no provisions for specifying timeouts. As a result, an HTTP request will hang indefinitely if the remote server fails to close the connection, thereby disabling the calling thread. Surprisingly, such ill-behaved web servers are not uncommon. To work around this problem, we wrote our own streamlined HTTP implementation. Our version uses the *java.net.Socket* class, which *does* provide methods for setting timeouts. Sun could remedy this problem by adding methods to the *URLConnection* class for setting timeout values.

- Our initial implementation of Mercator used *java.net.URL* to represent URLs. However, we found that URL creation was relatively costly. By implementing our own URL class[4], we sped up the URL creation process by 25%-30%.

7 Conclusions

We have used Java to implement a classical systems program, namely, a high-performance web crawler. We found that the Java language is well-suited to such a task. Its object-oriented model and its package system support a modular programming style; its exception system makes it easier to build robust applications; garbage collection prevents memory leaks, making it easier to write long-running programs; and language-level thread support makes it easier to write multi-threaded programs. (Java shares all of these attributes with Modula-3 [15], which has been our systems programming language of choice up until now.)

By sampling the hardware performance counters of the Alpha processor, we discovered that our system spends 63% of its cycles in the runtime or in native methods, and in fact 50% in the Unix kernel. We conclude that for I/O-intensive applications (which in Java result in a large number of native method calls), and in the presence of a reasonably good JIT compiler, Java is an attractive alternative to traditional systems programming languages such as C.

However, we also discovered that the Java core libraries were designed for ease of use, not for speed. We identified numerous performance problems within the core libraries. Two of the root causes are excessive synchronization and excessive memory allocation, but we also discovered other inefficiencies. We were able to program around most, but not all, of these problems.

[4]Using our own URL class made sense only because we had written our own HTTP implementation.

A cleaner solution would be to fix these problems in the core libraries themselves. For each of the problems we described, we have outlined how Sun could address them. The remedies we have proposed fall into three categories: providing unsynchronized variants of existing classes, improving the implementations of existing methods, and extending existing classes with additional methods. All the changes we have proposed are backward-compatible with the existing Java libraries.

Java's use as a development language for server-side applications is becoming more and more prevalent. The performance of such applications is often critical. One way to improve Java performance is to develop fast Java runtimes, and indeed, much research is being done in this area. Although such research is crucial, our experience shows that improving the performance of the core libraries is also important. We hope that Sun will therefore devote more effort toward improving the performance of the core libraries, and that it will augment the documentation of the libraries to highlight expensive method calls. Java's success as a systems programming language depends on it.

Acknowledgments

Thanks to Sanjay Ghemawat for creating the srcjava runtime, and to the DCPI team for developing DCPI.

References

[1] AltaVista Search Intranet.
`http://doc.altavista.com/business_solutions/search_products/-search_intranet/intranet_intro.shtml`

[2] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *ACM Transactions on Computer Systems*, **15**(4):357–390, November 1997.

[3] Berkeley Internet Name Domain (BIND).
`http://www.isc.org/bind.html`

[4] Andrew D. Birrell. An Introduction to Programming with Threads. SRC Research Report 35, Digital Equipment Corporation, Systems Research Center, January 1989.

[5] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *Proc. of the 7th International World Wide Web Conference*, pages 107–117, April 1998.

[6] Mark R. Brown and Greg Nelson. IO Streams: Abstract Types, Real Programs. SRC Research Report 53, Digital Equipment Corporation, Systems Research Center, November 1989.

[7] Mike Burner. Crawling towards Eternity: Building an archive of the World Wide Web. In *Web Techniques Magazine*, **2**(5), May 1997.

[8] Digital Continuous Profiling Infrastructure Home Page.
`http://www.research.digital.com/SRC/dcpi/`

[9] Sanjay Ghemawat. Srcjava Home Page.
`http://www.research.digital.com/SRC/java/`

[10] Monika R. Henzinger, Allan Heydon, Michael Mitzenmacher, and Marc Najork. Measuring Index Quality using Random Walks on the Web. In *Proc. of the 8th International World Wide Web Conference*, pages 213–225, May 1999.

[11] Allan Heydon and Marc Najork. Mercator: A Scalable, Extensible Web Crawler. To appear in *World Wide Web*, Baltzer Science Publishers, December 1999.

[12] Jinsight Home Page.
`http://www.alphaWorks.ibm.com/formula/jinsight/`

[13] JProbe Home Page.
`http://www.klg.com/jprobe/`

[14] Reinhard Klemm. Practical Guidelines for Boosting Java Server Performance. In *Proc. of the ACM 1999 Java Grande Conference*, pages 25–34, June 1999.

[15] Greg Nelson (editor). *Systems Programming with Modula-3*. Prentice Hall, 1991.

[16] OptimizeIt Home Page.
`http://www.optimizeit.com/`

[17] VTune Tool Home Page.
`http://developer.intel.com/vtune/analyzer/`