C and C++ are being replaced by Java for many applications, but they would never become completely obsolete.

Abstract

The exceptions, garbage collection and automatic type checking features of languages like Java outweigh most of the performance penalties these languages have. However, C and C++ would live on, as these languages are needed for real-time applications where timing and performance are crucial. But, once Java closes the gap in performance, fewer applications would be written in C++.

Introduction

C++ was created by Bjarne Stroustrup at AT&T Bell Labs in 1979. He needed a language that would combine the scaleable class concepts of the Simula programming language with the efficiency and portability of C. Several basic philosophies drove the design of the language: it must scale well on large projects; it must be based on an existing, successful language; and it must be general purpose and not impose any particular style of programming on its user. Above all, Stroustrup was influenced by the need to create a practical language.

Java was born in 1990. It was created to meet a different set of criteria: portability, reliability, and longevity. Java was created to be used in the rapidly changing consumer electronics industry. A different problem often requires a different approach. And so Java's evolution followed a different path. Java's design was also heavily influenced by industry experience with C++.

By using Java, developers can create software that is able to run on any computer chip and on any operating system. Creating software that runs on multiple platforms is incredibly costly and time consuming. Software that is written in C or C++ has to be adapted and recompiled for each computer that

it would run on. Quite often it is not a trivial task as C/C++ is very dependent on the library files and API's used in the code. Even within families of operating systems, like Microsoft's, it is difficult to support all platforms with a single set of code. Java's multi-platform capability makes it a natural choice for use on the Internet. The Internet is used daily by millions of people on a wide array of computing platforms. Internet developers need a way to create applications that will run on whatever platform the user is using today and tomorrow. Although Java's portability gives it a clear advantage over other languages, this feature also creates one of Java¹s biggest disadvantages - it's performance.

Table 1 shows the basic language features of Java vs. C++.

The exceptions, garbage collection and automatic type checking features of languages like Java outweigh most of the performance penalties these languages have. However, C and C++ would live on, as they are needed for real-time applications where timing and performance are crucial. But, once Java closes the gap in performance, fewer applications would be written in C++. But C/C++ would never completely disappear, similarly to other languages like Fortran and Cobol, which are still being used by some companies.

	Java	**C++**
Inheritance	Single (but with multiple subtyping)	Multiple
Preprocessor	No	Yes
Separate Interface/Implementation	No (interface generated from code)	Yes (header files)
Garbage Collection	Yes	No
Operator Overloading	No	Yes
Pointer Arithmetic	No	Yes
Generics	No (but extensive polymorphism)	Yes ("templates")
Exceptions	Yes	Yes

Native Multi-threading	Yes	No

Table 1

Exceptions

Exception specifications in Java are vastly superior to those in C++. Instead of the C++ approach of calling a function at run-time when the wrong exception is thrown, Java exception specifications are checked and enforced at compile-time. In addition, overridden methods must conform to the exception specification of the base-class version of that method: they can throw the specified exceptions or exceptions derived from those. This provides much more robust exception-handling code.

Garbage Collection

"Java frees memory automatically, by performing automatic garbage collection, so you never need worry about memory leaks, nor must you waste time looking for one. Thus, you are more productive, and less likely to be driven crazy via tedious, mind-numbing debugging." (On to Java, 3). There are no destructors in Java. There is no "scope" of a variable to indicate when the object's lifetime is ended – the lifetime of an object is determined instead by the garbage collector. There is a **finalize( )** method that's a member of each class, something like a C++ destructor, but **finalize( )** is called by the garbage collector and is supposed to be responsible only for releasing "resources" (such as open files, sockets, ports, URLs, etc). If you need something done at a specific point, you must create a special method and call it, not rely upon **finalize( )**. Put another way, all objects in C++ will be (or rather, should be) destroyed, but not all objects in Java are garbage collected. Because Java doesn't support destructors, you must be careful to create a cleanup method if it's necessary and to explicitly call all the cleanup methods for the base class and member objects in your class.

Garbage collection means memory leaks are much harder to cause in Java, but not impossible. (If you make native method calls that allocate storage, these are typically not tracked by the garbage collector.) However, many memory leaks and resource leaks can be tracked to a badly written **finalize( )** or to not releasing a resource at the end of the block where it is allocated (a

place where a destructor would certainly come in handy). The garbage collector is a huge improvement over C++, and makes a lot of programming problems simply vanish. Garbage collector is running in the background of all Java programs and taking care of returning to the memory heap all of the memory not used by the program. Experienced programmers see this feature as a remarkable one which improves productivity. It reduces additional coding/debugging/testing times during development. It might make Java unsuitable for solving a small subset of problems that cannot tolerate a garbage collector, but the advantage of a garbage collector seems to greatly outweigh this potential drawback.

Automatic Type Checking

Java's type checking is much stricter than that in C or C++. In particular, casts are checked at both compile time and run time. After that, type checking is repeated at link time again to detect version errors. This is better than C or C++ type checking as it prevents lots of errors. However, type checking in C and C++ also has an advantage of allowing programmer a lot of flexibility in writing the code.

Performance

Although Java's ability for producing portable, architecturally neutral code is desirable, the method used to create this code is inefficient. Unlike natively compiled code, which is a series of instructions that correlate directly to a microprocessors instruction set, an interpreter must first translate the Java binary code into the equivalent microprocessor instruction. Obviously, this translation takes some amount of time and, no matter how small a length of time this is, it is inherently slower than performing the same operation in machine code. According to PC Magazine, it appears very significant: "Compared with native code, Java VMs are excruciatingly slow. ... Java still cannot compete with natively compiled C++ code." (PC Magazine, April 7, 1998, 104).

Currently, there are some committees are working on trying to close the gap in the performance between C++ and Java. The biggest potential stumbling block is speed: interpreted Java runs in the range of 20 times slower than C. Nothing prevents the Java language from being compiled and there are just-

in-time compilers appearing at this writing that offer significant speed-ups. It is not inconceivable that full native compilers will appear for the more popular platforms, but without those there are classes of problems that will be insoluble with Java because of the speed issue.

Mads Pultz had compared the performance of Java and C++ for matrix computation and the results of this comparison are presented in figures 1~3. As it is seen from the figures and from table 2, C++ is much faster than Java. There is a range of different applications where the speed difference is crucial, and it would be a long time before Java will be able to compete with C or C++ for those types of applications. As an example, Java would not be very successful in tracking and then sending a missile to destroy an enemy war plane.

	java1.4.0	java1.4.1 RC	c++
console	33.58	37.58	3.6
hash	6.35	7.13	1.23
io	3.68	4.1	1.04
list	2.71	3.03	0.19
no	0.86	1.16	0.04
speed	1.18	1.63	0.18
native	1.4	1.67	0.09
Total sum	49.76	56.3	6.36

Table 2. Runtimes. [3]

Conclusion

Java has many advantages over C++. They include exception handling, garbage collection, and automatic type checking. But Java also has a large disadvantage. It performs many times slower than C++, especially when it comes to I/O operations. Although this performance difference cannot be ignored, it may not be too important for some applications. Modern day computers are very fast and the speed difference may often translate to waiting for 8 milliseconds for Java as opposed to 1 millisecond for C++. Java may be much slower, but, from the users perspective, there is relatively little difference. The advantages to Java outweigh its disadvantages in most cases. The ease of development, reusability and portability would make Java

replace C++ in many applications, where performance is not so important. But for real time applications, C++ would still be the language of choice.

References

1. http://hjem.get2net.dk/mpultz/
2. http://www.cs.colostate.edu/~cs154/PerfComp/
3. http://www.flat222.org/mac/bench/
4. Winston, Patrick Henry. 1997. "On to Java", Addison-Wesley.

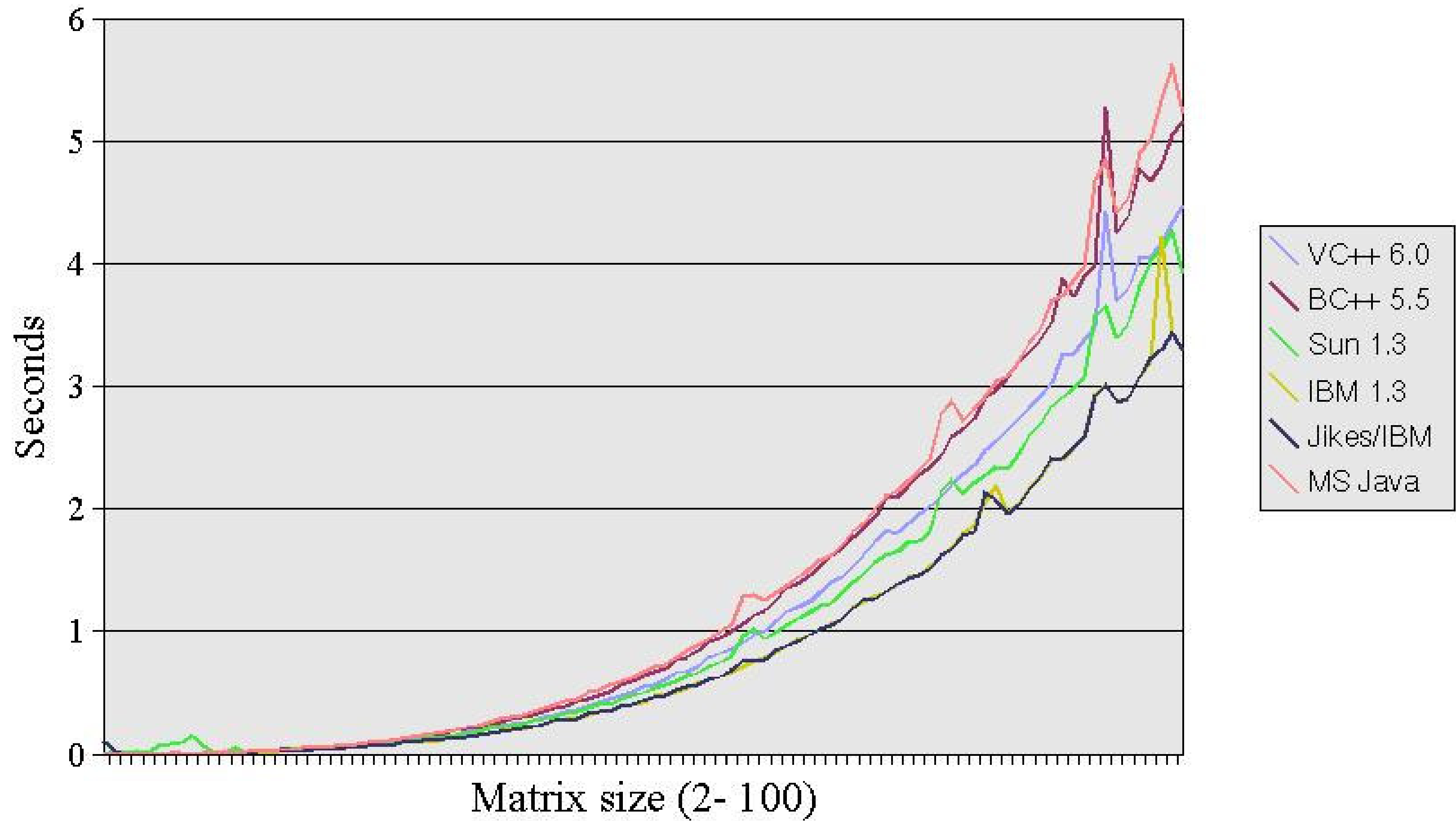

figure 1

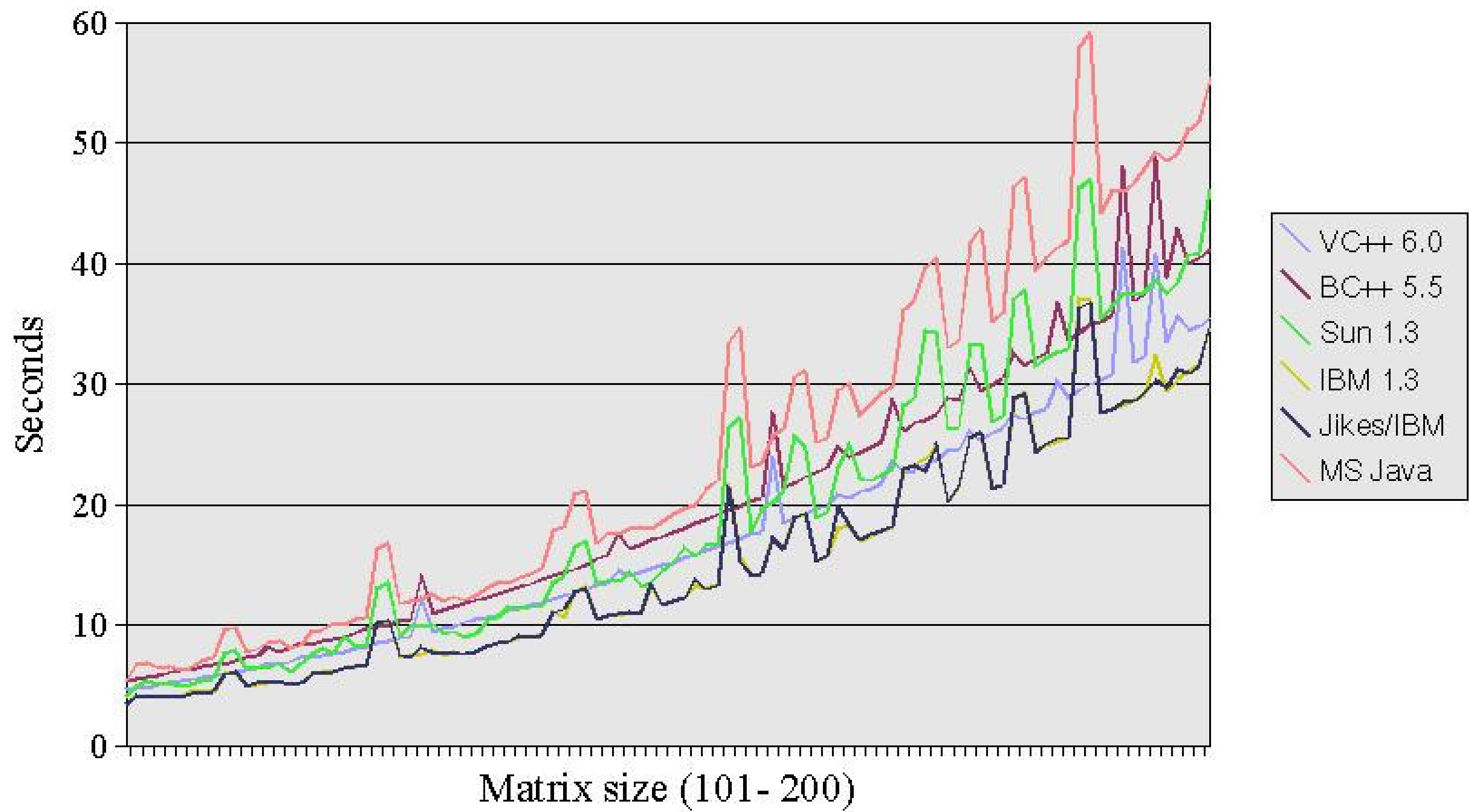

figure 2

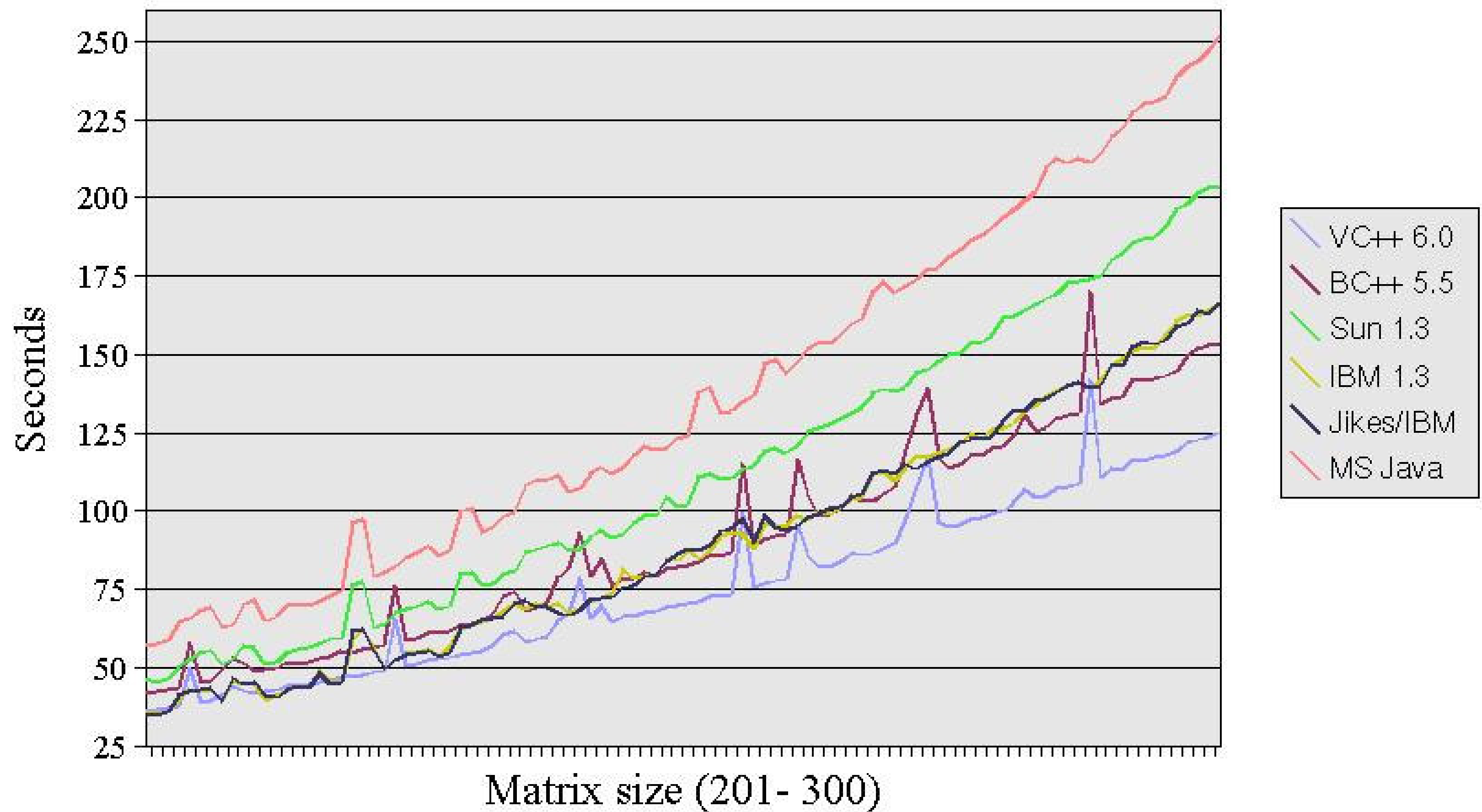

figure 3