WARNING: Java considered dangerous for high performance transaction systems. True or False?

M. C. Little

Computing Science Department, University of Newcastle,
Newcastle upon Tyne, England.

1. Introduction

Since the advent of commercial transaction processing systems, the aim has been to achieve greater and greater performance. First it was tens or hundreds of transactions per second, then a thousand transactions, and as transaction systems achieved these goals, newer and higher rates were set as the way of illustrating improvements and obtaining competitive advantage. Although some of this increase in performance has been due to the improvements in physical hardware speed, others have come from changes in programming languages.

As transaction systems have evolved, so have programming languages, moving from raw machine code, through COBOL and C, to C++, and beyond. When a new language appears and becomes accepted by the general community, so TP providers have typically look at whether or not it is appropriate for their use. Generally this has been the case, and because all of these languages are compiled, transaction systems have benefited in the improvements in compiler technology. The current language of choice appears to be Java, a language that is both compiled and interpreted [ORA00]. Few (if any) commercial transaction systems have appeared using interpreted languages. This paper attempts to address the question: Is Java a language that can be used to implement an efficient, and performant transaction system? Having implemented transaction systems in both C++ and Java [GDP95][MCL97b], we believe the experience we have gained can help to answer this question.

1.1 Compiled versus interpreted

Why are high-level languages used for implementing most applications rather than coding directly in assembler? Typically because the majority of programmers can write more lines of C++, say, than machine code, it's easier to debug and understand, and technology has advanced to such a point that compilers can generate machine code as efficient as some of the best machine code programmers. However, some transaction service implementations still use machine code written by hand for performance critical areas.

Applications written in interpreted languages (e.g., Basic or Lisp) are, by their very nature, slower than their compiled counterparts. However, unlike a compiled application, which by necessity is hardware specific, the interpreted application code can be moved

from machine to machine without requiring modification: the interpreter, which is part of the language runtime, is responsible for generating hardware specific instructions from the application code. Amongst other things, the trade-off between speed and portability means that compiled languages will likely never completely replace interpreted languages. However, the slowness of an interpreter's "compilation on demand" means that implementing a transaction system in such a language has been the domain of academic or research environments; for most commercial environments they present too much of an overhead.

2. The Web-effect on transaction systems

The Web and e-commerce in general promise a lot in terms of revenue for vendors and convenience for users. However, the Web frequently suffers from failures which can affect both the performance and consistency of applications running over it. For commercial services, such failures can result in loss of revenue and credibility, and make users wary of becoming involved in commercial ventures where their own money may be at stake. Therefore, there is a need for techniques which will allow applications to tolerate such failures and to continue to provide a service. The use of transactions to ensure the all-or-nothing effect is very important both for users and service providers [MCL97a].

As a result, the next generation of transaction systems are being developed for use within this environment, and several vendors see the use of Java to implement them as the right direction to take for a number of reasons including its cross-platform capabilities, and the high-level programming abstractions it offers which typically make development of complex, multi-threaded applications easier [HP01][GEM00]. As a result, existing transaction standards have been incorporated into the Java domain (e.g., the Object Transaction Service from the OMG [OMG95][SUN99a][SUN99b]).

However, there are 2 questions that need to be asked: (i) is Java the right language for e-commerce applications? and (ii) is it up to the task of implementing high-throughput transaction systems? In the following sections we shall attempt to answer these questions objectively.

2.1 100% pure Java or broke?!

Before answering the two questions posed previously, we should first mention that vendors using Java fall into two categories: those that simply use it to interface to existing languages and systems, and those that use it throughout their applications. Programs written in Java that do not rely upon linking in objects that have been written in other languages [SUN97] are called *100% pure Java.* With this in mind we shall rephrase the second question: is it possible to write a high-throughput transaction system using 100% pure Java that does not simply delegate its work to an existing transaction system?

2.2 Is Java the right language for e-commerce applications?

The answer to the first question is undoubtedly yes, simply because there is nothing currently better. The majority of vendors in the e-commerce arena see the ability to rapidly prototype and then develop applications as critically important to their success. In addition, the portability of Java applications means that vendors are not required to have access to the myriad of hardware and operating system variations available to their customers as they had to in the past. Subsequent programming languages (possibly C#, for example [CW00]) will learn from Java, just as Java learned from predecessor languages such as C++ and APL [BS97][EH77].

2.3 Is it possible to write a 100% pure Java high-throughput transaction system?

The answer to the second question, however, is much less straightforward. The efficiency of Java byte-code compilers and interpreters have improved over the years, and the advent of just-in-time compiler technology means that applications can typically run much faster than they did when Java first appeared. However, without breaking the 100% pure Java holy grail, it is unlikely that Java applications will ever consistently reach the same performance figures as, say, C or C++. But does that really matter when a programmer can produce more (and possibly better) code in less time? Possibly not in the Web domain, where *Web-years* dominate the schedules of developers, and hardware performance is still increasing. Being able to get an application to market quicker than a competitor is important.

However, it is our belief that the interpreted nature of Java is not the obstacle to Java being accepted into the exclusive circle of languages suitable for transaction systems. As mentioned above, hardware improvements mean that faster processors can be thrown at slower Java applications in order to achieve acceptable performance. We believe that the obstacle is the fact that Java is more than a language, it is also a platform neutral runtime environment (the virtual machine), and in order to make the virtual machine architecturally neutral, certain restrictions have been placed on the APIs: the lowest common denominator approach has been taken. Unfortunately, some of these restrictions can affect the implementation of a transaction processing system versus its traditionally compiled relatives. We include some of the restrictions we have encountered below:

- *File-level locking*: it is not directly possible for either a thread or JVM (i.e., process) to lock a file from another thread or JVM. It is possible to emulate file-level locking programmatically, but in a much less efficient manner than that provided by the operating system. This obviously affects the efficient implementation of certain types of transaction logs.
- *File-segment locking*: just as it is not possible to lock entire files, it is also not possible to easily lock sections within a file. Again, this can affect the efficiency and performance of transaction log implementations.

- *Thread synchronisation*: the Java language does not provide mutual exclusion and semaphore primitives as in advanced threading APIs, such as Posix Threads [DRB97]. The single *synchronized* keyword is often too coarse for efficient multi-threaded programming, requiring extra progammatical work to achieve more powerful primitives such as *trylock*. Unfortunately, as a result the *synchronized* keyword is often overused, with resultant degradation of system performance.
- *Memory mapped files*: in some transaction systems it is often more efficient to map log files directly into memory rather than use the raw file system APIs. This is simply not possible in Java since it maintains a strong separation of JVM memory from file system memory.
- *Shared memory*: it is not possible for one JVM to communicate with another residing on the same machine through shared memory. Although this may be argued as a security issue, it could be solved with suitable access permissions. This may affect the performance of, for example, shared lock stores and NVRAM object stores.
- *Garbage collection*: straddling the JVM and language divide, the garbage collection aspect of Java often proves a problem as much as it simplifies programming. The fact that it is not possible to delete objects when an application *knows* they are no longer required, means that the size of the JVM is sometimes larger than it needs to be. Obviously once the garbage collector runs, the size reduces but this may be some time later. Unfortunately, calling the garbage collector directly is still only interpreted as a "hint" that it should run, and cannot be a guarantee that it will run at the desired time.

Given the platform neutrality requirement of Java we can understand the reasons behind some API "weakening". However, in our experience of implementing a C++ transaction system on 5 different operating systems, and 7 different threading implementations [GDP95], as far as the file system and threading restrictions go, the core functionality that is missing is available on most (all?) modern operating system platforms. Therefore, it should be possible to support a more efficient and flexible API to these mechanisms without affecting the platform neutrality of the language.

Fortunately, although these limitations in the Java API present obstacles in the development of an efficient transaction system, it is possible to work-around many of them while still retaining 100% pure Java. However, such workarounds are not trivial (e.g., implementing file-level locking using JVM shared file-system objects), are almost certainly less performant than their operating-system counterparts, and could be solved simply by changing the language. Transactions form one of the core requirements for enterprise-level applications, regardless of the programming language. Therefore, this lack of functionality in a key area makes the development of efficient enterprise applications harder than it needs to be. As a result, we believe that changes in the API are necessary if Java is to be taken seriously for anything other than basic applications.

3. Conclusions

So, is it possible to implement an efficient 100% pure Java transaction system? Yes, in our opinion, but it requires more effort in some areas than should be necessary. Using Java certainly simplifies some aspects of development; for example from comparing our development of a C++ transaction system [GDP95] to one we wrote entirely in Java [MCL98] we can say that the latter took less time. However, objectively it is not possible to say how much of this was due to the advantages offered by the Java language, and how much was the result of having already addressed many of the more time consuming architectural issues in C++.

Given that transactions and enterprise-level systems are becoming more and more important in the Java world, performance and efficiency requirements will certainly have to be addressed. The language must continue to evolve in order to match the continually evolving requirements placed on it. However, there seems to be some resistance to changing the low-level language APIs, and some of the "core" functionality of the language such as threading and garbage collection. Why this is the case we do not know. In our opinion, without these changes, 100% pure Java may become less of a requirement, with the possibly that Java may even lose relevancy to enterprise applications.

4. References

[BS97] B. Stroustrup, "The C++ Programming Language, Third Edition", Addison Wesley, 1997.

[CW00] C. Wille, "Presenting C#", Sams Publishing, 2000.

[DRB97] D. R. Butenhof, "Programming with POSIX Threads", Addison Wesley, July 1997.

[EH77] E. Harms and M. P. Zabinski, "Introduction to APL and Computer Programming", Wiley, April 1977.

[GDP95] G. D. Parrington et al, "The Design and Implementation of Arjuna", USENIX Computing Systems Journal, Vol. 8., No. 3, Summer 1995, pp. 253-306.

[GEM00] "J Object Transaction Monitor", Gemstone Systems, September 2000 (see www.gemstone.com/products/j/otm.html)

[HP01] "Total-e-Transactions 2.1 Manuals", Hewlett Packard, March 2001 (see www.bluestone.com).

[MCL97a] M. C. Little, S. K. Shrivastava, S. J. Caughey, and D. B. Ingham, "Constructing Reliable Web Applications Using Atomic Actions", Proceedings of the 6th Web Conference, April 1997.

[MCL97b] M. C. Little and S. K. Shrivastava, "Distributed Transactions in Java", Proceedings of the 7th International Workshop on High Performance Transaction Systems, September 1997, pp. 151-155.

[MCL98] M. C. Little and S. K. Shrivastava, "Java Transactions for the Internet", Special Issue of the Distributed Systems Engineering Journal, 1998.

[OMG95] "CORBAservices: Common Object Services Specification", OMG Document Number 95-3-31, March 1995.

[ORA00] "Java in a Nutshell", O'Reilly and Associates, Inc., 2000.

[SUN97] "Java Native Interface Specification (JNI)", Sun Microsystems, May 1997.

[SUN99b] "Java Transaction Service 1.0 (JTS)", Sun Microsystems, December 1999.

[SUN99a] "Java Transaction API 1.0.1 (JTA)", Sun Microsystems, April 1999.