Building a Java virtual machine for server applications: The Jvm on OS/390

by D. Dillenberger
R. Bordawekar
C. W. Clark
D. Durand
D. Emmes
O. Gohda
S. Howard
M. F. Oliver
F. Samuel
R. W. St. John

As the use of the Java™ language and virtual machines proliferates beyond the sphere of applets into the space of server programs, developers are requiring better performance, availability, and transactional and scalability features. This paper describes the work done for the Operating System/390 (OS/390®) Java virtual machine to improve performance and serviceability, to introduce security and performance enhancements, and to redesign parts of the virtual machine to enable it to run server programs efficiently and safely. Although OS/390 was the motivating platform for these changes, Java server programs on any platform can benefit from these features.

A new era in application programming using the Java** language, virtual machine, and development kit came to life as a technology for the embedded devices market. The proliferation of embedded computer chips in the consumer product market begged for a new programming environment that was portable across chips, small in footprint, easy to use, and similar to programming languages used for business computer systems.[1] The Java language from Sun Microsystems fit the requirements.

Not only did this new technology fit the requirements for the embedded device market, but at about the same time it was introduced, the Internet and, more specifically, the browser market exploded on the scene. Here again, a new use of computers begged for a solution to a problem: the browser user interface needed more capability to do graphics, automation, and data presentation. Some of the same requirements existed for this market as for the embedded device market: the programming environment needed a small footprint, and it needed to be easy to use and portable across operating systems (and browsers).

The next evolutionary step for the Java technology came when many saw its potential for the server mar-

0018-8670/00/$5.00 © 2000 IBM

ket. However, the big question was: would a technology that was originally targeted for the embedded device and browser markets be applicable and useful on a server? And, for the Operating System/390 (OS/390*) market, would it be a viable technology that would satisfy the quality of services expected by customers using one of the most mature, comprehensive, and complex servers on the market? After some intense analysis, it was concluded that this technology, with some reasonable modifications and additions, could be applied to the server set of business application problems. If successful, one could then make the case for retraining a programming staff in one technology base that could be used for all business applications in an enterprise. Additionally, an enterprise staff could then build or buy applications for all aspects of the business—client, middle tier, and server—using the same technology. The potential for significant financial savings to our customers justified moving forward.

When we were faced with assessing this new technology for its applicability to OS/390 and the traditional OS/390 market, we were at first very skeptical, and so were several experts we consulted. But in spite of the initial negative reaction, we decided to proceed with a feasibility study. This prototype showed us the challenges we faced if we decided to go further. These challenges, and how we addressed them, are the subject of this paper.

The prototype showed that the base technology that we received from Sun Microsystems as a reference implementation for UNIX** systems suffered 60–100 times performance degradation when ported to OS/390. Moreover, the initial prototype did not scale up when run on machines with more processors available—in fact, performance degraded further. This was measured by using standard benchmarks[2] and by one of our early customers working with us on the feasibility study. Of course, this version of the prototype did not have a just-in-time compiler, it ran in "debug" mode, and had no C language optimizations applied to it, so good results were not expected. Admittedly, this was a crude prototype, built to prove feasibility, not to show good performance. But it did identify the magnitude of the job we would have in converting from this level of the code to a usable product for OS/390 customers. It also pointed out the need for reliable, useful benchmarks that could easily produce repeatable runs and reliable data for performance analysis. This requirement was addressed, and by our first release we had such a benchmark. We used this to establish a point of reference with the first release and then to measure our progress for all follow-on releases.

In addition to the poor performance, there were several functional deficiencies that would limit the use of this technology in a traditional OS/390 customer environment. Such functional deficiencies were:

Would a technology targeted for the embedded device and browser markets be applicable and useful on a server?

support for EBCDIC (extended binary-coded decimal interchange code) data, access to our transactional subsystems, access to our "legacy" data, and national language support (NLS).

When we examined the results of the feasibility study, it was clear that for this technology to be useful in the OS/390 marketplace, our first priority was to fix the performance problems. We asked ourselves (the key technical leadership community): can these problems be solved? We looked at past experiences with similar "ports" and decided that the problems could be fixed; we had done it before. So we decided that we would use the reference implementation as the base and fix the problems—this would be less expensive than attempting to build a Java-compliant virtual machine "from scratch" as the Operating System/400* (OS/400*) team had done.

The problems centered around initialization, locking techniques, memory management, excessive path lengths to execute service calls, inefficient coding techniques, excessive requests for system information, and inefficient handling of data types. Once the up-front analysis was completed, we put the teams in place to address these problems. In the next section we describe problems in two areas—performance and functional capabilities—and the solutions that we applied. Next we discuss the tools that were used to analyze the path length of the virtual machine functions. We then describe how we increased performance in the just-in-time compiler. The following section delineates fundamental design lim-

itations of the Java virtual machine that needed to be overcome to allow server transactions to run efficiently. We then outline a solution to allow the virtual machine to run server transactions. Finally, we summarize the work that has been done and discuss additional problems to be addressed.

The Java virtual machine on OS/390: Overcoming weaknesses

As originally ported to the OS/390 environment, the Java reference implementation (Java Development Kit 1.0.2) had a number of limitations in the general categories of efficiency, capability, and serviceability. In the first category are the areas of locking, memory management, and use of system services.[3] Each of these implementation-dependent areas may decrease the potential performance characteristics of the Java environment.

Examining the functional characteristics of Java and OS/390 exposes differences in what each environment expects. For example, the Java floating point data type was not natively available on OS/390 when Java was first ported. Not only did this impose a development cost to provide equivalent software emulation, but it also imposed a performance penalty for use of this data type prior to the introduction of native machine support.

Finally, the Java and OS/390 environments provide different techniques and assumptions for problem determination. As an environment that has provided industrial-strength availability and reliability for over a quarter century, OS/390 has developed technology that supports the capture of problem-related data, recovery, and problem determination while the existing workload continues to run. In order for the Java environment to be an effective server on OS/390, a number of enhancements were necessary, as summarized in Table 1.

Locking and serialization. Serialization within the Java environment for both applications and a number of run-time environment control structures is provided by a construct called a monitor. Monitors support only exclusive ownership. Since there is no shared serialization mechanism available in the Java environment, functions that access a number of structures in read-only mode are forced to obtain exclusive ownership of a monitor. This overserialization within the Java run-time environment (and applications) inhibits throughput and scalability due to increased contention, and increases overhead due to the cost of extra services being invoked.

Two of the most frequently obtained system monitors, i.e., the NameAndTypeHash lock and the LoadClass lock, were effectively eliminated by introducing read-only functions for the NameAndTypeHash table and by searching the BinClass table at first under only the BinClass lock (and without holding the LoadClass lock) when determining whether a class was already loaded or not (along with some other minor supporting changes).

In the Java reference implementation, the monitor *obtain* and *release* functions use the object handle address as a key into a table of system-specific monitors. Each monitor use requires a hash table search for the key prior to the monitor call. Not only is this preliminary hash table lookup costly, but the monitor functions themselves are also expensive. For example, the monitor mutex[4] must be obtained before any attempt is made to update the monitor owner field, so there are two serialization entities to be obtained for each monitor obtain request.

In the IBM Developer Kit for OS/390, Java Technology Edition, v 1.1.4, we introduced a locking scheme where a compare-and-swap technique was used to verify that no other thread currently held a monitor and that the monitor had not been inflated[5] in the past. If both these conditions were met, the thread obtained ownership of the monitor without other costs normally involved in obtaining a monitor. In the Developer Kit (DK) v 1.1.6, support was added to allow monitors that did not meet the conditions for "compare and swap" to be obtained in some cases without obtaining and releasing a mutex.

Compounding the problems described above, the reference monitor implementation contains a "spin loop" when there is contention for a monitor. This means that in contention environments application programs are spinning, consuming precious processor cycles, while other threads hold the mutex. In DK v 1.1.8, the OS/390 Java virtual machine has greatly reduced the number of iterations in this spin loop.

The thread queue (TQ) single lock is a synchronization mechanism used in garbage collection (GC) to coordinate the various garbage collection phases with all other threads. Originally, it was used in the "wait handshake" between GC and any thread going into or coming out of a wait; i.e., the TQ single lock was originally obtained and released by a thread both

Table 1 Java virtual machine improvements

Problem Category	Problem in Prototype Implementation	OS/390 Solution	IBM DK for OS/390
Locking/serialization	Unnecessary serialization	Some locks removed	v 1.1.1
	Excessive cost to obtain monitor/mutex	Compare-and-swap technique	v 1.1.4
		Mutex not always obtained	v 1.1.6
		Spin loop interactions decreased	v 1.1.8
Garbage collection/ memory management	Objects with handles	JIT compiler generated objects without handles	v 1.1.1
	Stop-the-world garbage collection	Used OS/390 concurrent garbage collection	v 1.1.4
		Used OS/390 generational garbage collection	v 1.1.6
Use of system services	More function in OS/390 services than Java needs	Interfaces added to low-level OS/390 functions	v 1.1.4
Threading	Extra overhead for thread support	Added function saves/restores thread pointer	v 1.1.4
EBCDIC/ASCII	Incompatible formats between Java and OS/390	ASCII converted to EBCDIC only as required outside JVM	v 1.1.1
Java data types	Floating point not compatible with OS/390, S/390 hardware	S/390 hardware changed, support added to OS/390	v 1.1.6
Security	Layers of defense vs principal-based access control	Added interfaces support principal-based access control	v 1.1.8
Performance management	No support for performance objectives	Added interfaces support performance objectives	v 1.1.8
Serviceability	Missing diagnostic information	JIT compiler changed to save diagnostic information	v 1.1.8

before and after going into a wait in order to know what phase GC was in. In DK for OS/390, v 1.1.8, threads use a compare-and-swap technique to coordinate with the GC thread, and the TQ single lock is primarily used only by the GC thread. This means that threads going into a wait will no longer cause contention on the TQ single lock.

Garbage collection and memory management. The Java reference implementation allocates objects with an associated, but discontiguous handle. This makes references and updates to the object slower, since it involves a level of indirection to first go through the handle. The use of handles facilitates garbage collection, but poses problems for memory management. For example, how should the heap be subdivided between space for objects and space for handles? Object allocation, which is a high-frequency event, is also slower since it involves two distinct allocations. Since the handle and the object may well be in distinct pages, increased paging may result; certainly the size of the working set will increase. Objects with handles are a disadvantage for the just-in-time compiler. According to estimates, avoiding handles will result in a 10 percent speedup in allocation,[6] and Java HotSpot**, Sun's new implementation of the Java virtual machine, does not use handles.

Since the Java language does not provide an explicit mechanism to free objects, it is dependent on the Java run-time environment to reclaim storage when needed. While there are many garbage collection algorithms, the Java reference implementation uses a "stop-the-world" technique—the garbage collection thread stops all other threads before it proceeds. While this avoids most serialization issues and is conceptually simple, it imposes a severe limitation on

scalability in a multithreaded environment, which the Java language supports. Starting with DK v 1.1.4, OS/390 provided a concurrent garbage collector. The garbage collector allows worker threads to continue processing while garbage collection is being done. Starting with DK v 1.1.6, the OS/390 garbage collector has also been generational, frequently cleaning up objects that were allocated recently and only occasionally doing cleanup for all objects. A large number of other improvements in garbage collection and object allocation have been made throughout the DK for OS/390, v 1.1 releases. For more information on these changes, please see the sections on garbage collection in other papers in this issue.[7,8]

Use of system services. One of the challenges in porting the Java environment to any platform is to implement Java application programming interfaces (APIs) in terms of native interfaces to actual system interfaces. In some cases, the existing system interfaces may provide much more function (and path length) than is actually needed (or desired) for the Java environment. To cite a concrete example, the identity and state of a Java thread must be maintained independently of any underlying system thread construct. One frequently executed service is sysThreadSelf, which returns the address of the Java control block for the currently executing thread. A standard UNIX API, pthread_getspecific, may be used to extract a value previously saved via pthread_setspecific. One of the changes to DK for OS/390, v 1.1.4 was to create an assembler interface to save and restore the Java thread pointer in a UNIX-related thread control block, resulting in about a 10 percent performance improvement.

Threading. The Java language supports a multithreaded environment, so it is important for each platform to provide an efficient mapping of the Java thread construct to the underlying system thread construct. Thread support traverses many layers: from the application, through various Java classes (e.g., Thread and ThreadGroup), then through the platform-independent layer, the platform-specific layer, and finally to the operating system itself. While this layering provides a clean separation of responsibilities and allows many platforms to "plug in" their support, it can also contribute to extra overhead. Since the Java run-time environment has its own view of a Java thread, which is independent of the operating system view, there are some redundancies in control information and some cross communication for coordination. This causes the performance problem described earlier relative to saving and restoring the Java thread pointer.

EBCDIC code page used by OS/390. An area of consideration when porting any C code from another platform to OS/390 is the difference in the encoding of character data. Many platforms support character strings in ASCII, in fact most Transmission Control Protocol/Internet Protocol- (TCP/IP-) based services, including the Internet, and most other platforms that the Java environment has been ported to are ASCII-based. The ASCII format[9] is so pervasive it is almost considered to be platform-independent.

The OS/390 platform uses the EBCDIC format, and this presented one of the biggest challenges in porting the C portion of the Java reference implementation to the OS/390 platform. We decided to encode character strings in the C code in ASCII, rather than EBCDIC format. This approach involved drawing a virtual boundary around our Java virtual machine (Jvm); all characters within the virtual machine were in ASCII and were only converted to EBCDIC when they needed to be, such as when printing to the screen. This enabled the Jvm to operate as though it were running on an ASCII-based platform and circumvented any assumptions within the code that characters were in any particular format. This also benefits Java programmers, since the Jvm handles the differences between ASCII and EBCDIC formats.

Character strings in Java code are not in either ASCII or EBCDIC format, but in a platform-independent character encoding called Unicode**. The Java programming language is unique in having chosen this encoding. Since most platforms do not support Unicode, data passed into and out of a Java application must be translated between the local machine encoding and Unicode.

In the Sun Java Development Kit (JDK**), v 1.0, streams were available to read in or write out data, but they handled only binary data that required no translation. They could be used to read in ASCII characters, because ASCII and Unicode are for the most part very similar and therefore translation was often not necessary. They did not provide the means to read in EBCDIC data.

JDK v 1.1 introduced the internationalization feature, which provides national language support (NLS). This is ideal for OS/390, since this feature provides the mechanisms to translate between a whole range of

different encodings and Unicode, thus allowing Java applications to access EBCDIC data. These data could be in existing OS/390 subsystems, such as DATABASE 2* (DB2*).

Support for Java data types. Another platform characteristic that we had to deal with was the different way that the OS/390 platform represents various types, specifically single- and double-precision floating-point values. The Jvm requires that floating-point numbers conform to the IEEE (Institute of Electrical and Electronics Engineers) Standard for Binary Floating-Point Arithmetic (ANSI [American National Standards Institute] IEEE Standard 754-1985), which defines the format of 32-bit and 64-bit floating-point numbers and the operations on those numbers. Native OS/390 floating-point numbers are different from this standard.

The Java security model differs from that of the OS/390 environment in a number of significant aspects.

Our solution was to provide emulation for these numerical types in the OS/390 version of the Jvm. This meant writing code to create and manipulate our own versions of these numerical types that conform to the IEEE standard. As with our ASCII emulation, the Jvm implementation operates internally with these emulated types, only converting between one type and another when needed.

Java developers only need to be aware of this if they are coding native methods, that is, they need to be aware that these types are different and convert between the two types as appropriate. For a user running a Java application, this emulation of floating-point numbers is transparent. However, there is the consideration of performance. It is undoubtedly slower to perform manipulation of floating-point numbers in software rather than hardware. With the System/390* (S/390*) Generation 5 (G5) hardware, in conjunction with OS/390 Version 2 Release 6, support for these numerical types has been provided, removing the need for our emulation. This native support for IEEE floating-point operations resulted in a huge performance improvement for floating-point-intensive Java applications.

Security. The Java security model differs from that of the OS/390 environment in a number of significant aspects. Java security is based on several layers of defense. First, there are the restrictions in the language itself. For example, pointers are not allowed, which implies that memory cannot be directly accessed, and array limit checking is enforced so that programs cannot go outside the true range of the array. Second, there is byte-code verification of class files during class loading. This verification ensures that the program cannot branch outside its current method except via a formal method call, cannot overflow (or underflow) the stack, and must pass correct data types on method calls. The third mechanism involves the use of a different name space for each class loaded by a different class loader. The fourth line of defense lies in the SecurityManager class, which enforces access permissions. A security policy may be created that keys off the class loader or that keys off certain class file attributes, such as where the class came from.

Security in an OS/390 environment depends heavily on the concept of principal-based access control, i.e., the association of an identity or "userid" with a request to access resources. Despite the existence of the SecurityManager class, the base Java environment does not provide APIs to native security interfaces that implement principal-based access control. There is no Java API to extract the userid in effect for the currently running thread or to check that the userid has access authority to a given resource.[10]

When a Java application creates other threads in OS/390 environments prior to Release 8, there is also no way for the principal userid of the parent thread to be propagated to new threads that are created, even though these threads are all processing against a single work request on behalf of the same userid. Instead, these new threads will have the security permissions of the server address space in which they are running, which is generally different from the permissions accorded the userid associated with the parent thread.

The IBM Developer Kit for OS/390 has been enhanced in the area of security, both in respect to how the Jvm itself operates and with the provision of Java APIs to allow Java programmers to provide security

functionality within their Java applications. A number of UNIX services and the Java Event Handler was provided in DK v 1.1.8 to allow the security identity of the primordial thread to be propagated to new Java threads. This even allows the primordial thread to create the new threads for a given work request to be completed asynchronously, so the primordial thread could pick up a second work request (with a new security identity) and create new threads for the second work request with the second identity.

We also provided two sets of Java APIs. The first is the security migration aid, which allows users to exploit the security enhancements provided by the Java 2 Software Development Kit, Standard Edition, from within the Java Development Kit (JDK) 1.1 implementation. This feature assists users who wish to migrate from the relatively simple JDK 1.1 security model to the finer-grained Java 2 model. Also available is a set of Java APIs called "Java for OS/390 Security Services." Initially these APIs provided access to a basic set of existing OS/390 UNIX APIs that are required to implement principal-based access control in a Java application, for example, an application that implements a Java SecurityManager class. These APIs have been enhanced to provide user authentication functions, and further additions and improvements are expected.

Performance management. One of the strengths of OS/390 is its ability to manage and balance the use of system resources toward a set of installation-defined performance goals for the entire workload that happens to be present at any given time. This management applies to all work requests in the system. However, within the original ported Java environment running in OS/390 environments prior to Release 8, there is no way for the performance objectives for one thread to be propagated to new threads that are created, even though these threads are all processing against a single work request. Such propagation is standard for existing transaction managers, such as CICS* (Customer Information Control System), IMS* (Information Management System), and DB2, batch environments, and interactive environments such as TSO/E (Time Sharing Option Extensions), UNIX/390 processes, and Web servers. While this may not be an issue for a Java client environment running on a workstation, it is a critical attribute for a server environment such as OS/390.

The support described in the subsection on security—new OS propagation services and the Java Event Handler—allows the performance goals of the primordial thread to be propagated to new Java threads, with the same ability to support asynchronous work.

Serviceability. As OS/390 provides the stability, availability, and reliability required by corporate business environments, it is critical that a variety of workloads be able to coexist without compromising these attributes. It is not acceptable for the introduction of a new Web page, accessed by tens of thousands of Internet users, to affect corporate applications running on the same machine. It is furthermore necessary that problems that arise in a Java server environment be capable of having sufficient diagnostic information captured to allow meaningful problem analysis to be performed.

A number of problems did arise from the introduction of the early Java environments into existing OS/390 environments. Several of these relate to Jvm messages and other diagnostic information, which are more oriented toward a client workstation than a high-volume server environment.

The first problem is that Jvm messages, which have no unique identification such as message numbers, intermingle with messages produced by other applications and the operating system. This may not be a problem in a Java client environment running on a workstation, but it is definitely a problem in a server environment where thousands of messages are produced each second. OS/390 provides a message automation facility that depends on message identifiers to filter and then act on messages so that they need not be presented to a human operator. The absence of message identifiers impedes message automation and unattended operations.

One of the key serviceability tools available in the OS/390 environment provides the ability to look up known and ongoing problems to determine duplicate problem conditions, to learn the status of fixes, bypasses, available and target fix dates, and so forth. However, Jvm messages are too generic to use as a search argument, which renders the problem database essentially useless for this purpose.

Finally, Jvm messages are not NLS-enabled. One of the tools in the OS/390 environment allows message text replacement in alternative languages, which facilitates the use of programming products outside the English-speaking community. Lack of this capability for Jvm messages increases the difficulty for the non-English-speaking community to fully and ef-

fectively attain the benefits available to the Java community.

A further problem is that Jvm return codes are not unique. For example "out of memory" can represent any one of 27 different problems, including "out of threads." Further effort is required to find the exact cause of failure. More granularity is needed so that problem recreation is less likely to be needed for problem determination.

The original Java reference implementation, JDK 1.0.2, did not have complete information about the build level. For the person trying to analyze a problem, the difficulty is compounded because several versions may be running concurrently on a single system. When the build level is unknown the code base is unknown, making it much more difficult to find duplicate problems in the problem database and impossible to know what Developer Kit source code to scan for potential new problems.

Another problem concerns application debugging. The just-in-time compiler does not keep line numbers. Thus, even if the Java source code is available, all that can be seen is the compiled code. In the base Java environment, JNI_PANIC may be issued as a result of errors in the user's Java Native Interface (JNI) code, but this does not cause the environment to be captured in a dump.

Once a dump is taken in an OS/390 environment, IPCS (Interactive Problem Control System) is used to browse it on line. However, in versions of OS/390 prior to Release 5, IPCS assumed that it was interpreting either hexadecimal or EBCDIC data. When the Java reference implementation was first ported to the OS/390 environment, there was no way to format ASCII data (e.g., control blocks) within IPCS.

OS/390 system dumps are viewed using IPCS. IPCS provides a set of standard formatters to analyze system data structures, such as stack and heap control blocks, and to format the data for easier display. IPCS also supports custom formatters for user data. In earlier versions of OS/390, there were no formatters for Java data structures to aid in debugging Java problems through an OS/390 dump.

To improve serviceability for Java problems on OS/390, a series of features were added. Traces produced by the just-in-time compiler are now held in memory buffers for improved efficiency, and for ease of problem determination, trace output from different threads is written to separate files. Additional options can now be specified to generate more information in dumps. Dumps can show all active threads as opposed to just the current thread. User ability to generate an abnormal termination and produce a dump has been also provided. Further improvements will continue to be added, including improving messages issued from within the Jvm, and providing greater granularity to common messages, for example, OutOfMemory exceptions.

Use of OS/390 facilities to improve Java for the OS/390 platform

To address the performance problems discussed in the previous section, we needed to find the areas where the OS/390 Jvm was spending most of its time. This section describes the strategies and path-length tools used to determine the areas on which to concentrate. Since the OS/390 Jvm is implemented primarily in C, we can use OS/390 tools to investigate and improve Java performance on the OS/390 platform. We followed a simple but effective methodology to identify areas of potential improvement. First, we identified benchmarks and microbenchmarks that were appropriate to our platform. For example, we focused on benchmarks that were server-based rather than presentation-based, since we expect presentation to be done in a client environment rather than on the server host. We were aided in this by IBM's cross-platform Java Performance Team.

We next fixed the OS/390 platform as a baseline to understand our starting point and to serve as a basis for determining the effect of any improvements. We then looked at general platform facilities, such as advanced code generator optimization and "inlining," as a first step to performance improvements. Next we looked for code "hot spots" of CPU usage and "cold spots" where code was waiting for locks or other synchronization. The goal was to identify candidate changes where we could move frequently called routines in line, eliminate the calls completely, or speed up the routines by improved algorithms, such as substituting hashing for linear searches, eliminating unneeded locks, and generally reducing CPU time spent in the locking process.

Once candidate changes were identified, alternative solutions were proposed. For larger changes expected gains were estimated, and the most promising were modeled and introduced into the product through the development process. Estimating gains

was a challenge because of the many interacting effects of the changes under investigation.

Our primary benchmark for DK v 1.1.x performance investigations models the internal scalability aspects of a warehousing application, without doing real database I/O or having real terminal interactions. The OS/390 C/C++ Performance Analyzer, currently available in a beta version, was important to our analysis of benchmark behavior. This tool supports both function profiling and interval sampling. Function profiling was very helpful in determining how often routines were invoked, how much CPU time was spent in each, and how long the routine was on the call stack. The output from the performance analyzer was useful as input to spreadsheet models that we used to estimate the effect of improvements on throughput. Interval sampling was also helpful in finding hot spots and cold spots in execution and is generally less intrusive than profiling, which requires modification of the executable code. We also used proprietary, confidential tools that duplicate the function of profiling and tracing but are even less intrusive. One set of proprietary tools allowed us to determine, on a system-wide basis, what modules, control sections, C functions, and methods compiled "just in time" consumed the most CPU time. We examined these functions first to search for performance improvements. Another set of proprietary tools allowed us to monitor the instruction paths taken to perform a specific Java function. Instruction trace data generated by these tools made it much easier to locate and correct specific performance problems. For certain types of high activity we also generated our own instrumentation in the Jvm. One such special Jvm allowed us to collect information about the activity of various Java monitors during benchmark runs.

An example of the kind of improvement opportunities we found was in looking up thread identifiers. We noticed a relatively high percentage of CPU time spent in calls to a routine that extracts the Java thread identifier, which on OS/390 is the TCB (task control block), to an index into an internal Java thread table. The search for the thread identifier was linear. Once the area was identified, we were able to take advantage of OS/390 facilities to provide the Java thread identifier after a number of "fetches" anchored by a pointer in the TCB, rather than by searching. So the call and the search were both eliminated, for a significant performance gain.

Our use of these tools has allowed us to find significant throughput improvements in the OS/390 Jvm and to identify improvements in the underlying OS/390 operating system as well.

The Java for OS/390 just-in-time compiler

A just-in-time (JIT) compiler is a code generator that converts Java bytecode into machine language instructions.

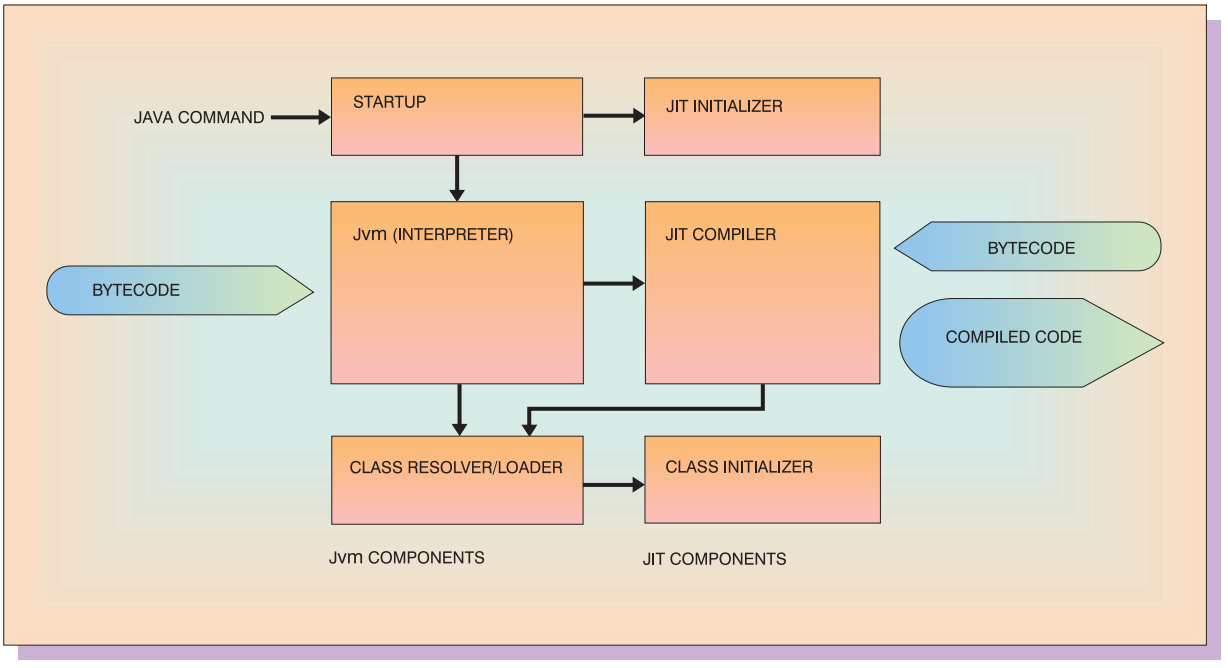
Within the Jvm (Figure 1), two different techniques exist for executing programs in Java bytecode form. The first and most basic alternative is bytecode interpretation based on a rote implementation of behavior as defined by the Java Virtual Machine Specification. With an interpreter, bytecode instructions are fetched and then executed sequentially, one at a time. This is generally an expensive approach. A just-in-time compiler provides a higher performance alternative.

With JIT compilation, a two-phase approach is used. First the program's bytecode, or a subset of it, is compiled into machine language instructions, then the newly generated instructions are executed. The JIT compilation applies many of the same optimization techniques used by standard compilers, including the elimination of redundant code. Often the generated results are cached for subsequent reuse.

JIT compiler structure. The JIT compiler for OS/390 (Figure 2) is structured in the same way as many traditional compilers supplied by IBM. There is a common front end that applies platform-independent optimizations, such as code motion, loop optimization, and "inlining" analysis. Its output, in an internal format, is input to the JIT back end. The JIT back end is specific to each platform. It translates the internal format into executable S/390 machine instructions. These machine instructions are saved. Methods that are re-executed use the saved code.

For S/390 the specific instructions generated in the back-end translation are machine-dependent. The S/390 JIT compiler is common to all levels of the OS/390 operating system. It supports any of the machine families where OS/390 can run. Differences in machine designs and available instruction sets are recognized. These are reflected in the generated machine code, through adaptations in instruction choice and instruction ordering. Instruction choice and ordering are based on machine design, taking into account characteristics such as pipelining effects, cache behavior, and relative instruction execution speed. An additional factor in instruction choice is the avail-

Figure 1 Just-in-time compiler

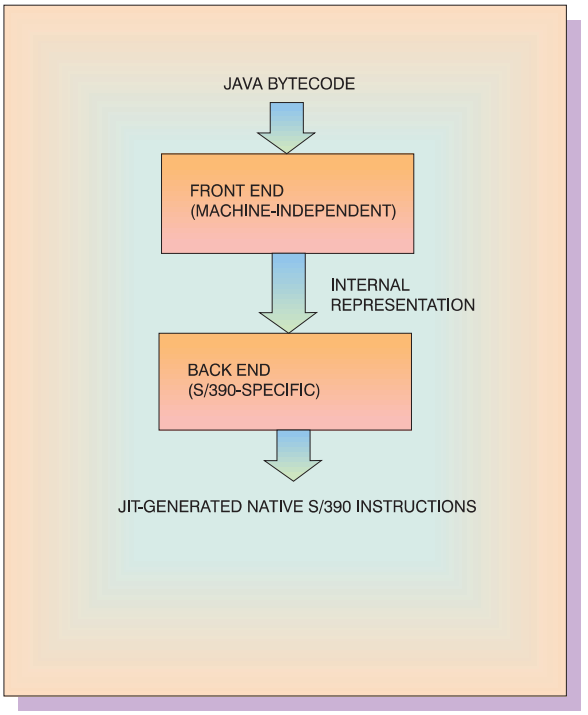


ability of specific instructions within each machine family. Over time the S/390 architecture has grown.

During its initialization the OS/390 JIT compiler tests for the availability of instructions like the relative branches and immediate operations first available on the Generation 2 processors, or the IEEE floating point standard instructions added with Generation 5 processors. In the first case, the result will be changes to register assignments done in the back end, which effectively expand the register set, and in the second case, the emulation routines are replaced by native instructions.

Design trade-offs. Space and time design trade-offs within the JIT compiler for S/390 have been made based on the normal expected usage profile, for OS/390, of the Java language. The expectation is that it will typically be used for relatively long-running applications, often in conjunction with a Web server as, for example, a Java servlet. There is an emerging demand, as well, to run more traditional batch-like OS/390 applications in the background, such as report generators, which are often batch-like. In these cases heavy code reuse is common. Given this, the emphasis is on generating high-performance machine code that is cached for re-execution. This higher up-front

Figure 2 Just-in-time compiler for OS/390



cost in code generation results in lower execution time.

For example, in the front end, loop versioning and inlining are commonly done at the expense of increased memory use. As method inlining is done more aggressively, the scope of data flow analysis can become larger, potentially crossing multiple methods. Endleaf routine[11] analysis for stack and linkage optimization is another front-end technique that trades compile time for execution time.

In the back end, code is ordered to optimize branch behavior, code and data are separated, additional inlining including monitors, object allocation, and other system methods is done, and the back end now includes a multipass code generation and compression algorithm based on optimizations on a linked list of pregenerated instructions that are copied to the execution buffer.

These optimizing techniques can lead to both a relatively long compilation time and a relatively large memory footprint. The approach begins at Jvm startup, at which time all methods are loaded and compiled. Virtual memory consumption is traded off in anticipation of reuse. These are not the correct tradeoffs for applications with limited reuse; consequently applications with limited reuse get little benefit from the extensive improvements made since the JIT compiler's inception. Common cases where there is limited reuse are (1) source code compilation, (2) Jvm initialization, (3) applications with short duration, and (4) parts of applications that may be executed infrequently, such as prologue and initialization code, administration code, methods used by exception handling, and termination code.

The addition of a mixed mode interpreter (MMI) with the DK v 1.1.8 level is intended to improve performance by as much as a factor of two for typical Java applications, expanding the set of well-performing Java applications on OS/390. MMI keeps a history of methods executed, and until a threshold value is reached, methods are interpreted. For methods that are executed a small number of times, the cost of compilation can outweigh the benefit of limited (or no) reuse. After reaching the execution threshold, the method is compiled and cached. Before the introduction of MMI, because all methods were compiled before first use, interpreter performance was largely irrelevant. With MMI, the interpreter will be used until the threshold is reached. Consequently, the interpreter was rewritten in assembler code to optimize its performance.

OS/390 JIT compiler chronology. There have been extensive improvements over time throughout the development of the JIT compiler. Their net cumulative effect is a typical performance improvement for compiled code vs interpreted code. The basis for OS/390-specific items is described in other sections of this paper. General optimization techniques are described by Suganuma et al.[12] Table 2 gives a chronology of the changes to the JIT compiler since its inception. The OS/390 optimizations concentrate on key themes: to minimize linkage costs, to optimize code generated against the S/390 hardware, and to use the lowest cost options for system or library services. There have been iterative improvements in all these areas, and for this reason some items in the table occur multiple times.

JIT compiler future. We anticipate making further enhancements.

On the immediate horizon are:

- The use of an expanded intermediate language similar to conventional IBM optimizing compilers. This will increase the scope of optimization possible in generating machine instructions in the JIT compiler back end.
- Continued refinements to instruction-level optimization, in particular to support the machine organization and execution characteristics of new processor families. These will include: code scheduling to include cache line size recognition and AGI (address generation interlock) avoidance, branch optimization, and register usage optimization.

In the longer term there will be continued refinements in support of our design goals. New enhancements may result from current research. These may include:

- Profile-directed translation that will apply compilation to areas where greatest benefit is expected. This might be accompanied by multilevel dynamic compilation based on a dynamic cross-system cost/benefit estimation.
- Background compilation using dedicated Java accelerators (additional hardware dedicated to background JIT translation).
- "Persistent" code, with additional compile-time optimizations borrowing from conventional compiler technology applied. The goal is to retain optimized

Table 2 JIT compiler improvements. The initial version, available 9/97, included a tightly connected front and back end with basic code generation and caching ability.

IBM DK for OS/390	Date Available	Cumulative Front-End Improvements	Cumulative Back-End Improvements
v 1.1.1	02/98	Bytecode translation by idiom Common subexpression elimination Loop detection	Use of handleless object model, eliminating one level of indirection for object access, and optimizing array checks (Note: This was done in conformance to the object model then in use within all IBM Jvms.) Locking improvements Method call improvements Array handling improvements Register assignment for loop handling
v 1.1.4	05/98	Flow analysis to remove redundant null pointer checking	Class unloading Improved instruction choice and branch optimization to fit machine design Monitor inlining
v 1.1.6	12/98	Limited static and virtual function inlining Increased data flow analysis	Use of immediate/relative instructions for G2 and above machines Use of native IEEE floating point instructions Optimized linkage for JIT-compiled method calls Code and data separation Further register use optimization Extraneous instruction elimination Tailored instruction sequence to pipeline behavior of G5 machine family
	04/99		Frequently used functions built in Use of native libraries for function calls Caching of method results
v 1.1.8	07/99	Extensive inlining for static and virtual functions Endleaf routines in stack Data flow analysis including array bounds check	Extensive linkage generation refinements Multipass code generation optimization with extraneous instruction removal Exception handling with native services Object allocation inlined Mixed mode interpreter with converged stack use "Peephole" analysis

translated code for reuse across multiple execution instances. This is motivated by server workload characteristics, where there is repeated execution of self-contained application code. This is particularly applicable for transactional content where relatively small pieces of application code can be frequently executed by, or across, multiple Jvm instantiations. For such applications, the increased processor time investment at translation time will be returned in good measure by the high frequency of execution of the translated and stored application code.

JIT compilation. JIT compiler performance is central to Jvm performance on OS/390. Since its inception in late 1997, there has been a multifold improvement in execution time for JIT-generated code. We expect to continue making progress based on the goals that have driven the improvements so far. These will be integrated with technology derived from current IBM research efforts focused on improvements common across IBM JIT compilers and specific to S/390.

A Jvm for server applications

We looked at the characteristics of existing transactional programs in OS/390 environments (CICS, DB2, IMS) to drive the requirements for a server Jvm. The requirements of reliability, security, clean Java heaps, and reinitialized writable static areas drove the need for a Jvm that can span multiple processes with a

transaction running in its own process. The requirement for transaction-level throughput, not wasting CPU cycles reloading, relinking, recompiling, reverifying, and reinitializing hundreds of static variables and Java classes over and over again, drove the invention of a *shared heap* that worker Jvms could initialize from and execute out of. The demand for transaction-level isolation, and the need to quickly clear the working heap of a worker Jvm, motivated the design for a private heap that could instantly be cleared to save the cost of garbage collection. These requirements from commercial transactional monitors remain the same for Enterprise JavaBeans** workloads and indeed benefit Java server applications on all platforms.

Sun's Java virtual machine is created within a process to load and run an application. At the end of the application the entire process and run-time environment is torn down. As more transactions in Java are written, it is desirable to have the virtual machine stay up to execute the next application. This would save the cost of tearing down and starting up the process, which is expensive. Having a long-running, reusable virtual machine would also help save the cost of class linking, loading, and initialization, which are also expensive. Not having to create the virtual machine environment for every application means that a system can manage more volume and throughput of applications. Currently, the Java virtual machine is used for a single process; it begins when the application begins and ends when the application completes. Earlier ideas for keeping the state of a virtual machine are: maintaining a pool of virtual machine processes,[13] checkpointing a virtual machine's state,[14] reusing the same virtual machine process for multiple applications, and maintaining a pool of virtual machines and sending objects created in one machine to the heap of another machine.[15]

Maintaining a pool of virtual machines does not diminish the initialization path length of a virtual machine. Scheduling applications in a previously created virtual machine hides the path length from a client request, but does not reduce class linking, loading, or initialization requirements, nor does it obviate the need to bring up and tear down a process for each application.

Checkpointing a virtual machine's state and applying the state to new processes require pointer and offset readjustments that are extremely costly in path length and may not be possible in systems where the range of addresses a process will command cannot be guaranteed.

Reusing the same virtual machine process for multiple applications does not guarantee a clean heap or writable static area for each application that runs consecutively in the virtual machine. An erase-memory mechanism that tracks all variables or objects that were changed, created, or deleted in the Jvm with the goal of restoring them to their original values would involve intervention on every "putxxx" operation code in the Jvm. This would degrade performance.

Maintaining a pool of virtual machines and shipping the application to the correct virtual machine, or sending the correct object to the virtual machine where the application is running, requires a cache coherency scheme and incurs extra overhead and possibly network flows to pass the application or object. Moreover, this scheme does not guarantee that the memory space in each virtual machine is free of values left by previous applications.

An implementation that can keep Jvm initialization state across invocations, amortize class loading, linking, and initialization cost, avoid garbage collection and process termination, and provide greater application security, isolation, and availability is described in the next section.

A scalable Java virtual machine implementation

Execution of a Java application involves creation and initialization of a new instance of the Java virtual machine and execution of the application class file within the Jvm's context (see Figure 3). The Jvm thread bootstraps itself with the callee process (e.g., a JNI application) and loads various Java system classes to set up the context. Studies of Java applications running under OS/390 UNIX have revealed that Jvm initialization loads and resolves 60 system classes, allocates over 1000 objects that are not arrays and uses over 700 array objects. The Jvm initialization contributes significantly to the overall execution cost of the Java application. Further, the Jvm initialization process is the same regardless of the Java application being executed, and the majority of classes used in the Jvm initialization are not used by the target application.

One way of reducing the Jvm initialization cost is to create new Jvm processes using already-resolved sys-

tem classes. In this approach, only one Jvm process will load and resolve the necessary system classes (resource-owning Jvm). Future Jvm processes (worker Jvms) can reuse the resolved classes.

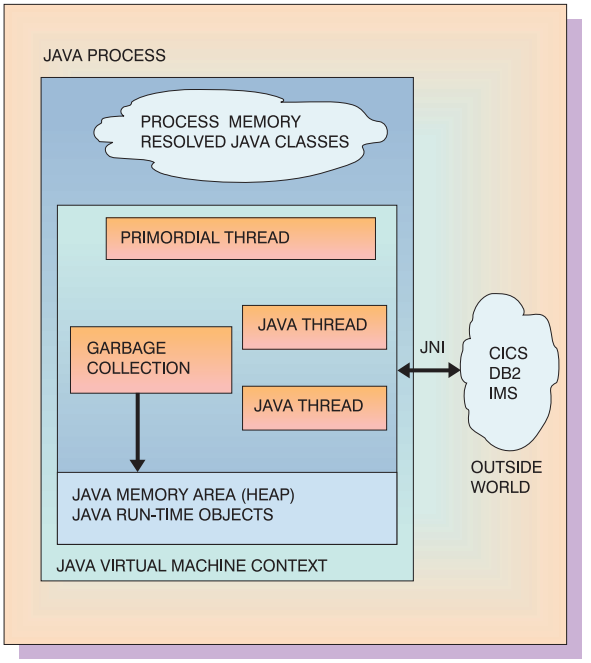
Multiple worker Jvms can now share system data (e.g., classes, method tables, constant pools, field blocks, etc.) using a shared memory region, called the shared heap (see Figure 4). The shared heap is designed to store system data; however, it can also store application data that can be reused across multiple workers. In addition, individual worker Jvms maintain local memory regions to store data private to the Jvm process. These regions are called the local heaps. For the initial scalable Jvm design, shared heaps will not be subject to garbage collection or be expanded at run time. When CICS or DB2 starts a resource-owning Jvm, the shared heap will be large enough to hold the Java system and application classes needed to run a CICS Java or DB2 SQLJ[16] program. CICS and DB2 SQLJ programs are expected to run for short durations (seconds or milliseconds) and not allocate thousands of objects. When the application is finished, the local heap will be reinitialized.

The shared heap is partitioned into two regions: (1) the data structure (DS) area stores static data structures and (2) the shared run-time (RS) area stores the shared system and application classes and other auxiliary data structures. The local heap contains data private to an individual Jvm instance. The local heap can be modified only by the threads created in the process (worker Jvm) that created this local heap.

The modified Developer Kit supports three Java execution modes: the traditional mode that bypasses the modifications, the Java resource-owning mode, and the Java worker mode. All three modes are entire Jvms in themselves and are Java-compliant. The resource-owning Jvm and worker Jvms can be selected either at the UNIX/390 command line prompt, e.g., Java -server or Java -client, or in the JNI mode, by setting the appropriate JNI argument (i.e., is_server or is_client) in the native program. The resource-owning Jvm allocates the memory region that serves as the shared heap and creates the data structures needed for locking, class location, and instantiation. Worker Jvms can use a common class loader to share name spaces across a set of Jvms.

A worker Jvm allocates a local heap for storing temporary hash table entries, temporary strings, and data associated with local objects. A worker Jvm uses the shared heap to load, link, verify, and compile classes.

Figure 3 Traditional Java execution environment

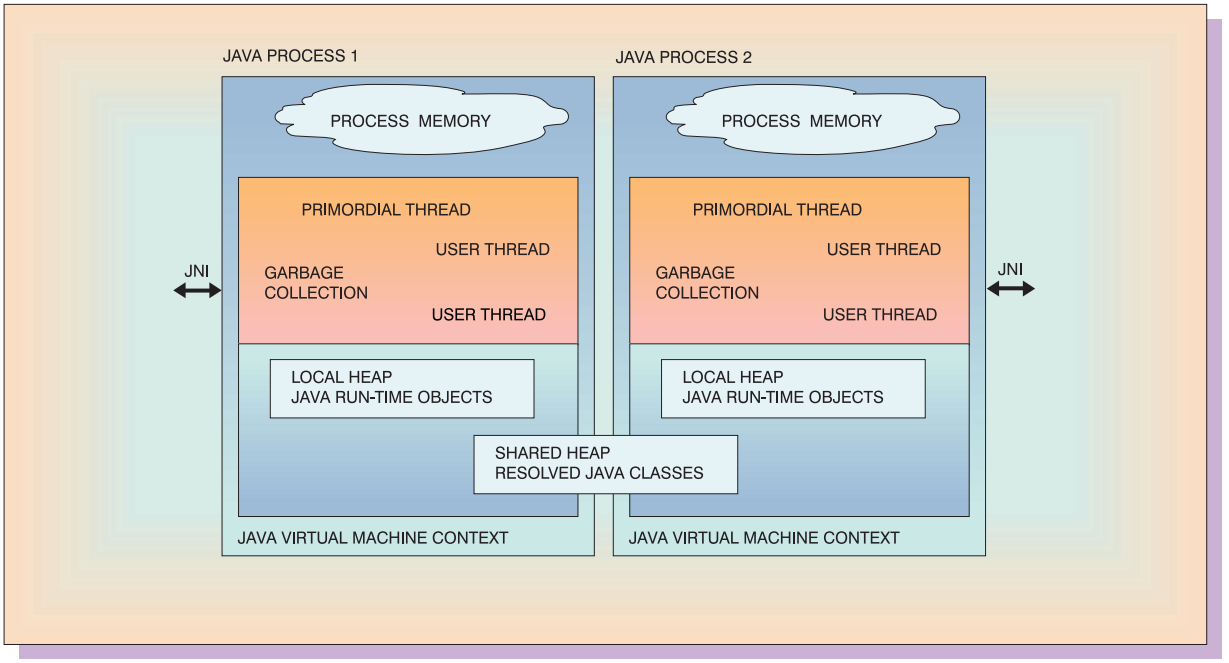


A worker Jvm need not load, link, and instantiate any class that has already been loaded by another worker Jvm. This includes the 60 system classes every Jvm needs for initialization—these have already been loaded by the resource-owning Jvm. Thus, all worker Jvms are saved that costly initialization sequence.

To further reduce the path length of starting up a worker Jvm, a "clean slate" design is used to ensure a clean heap. Each worker task executes a large outer loop where it creates the worker Jvm and its environment. Within an inner loop, the worker Jvm will execute a Java application, and then determine whether that application was well-behaved. An application is considered well-behaved if it has left no residual resources behind (e.g., threads, open files, storage, etc.).

If the application was well-behaved, then the Java heap and other control variables will be reinitialized to set up the correct environment for the next work request. The Jvm worker process is not terminated—it is reused. This inner loop continues, reusing the Jvm worker process until some ill-behaved application runs that causes the inner loop to termi-

Figure 4 Multiheap multiprocess Java virtual machine



nate. At this point the Jvm worker process will be terminated in order to reclaim all related resources, and a new Jvm worker process will be created. The new Jvm worker process need not re-execute the millions of instructions to reload, link, and allocate the 60 system classes required for initialization.

In an ordinary Jvm, the static initializer for a given class runs once. Since the static initializer represents application code, it is capable of doing almost anything a Java program can do. For example, it may invoke methods in other classes, create objects, and set static variables in its class (or other classes, if it has proper access authority).

Once work requests start running, they may update the static variables, or update object fields that are anchored by static variables. Other Jvm instances running Java code perceive that they are running in their own dedicated Java virtual machine and are not affected by other Jvm instances. In order to preserve this illusion, each worker Jvm must be given its own logical copy of the static variables to update as it sees fit.

The scalable Jvm has the effect of a long-running, reusable virtual machine because classes, once loaded, need not be reloaded, relinked, recompiled, or reinitialized. This Jvm structure has been run both on AIX* (Advanced Interactive Executive) and OS/390. Other platforms would also benefit from not having to tear down and start up a Jvm for each application, as well as from being able to share classes that have already been loaded, linked, and initialized.

Conclusions

In this paper we looked at functional and design problems of a typical Java virtual machine and described methods to reduce path length, eliminate unnecessary synchronization, and generate more efficient code. We looked at the requirements of server programs (reliability, availability, security, start-up costs below 50 000 instructions) and designed a Jvm that is able to amortize class loading, linking, and initialization, avoid garbage collection, and preserve process isolation among Java applications while not incurring process teardown for reinitialization.

As Jvm performance, availability, and scalability become more suitable for server programs, problems to address are serviceability, security, and performance management. How can we provide data cap-

ture at first failure so that problems can be diagnosed without having to be recreated? As dynamic compilation methods become viable, what information needs to be captured (dumped) from the Jvm at the time of failure to be able to relate back to the compiled and Java language classes that caused the error? As stated earlier, security in OS/390 depends on the concept of principal-based access controls. Java APIs are needed to extract the principal in effect for the current running thread to check that the principal has access authority to a given resource. To guarantee levels of service, performance management contexts need to be associated with Java threads. Work is being done to address these issues, further facilitating the use of Java technology for commercial applications.

Acknowledgments

The authors wish to thank members of the IBM Java Technology Center in Hursley, and Jeff Aman and Don Schmidt, IBM Poughkeepsie.

Cited references and notes

1. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Publishing Company, Reading, MA (1997).
2. S. J. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. J. Munroe, "Java Server Benchmarks," *IBM Systems Journal* **39**, No. 1, 57–81 (2000, this issue).
3. B. Venners, *Inside the Java Virtual Machine*, McGraw-Hill, Inc., New York (1997).
4. A "mutex" is a synchronization object that provides mutual exclusion among threads. A mutex is often used to ensure that shared variables are always seen by other threads in a consistent state.
5. When a monitor for an object is "inflated," a monitor control block is allocated and added to a hash table, and the monitor index is saved in the object header.
6. Correspondence from Toshio Nakatani, manager, JIT compiler development, IBM Tokyo Research Laboratory.
7. R. Dimpsey, R. Arora, and K. Kuiper, "Java Server Performance: A Case Study of Building Efficient Scalable Jvms," *IBM Systems Journal* **39**, No. 1, 151–174 (2000, this issue).
8. W. Gu, N. A. Burns, M. T. Collins, and W. Y. P. Wong, "The Evolution of a High-Performing Java Virtual Machine," *IBM Systems Journal* **39**, No. 1, 135–150 (2000, this issue).
9. ASCII ISO 8859-1.
10. D. Balfanz and L. Gong, "Experience with Secure Multiprocessing in Java," *Proceedings of International Conference on Distributed Systems*, Amsterdam, the Netherlands (May 26–29, 1998).
11. Endleaf routines are small methods (less than 256 bytes) that call no other methods. They can be contained within the current stack frame and branched to immediately, without the normal linkage cost.
12. T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Just-in-Time Compiler," *IBM Systems Journal* **39**, No. 1, 175–193 (2000, this issue).
13. Y. Gu, B. S. Lee, and W. Cai, "Evaluation of Java Thread Performance on Two Different Multithreaded Kernels," *Operating Systems Review* **33**, No. 1, 34–46 (January 1999).
14. J. Howell, "Straightforward Java Persistence Through Check Pointing," *Advances in Persistent Object Systems*, R. Morrison, M. Jordan, and M. Atkinson, Editors, Morgan Kaufmann Publishers, San Francisco, CA (1999), pp. 322–334.
15. Y. Aridor, M. Factor, and A. Teperman, "cJVM: A Single System Image of a JVM on a Cluster," *Proceedings, International Conference on Parallel Processing (ICPP)*, Fukushima, Japan (September 21–24, 1999).
16. SQLJ (also called Embedded SQL for Java) is a standard proposed by a consortium of database vendors (see http://www.sqlj.org/). The standard has been implemented by IBM in DB2 (see http://www.software.ibm.com/data/db2/java/sqlj/).

Accepted for publication September 10, 1999.

Donna Dillenberger *IBM Research Division, Thomas J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, New York 10532 (electronic mail: engd@us.ibm.com).* Ms. Dillenberger joined the IBM Data Systems Division in Poughkeepsie in 1988 and transferred to the IBM Research Division in 1994. She has worked on future hardware simulation, workload management, Web serving, distributed objects, Enterprise JavaBeans containers, and Java virtual machines.

Rajesh Bordawekar *IBM Research Division, Thomas J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, New York 10532 (electronic mail: bordaw@us.ibm.com).* Dr. Bordawekar joined IBM in 1998 and has worked on scalable Java virtual machines, virtual machine memory management, and persistent JIT compilers. Before coming to IBM, Dr. Bordawekar worked as a postdoctoral researcher at the California Institute of Technology, investigating distributed and parallel file systems for large clusters and distributed shared memory machines. He received his Ph.D. degree from Syracuse University in 1996. His Ph.D. dissertation dealt with optimizing memory-intensive parallel scientific applications.

Clarence W. Clark III *IBM S/390 Division, 2455 South Road, Poughkeepsie, New York 12601 (electronic mail: clarence@vnet.ibm.com).* Mr. Clark currently works in the S/390 e-business area, specifically on Java technology. Prior to this assignment, he spent many years working in technical and managerial positions related to systems and database performance, analysis, and design, For the past several years his work has focused on systems implications of object technology and related application development issues.

Donald Durand *IBM S/390 Division, 2455 South Road, Poughkeepsie, New York 12601 (electronic mail: durand@us.ibm.com).* Mr. Durand has over 20 years of professional experience as a systems programmer, network architect, and operating system software designer. Most of his experience has been within the information technology and insurance industries. In recent years he

has focused on development of parallel software technologies, industry solutions, and Java architecture.

David Emmes *IBM S/390 Division, 2455 South Road, Poughkeepsie, New York 12601 (electronic mail: emmes@vnet.ibm.com).* Mr. Emmes joined IBM in 1978. He has worked in processor storage management, multisystem-coupled communication, and workload management in MVS development for over 18 years. His recent work includes a high-performance TCP (Transmission Control Protocol) stack on MVS that set a new SPECweb benchmark record and performance enhancements in the Java run-time environment for MVS.

Osamu Gohda *IBM Tokyo Research Laboratory, 1623-14 Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502 Japan (electronic mail: gohda@jp.ibm.com).* Mr. Gohda is a senior advisory researcher in the Network Computing Platform group. In 1979 he received an M.S. degree in computer science from the University of Electronic Communications in Japan. He joined IBM in 1984. Since 1990, he has been working on research and development of compilers, including HPF (high-performance FORTRAN) and Java JIT compilers.

Sally Howard *IBM Hursley Development Laboratory, Java Technology Centre, Hursley Park, Winchester, Hants S021 2JN (electronic mail: sally_howard@uk.ibmmail.com).* Ms. Howard is the lead developer of the porting team for the IBM Developer Kit for OS/390, Java Technology Edition, in the Java Technology Center at Hursley Park. She has worked on this project since its inception in 1996.

Michael F. Oliver *IBM S/390 Division, 2455 South Road, Poughkeepsie, New York 12601 (electronic mail: mfoliver@vnet.ibm.com).* Mr. Oliver, program director for the IBM Developer Kit for OS/390, Java Technology Edition, has worked for IBM for 34 years in various technical, planning, and managerial positions in software development. He was recently the program manager responsible for the strategy, planning, and delivery of object technology for OS/390. His current responsibilities include complete product management of DK for OS/390, including planning, coordination of development, and delivery of Java products and technologies for OS/390 from IBM development laboratories in England, Israel, Japan, Toronto, and the United States. He is a recipient of several IBM excellence awards for management and technical achievement. He received a B.S. degree in electrical engineering in 1965 from Worcester Polytechnic Institute, Worcester, Massachusetts.

Frank Samuel *IBM Server Group, P.O. Box 6, Endicott, New York 13903 (electronic mail: samuelfs@us.ibm.com).* Mr. Samuel joined the former IBM Systems Products Division in 1973. He worked on hardware performance and developed microcode for the 4331, 4361, and 9370 processors for 18 years, before beginning performance work and technical ownership for MUMPS/VM (Massachusetts General Hospital Utility Multiprogramming System/Virtual Machine) in 1991. In 1994 he joined the OS/390 OpenEdition DCE (Distributed Computing Environment) development team. He has worked on performance for the IBM Developer Kit for OS/390, Java Technology Edition, since 1998.

Robert W. St. John *IBM S/390 Division, 2455 South Road, Poughkeepsie, New York 12601 (electronic mail: rstjohn@us.ibm.com).* Mr. St. John joined IBM in Poughkeepsie in 1982. After a number of years designing and developing MVS (multiple virtual storage, now OS/390) performance tools, he turned his attention to performance analysis. In 1990, he joined a group with the seemingly impossible mission of providing UNIX services under MVS. He worked through 1997 on design, development, and performance analysis of what is now called OS/390 UNIX System Services. In December of 1997, Mr. St. John began focusing on Java performance analysis. He now serves as the technical leader of a team of performance analysts working on the IBM Developer Kit for OS/390, Java Technology Edition.

Reprint Order No. G321-5723.