The evolution of a high-performing Java virtual machine

by W. Gu
N. A. Burns
M. T. Collins
W. Y. P. Wong

Early Java™ virtual machines (Jvms) possessed several significant performance bottlenecks that inhibited the speed of Java workloads. This paper presents the methodology that was used by IBM to identify and eliminate these bottlenecks for improving the performance of Java applications running on several operating system platforms. In addition, several of the key performance problems that were common to all early Java virtual machine implementations and how they were solved for IBM enhanced Jvms are described in detail. The issues discussed in this paper are focused on problems found in core Jvm components, such as object synchronization, object allocation, heap management, text rendering, run-time resolution, and Java class library methods. The results obtained from applying the described methodology and eliminating the identified performance bottlenecks increased the performance of IBM Java virtual machines by as much as four times on some workloads. The technology discussed in this paper is applicable to other Jvm implementations.

The Java** programming language spread rapidly in the computing industry after it was created by Sun Microsystems, Inc., in 1995. Its ease-of-use and "write once, run anywhere"** capability fascinated many software developers craving a truly portable language and a run-time application execution environment. Since the Java language[1] was initially designed as an interpreted language, programs written in it were translated into platform-independent bytecode, and then they were interpreted at run time according to the Java Virtual Machine[2] (JVM) Specification. Consequently, applications running on the first Jvms were slow. Java performance was substantially improved when just-in-time (JIT) compilers[3–5] were introduced into Jvms. With a JIT compiler, the bytecode of Java applications is compiled into machine code "on the fly," and then the generated machine code is executed, resulting in better performance. But several significant bottlenecks remained.

The promise of Java to write once, run anywhere fits particularly well with the overall strategy of IBM, which manufactures and supports heterogeneous hardware and software platforms. As early as 1996, IBM shipped a Jvm implementation as part of Operating System/2* (OS/2*) Warp 4. Unfortunately, the decisions to enable quick porting and ensure 100 percent compatibility with the Java specification caused IBM's Java performance to initially lag behind that of other Jvm vendors as measured by emerging industry standard benchmarks. In early 1997, after the IBM Developer Kit (DK) for OS/2 Warp, Java Technology Edition, Version 1.0.2, was shipped, a team was formed in the IBM Austin facility to identify and eliminate performance problems of Jvm implementations on OS/2 platforms, with the objective of delivering industry-leading Java performance by year end and to narrow performance gaps with other ob-

ject-oriented programming languages such as C++. The results of that work were delivered on IBM Java platforms beginning in 1997.

The remainder of the paper discusses how performance problems were identified and how they were solved. The next section describes the methodology we developed to identify and isolate performance bottlenecks in Jvms. In the subsequent section, we describe in detail solutions for some of the identified performance bottlenecks we eliminated in core Jvm components, such as synchronization, object allocation, heap management, text rendering, run-time resolution, and Java class library methods. Some conclusions of our Jvm performance work are discussed in the last section.

Performance is a function of the workload being executed on a particular system.

Identifying performance bottlenecks

To improve the performance of a Java virtual machine, we developed a simple but effective methodology. The methodology uses a "divide-and-conquer" technique to isolate and then pinpoint the most pressing performance issues in the current implementation of the Java virtual machine. First, we select a workload that requires improvement. Next, we use a variety of performance tools to analyze the runtime behavior of the workload under study. By periodically sampling the execution[6] of the workload, we get a high-level profile of where time is spent during the workload execution. Then we generate a runtime call graph[7] to understand the flow of the program execution. For a detailed understanding of a "hot" routine, instruction-level traces might be generated to drive reduction in the path length of the routine. This analysis allows us to identify and evaluate performance bottlenecks in the Jvm implementation. Once isolated, these bottlenecks are eliminated by iteratively prototyping solutions and reevaluating the solution.

Selecting a workload to improve. Performance is a function of the workload being executed on a particular system. In order to improve performance, a target workload must be identified for systematic improvement. For improving the performance of the Java virtual machine, that workload may be an industry standard benchmark, such as CaffeineMark**,[8] JMark**,[9] SPECjvm98,[10] and VolanoMark**.[11] It could be an application scenario based on a real-world application, such as Lotus eSuite** or the Web-serving application used at the Olympics in Nagano. In some cases, it might even be a specifically designed microbenchmark to exercise a specific Jvm component or function, such as threading, object allocation, or text drawing.

The primary requirements for the workload are that it must be automated, measurable, and repeatable. Automation is necessary so that there will be no variable idle time while waiting for user interaction. The workload must be measurable, and the results from the measurement should converge within a predefined confidence level. Thus, before and after results can be compared to see whether any improvement has been made. For many aspects of performance analysis, it is also beneficial to have a workload that is CPU-intensive instead of I/O- or network-intensive, so that there will be little wait or idle time.

Profiling execution time. When a workload is CPU bound, it is useful to understand what modules and which functions in those modules are executing the most number of instructions. For this we use a sampling-based program profiling technique.[6] The operating system kernel is instrumented with hooks to capture execution information periodically.[7] This information includes the addresses of the instructions being executed at all sampling points, and these addresses are mapped to specific modules and functions when the sampled trace is postprocessed. Extracts of one sample profile of a Java workload running on IBM DK for OS/2, v 1.0.2 are listed in Figure 1. The workload is a microbenchmark designed to exercise the object allocation and garbage collection facilities of Java. According to the information, you will see that while the Applet process (process ID = 83) was executing, the instructions that were executed came from one of six different modules.

Peeling back one more layer of detail of the above sample output in Figure 1, the performance analyst can determine which routines within the modules of interest to him or her were consuming the most execution time in the sampled workload. The listing

Figure 1 Sample profile of Java workload running on IBM DK for OS/2 1.0.2

```
PID 83, 436 ticks (23.945823 seconds, 73.6%), process name: APPLET
    File: E:\JAVAOS2\DLL\JAVAI, 338 ticks (18.563505 seconds)
    File: JIT compiled code, 77 ticks (4.228964 seconds)
    File: E:\JAVAOS2\DLL\JAVAX, 15 ticks (0.823824 seconds)
    File: DOSCALLS, 4 ticks (0.219686 seconds)
    File: E:\OS2\DLL\PMMERGE, 1 ticks (0.054921 seconds)
    File: E:\OS2\DLL\IBMS332, 1 ticks (0.054921 seconds)
```

Figure 2 Detailed information captured for DLL module JAVAI

```
File: E:\JAVAOS2\DLL\JAVAI, 338 ticks (18.563505 seconds)
  Segment: CODE32, 338 ticks (18.563505 seconds)
      Subroutine: sysMonitorEnter, 45 ticks (2.471472 seconds)
      Subroutine: WinMtxRequest, 45 ticks (2.471472 seconds)
      Subroutine: AllocHandle, 38 ticks (2.087021 seconds)
      Subroutine: gc0_locked, 34 ticks (1.867334 seconds)
      Subroutine: mutexLock, 31 ticks (1.702569 seconds)
      Subroutine: WinMtxRelease, 27 ticks (1.482883 seconds)
      Subroutine: sysMonitorExit, 24 ticks (1.318118 seconds)
      …
```

in Figure 2 shows more detailed information captured for the DLL (dynamic link library) module JAVAI in the Applet process.

On the basis of the profile results, we divide the application and the run-time system into five categories: the operating system kernel, the operating system graphics subsystem, the Java virtual machine, the application code (including JIT compiled code), and the Abstract Window Toolkit (AWT). Depending on where the majority of time is spent, different performance and development skills and expertise are used to tackle the problems. For example, the pie charts shown in Figures 3, 4, and 5 demonstrate three different types of problems in three different workloads.

Figure 3 shows a workload in which most of the execution time was spent in the graphics engine. The workload was an automated test of drawing graphic shapes of different sizes and different colors on the display screen, including items such as lines, rectangles, ovals, arcs, and text strings. Note that in this particular workload, most of the execution time (83.0 percent) was spent in the OS/2 graphics subsystem, denoted by the largest (red) slice of the chart, whereas the time spent in the Java code of the AWT, denoted by the light purple pie slice, is only a very small fraction of the total time (2.0 percent). A later subsection entitled "Graphics" describes the methodology and optimizations for improving such functions.

Figure 3 Workload predominantly running graphics code of the operating system

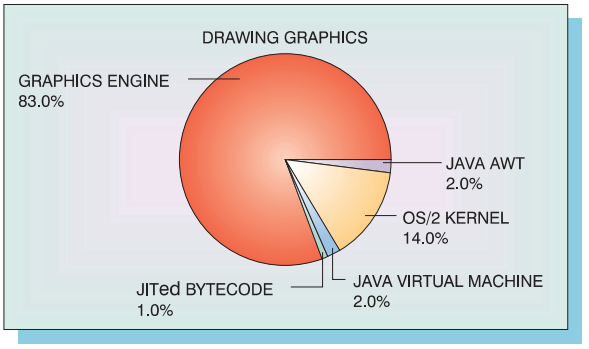


Figure 4 Workload predominantly executing application code (JIT compiler output)

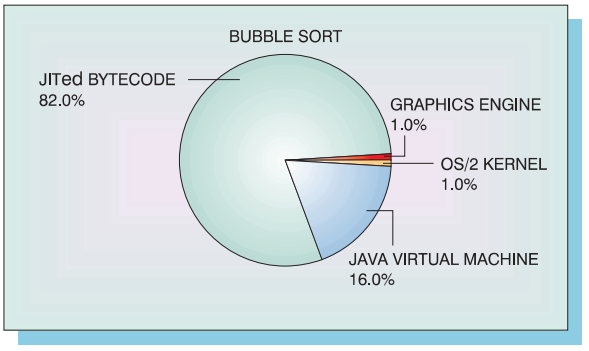


Figure 4 shows a workload in which most of the execution time was spent in the JIT compiled methods. This particular workload implements a bubble sort algorithm for a given set of data items. Intuitively, improvements to such workloads must come from generating more efficient code by the JIT compiler because of the dominance of the total execution time.

Interestingly, changing the workload described above by replacing the bubble sort with a quick sort changes the profile of the new workload dramatically, as shown in Figure 5. Instead of the majority of time being spent in the native code generated by the JIT compiler, most of the time is spent in the Jvm modules: javai.dll (the core Jvm, including object allocation, synchronization, garbage collection, etc.), javar.dll (C run time supporting the rest of the Jvm), and javax.dll (the JIT module that produces the native machine code and provides run-time linkage between the Jvm and the JIT compiler). As a result, attention should be focused on these modules when trying to improve the performance of this new workload. The next section addresses the changes made to improve these Jvm modules and workloads with profiles similar to this workload.

To further isolate problem areas in the Jvm, a more detailed breakdown is required. Look, for example, at the allocation and garbage collection micro benchmark, shown in Figure 6, that tests the efficiency of memory allocation and garbage collection. Profile output, like that listed above, can tell us exactly which routines in the Jvm modules are being executed most frequently. When similar routines are coalesced into common functions, the dominant Jvm modules can be categorized into how much time is spent compiling the Java application bytecode into native code (JIT compiler), synchronizing methods and objects (synchronization), resolving object ownership and instances (run-time resolution), allocating objects (object creation), and garbage collection. Notice that almost 57 percent of the time spent in Jvm routines was spent executing synchronization routines (synchronization).

Generating call graphs for understanding calling sequences. Knowing the amount of time spent in a routine is not enough to be able to significantly improve performance. It is necessary to understand the program flow as well. If there are many small subroutines in a program, as is often the case in an object-oriented design, no single routine may consume a large percentage of the total time, but when coalesced with calling routines and other routines it calls,[7] the combined time can be a significant bottleneck.

For this type of analysis, we added instrumentation to the Jvm and the JIT compiler to generate information about method entries and exits. During the execution of a workload, this information is captured and later postprocessed. The results of the postprocessing show the dynamic call-graph structure of the traced program. It shows how much time was consumed by a particular routine, who was calling that routine at that particular instance, and what other routines were called from the routine under study. For more detailed information about this tool, we refer readers to Reference 7. This tool was used extensively in our performance work for the Java class libraries, as described later in this paper.

Understanding instruction-level path lengths. Conversely, if a particular routine consumes a large portion of the CPU time, a more detailed view of what is happening during the execution of a program is required. At times it is desirable to actually trace the instructions that an application or a portion of an application executes. Tracing is particularly valuable in assessing path length problems of frequently called functions and opportunities for their optimization. For this type of analysis, we developed a tool to exploit the single-stepping capability of the Intel architecture. During execution of a workload, all machine instructions that a selected portion of an application executes are captured and recorded.

This information is postprocessed to sum the number of instructions executed between branches or function calls. It can also show the actual instructions executed for each branch or function. This information is useful for understanding the most frequently used paths through the code and which branches are most often executed in real workloads. This knowledge enabled us to streamline the paths through the Java synchronization code and the runtime resolution code, as described in the next section.

Eliminating performance bottlenecks

Using the methodology discussed in the previous section, performance problems were identified in core Jvm components, including object synchronization, object allocation and heap management, the Jvm run-time system, Java class libraries, and the AWT graphics. In this section, we describe in detail some of the identified performance bottlenecks and how they were eliminated in Jvm implementations for OS/2 systems. Note that the methodology and techniques discussed in this paper were refined and applied to the latest releases of IBM Jvms, such as IBM DK for OS/2 v 1.1.8 and IBM Developer Kit for Windows**, Java Technology Edition, Version 1.1.8, and to upcoming IBM enhanced Java 2 Jvms. Also note that Java performance has been the focus of numerous research and development projects in the past few years. As we were attempting to provide solutions for the identified performance problems in IBM Jvm implementations, others were drawing similar conclusions about these Java performance problems and offering solutions for them.[4,5,12–15]

Object synchronization. With use of the performance analysis methodology discussed previously, a serious performance bottleneck was quickly identified in the

Figure 5 Workload predominantly executing Jvm code

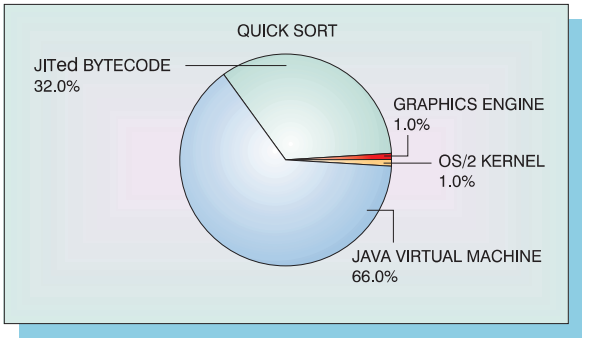
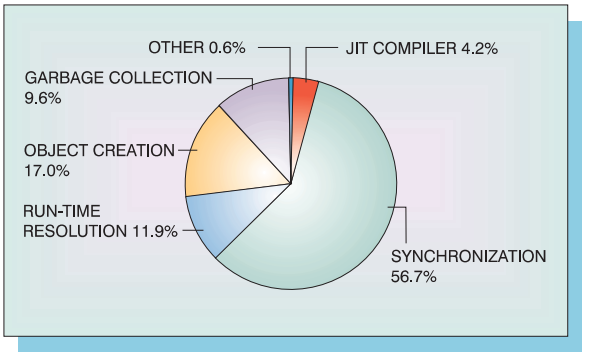
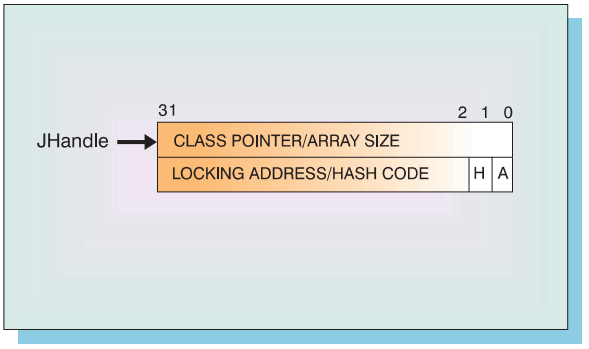


Figure 6 TPROF breakout for an allocation and garbage collection microbenchmark (running with DK for OS/2 1.0.2)



virtual machine implementation of IBM DK for OS/2 v 1.0.2. Analysis of TPROF (timing-based profiler) output for a microbenchmark test (shown as a pie chart in Figure 6) clearly indicates that a significant portion of Jvm execution time is spent in monitor operations. Java supports thread synchronization by means of monitor primitives.[16] More specifically, Java monitors implement the Mesa style[17] of wait and signal. By definition, each Java object has a monitor logically associated with it. The lock mechanism of the monitor is used to synchronize object data accesses. More specifically, monitor enter and exit operations are used to implement the synchronized methods and synchronized blocks of Java code that are used frequently by Java applications and Java class libraries. Therefore, the performance bottleneck is the lock mechanism of the monitor. In the subsequent discussion, we focus on the performance problems found in the implementation of monitor enter and exit operations.

Figure 7 Object header layout of the new monitor design in DK for OS/2 1.1.1



Sun's original implementation. A more detailed look at how monitors are implemented in the Java virtual machine in the IBM DK for OS/2 v 1.0.2 helps to explain the cause of the problem. Note that the implementation is based on Sun's original monitor design in their reference implementation for Windows 32-bit platforms. Therefore, this monitor implementation is also referred to as Sun's monitor implementation in the rest of this section. Also note that the monitor implementation is improved in Sun's more recent Jvm releases.

In Sun's original implementation, a system-wide pool of monitors is used. On entering an object monitor, the thread locks the system-wide monitor pool. Next, a hash algorithm is used, with object handles as the key, to look up and then associate one of the monitors from the monitor pool to the object. The thread then acquires the lock associated with the monitor. The system-wide monitor pool is then unlocked. Exiting an object monitor is similarly cumbersome. The owner thread again needs to lock the system-wide monitor pool and use the same hash algorithm to look up the monitor associated with the object. The thread then releases the lock associated with the monitor. Finally, the system-wide monitor pool is unlocked.

Although this monitor design is rather straightforward to implement and does not require a large number of system monitors (only a system-wide pool of monitors is needed), the approach has several drawbacks. First, it is slow because of the lengthy path required to perform monitor entries and exits. A matching pair of monitor enter and exit operations requires at least 730 Intel instructions in IBM DK for OS/2 v 1.0.2. Second, the single lock to the shared system-wide pool of monitors can quickly become a contended resource. Any thread performing any monitor operation needs to lock and unlock it. Third, the locking mechanism of the original monitor is effectively a wrapper around an operating system semaphore. The use of an operating system semaphore requires calls to the operating system kernel on OS/2, which is very expensive.

The improved monitor implementation. For better performance, a monitor needs to be directly associated with each Java object involved in any monitor operation. In the absence of contention, monitor entries and exits should be shallow operations without calling kernel-level semaphores. The new monitor design in IBM DK for OS/2 v 1.1.1 and beyond was based on these principles. Briefly, the new mechanism binds a monitor to an object when synchronization is first requested for the object. As a result of this change, an additional word, called the *lock* word, is needed for each Java object to store the monitor pointer—the binding between the Java object and its monitor. A new object model is used in IBM DK for OS/2 v 1.1.1 with the object header attached directly to the object body, freeing a word in the object header that was used for linking the object header to the object structure.

Figure 7 shows the new object header layout. The first word is used for storing the class pointer for non-array objects or the array size for arrays. The second word, the lock word, is also overloaded. Bit A of the lock word indicates whether the object is an array or a normal Java object. Bit H indicates whether the lock word contains a hash value or a pointer to the monitor structure of the object. Initially, the lock word contains the hash value of the object, which is calculated at creation time of the object. When the object is synchronized for the first time, a monitor structure is allocated and initialized on behalf of the object. The pointer to this monitor structure is then stored in the lock word of the object, replacing the hash word originally stored there. Note that the hash value is saved in the new monitor structure as part of monitor initialization.

The actual monitor data structure contains the following fields: (1) *Sema4*—a semaphore identifier (ID) field containing the system ID of an operating system semaphore associated with this monitor. Note that this field is needed only if contention for accessing the monitor occurs, or blocking on the monitor is requested. (2) *TakeLock*—a field used for locking and unlocking the monitor structure. It is used for

threads entering and exiting the monitor. (3) *Owner*—an owner field indicating the current thread working with the locked data. Owner is set to null if no thread has entered the monitor of the object. (4) *EntryCount*—a count indicating how many times the current owner thread enters the monitor recursively. (5) *HashValue*—a field for storing the hash value of the object.

In this new monitor implementation, monitor_enter is implemented as follows. Bit H of the lock word is first tested. If the monitor of the object is accessed for the first time, a monitor structure is allocated and initialized, and then the lock word is updated to point to the new monitor structure. Otherwise, the monitor structure is fetched through the lock word. The next step is to check whether the current thread already owns the monitor. If it does, EntryCount is incremented by one, and then the monitor_enter operation is done. Otherwise, the monitor is either free or owned by another thread. For either case, TakeLock is atomically incremented by 1. Since TakeLock contains value -1 when the monitor is free, the first thread that turns the value of the field to 0 obtains ownership of the monitor. The ID of the owner thread is then written to the Owner field of the monitor structure. If the monitor is already owned or contention occurs on entering the monitor, TakeLock contains a positive number. For such cases, the kernel semaphore stored in Sema4 is invoked to force the requester thread to wait on the monitor. Note that atomically incrementing TakeLock is a technique to acquire a lock quickly, with costs as low as one clock cycle on uniprocessor Intel machines.

Monitor_exit is implemented in a similarly efficient way in this new monitor design. The monitor data structure is first fetched through the lock word. The next step is to decrement the recursion count field EntryCount. If the count is not zero after the decrement, then the lock is still held by the current thread, and no further action is needed for the current monitor exit operation. Otherwise, the current thread releases ownership of the lock by clearing the Owner field of the monitor. It then atomically decrements the TakeLock field and checks whether it goes to -1. If not, contention on entering the monitor occurs. The associated kernel semaphore is invoked to wake up one of the threads waiting to enter the monitor. Otherwise, no further action is needed. Note that H-bit is not tested at the beginning of the monitor exit operation, because any monitor exit must follow a monitor entry. Therefore, the lock structure of the object must have already been allocated and initialized, and the lock word contains the pointer to the lock structure.

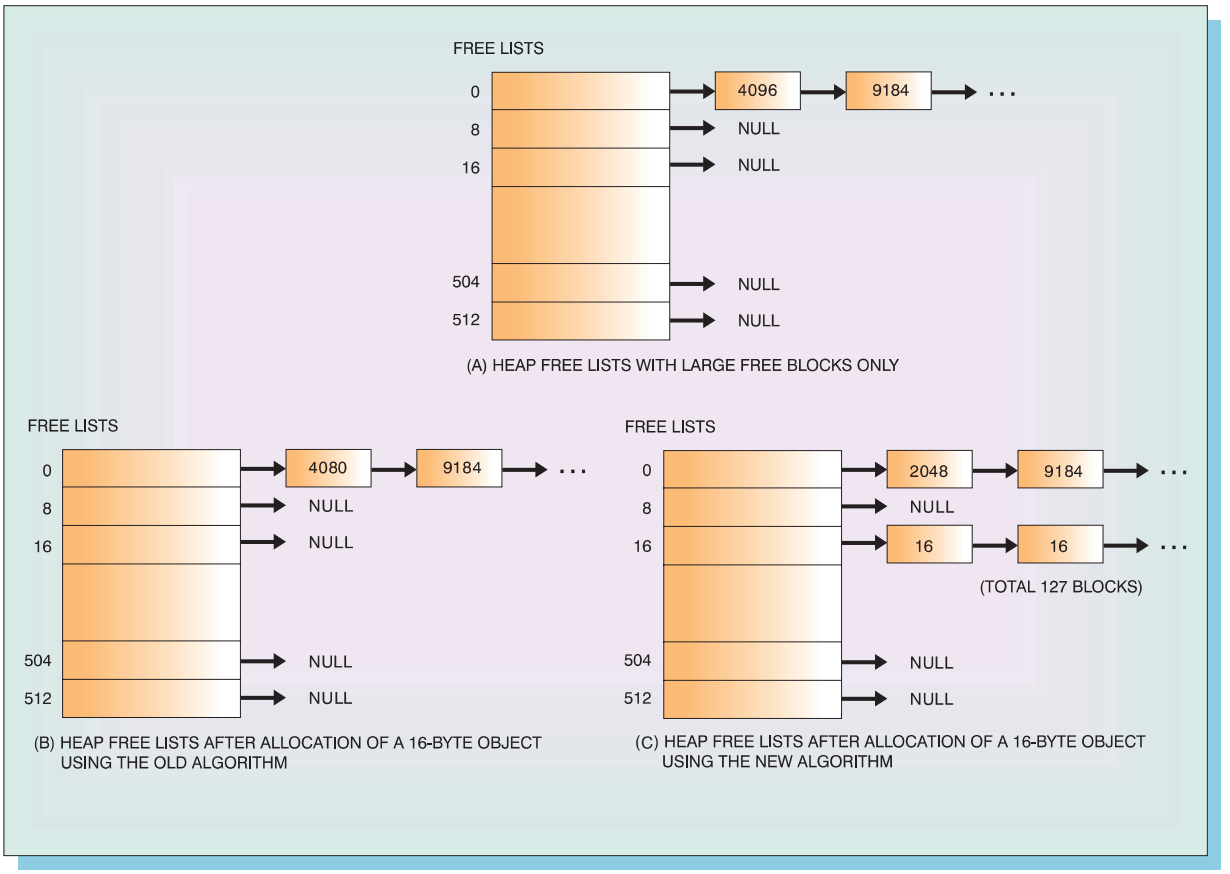
Evaluation of the new monitor implementation. The new monitor implementation is efficient. It eliminates several time-consuming steps found in the original mechanism. The locking mechanism is bound to each object by defining a memory area within the header of the object that contains a pointer to a monitor structure. The monitor structures are not shared, thus eliminating the need to lock and unlock a system-wide monitor pool. Once created, a monitor remains bound to the object for the life of the object. Further efficiency is gained by limiting the use of operating system kernel semaphores. In the absence of contention on monitor operations, operating system semaphores are not used. Rather, the lightweight locking mechanism bound to the object is used during nonblocking situations. On such fast paths without any contention, the path length of entering and then exiting a monitor is reduced from 730 Intel instructions in DK for OS/2 v 1.0.2 to around 30 instructions for later versions of DK for OS/2. Additional efficiency is also achieved by not assigning or initializing the locking mechanism of an object until the first synchronization request is received for the object.

As a result of significantly improved monitor performance, the cost of object synchronization no longer shows up prominently in TPROF output for the allocation and garbage collection benchmark test (see Figure 6) when the workload runs on later versions of DK for OS/2.

The new monitor design in DK for OS/2 v 1.1.1 still has some shortcomings, however. A monitor structure is needed as soon as an object is first involved in any synchronization operation, and it is kept in the Jvm throughout the lifetime of the object. For Java programs containing many objects with synchronization operations, the number of live monitor structures may consume a significant number of system resources. Therefore, a finite number of monitors were made available to be dedicated to Java objects. When this finite supply of monitors is exhausted, the Jvm falls back to the original Sun implementation.

Note that the shortcomings discussed above were partly a result of our initial focus on client-based Jvm performance. The problems were, however, addressed in subsequent releases of the IBM DK for OS/2

Figure 8 Heap management with multiple free lists



and Windows. Details of those implementations can be found in References 14 and 15.

Object allocation and heap management. In typical Java applications, as with any object-oriented program, large numbers of objects are created and used. Therefore, an important objective of any Java heap management mechanism is to allow efficient, high-performing object allocation. Note that dynamic allocation is a well-addressed research topic.[18] The techniques discussed here only attempt to address specific performance requirements of IBM Jvm implementations.

Heap management for IBM DKs was improved incrementally. The implementation in IBM DK 1.0.2 was inherited from Sun's Java Development Kit (JDK**), which organizes the free heap space as a linked list of free blocks. For each object allocation, the list is linearly searched for a free heap block of appropriate size. As a result, the time spent in object allocation is a significant portion of some workload, as illustrated in Figure 6. An improved scheme was implemented in IBM DK for OS/2 1.1.1. The IBM algorithm uses multiple buckets of memory blocks (connected through linked lists) of different sizes to manage the free heap space. There is one free list for every size of free heap block from 8 bytes to 512 bytes with an increment of 8 bytes. All free heap blocks larger than 512 bytes are maintained by a special free list of large blocks (see Figure 8A). Note that in IBM heap management schemes, objects and free blocks are always aligned on 8-byte boundaries. This heap management scheme uses a best fit algorithm to minimize fragmentation of the heap space.

Using the methodology discussed in the last section, we determined that the sluggish performance of ob-

ject allocation is caused by a flaw in the allocation algorithm used in DK for OS/2 1.1.1. For example, consider the state of free lists as depicted in Figure 8A: All free lists are empty except the free list for large objects. If a 16-byte object is allocated, the free list for 16-byte free heap blocks is searched first. Since it is empty, the algorithm moves on to the next free list for 24-byte free heap blocks and finds that it is also empty. In the same manner, all the free lists after that of 16-byte objects are searched before the algorithm determines that there is no free space available in these free lists. It then goes to the free list for large chunks and finds that it is not empty. A large block, say of size 4096 bytes, is removed from the free list and is split into a 16-byte object and a free block of size 4080 bytes. The 16-byte part is returned to the requester, while the rest is returned as a new free block to the free list of large free objects since it is still larger than 512 bytes. The resulting free lists after the object allocation turn into the state as depicted in Figure 8B.

Note that the above process is slow because all free lists are linearly searched. Also note that all the free lists except the top one are still empty after the object allocation of a 16-byte object. If there is another request for a 16-byte object, the same process repeats.

Once the problem was identified, it was not difficult to fix. The improved algorithm incorporated into IBM DK for OS/2 1.1.1 is described as follows:

Step 1. If the free list of size n is not empty, remove a memory block from the head of this free list and return it to the requester. Algorithm done.

Step 2. Go to the free list for large heap blocks (i.e., sizes greater than 512 bytes). If the free list is not empty, remove a free memory block from the list and go to Step 4.

Step 3. Trigger the garbage collection of the entire heap in order to free some heap memory space by reclaiming "dead" objects. Go back to Step 1.

Step 4. If the free block is less than some fixed size, say 2048 bytes, go to Step 5. Otherwise, break the free block into a 2048-byte part and a remaining part. Return the remaining part to the proper free list, and then go to Step 5.

Step 5. Break the free block (2048 bytes or less) into several parts of the requested object size of n bytes. The first part of n bytes is returned to the requester, and the rest are put back to the free list of the requested object size. Note that there might be a left-over block smaller than the requested size, and it needs to be returned to the appropriate free list. Algorithm done.

The difference between the improved and the original object allocation algorithm is dramatic when the corresponding free list of the requested object size is found empty. Instead of stepping through all the free lists for blocks larger than the requested one, the improved object allocation method directly goes to the free list for large objects and allocates a large chunk (not larger than 2048 bytes). It then breaks the large chunk into a number of memory blocks of the requested object size, returns one of them to the requester, and adds the rest to the free list of the requested object size. Therefore, subsequent requests for objects of the same size will be found in the free list. Many programs will allocate multiple objects of the same size (such as Java String and StringBuffer objects, for example); this algorithm optimizes those object allocations. This technique also avoids scanning the free list of large-size objects, which can take a significant amount of time.

An important objective of any Java heap management mechanism is to allow efficient, high-performing object allocation.

Consider the example of allocating a 16-byte object, using the improved object allocation; the free list turns into the state as depicted in Figure 8C. Note that after the allocation, 127 objects still populate the free list of 16-byte objects. The next request for another 16-byte object can quickly obtain it from this free list.

An advantage of the algorithm is that the object allocation process can be broken into a frequent case and an infrequent case. In most cases, a free memory block can be found in the free list of the requested object size. Allocation can be done quickly, and hence called the *fast path*. Periodically, a *slower path*

Figure 9 A trimmed TPROF output from a benchmark test using vector class

```
PID 83, 534 ticks (29.332153 seconds, 95.4%), process name: APPLET
    File: E:\JAVAOS2\DLL\JAVAI, 325 ticks (17.851965 seconds)
        Segment: CODE32, 325 ticks (17.851965 seconds)
            Subroutine: EE, 46 ticks (2.526739 seconds)
            Subroutine: is_subclass_of, 42 ticks (2.307023 seconds)
            Subroutine: threadSelf, 37 ticks (2.032377 seconds)
            Subroutine: sysThreadSelf, 37 ticks (2.032377 seconds)
            Subroutine: mutexLock, 24 ticks (1.318299 seconds)
            Subroutine: sysMonitorEnter, 22 ticks (1.208440 seconds)
            Subroutine: is_instance_of, 21 ticks (1.153511 seconds)
            Subroutine: WinMtxRequest, 21 ticks (1.153511 seconds)
            ...
```

may be taken. Optimization can be focused on the fast path to improve the overall performance of object allocations. For example, an assembly language implementation may be written if high-level language compilers cannot generate the best machine code for the fast path. Moreover, the fast path of object allocation can be "inlined" in the JIT compiler generated code. In fact, the idea of inlining the fast path of object allocation is adopted by all JIT compilers in IBM enhanced DKs.

Note that the improvement of the heap management in DK for OS/2 1.1.4 was focused on reduced path lengths of the object allocation, in particular its fast path. Since then, IBM DK heap management has continued to improve. For example, in IBM DK 1.1.6 each thread is allowed to have its own small area in the heap, called *thread-local-heap*, from which it can allocate small objects without grabbing the heap lock. The reduced contention on a global resource, the heap lock, helps to improve scalability of IBM Jvms running on multiprocessor servers. In IBM DK 1.1.8, the heap management mechanism is again improved by reducing the need for expensive heap compaction.[14]

Jvm run-time system. The analysis of earlier versions of the Jvm revealed that a significant amount of time is spent in the run-time system of the virtual machine.[13] In addition to functions related to object synchronization, object allocation, and garbage collection, several other frequently called Jvm functions were identified, including EE, is_instance_of, and is_subclass_of, as shown in the TPROF output in Figure 9. EE is a Jvm function that obtains the execution environment of the current thread. Since the execution environment of a thread contains such important information as stack pointers (for both the JIT compiler and the interpreter), current method frame, exception-handling object, etc., EE is understandably a "hot" function that is called frequently from other parts of the Jvm and the JIT compiler. The function is_instance_of is used to determine whether an object is an instance of a given class. Because of the object-oriented nature of the Java language and the prevalent use of "virtual" functions in Java programs, is_instance_of is called frequently to determine the actual type of an object at run time. Is_instance_of almost always calls is_subclass_of, making it another "hot" Jvm function.

We developed several methods of improving the performance of "hot" functions. The first is to reduce the path lengths of the concerned functions. This approach emphasizes reducing the number of instructions that are executed every time a routine is called. It might mean streamlining code paths for the most frequently executed cases, using more efficient data structures, or simply eliminating unnecessary instructions. The second method is to avoid or delay calling these functions if possible. Some results can be cached to avoid calling the same functions again later in the execution. Some function calls can be delayed until the results are absolutely needed. The third

method is to redesign the algorithm that implements the functions. That is, the performance problems found in the old algorithm are avoided in the new design.

For each specific problem, a combination of the above approaches may be used. For example, the impact of EE on performance is reduced by enhancing its implementation as well as by not calling the function. In its original implementation, EE in turn called several other functions. All of the overhead of calling or returning from a function can be avoided by using macros. In IBM DK for OS/2 1.1.4, we reduced the cost of calling EE to as low as 10 Intel instructions. We also reduced the number of times EE is called by revising some key Jvm functions to include the pointer to the execution environment of the thread as one of their arguments. In addition, EE can also be cached in the JIT compiler environment to further reduce the number of calls to the function.

It is more difficult to avoid calling is_instance_of and is_subclass_of in the Jvm. It is also not straightforward to improve their C code implementations. However, we observed that these two functions are very generic; they handle all kinds of instance type checking and subclass checking. No knowledge is assumed of objects or classes involved. Therefore, they need to check whether an object or class pointer is null, whether an object is an array or a "normal" object instance, and whether a class is an interface or a regular Java class. It is difficult for even the best C compiler to optimize each execution path of the generated code when there are as many cases as these. We noticed, however, that typically a specific execution path, that we call the "fast" path, is taken by is_instance_of and is_subclass_of. In the majority of cases, an object is a normal object, i.e., not an array, and the object is an instance of the given class. In order to "force" the best optimization on the fast path of execution, we chose to implement the two small functions in Intel assembly language in DK for OS/2 1.1.4. In particular, register usage is fully optimized for the fast path, and a call from is_instance_of to is_subclass_of is avoided by jumping directly from the caller's context to the callee's context because of the smart allocation of registers and similarity of the arguments of the two functions. As a result, the length of the fast path is significantly reduced, and the overall performance is improved.

The effect of improving the frequently called Jvm functions listed above is significant to the overall performance of the Jvm, as demonstrated by the results[19] shown in Table 1 for several micro Jvm benchmark tests. These benchmarks exercise several key Java class libraries. As another indication of improved performance, these Jvm functions no longer show up prominently on the TPROF output, which means they no longer take a significant amount of time to execute.

Table 1 Results for benchmark tests using Vector, Hashtable, and Stack of Java

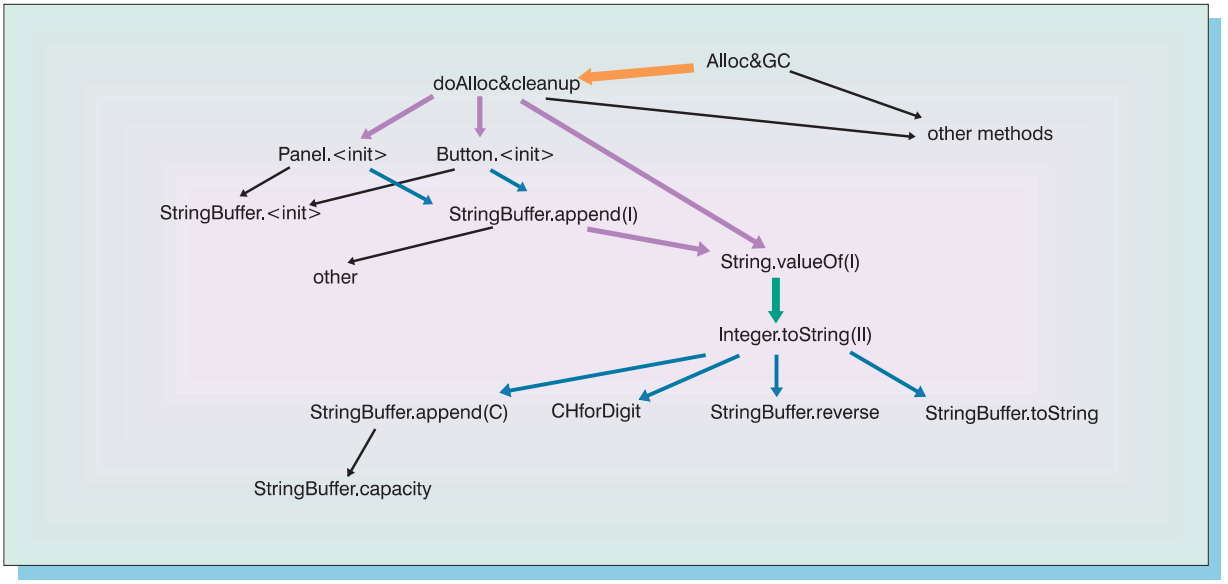
	JDK 1.0.2	Improved Jvm	Improvement (percent)
Vector	1172	1608	37
Hashtable	3624	4447	23
Stack	1068	1422	33

It is important to note that the specific functions discussed in this section are not the only hot functions in all Jvms. Instead, they are used to exemplify how hot Jvm functions are identified and improved. Different hot functions may be identified and improved in other Jvm implementations by using similar approaches.

Java class libraries. The class libraries are another part of the Java virtual machine where performance problems exist.[12] We used the same methodology to identify performance bottlenecks in the class libraries of the Jvm as we did for other areas of the Jvm. However, in this area, the arcflow analysis tool[7] is more effective, because it provides richer run-time information for each method and class. For example, it keeps track of how much time, in terms of executed bytecode or actual CPU time, is spent executing each method and the methods it invokes. The tool presents results in different views (Figure 10 shows an intuitive tree view), which helped us to identify heavily used classes or methods in the class libraries. Such insightful run-time information directs attention to those paths in the class libraries where performance is critical for most Java applications and applets. It also helps us to decide whether improvement is needed for the identified heavily used classes and methods.

Figure 10 shows a visualized tree view from an arcflow analysis of an allocation and garbage collection microbenchmark running on IBM DK for OS/2 1.0.2. The arrows represent method invocations, and line thickness roughly indicates the relative amount of time spent in that method. For example, method

Figure 10 A visualized arcflow tree view of an allocation and garbage collection benchmark running on DK for OS/2 1.0.2 (unessential details omitted)



String.valueOf is invoked many times by this Java benchmark test. A significant amount of time is spent in this method and the methods it invokes, including Integer.toString, StringBuffer.append(C), StringBuffer.reverse, etc. Therefore, attention should be paid to the performance of String.valueOf(I) and the methods it invokes.

A closer examination of the trace data and the method invocation pattern reveals some inefficiency in the String.valueOf(I) implementation. This method converts a decimal integer number into a character string representation. In the original implementation, this method simply invokes method Integer.toString(II) and relegates the task to the new method. Note that the invoked method is a generic implementation of converting an integer value into any format of character string representation (e.g., binary, decimal, or hexadecimal). Therefore, the algorithm requires quite a few expensive operations, such as three object creations and 10 to 30 method invocations, 2 to 12 of which are synchronized.

It is clear that String.valueOf is a "hot" method for this workload. The method is also used to implement concatenation of a string and an integer, such as "a_string" + 100. Hence the method is used frequently by typical Java applications or applets. In fact, AWT uses it to create default names (used internally by AWT) for basic AWT objects, such as Label, Button, and Panel. The following expression is used to generate a unique name for each object

```
This.name = base + nameCounter++;
```

where the base name and nameCounter are associated with each AWT class. For example, new AWT label objects are named label0, label1, label2, and so on, if not explicitly specified when they are created.

There are many different approaches for improving the performance of the implementation of any given class, but they are all restricted by the requirement that Java compatibility cannot be broken by any changes to the system class library of the Jvm. For String.valueOf(I), one straightforward approach is to improve the implementation of the method. A native C implementation was first tried, and good performance was achieved through avoiding unnecessary object creations and method invocations. It was later determined that similarly good performance can be achieved through a Java implementation. Because of the portability of Java, the Java implementation was chosen so that other platforms would benefit as well. The improvement was incorporated into

DK for OS/2 1.1.4 and later into the class libraries for other platforms. In the improved implementation, only two object creations (for the string to be returned) are needed, and all method invocations are avoided.

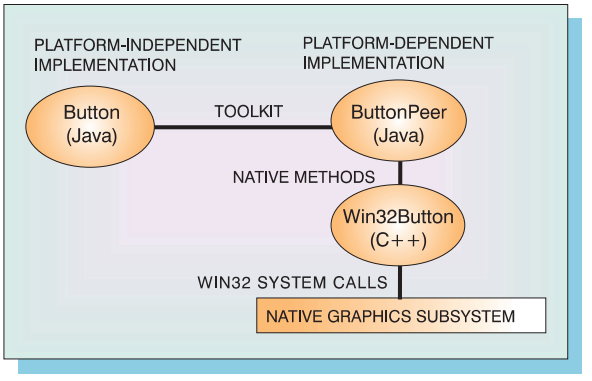
Another "fix" for some expensive methods in class libraries is to avoid calling these methods, if possible. In the above example, String.valueOf(I) is invoked to generate default names of new AWT objects at their initialization time. However, the names of AWT objects are not always used by applications. Therefore, creation of these names may be delayed until they are actually used, saving time for applications that do not make use of the names.

We explored the examples above in some detail to illustrate the improvements we made in the class libraries. We also worked with Sun and Netscape Communications Corporation to develop many other enhancements to the system Java classes, including String, StringBuffer, Vector, Hashtable, etc. These enhancements were delivered as part of the Sun JDK 1.1.6 reference platform and were therefore made available to all platforms that have incorporated this release.

Graphics. Java graphics[20] is another important area in early Jvms where performance enhancements were needed. Using the tools and methodology discussed in the previous section, performance problems in the implementation of AWT were identified, and extensive efforts were made to improve the overall graphics performance of IBM Jvm implementations. Areas that were significantly improved include text drawing, AWT event handling, image conversion, and basic graphics primitives such as lines, ovals, and polygons. The performance work was carried out on different operating system platforms, including OS/2 and Windows systems. Limited by space, we only discuss in detail the work on text drawing where significant performance improvements were achieved. For brevity, discussions below assume a Windows 32-bit, or Win32**, platform.

To facilitate the following discussions, let us use the AWT Button class as an example to briefly explain how the AWT components are implemented on Win32 platforms. For each AWT component class like Button, there is an associated class called a Peer class that is designed to bridge the platform-independent AWT class to the underlying native graphics subsystem. Peer classes contain mostly native methods that are implemented in C and C++. For instance,

Figure 11 The interaction of Java Button class with its peer class



as shown in Figure 11, a C++ class Win32Button is used to implement the native methods of ButtonPeer. Figure 11 also shows interactions among AWT Button, ButtonPeer, and Win32Button. Because of this peer architecture, a number of objects (in both Java and C++) are created for each AWT component. Similar data structures are maintained in different locations. Events and messages are queued and processed in both Java and the native graphics subsystem. The complexity of this peer architecture is responsible for many graphics performance problems.

We now discuss in more detail performance problems and fixes for text drawing. Text support is an important aspect of Java graphics. Drawing a text string seems simple: A font is first created and set for the graphics context, and then the string is drawn by calling the string drawing method. In early Jvm implementations, however, these operations had long path lengths. Although problems were found in text rendering on some platforms, the major performance problems were in the areas of font management and Unicode**[21] support on Windows platforms.

Similar to Button, AWT Font also has its own peer class called FontPeer, and a C++ class called AwtFont. Whenever an AWT Font object is created at the application level, an instance of FontPeer is created as well. As identical font objects are created in different parts of the applications, multiple instances of its font peers are created accordingly. In other words, Java peer fonts are not shared. For applications that create fonts frequently for large amounts

Table 2 Improvement of font management and Unicode support for font creation and text drawing

	Original Jvm (sec.)	Improved Jvm (sec.)	Improvement (percent)
Font creation	4.7	0.8	83
Text drawing	3.8	2.3	39

of text drawing, identical instances of peer fonts are frequently created and then thrown away, causing performance problems.

Two improvements to the font management were implemented in IBM DK for Windows 1.1.7. Basically, Java peer fonts are cached in a hash table and reused for font references as much as possible. That is, many identical font instances at the application level share one instance of the peer font. In addition, we also modified the implementation in the native code so that the creation of Windows fonts is delayed until the fonts are actually needed. We noticed that there are situations where font setting is not immediately followed by text drawing, causing unnecessary font creation. As a result of these changes, the path length is shortened significantly in most cases. Moreover, memory usage and garbage collection activities are also reduced because fewer peer objects and Windows fonts are being created. The benefit from the font caching at the Java peer level is demonstrated by improved results[22] of a font creation microbenchmark shown in Table 2.

The next major performance problem of text drawing was found in the handling of Unicode characters.[21] Unicode uses a 16-bit character encoding system designed to support text written in diverse human languages. Currently, it is not common, however, to have font sets capable of displaying all Unicode characters. Most fonts for Indo-European languages display only the first 256 out of the total of 65 535 Unicode characters. If an application wants to use two languages in one sentence, or displays a string that contains English text with mathematical symbols, a font such as Times Roman could not display all characters by itself. To deal with such situations, Sun Microsystems, Inc., introduced the concept of multiple host fonts[23] into Jvms starting from JDK 1.1.5. Figure 12 uses a portion of a font property file to illustrate how multiple host fonts are used. In the font property file, Java Font Dialog is in fact mapped into a series of host fonts, namely Arial, WingDings, and Symbol. For characters that cannot be displayed by a particular font, exclusion ranges of that font are specified. For example, the exclusion ranges for Arial are $0\times100-0\times20ab$ and $0\times20ad-0\times ffff$, and no exclusion ranges for WingDings and Symbol are specified. In addition, each font is associated with a converter that maps each Unicode character into bytes understood by a certain underlying encoding scheme. The idea is that if a Unicode character cannot be handled by the first host font, the second one should be tried, or else the third one, and so on. The goal is to cover as much of the Unicode character set as is desired.

The introduction of multiple host fonts caused significant performance degradation for drawing text strings. For every character in the string, two checks must be performed: (1) Is the character an excluded character for the given font? (2) Can this Unicode character be mapped into the underlying encoding scheme, such as ISO (International Organization for Standardization) Latin-1? These checks become an expensive part of text drawing. However, they are unnecessary for strings containing only ASCII characters because usually only the first host font is involved.

For better performance, a fast path is introduced into IBM Jvms for drawing strings with single-byte characters only. First, the string to be drawn is checked quickly for the existence of excluded characters. If it contains no excluded characters, the whole string is passed to another routine to check whether all characters in the string can be converted. The converter used in this case specifies convertible characters in ranges. If the maximum character and minimum character of the string fall into one of the ranges, we can quickly conclude that all the characters of the string can be converted. Thus, a significant amount of time is saved by not checking individual characters, and the string can be passed to the lower-level drawing routine right away. With the fast checking routines, the performance of drawString for ASCII characters, e.g., English text, improves significantly. The benefit from this improvement is depicted by improved results (in Table 2) for a microbenchmark of drawing English text. Note that there is a slight path length increase for strings containing double-byte characters, but the increase is outweighed by the gains in the single-byte cases.

Figure 12 A portion of a font property file illustrating how multiple host fonts are used

```
dialog.0=Arial,ANSI_CHARSET
dialog.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialog.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
# Exclusion Range info.
exclusion.dialog.0=0100–20ab,20ad–ffff
```

Conclusions

Our early efforts to improve performance of IBM Jvm implementations on IBM Intel-based platforms led us to develop techniques and methodologies to identify, isolate, and solve performance bottlenecks. Among the first key performance issues critical to the overall performance of Jvms we identified were object synchronization, object allocation, heap management, run-time resolution, Java class libraries, and AWT graphics. The enhancements we developed helped IBM deliver high-performing Java virtual machine implementations on all IBM platforms.

This methodology and process for identifying and solving Jvm performance problems are applicable to any Jvm implementation. In fact, the performance of Jvms has been continuously improved by teams across IBM using the same approach, including further enhancements to parts of the Jvm addressed in this paper, such as object synchronization, object allocation, and AWT implementation. As a result, some technologies discussed in the paper have evolved into new schemes, and some others were replaced with similar but more comprehensive technologies. For example, the monitor implementation discussed in this paper was implemented in IBM DK for OS/2 1.1.4, but it was replaced on later IBM Jvms by a more scalable monitor design from IBM Research,[15] which is also based on the idea of associating a monitor directly with each Java object. The improved object allocation scheme was also implemented in DK for OS/2 1.1.4. It was extended by the Thread-Local-Heap design on more recent Jvm implementations.[14]

Acknowledgments

We thank the tools team (Robert Berry, Chester John, Riaz Hussain, and Bob Urquhart) of IBM Network Computing Software Division System Performance for their tremendously useful Java performance tools, Mike Brown for the implementation of the improved monitor design on OS/2, Andrew Johnson and James L. Peterson for their contributions to the object model improvements, Peter Kessler of Sun Microsystems, Inc., for his assistance in incorporating the class library changes into the reference JDK, and Mike Cooper, R. Ravisankar, and Bruce Worthington for their assistance in Java graphics improvements. We thank Bill Alexander, Robert Berry, Maria Jenny, and Patricia Orlosky for reviewing and commenting on the paper. We would also like to thank Jerry Kilpatrick for his leadership in Java performance.

Cited references and notes

1. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley Publishing Co., Reading, MA (1996).
2. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Publishing Co., Reading, MA (1996).
3. T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Java Just-in-Time Compiler," *IBM Systems Journal* **39**, No. 1, 175–193 (2000, this issue).
4. C.-H. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal, W.-M. W. Hwu, "Compilers for Improved Java Performance," *Computer* **30**, No. 6, 67–75 (June 1997).
5. T. Newhall and B. P. Miller, "Performance Measurement of Dynamically Compiled Java Executions," *Proceedings of the ACM 1999 Conference on Java Grande* (1999), pp. 42–50.
6. S. L. Graham, P. B. Kessler, and M. K. McKusick, "An Execution Profiler for Modular Programs," *Software—Practice and Experience* **13**, No. 8, 671–685 (August 1983).
7. W. P. Alexander, R. F. Berry, F. E. Levine, and R. J. Urquhart, "A Unifying Approach to Performance Analysis in the Java

Environment," *IBM Systems Journal* **39**, No. 1, 118–134 (2000, this issue).
8. CaffeineMark, Pendragon Software Corporation, Libertyville, IL, http://www.pendragon-software.com/pendragon/cm3/index.html.
9. JMark, Ziff-Davis, Inc., New York, http://www.zdnet.com/zdbop/jmark/jmark.html.
10. SPECjvm98, The Standard Performance Evaluation Corporation (SPEC), Manassas, VA, http://www.spec.org/osg/jvm98/.
11. VolanoMark, Volano LLC, San Francisco, CA, http://www.volano.com/guide21/mark.html.
12. A. Heydon and M. Najork, "Performance Limitations of the Java Core Libraries," *Proceedings of the ACM 1999 Conference on Java Grande* (June 1999), pp. 35–41.
13. F. G. Chen and T.-W. Hou, "Design and Implementation of a Java Execution Environment," *Proceedings of the 1998 International Conference on Parallel and Distributed Systems* (1998), pp. 686–692.
14. 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).
15. D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano, "Thin Locks: Featherweight Synchronization for Java," *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation* (1998), pp. 258–268.
16. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM* **17**, No. 10, 549–557 (October 1974).
17. B. W. Lampson and D. D. Redell, "Experience with Processes and Monitors in Mesa," *Communications of the ACM* **23**, No. 2, 105–117 (February 1980).
18. P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review," *International Workshop on Memory Management*, Kinross, Scotland (September 1995).
19. The results are obtained on an OS/2 system installed on an IBM PC 750 with 166 MHz Pentium® processor with MMX and 48 MB memory. The Jvm involved was DK for OS/2 1.0.2 GM. The improved Jvm is a prototype based on DK 1.0.2 with the improvements discussed in this section. The reported scores are averages of several runs of the tests. Higher scores are better scores in all tests.
20. D. M. Geary and A. McClellan, *Graphic Java 1.1: Mastering the AWT*, Prentice Hall, Englewood Cliffs, NJ (July 1997).
21. Unicode Consortium, *The Unicode Standard: Version 2.0*, Addison-Wesley Publishing Co., Reading, MA (September 1996).
22. The results are obtained on a Windows NT™ system installed on an IBM PC 300PL with a 450 MHz Pentium II processor, 48 MB RAM memory, S3 Trio 3D chip set with 4MB VRAM at 1024 × 768 × 256 colors. The Jvm involved was IBM Developer Kit for the Windows 1.1.7 platform. The measurements of the original Jvm are based on the code rolled back from the improved code. The reported scores are averages of several runs of the tests. Lower scores are better scores in all tests.
23. *JDK 1.1 Internationalization Specification*, Sun Microsystems, Inc., documentation (December 1996). http://java.sun.com/products/jdk/1.1/intl/html/intlspecTOC.doc.html.

Accepted for publication August 23, 1999.

Weiming Gu *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: wgu@us.ibm.com).* Dr. Gu is a senior software engineer in the System Performance Group of the Network Computing Software Division. He currently works on Java and JIT compiler performance. After joining IBM in 1995, he worked on building performance tools and analyzing performance of the OS/2 system kernel. Dr. Gu received his Ph.D. degree in computer science from the Georgia Institute of Technology in 1995.

Nancy A. Burns *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758.* Mrs. Burns is a senior software engineer, currently serving as the IBM Corporate Java Performance focal point, coordinating the performance work done across all IBM platforms. She has served as the technical lead for the performance work on the Developer Kit for OS/2 Warp, Versions 1.1.1 and 1.1.4. She led the performance team that delivered JavaOS for Business Release 2.0 and 2.1. She joined IBM in 1983 and has worked on a variety of software development projects involving expert systems, multimedia, database management, and parallel processing, before focusing on performance of the Java language and Java virtual machines. Mrs. Burns received a B.S. degree in statistics and quantitative methods from Louisiana State University and an M.S. degree in mathematical sciences from the University of North Florida.

Michael T. Collins *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: mcollin@us.ibm.com).* Mr. Collins graduated from Purdue University with a B.S. degree in computer science in 1982. He joined IBM in Austin in 1982, initially working on software design and development for the 5520 office system, a dedicated minicomputer for office groupware applications. He later designed and developed software for communication and networking software systems on the DOS and OS/2 platforms, including the PC LAN program and the directory and security services feature of Warp Server. He has also developed software for IBM word processor and office groupware applications at IBM's former facility in Westlake, Texas. In these assignments, he was the development lead for DisplayWrite 5 (a popular DOS word processor) and the IBM WorkGroup Directory product. Most recently, he has worked as part of the team dedicated to improving the performance of IBM's Java products on Intel processors, with a focus on the Java virtual machines and JIT compilers for Windows and OS/2. He has received several IBM awards, including two Outstanding Technical Achievement Awards and a Division Award. He is also a coinventor on three filed patents and is a member of the Phi Beta Kappa Honor Society.

Wai Yee Peter Wong *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: wpeter@us.ibm.com).* Dr. Wong is an advisory software engineer, working on performance improvement of the graphical user interface since 1995. In the past two years, his main work has been on Java graphics. He holds a B.A. degree in mathematics and computer science from the State University of New York at Buffalo, an M.S. degree in computer science from the University of Michigan at Ann Arbor, and a Ph.D. degree in computer science from Ohio State University. His current interests include graphical user interfaces and object-oriented systems. Dr. Wong received two IBM Outstanding Technical Achievement Awards for his work on Java graphics performance.

Reprint Order No. G321-5720.