Java server performance: A case study of building efficient, scalable JVMs

Article *in* Ibm Systems Journal · February 2000

DOI: 10.1147/sj.391.0151 · Source: IEEE Xplore

3 authors, including:

Rakesh Arora
Yet to be Named
90 PUBLICATIONS **999** CITATIONS

SEE PROFILE

Java server performance: A case study of building efficient, scalable Jvms

by R. Dimpsey
R. Arora
K. Kuiper

The importance of the Java™ platform has shifted from a client-centered paradigm to the server. In particular, the Java language has matured into a viable programming model for server applications. Correspondingly, the requirements on the Java virtual machine (Jvm) have shifted. This paper details the server-specific performance enhancements made to the core Jvm and just-in-time (JIT) compiler, which have allowed the IBM Developer Kits that implement Java code for Intel processors to become industry performance leaders. The paper focuses on synchronization implementation and granularity improvements that have greatly increased the scalability of the Java language on multiprocessor machines. Focus is also given to memory management, specifically, object allocation, garbage collection, and heap management. Details of communication and connection scaling are also provided. Finally, server-specific enhancements to the JIT compiler are discussed. All component enhancements in the paper are explained, and their performance implications are quantified with results from representative multithreaded server workloads. The paper summarizes work from across IBM. The authors' specific contributions include the three-tier spin lock, the thread local heap and freelist merge, the dynamic heap growth algorithm, bitwise sweep, compaction avoidance, and the suite of network enhancements.

Initially targeted at the client market, the Java** platform has emerged as a compelling base for server applications. The use of the Java platform on the server is diverse and, not surprisingly, produces workloads significantly different from those found on the client. Correspondingly, performance enhancements generated by client workloads may not translate to increased server performance. Examples of Java platform use on the server include servlets, with their efficient implementation of dynamic Web content, database access through a variety of connectors, chat servers, terminal servers, Enterprise JavaBeans** components, and frameworks for persistent business objects. Although these workloads are quite distinct, they all place certain common stresses on the core Java virtual machine (Jvm), just-in-time (JIT) compiler, and class libraries. For instance, server workloads require that the Jvm manage large Java heaps efficiently, both through optimized object allocation and efficient garbage collection. Also, server workloads, and their corresponding underlying infrastructure, must be designed to scale to larger systems with more processors and memory. This requirement places a premium on an efficient lock implementation, as well as the granularity of the locking within the Jvm. Another common attribute of Java server workloads is their reliance on efficient, robust networking and their need for a large number of simultaneous connections and threads.

Given IBM's long history with server systems, it is not surprising that IBM's Java virtual machines have met the above requirements to become the highest-performing server Jvms. Proof of this statement comes from both external and internal benchmarks.[1] For instance, *JavaWorld* magazine has identified through the use of the server benchmark VolanoMark** that the two highest-performing Jvms are the Jvm in the IBM Developer Kit for OS/2* (Operating System/2*) Warp, Java Technology Edition, version 1.1.7, and the Jvm in the IBM Developer Kit for Windows**, Java Technology Edition, version 1.1.7.[2] In addition, *InfoWorld*, using a proprietary server workload, identified the Jvm in the IBM Developer Kit for AIX* (Advanced Interactive Executive), Java Technology Edition, version 1.1.6, to be the top performer with the Developer Kit (DK) for OS/2, v 1.1.7 next in line.[3] Other sources validating the superiority of IBM Jvms are also available.[4,5]

Considerable effort has been expended across IBM in designing and implementing monitors for each Jvm.

The above results have generated significant interest in the internal implementation of the IBM Jvms. This paper presents a synopsis of the major enhancements made to the IBM Jvms in the Developer Kits for Windows and OS/2. The enhancements described are intended for server workloads but will benefit all workloads with a significant Java component. This paper is not intended to be a theoretical research paper. The nature of theoretical research often precludes the scrutiny of implementation and the marketplace. This paper is an attempt to summarize the key ideas, whether from research or invention, that have been implemented and are currently enjoying success in the field. One may view this paper as a case study of a Jvm implementation aimed at servers. It is a summary of the pragmatic decisions that must be made when research and product deadlines collide. In addition, the paper can be seen as a companion piece to the paper written by Gu et al.,[6] which describes initial changes to the Jvm in the IBM DK for OS/2 focused on client workloads and encompassing enhancements up to version 1.1.4.

The contribution of this paper is that it describes in detail and supports with data the enhancements made to the current IBM Jvms on Intel processors. Considered first are the design and implementations of the Java lock (monitor). Next the object allocation subsystem and its evolution are described. Subsequently, there is a discussion of enhancements made to heap management and heap growth as they pertain to a server workload. This discussion is followed by a summary of the changes made to the mark/sweep/compact garbage collection technology shipped by the reference platform. Because efficient I/O, especially network I/O, is crucial for server workloads, a section is dedicated to enhancements IBM has made in this area. Finally, server-specific changes to the JIT compiler are discussed.

Monitor implementation

It soon becomes evident to all users, developers, or performance analysts working with the Java platform that the implementation of monitors is the key to good performance. An article in *JavaWorld* claimed that synchronization accounts for over 19 percent of Java application time while running on the Sun Microsystems reference platform.[7] The prodigious use of monitors for synchronization is especially evident in server applications running on multiple processors. Heavy monitor usage can be found in the Java classes and user applications. In addition, the locking implementations described in this section are also those used by the core Jvm to guard internal structures. Monitors are used for synchronization in one of two ways. The most common is the synchronized method call, whereby the monitor is obtained before execution of the method and released after completion. The second is by using the *synchronized*(Object) call to enforce a recursive mutex lock around a critical section. In both cases, synchronization is done at the object level, and therefore a lock structure is required per object instance.

Given the importance of monitors to performance, considerable effort has been expended across IBM in designing and implementing them for each IBM Jvm. The work has been completed in an iterative fashion with one improvement built upon the last. This section describes the key performance issues associated with Java monitor implementation and sketches the current IBM solution based on Intel processors.

The key performance issues to attend to when implementing the Java monitor are:

1. The acquisition and release time for an uncontended monitor
2. The action to take when contention occurs (e.g., spin, yield, block)
3. The mechanism used to determine which thread obtains the monitor when there are multiple waiters

Uncontended lock acquire or release. The time to acquire or release the monitor was optimized through a design proposed by Bacon et al.[8] referred to as *thin* locks. These locks are inspired by the MCS locks proposed by Mellor-Crummey and Scott and named from the initials of their last names.[9] The thin locks are the second iteration of monitors implemented by IBM. The first implementation is described in the paper by Gu et al.[6] The thin lock design is based on 24 bits stored in the object header that act as a lock word.[10] The high bit in the word delineates between two basic lock states: inflated and flat. A flat lock is one that has never been the source of contention (i.e., all acquires have been successful on the lock up to the point of the current acquisition). An inflated lock is one that has sustained contention in the past. For the flat lock, the 24 bits are defined as follows: one to indicate lock type (inflated or flat), 15 for a thread identifier, and eight for a recursion count. The 24 bits in the inflated case contain the one-bit lock type indicator plus an index to an array that holds a pointer to the inflated lock structure.

The inflation of a lock occurs on the first instance of contention, and the lock remains inflated for the lifetime of the object. However, because most monitors are never contended, they remain flat. Therefore, the acquisition and release time for the uncontended flat case is crucial. Fortunately, the structure of the flat lock enables a very efficient acquire and release. In essence, to acquire the lock, a unique thread identifier needs to be atomically swapped into the lock word, or if it is a recursive acquisition, the recursion count needs to be incremented. A release is nothing more than a zeroing of the ID (identifier) or a decrement of the recursion count. On Intel systems the *cmpxchg* instruction is employed for the atomic instruction, and an acquire-release pair takes just 7 and 5 instructions in the nonrecursive case, and 11 and 6 for the recursive case. Note that this count does not include obtaining the thread identifier and chaining monitors for exception processing. Because of this, and also because of the granularity of the measurement, a processor cycle count for the acquire-release operation was difficult to accurately collect. However, cycle count is the true measure of performance, so for improved cycle-per-instruction (CPI) behavior, the code only locks the system bus for atomic operations on multiprocessor systems, thus saving this expensive operation on a uniprocessor.

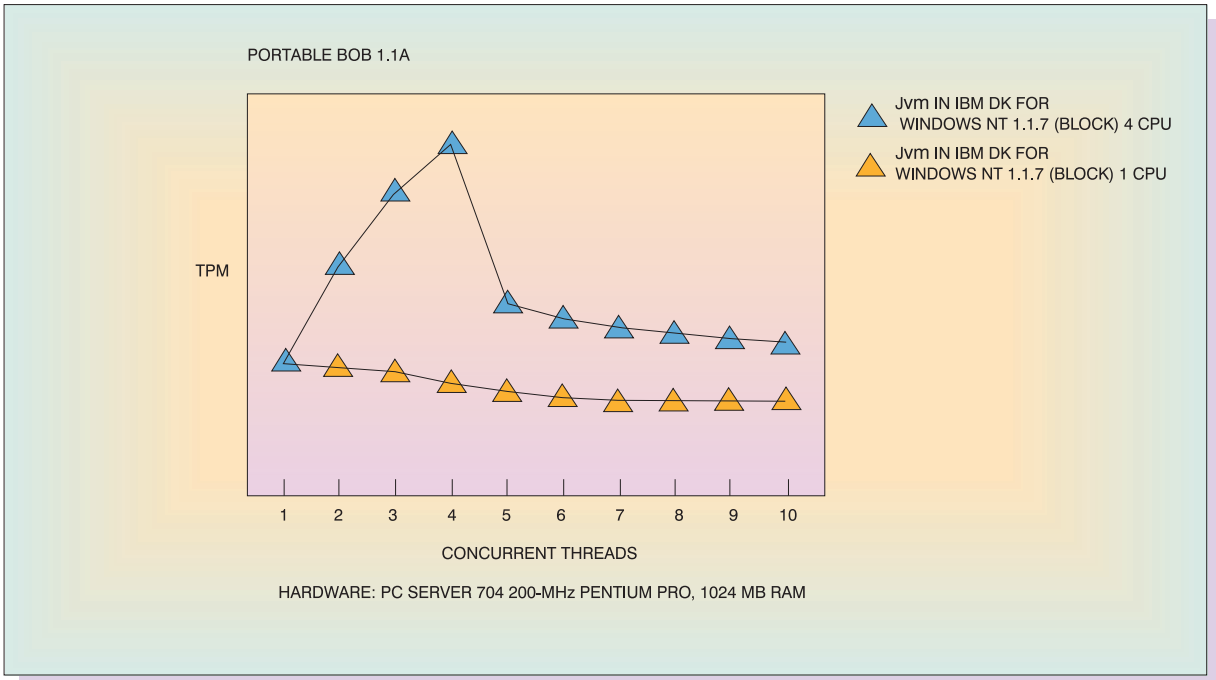
The acquisition and release time for an inflated lock is relatively more expensive. Although this operation occurs at a lower frequency than the acquisition and release of a flat lock, it is an area of concern and is under investigation.

Contention behavior. In general, when a thread tries to acquire a lock held by another thread, a number of possible actions can be taken. The most common are to (1) block waiting to be notified for the lock to be released, or (2) yield the processor until the thread is rescheduled and then try to acquire the lock again, or (3) spin consuming CPU cycles until the lock is released. In addition, combinations of the above can be implemented. For instance, one popular methodology is to spin for a given amount of time and then block. To add to this set of technologies there have been implementations that "sleep" for increasing periods of time when locks become "hot," those that dynamically set the spin time, and those that disable interrupts while the lock is being held.

Each of the different implementations has varying strengths and weaknesses that in turn depend on the workload and the environment in which the locking is being done. For instance, it has been found through empirical database studies done with DB2* (DATABASE 2*) that the optimal length of time to spin before blocking depends on the number of processors on the system. If there are more processors on the system, longer spin times yield increased performance. The appropriate spin length also depends on whether it is possible for a thread to be blocked while holding a lock. For instance, if it can be guaranteed that a lock will not be held by a blocked thread (possible with kernel-level locks that can turn off interrupts and guarantee that code is in memory to avoid page faults), then spin times of unconstrained length are often the right choice for optimum performance.[11]

The literature on this subject is also vast. Of note is a technique proposed by Mukherjee and Schwan concerning dynamically configurable kernel locks.[12,13] In addition, by using analytical models, Zahorjan and Lazowska found that multiprogramming and data-dependent uncertainty do not affect the choice of spinning versus blocking for simple mutex locks.[14]

Figure 1 Portable BOB (pBOB) throughput with blocking contended monitors



Empirical results for the spinning versus blocking are presented by Karlin et al.[15] Finally, the reference by Anderson compiles a good summary of synchronization methods with performance data.[16] However, none of these references presents empirical results from Java-driven applications or is concerned with product-level code, and for this reason, the following data are valuable.

Before the contention mechanism employed by the IBM Jvms is described, it is necessary to detail the structure of an inflated lock. In essence, the inflation of a lock provides the lock word with a mechanism by which it can suspend a thread; it associates a system semaphore with the lock structure. In addition to this system semaphore, the inflated lock structure contains the owning thread ID, the recursion count, a condition variable, and a pointer to the next lock on the chain (for inflated lock management and reuse). For the IBM DK for Windows implementation, the mutex lock that is used by an inflated lock is the Windows *CriticalSection*; for OS/2, *16-bit RAM semaphores* are employed. Both of these mutex implementations immediately block on a failed lock acquire and are notified when the lock is released by the owning thread.

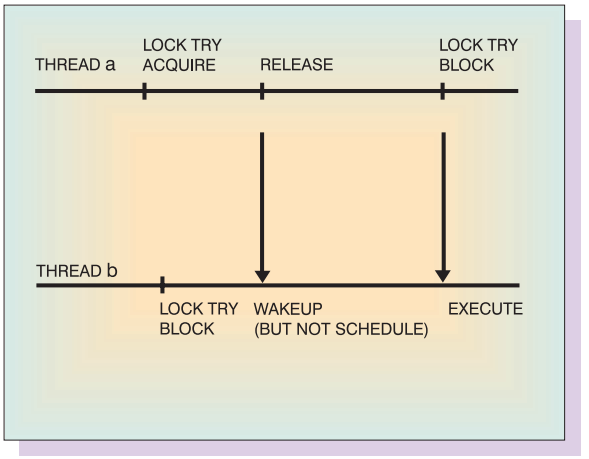
Figure 1 illustrates the results obtained with this blocking type lock under the stress of a multithreaded, highly contended workload on a multiprocessor. The workload used, called *portable BOB* (pBOB), was created by the SanFrancisco* project group within IBM and is representative of user applications implementing business logic using the SanFrancisco framework. pBOB stresses object allocation, garbage collection, threading, core Jvm scalability, JIT technology, and monitor implementation. The figure shows throughput as a function of the number of threads. The workload was run on a four-CPU, 200-MHz Pentium Pro** system with 1 GB of memory. As expected, throughput increases as the number of threads increases to equal the number of processors. At this point (four threads), one can simplistically view each independent thread running unencumbered on a dedicated processor. When more threads than CPUs are executing, the throughput precipitously drops off—45 percent less throughput at five threads when compared to four. The ideal behavior (assuming full CPU utilization) would be either slightly increased performance or a leveling off of performance as more threads are added.

Why does the performance degrade so much with the introduction of the fifth thread? Like many Java workloads, pBOB contains a few extremely highly contended locks.[17] When many threads are executing, the system operates in a mode where at most times threads are blocked waiting on one of these hot locks. This behavior is exacerbated by the fact that a lock can actually be held by a thread that is not currently active. For the sake of fairness, the Windows CriticalSection forces the thread that holds a lock to release (or hand off) the lock to a waiter before it can reacquire the lock after release (see Figure 2). The figure illustrates this behavior and shows thread b waiting until thread a tries to reacquire the lock before it is allowed to run. (Note: It is assumed that the CPU being used by *thread b* is utilized by a thread of equal priority while thread b waits.) This behavior, which is referred to as the convoy problem, thrashes the system by only allowing a thread to execute for a short time (thread a in the figure) before a context switch is required.

The solution implemented in the IBM Jvm is to alter the behavior upon a failed acquire attempt to avoid the convoy. A simple infinite-spinning lock, though, is not appropriate because a lock can be held by a blocked thread. In addition, yield-based solutions are not optimal because they are inherently unfair and result in too many context switches. The solution reached, through much empirical testing, was a three-tier spin/yield/block solution that was first proposed by a database group within IBM (see Figure 3). Note that the solution required that the operating system semaphore be exposed so that a failed acquire does not block immediately.

The solution exposes three separate counts for looping. The first is the amount of time to spin wasting CPU cycles before the lock is retried. The second is the number of retries before a system yield is attempted. Finally, the last parameter is the number of yields to attempt before a system block. The values that the IBM Jvm uses for these counts depends on the number of CPUs in the system and is proprietary. However, it is the case that for almost all locks in the system, the final block is avoided. The results are impressive, as Figure 4 shows. Not only has raw performance improved, the drop after four threads has been removed. The actions for contention on a single-CPU system are the same as that for the multiprocessor case (Figure 3) with the inner "while" loop removed. The improvement on the single-CPU system as shown in Figure 4 illustrates that the convoy effect can even occur in a limited way on a single-CPU system.

Figure 2 Summary of fair lock scheduling behavior



Monitor fairness. Monitors, or more correctly, the sections of code or data structures that monitors synchronize, are shared resources and, as such, the fairness with which the resources are provided to the requesting threads becomes an issue. Enforcement of lock access fairness is centered on the action taken when a lock is released. In general, implementing a higher degree of access fairness reduces overall throughput but improves response time. In this subsection we introduce and provide performance data for three types of monitor-lock fairness: strict, hybrid, and greedy-random fairness.

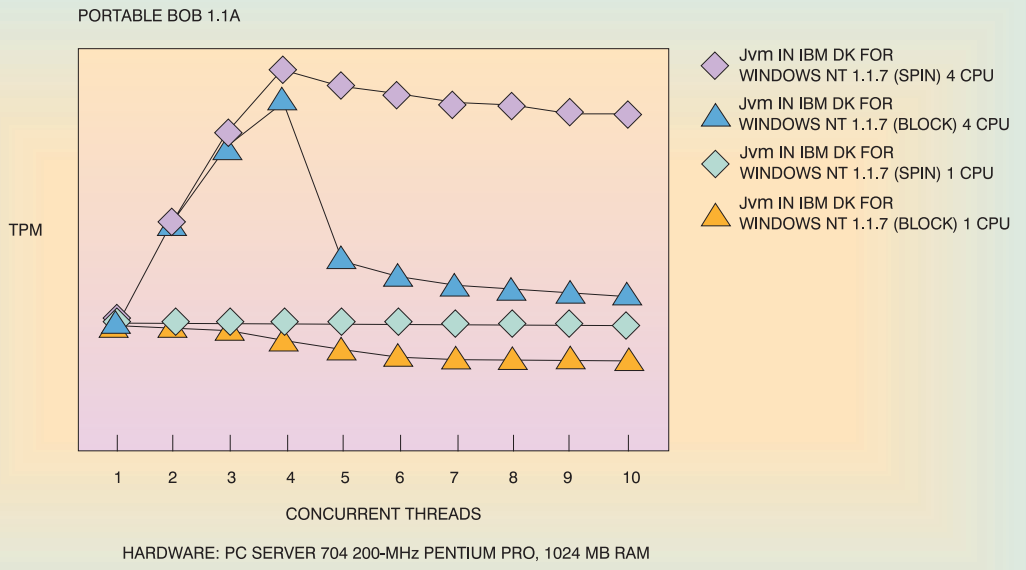
In a strictly fair locking implementation, the next owner of a released lock will be the thread that has been waiting the longest for it (or has the highest priority). If none is waiting, the current thread may reacquire the lock. The Windows CriticalSection underlying the spin structures described in the last subsection is strictly fair (see Figure 2). The monitor implementation provided in the Jvm in the IBM DK 1.1.7 is a perturbation of the strictly fair lock because of the three-tier spin lock. It will be referred to as the hybrid implementation.

In a random fairness implementation, all waiters on the lock are made runnable when a lock is released, and the next one scheduled gets the lock if it is free. This condition is sometimes referred to as the "thundering herd" problem, where all the threads stampede to obtain the lock.[18] If the thread that released

Figure 3 Pseudo-code for three-tier blocking lock

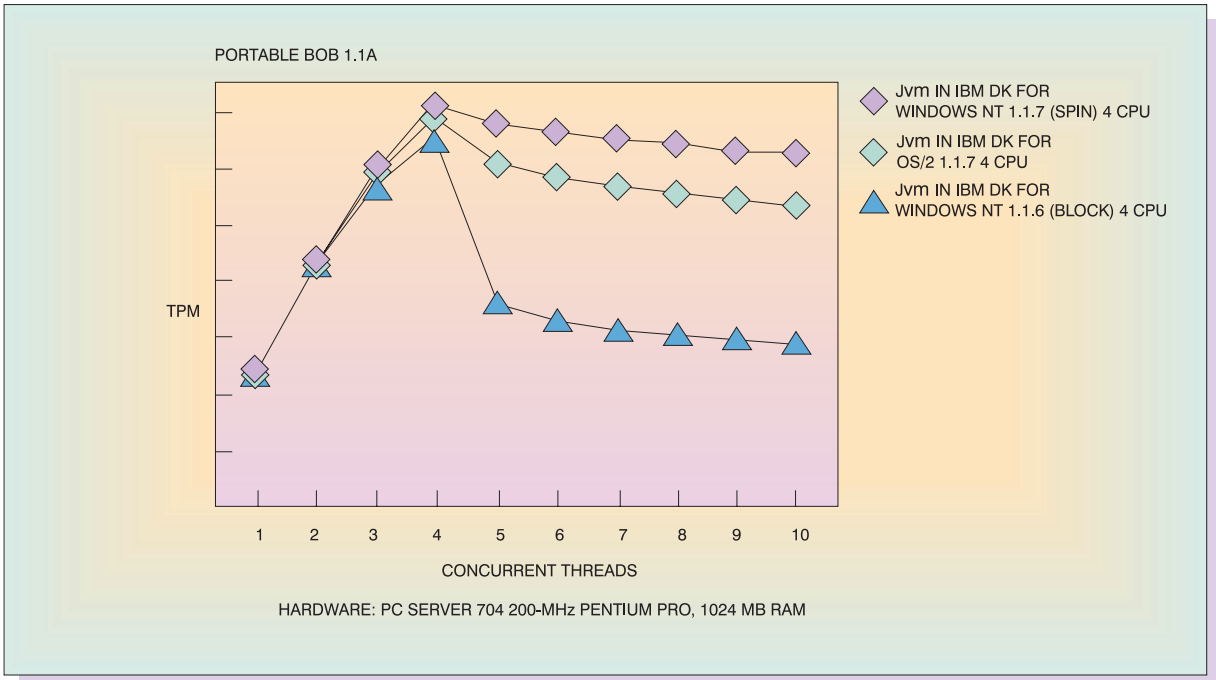
```
while ( (! Acquire(lock-exposed) && (count3 <LOOP3) {
        Spin(wasting count1 CPU cycles);
        while ( (! Acquire(lock-exposed) & (count2 <LOOP2)) {
                Spin(wasting count1 CPU cycles)
                count2++;
        }
        system_yield( );
        count3++;
}
system_block(lock);
```

Figure 4 pBOB throughput with blocking and spinning contended monitors



the lock is eligible to be the next owner of the lock when there are other threads waiting, the policy is described as greedy. In a greedy system, the running thread has the highest probability of acquiring the lock if a lock is hot and the system is busy. This aspect of the policy tends to reduce context switches (used to enforce fairness) and increase throughput because it allows a thread to run for a long time without context switching. The result is often excellent cache behavior. However, the greedy aspect allows

Figure 5 pBOB throughput for fair (NT block), hybrid (NT spin), and random-greedy (OS/2) monitor scheduling



for unfairness and thread starvation, especially when, as in the case of the Jvms, a thread holding a lock is not guaranteed to be running. The 16-bit RAM semaphores provided by OS/2 and used by the Jvms in the inflated locks are implemented in the greedy-random fairness style.

Figure 5 shows four-way results on three Jvms that are similar in most respects except for their locking fairness policy. The Jvm in the IBM DK for Windows, v 1.1.6 uses a strict fairness policy, the Jvm in the IBM DK for Windows, v 1.1.7 uses a hybrid policy, and the Jvm in the IBM DK for OS/2 uses a random-greedy policy. It should be noted that there are other differences between the Jvms in the DK for OS/2 and DK for Windows. For instance, the OS/2 implementation does not exploit the three-tier spin code (Figure 3). However, even with these differences, it is instructive to compare, with the proper caution, the performance of the Jvms. Note that from a throughput point of view, the hybrid and greedy policies are much more efficient than the strict policy. The apparent greedy policy loss as compared to the hybrid solution may more likely be the result of the thundering herd problem, so with the above data it cannot be concluded that the hybrid approach is more efficient than the greedy approach.

The differences between the strict Jvm in the IBM DK for Windows, v 1.1.6 policy and the random-greedy OS/2 policy are highlighted by the *InfoWorld* server benchmark.[3] This benchmark runs a highly repetitive, contended workload on hundreds of threads in a Jvm. The benchmark shows the greedy policy found in OS/2 able to sustain three times more throughput than the strictly fair policy of the Jvm in the IBM DK for Windows 1.1.6. The performance throughput advantage of this policy is even more compelling given that the OS/2 implementation suffers from the thundering herd problem. The strict policy ends up causing excessive context switches, whereas the OS/2 system allows work to complete on given threads. In contrast, the Windows solution allows all threads to complete approximately an equal amount of work in a given time. The OS/2 solution had shown some threads completing two to three times more work than others.

Object allocation

The evolution of the object allocation implementation within the Jvm is an excellent example of improving system software through iterative analysis and enhancement. Changes to this crucial component have been made by various divisions within IBM and also by Sun Microsystems. The changes were made to address two related, but distinct, concerns. First, due to the frequency of allocation, improved allocation efficiency was required for good performance on both client and server platforms. Second, the lock under which allocation is done, referred to as the *heap* lock, is one of the most contended locks in the system. Early versions of the Jvms exhibited excessive heap lock hold times on four-way systems. Given that hold times greater than 20 percent lead to exponential performance degradation,[11] it was observed that the object allocation implementation would not scale on a multiprocessor system. This section briefly describes the stages of object allocation evolution, the current implementation, and possible future directions. Direct measures of object allocation time are intentionally avoided, because they depend on many factors (i.e., heap fragmentation, object size).

Freelist introduction. Simply put, object allocation requires the Jvm to find a contiguous chunk of space within the heap large enough for the requested object and an object header. The first implementation kept a linear list of open chunks on the heap and would traverse this list looking for a large enough chunk. If none was found, then garbage collection would be tried. Not surprisingly, the performance of this code was poor.

This simple approach was first improved by introducing an array of lists that chained together free heap chunks of predetermined size (for details consult Gu et al.[6]). In other words, each freelist in the array kept the list of currently free objects of a given size. When an object of a certain size was requested, the correct freelist was consulted, and if it was not empty, the request could be satisfied quickly. If the list was empty, freelists of larger size were consulted. If all proper-sized freelists were empty, or if the initial request was larger than a certain threshold (512 bytes), a freelist of all objects larger than the threshold was traversed.

Thread local heaps. Because almost all objects requested are small (measurements indicate greater than 99 percent of the objects requested are less than 512 bytes in size), the introduction of freelists drastically improved the time required to satisfy an object allocation. However, it still required that the heap lock be acquired on each object allocation or whenever the freelists were consulted. Not surprisingly, this design did not scale when multiple CPUs were trying to allocate objects simultaneously. Sun addressed this problem in the 1.1.5 reference platform with the introduction of thread local heaps (TLHs).

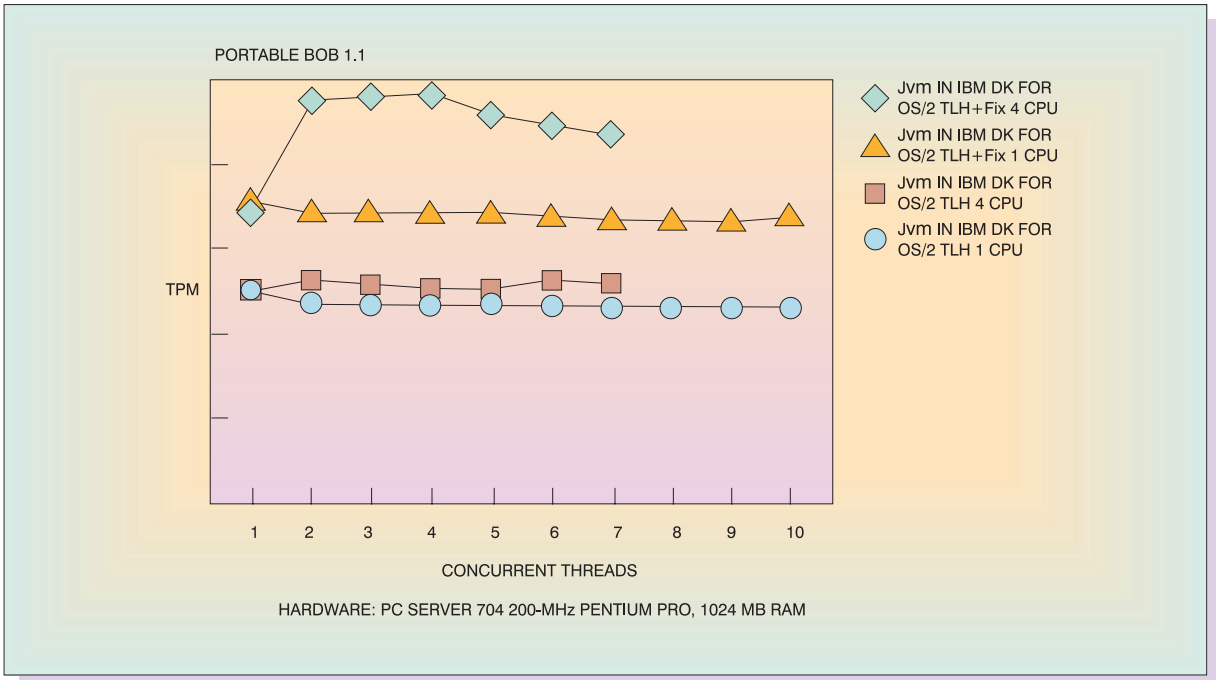
Contrary to expectations based on the name, the TLH design is not a two-level store with the heap divided among multiple threads. A thread local heap is nothing more than an object of a certain size (1024 bytes in the reference platform) to which a thread has exclusive access. To satisfy a typical small object allocation, the TLH is split at one end to release the required space. If the TLH is not large enough to satisfy an allocation, it is discarded, and a fresh TLH takes its place. Large allocations bypass the TLH mechanism and are satisfied directly from the heap. Because the thread has exclusive access to the TLH, the heap lock is not required for allocations satisfied from it.

Measurements have shown that performance can be improved by increasing thread local heaps from 1 KB to 1.5 KB in size, and this was done on IBM implementations. However, it is interesting to note that beyond this size not much improvement has been measured. In fact, if the required TLH size is too big, overall performance degrades. This issue is addressed in more detail shortly.

Freelists with thread local heaps. The introduction of TLHs made the initially introduced freelist structures ineffective. Most allocations would come from the TLH and not the freelists. Those allocations that did come directly from the heap manager were most frequently requests for new TLHs. Of course, there were some exceptions. For instance, objects greater than 0.25 of the TLH (384 bytes) went straight to the heap manager, and certain class objects also bypassed the TLH. But, for the most part, all requests to the heap manager were for TLHs themselves and were satisfied with a linear search of the large-object freelist.

For large heaps this method led to very long searches of the large-object freelist. This problem was exacerbated by the fact that with time, free chunks that could not satisfy a request of TLH size would gather at the front of the list, being passed over on each

Figure 6 Performance improvement of pBOB throughput with TLH freelist enhancement



TLH request. For certain runs of pBOB that were run in a 256-MB heap, repeated searches of over 3000 chunks were not uncommon. In addition, because the search was on the shared freelists, it was done under the protection of the heap lock.

Taken separately, the freelist and TLH fixes were both beneficial. However, when combined they produced the adverse side effect noted above. To address this inefficiency, a freelist was added that contained chunks exactly the size of the thread local heaps (1.5 KB plus object header). In addition, a separate list was added for requests between 512 bytes and TLH size. The large-object list now contained only objects greater in size than the thread local heaps. When the freelist containing chunks of TLH size ran dry, a chunk was obtained from the large-chunk list and split into multiple (up to 100) TLHs.

Figure 6 illustrates the improvement that this change yielded in pBOB performance (data for only seven warehouses are included because of temporary instability of the code base). Note that the memory work was done during an early stage of Jvm development so the pBOB curves look different than those shown in the previous section. The TLH specific pBOB fix resulted in a 46 percent improvement on uniprocessor systems. The improvement was even greater on a four-CPU system; the peak performance nearly doubled. This large improvement was a result of both raw path length improvement and enhanced scaling on a multiprocessor because hold time of the heap lock was reduced.

Dynamic heap growth. The initial reference implementation of the core Jvm was geared for client workloads and, as such, placed a high priority on limiting memory consumption. Often a major consumer of memory for Java applications is the heap. Therefore, initial Jvm implementations (i.e., the Jvms in the IBM DKs prior to version 1.1.4 and all 1.1.X Sun Jvms) were initialized with very small heaps (1 MB), and these heaps grew very slowly. To offer some respite from this constrained setting, which is wholly inadequate for a server workload, Sun provided the following two command line options: -ms and -mx, allowing the user to specify the minimum and maximum heap size, respectively. The minimum setting is the amount of memory allocated and committed for the initial heap. The maximum value is the

amount of virtual space reserved for the heap. (For reasons of efficiency and simplicity, it is important to keep the heap contiguous in virtual memory.)

These parameters provided some relief to server applications, but they had significant drawbacks. The stinginess of heap growth was still enforced, even at large heap settings. This would force the Jvm user to understand the heap working set size of an application *a priori* and set its minimum size to this value because the Jvm could not be relied upon to realize when a larger heap would benefit performance and grow accordingly. In addition, the Jvm would not recognize when the application working set size had decreased and reduce its heap accordingly. The latter is especially important given the mark/sweep garbage collector in use, which traverses the entire heap on each pass. Furthermore, the heap growth mechanism did not take into account the physical memory available and could indiscriminately grow larger than physical memory onto paging space. When such growth occurred, the performance of the Jvm and the system suffered tremendously. Finally, the average user does not want to set multiple settings for good performance but wants the system to dynamically react to the current workload. What was needed was a mechanism that would take the above considerations into account and dynamically grow the heap fast enough for optimum performance without overgrowing for the current system and workload.

IBM addressed the above concerns with a number of innovations. First, a simple change was made to link the initial heap size allocated in the default case to the total system memory available. The rationale was that a system with 4 GB of physical memory was more likely to be running a server application than one with 32 MB of memory and could more easily afford more memory per Jvm. The default -mx value was also set relative to system physical space available to avoid the case of the heap growing outside of physical memory and causing performance problems through excessive paging. This algorithm has proved useful on systems running a single Jvm but has caused problems that are being addressed for systems running many independent Jvms.

The heap growth algorithm was also altered with a bias toward server applications. This alteration was done without incurring a large detriment to memory-constrained systems. Before the enhancements are described, a cursory description of the reference platform implementation is now presented. The reference platform will expand the heap when less than a certain percentage of the heap is not free after a garbage collection. For the 1.1.X releases this value is set at 25 percent; for Java 2, version 1.2 this value is set at 35 percent. When the decision to grow the heap is made, the heap is grown the necessary amount to reach this required free space threshold. Analysis exposed two major weaknesses with this approach. First, if a workload creates a preponderance of short-lived objects (which is common), it is often the case that the 25 percent free goal is met. However, meeting the goal does not indicate that the heap is ideally set for the *working set* of the particular application. Second, heap growth itself is a costly operation and should be done infrequently. However, it was found that in the field, the heap would sometimes grow minuscule amounts (sometimes only a handful of system pages) and then be required to grow again in a few garbage collection iterations.

To address these two shortcomings, IBM redefined the concept of a *heap working set size*. The Sun reference implementation implicitly views the heap working set size as the space consumed by live objects (those not collected) after garbage collection. We enhanced this definition to also take into account the percentage of time spent in garbage collection at a given heap size. If a workload spent 50 percent of its execution time collecting garbage in a certain heap, its heap working set size would be considered larger than if it only spent 30 percent of its time collecting garbage. This perspective would be true even if the space consumed by the live objects after garbage collection was the same. This fundamentally different approach to gauging heap working set size allows the Jvm to treat workloads that create short-lived objects at a high rate differently than those that do not. It also allows the Jvm to dynamically tune the heap to both live object consumption and short-lived object creation rate.

In the Jvm in the IBM DK for Windows, v 1.1.7, heap growth was triggered if the given percentage of free space was not available after a garbage collection or if the ratio of garbage collection time to mutator time (Java application execution time) was greater than 0.13. This threshold was determined through experimentation. Ratios smaller than this number (faster growing heaps) did not show an appreciable difference in overall throughput.

In addition, because the act of growing the heap is relatively expensive, the IBM implementation increases the heap by a fixed percentage (17 percent)

once it is determined that the heap should grow. This percentage allowed the heap growth mechanism to scale with larger heaps and correspondingly larger workloads. The heap growth would still be capped by the specified -mx value and, for the sake of conservatism, the ratio heap growth trigger was disabled when the heap approached 75 percent of physical memory size.

Sun has also recognized the inefficiency of the original heap growth mechanism and has provided four new user-specified values to allow more user control. The Jvm in the IBM DK, v 1.1.8 implementations[19] also provides these new parameters, but it is recommended that users allow the default heap growth mechanism to adapt to the workload unless they are very familiar with their workload. Currently, IBM is enhancing the implementation to shrink the heap as the IBM defined heap working set size shrinks.

The effect of the new implementation can be measured by running server workloads with small or default initial heaps before and after the fix. For instance, when VolanoMark[20] is run in loop-back mode with an initial heap of 2 MB on a uniprocessor, the enhanced heap growth mechanism increases the throughput by 28 percent. This improvement is even greater on a four-way system, netting a gain of 70 percent. These results indicate that the working set size of a Java application, when defined in the new sense, increases as relative throughput increases (i.e., more short-term data are created). Similar experiments using pBOB with an initial heap of 2 MB show throughput increases of 28 percent and 66 percent for a uniprocessor and four-processor system, respectively. This improvement is solely due to the Jvm recognizing the correct working set size of the application quickly and setting the heap to that size.

Garbage collection

Not surprisingly, the inverse of object allocation, referred to as "garbage collection," is also a key area in the performance of Java server workloads. The impact of garbage collection on performance is exacerbated by the fact that the current Intel implementations "stop the world" by allowing only a single thread within the Jvm to be active while garbage is collected. The possible adverse impact on scalability of this implementation is obvious, so in response, IBM has made significant server-specific enhancements to the garbage collection implementation. These enhancements are summarized in this section, and as will be seen, some involve further enhancements to object allocation.

An overview of the mark/sweep/compact collector. The IBM Java garbage collector is derived from the reference implementation provided by Sun. It is a fairly traditional collector that marks live objects, sweeps to find free space, and compacts objects within the heap as required. For background, a quick overview is provided here; see Jones and Lins[21] for additional general information.

Garbage collection is a key area in the performance of Java server workloads.

Java objects are *marked* (noted as live) by following chains of references from a set of *root* objects to all other objects they reach. Marks are recorded in an area of memory allocated outside of the heap, referred to as *mark bits*. A single bit in the mark bit array is set as each new live object is discovered.

Root objects, the initial set of known live objects, are identified by a set of global references (such as objects referenced by JNI, the Java Native Interface) and through inspection of the dynamic state of the program. The entire run-time stack of each thread involved in the Java program is scanned, looking for pointers to objects. Because the location of pointers within the stack is uncertain, a combination of tests is applied to each value on the stack to determine whether it may be a pointer to an object. For instance, if the stack variable in question points outside the range of the Java heap, it certainly is not a pointer to a Java object. If, after these tests, a value appears to be a pointer to an existing object, that object is *marked* and included in the root set.

When an object is marked, a reference to it is pushed on a stack called the *marking stack*. The marking stack is processed by popping an object reference and scanning the associated object for references to other unmarked objects. Newly discovered objects are placed on the stack for future processing. The

process completes when all root objects have been examined and the mark stack is empty.

Once all live objects have been marked, free space is reclaimed during the sweep phase by coalescing sequences of dead objects and spaces that are not marked as live. In some cases, fragmentation caused by the live objects within the heap motivates a step following the sweep phase called *compaction*. Compaction moves objects toward one end of the heap with the goal of creating the largest possible contiguous free area or areas.[22] Object references are updated using a technique called *pointer reversal*, where an object is moved only after temporarily inverting any pointers to it. Because moving objects and the calculations required for compaction are CPU-intensive, compaction is performed sparingly. An example is when a large object must be allocated, but fragmentation of the free space prevents allocation from succeeding. In this case, compacting the heap avoids allocating new memory to expand its size.

Bitwise sweep. Given the importance of garbage collection to overall performance, a number of relatively straightforward enhancements devised across the IBM Corporation were included in the 1.1.6 implementations of the Jvms in the DKs on Intel processors. For example, a technique proposed at IBM Research[23] exploited structural information created when each class is loaded to accelerate the scanning process in both the mark and the compaction phases of the garbage collector. The peak pBOB benchmark score improved by more than 5 percent as mark time dropped by 55 percent and compaction time fell nearly 20 percent.

Before this enhancement to object marking, the mark and sweep phase of garbage collection took roughly the same amount of time. After the enhancement, the sweep phase dominated the time for garbage collection. This domination was especially true for large heaps with few live objects. Mark time is proportional to the number of live objects in the heap, whereas sweep time, because it must enumerate over all objects, is proportional to the sum of live and dead objects.

To reduce sweep time, a methodology that avoids inspecting individual dead objects was implemented. The idea was inspired by traditional copying collectors[24] and involves scanning the mark bits directly instead of enumerating the objects in the heap. During the mark phase every live object has a single mark bit set corresponding to its header. Therefore, a sequence of zero mark bits either describes a large live object or free space following an object. To distinguish these cases, the size of the live object under the set mark bit is inspected. If the object is smaller than the size implied by the sequence of zero mark bits, the zeros indicate free space following the object. This free space may have originally been occupied by many live objects, but there is no longer any need to inspect them.

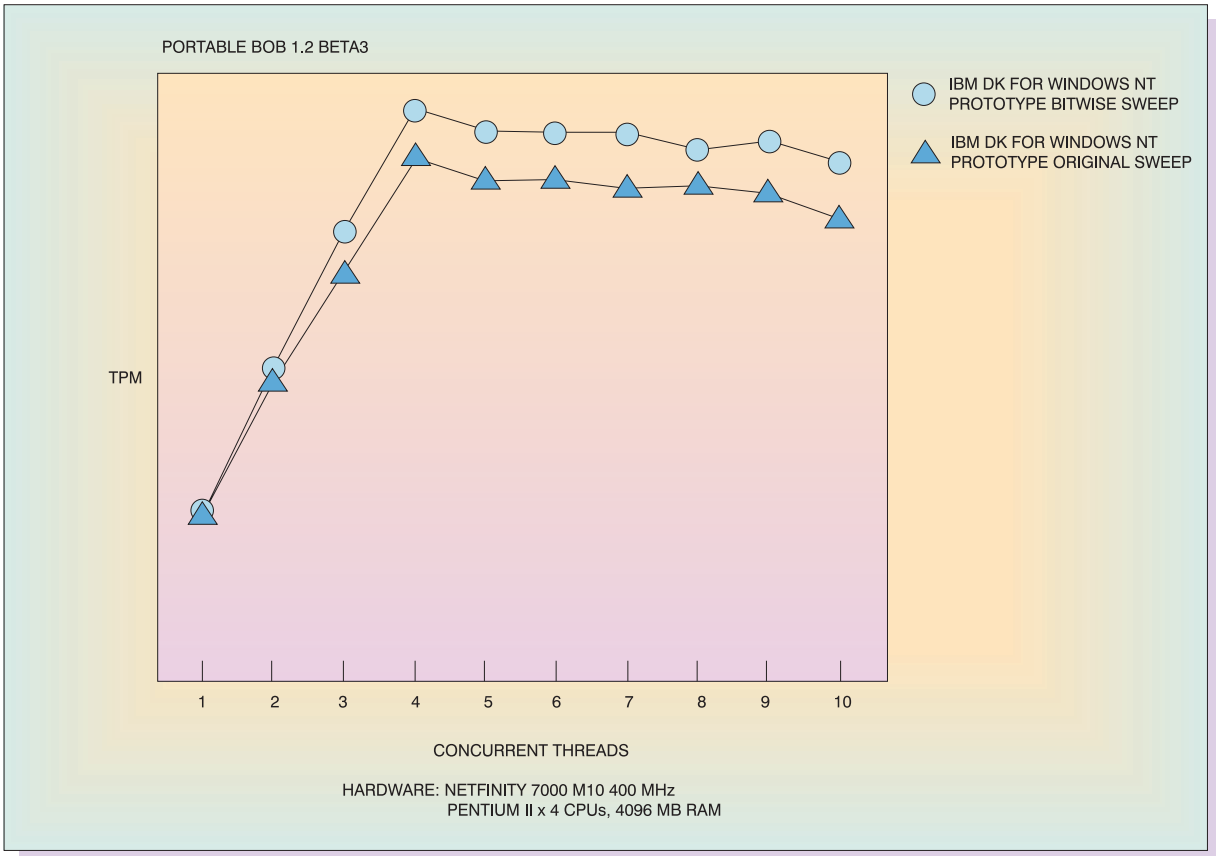
To accelerate the traversal of the mark bits, a threshold is established below which free space is ignored. That is, free areas less than a certain size are not reclaimed. This threshold establishes a corresponding lower bound on the number of consecutive zero mark bits needed to meet the threshold and require object inspection. Without this lower bound, the size of every live object would have to be inspected, including the common and unproductive case of small adjacent live objects. This free space threshold policy mirrors on the collector side the actual behavior imposed by thread local heaps on the allocation side. The smallest objects actually allocated from the heap are too large to allocate from a TLH directly.

An additional optimization speeds the process of finding an interesting sequence of mark bits (one with the proper number of zeros). The inner loop of the search inspects mark bits a byte at a time, and a word at a time when possible. To help terminate the search, a sentinel pattern with a leading set bit and a sequence of zeros is appended to the mark bit array. A pair of small tables translate arbitrary byte values into counts of leading and trailing zero bits.

Using these techniques, the bitwise sweep algorithm ends up spending most of its time in one of two productive activities. The first involves skipping runs of live objects, possibly intermingled with small unusable free areas. The second skips runs of dead objects, allowing them to be coalesced at very low cost. The result is a dramatic reduction in sweep time, often by more than a factor of five, yielding improvement in pBOB scores as shown in Figure 7.

Compaction avoidance. With the above enhancements, mark and sweep costs were improved to the point where compaction stood out dramatically in overall garbage collection time. Empirical results showed that the best overall performance was obtained by using a fragmentation heuristic that typically triggered compaction during every second or third garbage collection. But this led to compaction contributing approximately half of all garbage col-

Figure 7 Performance improvement of pBOB throughput with bitwise sweep enhancement



lection time. Without ruling out improvements to compaction itself, the next focus was on performing compactions less frequently. The inspiration for this tactic, nicknamed *compaction avoidance*, came from studying memory allocators in environments where objects cannot be moved. For example, C language heap managers have been available for decades and often perform remarkably well.[25]

In his Ph.D. dissertation,[26] Mark Johnstone studied memory fragmentation introduced by various allocation policies and made several key observations, some of which apply to Jvm allocators. First, he noted that objects that are born together tend to die together. Thus, if objects allocated at the same time are also physically near one another, there is a better chance that they can eventually be coalesced into a large free area to satisfy future allocations. Johnstone pointed out the potential locality advantages of this scheme both for caches and for virtual memory. Of the many allocation policies he studied, one of the best was also surprisingly simple—address-ordered first fit, where allocation requests are satisfied greedily from a single freelist maintained in address order. Johnstone asserted, "For a large class of programs using well-known allocation policies, we show that fragmentation costs are near zero."

Johnstone also discussed a key concept called "wilderness preservation," first introduced in a 1985 paper by Korn and Vo,[27] whereby memory allocators try to preserve a portion of memory in a pristine state. The wilderness improves the ability of the allocator to satisfy the occasional large allocation request.

Compaction avoidance is a Java memory management strategy built on these two fundamental con-

cepts: (1) choose a good placement policy, and (2) implement a form of wilderness preservation. By pursuing this approach, we have found that objects rarely need to be moved and that compaction is often unnecessary.

Object placement. Following Johnstone's advice, an address-ordered first fit allocator was created underlying the present TLH implementation. By greedily placing new objects in the first available location, address-ordered first fit inherently works to place consecutive allocations near one another. This policy supports sustained operation of the allocator with low demand for memory compaction.

Because of bitwise sweep, free areas smaller than a threshold size never appear on the freelist. This limitation reduces the size of the list, shortening searches for usable free areas. In keeping with this policy, the allocator discards remainders of split free blocks that fall below this threshold size. In a sense, these small free areas are dropped until they "grow up" and again become usable.

Another factor makes significant contributions to the success of this allocation scheme. The introduction of variable-sized TLHs has a profound effect on overall performance. The initial motivation was that fixed-size TLHs increased fragmentation by ignoring significant memory available at slightly smaller sizes. It also seemed reasonable to allow TLH sizes to be somewhat larger than usual when such allocations were possible.

Variable-sized TLH support is based on the concept of a desired size, "T." Integration with address-ordered first fit placement is as follows. First, the minimum acceptable TLH size matches the threshold of minimum size for anything on the freelist. Then by definition, the first item on the list is either an acceptable TLH or can be split to produce one. If the first item is of any size up to T, it is used "as is." If the size of the first item is between T and 2T, it is split evenly; the remainder is likely to be taken as the next TLH allocation. If the first item is larger than 2T, it is split to yield a TLH of size T.

Data gathered from several benchmarks using various heap sizes showed that choosing 6K bytes for T results in a good balance between making large TLH allocations when possible and avoiding excessive memory consumption in small heaps with large numbers of threads. Finding a suitable threshold was not difficult; the allocator is largely self-tuning, reducing average TLH size as the heap becomes fragmented.

An obvious concern when implementing the address-ordered first fit policy is the cost of searching the linear list of free areas. In a typical first fit scheme, free areas early on the list rapidly become small as allocations are greedily satisfied from them. Thus, the tendency is for relatively large allocation requests to experience long searches as small items begin accumulating at the beginning of the list. But because TLH allocations typically dominate other allocations from the freelist, often by more than ten to one, there is little accumulation of smaller free space entries at the head of the list in practice. The TLH allocator eagerly consumes the first free space on the list, keeping average search lengths for other objects small in the general case.

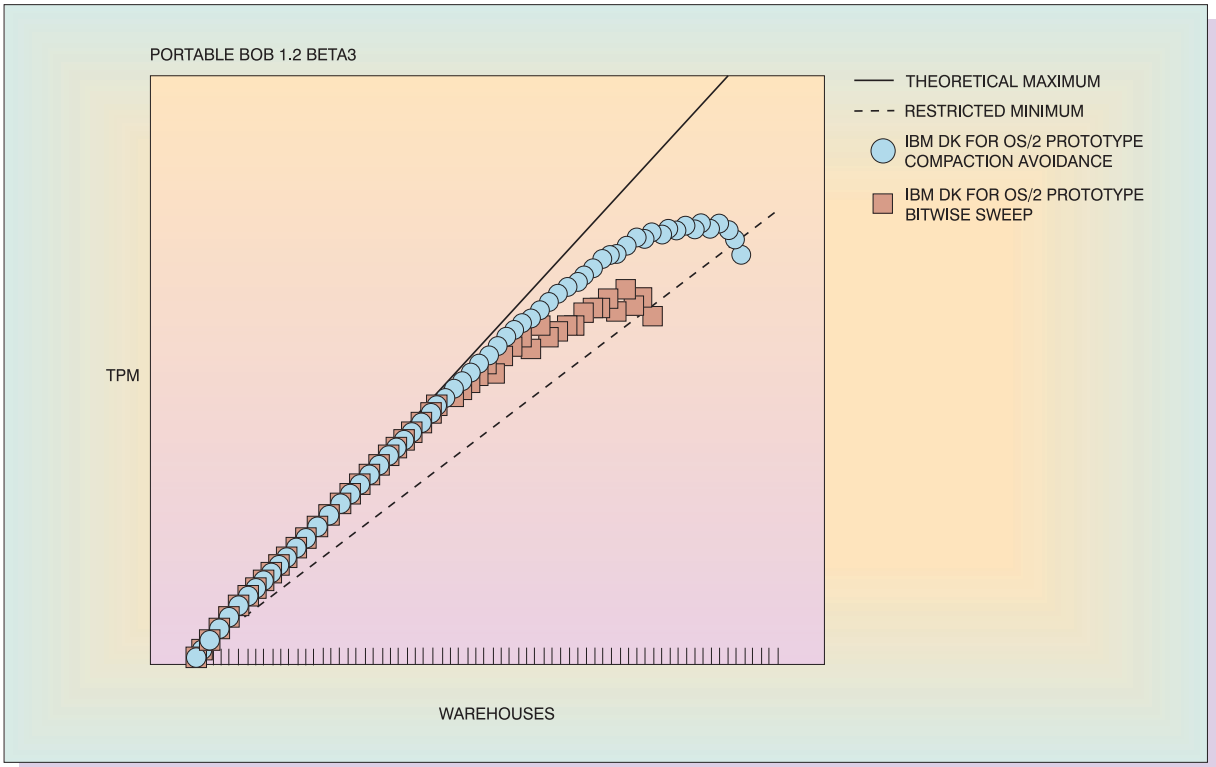
Managing the wilderness. Without further modification, the existing garbage collector automatically triggers a collection cycle when the entire heap has been exhausted. The C language run-time analogy is to artificially defer all calls to the free( ) function until after malloc( ) encounters low free memory. This does not lend itself to wilderness preservation, as continuing allocation activity carves up all available memory. Adapting wilderness preservation to Java memory management led to exploring methods of triggering garbage collection before the wilderness is consumed.

A key issue in implementing wilderness preservation is determining the proper size of the reserved area. Clearly, this determination involves a balance between competing demands. Wilderness area is (by design) largely unused by the program and should be of minimal size. But without some wilderness area on hand, large allocation requests are more likely to force a compaction. After measuring a variety of Java programs, a reasonable compromise was determined empirically. The target wilderness size is 5 percent of the heap, with an upper bound size of three megabytes. Wilderness size is dynamically reduced when the heap becomes excessively full.

Wilderness area is preserved by setting aside free areas that are beyond the wilderness boundary. Only nonwilderness memory appears on the freelist. When an allocation request cannot be satisfied from the freelist, the allocation progress since the previous garbage collection is checked. If sufficient progress has been made, garbage is collected by marking and sweeping the heap. Otherwise, an attempt is made to allocate the object from the wilderness.

Compaction avoidance results. A variant of pBOB, its *auto server* mode, challenges a Jvm by simulating

Figure 8 Performance improvement of pBOB autoserver with compaction avoidance



many users via multiple threads and simulated pauses (think times). Twenty-five additional threads interact with the data in each new warehouse. The test completes when throughput falls below a *restricted minimum* threshold derived from the number of active threads. In this environment, compaction avoidance makes an especially notable improvement. Compaction events taking tens of seconds are eliminated from the run, which aids throughput and dramatically reduces pause times. Compaction avoidance shifts this benchmark from being CPU-limited to being memory-limited. In Figure 8, throughput falls below the restricted minimum only as the heap becomes full.

Java network I/O

The Java network subsystem is a crucial part of the Java run time for good server performance. This subsystem includes the Socket and ServerSocket classes in java.net and the associated run-time code that maps these abstractions to the native sockets application programming interface (API). Additionally, this subsystem can be thought to extend and overlap with the java.io subsystem since the Java abstraction for data transfer is input and output streams. Finally, some aspects of threading are also included in this section in acknowledgment of the fact that the Java platform commonly requires the server to devote a thread to each socket that it services.

Performance goals and metrics. Since the Java socket APIs and run time provide an abstraction above those provided by the base operating system, it is actually possible to set tangible goals for Java network performance relative to the base operating system. The Java sockets API *should* impose a minimal overhead above the native sockets API. Thus one metric of Java socket performance would be the percentage of native performance obtainable through the Java sockets API, with the upper bound being 100 percent.

The above metric represents the cost of the API and would be measured by an API microbenchmark described in the next subsection. Also of interest is how improvements to the Java networking subsystem would improve the throughput and scalability of real applications. Scalability can be expressed both in terms of throughput speedup with multiple processors, which we refer to as *symmetric multiprocessor* (*SMP*) *scaling*, and the aggregate number of clients or connections that can be supported, which we refer to as *connection scaling*.

Performance benchmarks. An IBM sockets microbenchmark called SockPerf[28] was used to measure the cost of the Java sockets API. SockPerf is a peer-to-peer socket benchmark written completely in the Java language. The benchmark is multithreaded, and concurrent sessions can be run over the same network interface or over multiple network interfaces.

For the native sockets measurements, an internal IBM socket benchmark was used that runs on OS/2 and Windows NT** called *XMPT*. XMPT is functionally identical to SockPerf and indeed formed the template for SockPerf when it was created.

For this study, the following four tests were defined:

1. TCP_RR_1: Client sends a one-byte message (request) to the server over a Transmission Control Protocol (TCP) socket, which echoes it back (response). The result of the test is reported as a throughput rate of transactions per second, which is the inverse of the round-trip time for request and response, as well as the CPU utilization of the client and server.
2. UDP_RR_1: Client sends a one-byte message (request) to the server via a datagram with the User Datagram Protocol (UDP), which echoes it back (response). The reported result is in transactions per second and CPU utilization of the client and server.
3. TCP_Stream_8k: Client sends continuous 8-KB messages to the server, which continuously receives them. The reported result is bulk throughput in kilobytes per second and megabytes per second and client and server CPU utilization.
4. CRR_64_8k: Client sends a 64-byte message (request) to the server over a TCP socket, and the server sends back an 8-KB response. The connect, request, response (CRR) test includes the connection setup and teardown costs in the timing loop and is designed to simulate an HTTP (HyperText Transfer Protocol) 1.0 transaction.

Internal versions of the SockPerf and XMPT benchmarks were used that had access to OS/2 kernel instrumentation for CPU utilization. Since CPU utilization could be very accurately collected, a metric called *scaled throughput* is defined as the figure of merit for each test. Scaled throughput was computed by dividing the raw throughput by the average of the client and server CPU utilization. Dividing the Java scaled throughput for a given test by the corresponding native scaled throughput results in a metric of Java performance as a fraction of native performance. In addition to each of the individual tests above, an overall score for both Java and native performance was also computed. This score was the geometric mean of the scaled throughputs in each case.

Scaled throughput is an estimator of software efficiency or path length. Thus, the ratio of Java-to-native scaled throughput is a measure of the additional path length imposed by the Java platform on top of the native sockets API.

A popular industry benchmark, VolanoMark 2.1.2,[20] is used for the throughput and scaling study. VolanoMark is suitable because it is a benchmark that does real work, i.e., Internet chat serving, and it heavily stresses the Java network subsystem.

VolanoMark simulates many clients using an Internet chat server. As part of the setup, a specified number of "chat rooms" are established, with 20 users per chat room. The total number of users can be controlled by varying the number of rooms. Each user sends a message to the server, which echoes it back to the other users in the room. The result of the test is a score that represents the throughput, i.e., the number of messages transferred per second.

VolanoMark was also used to measure the connection scaling of our Jvm by measuring the maximum number of connections that could be opened by clients to the server.

Performance improvements. Figure 9 shows SockPerf and XMPT results gathered between a pair of IBM PC Server 704 systems, each with a 200-MHz Pentium Pro processor and 1-GB RAM, running OS/2 Warp Server SMP with TCP 4.1. The machines were connected to an isolated 100-Mb Ethernet network. Each of the tests listed above is shown as well as an overall geometric mean. The native results are normalized to one, and the Java SockPerf results are shown relative to the native results.

Figure 9 SockPerf results

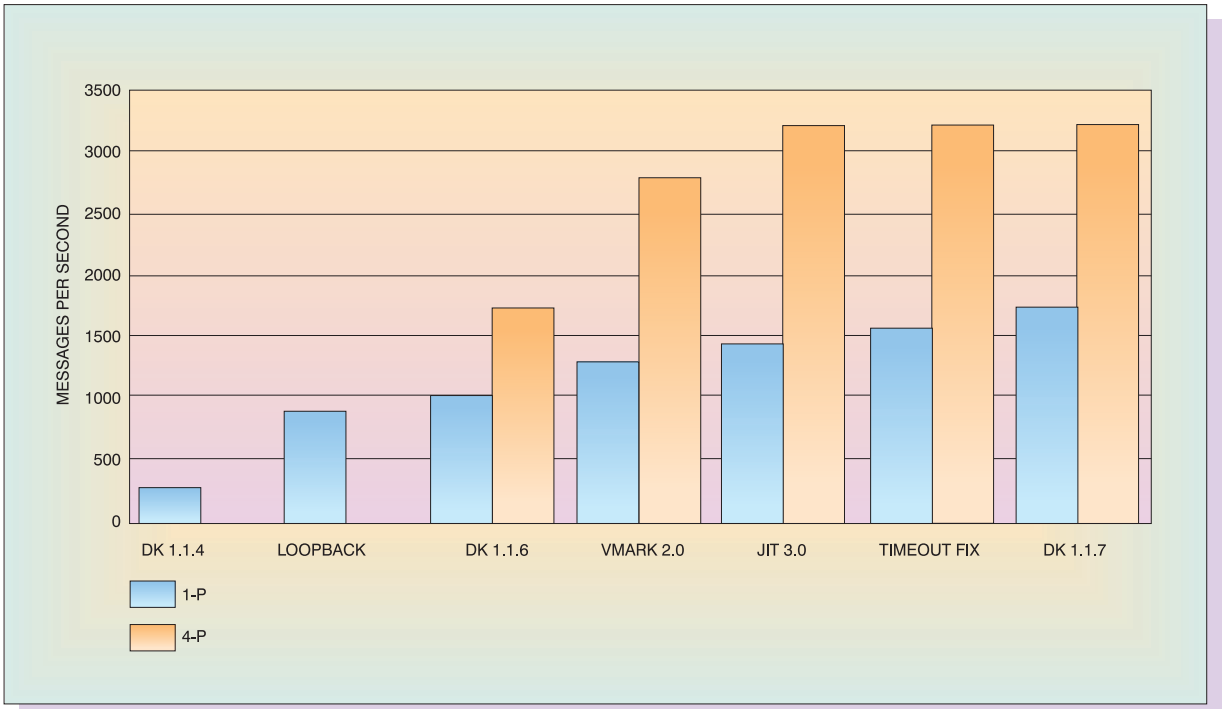
The performance improvement from an initial low in IBM DK, v 1.1.4 to DK, v 1.1.7 is very clear. In DK, v 1.1.4, Java socket performance was only 60 percent of native performance. The set of improvements that took performance up to 95 percent in DK, v 1.1.7 is described below.

The first problem found was in platform-specific code. In an attempt to obtain the thread context, the function preBlock( ) in the Java run time was setting an OS/2 exception handler and then firing the exception. Obviously, this way was very expensive to obtain a thread context. This condition was fixed by using a better API, DosQueryThreadContext( ). Additionally, a check was put in place whereby this call was avoided altogether in the common case.[29] The result of this change is shown by the bar representing Proto 1 in Figure 9.

Another improvement that affected performance dramatically was the FindLoadedClass( ) routine in the Jvm. This routine locates a reference to a class that has been loaded by the Jvm using a linear search which consumed CPU cycles as the number of classes grew large. This problem was originally discovered at IBM Research. Changing the linear search to a binary search helped performance considerably as shown by the Proto 2 bar in Figure 9.

UDP appeared to have an additional performance problem. The problem turned out to be as follows: On every invocation of DatagramSocket.receive( ) in Java, the Jvm function java_net_PlainDatagramSocketImpl_receive( ) would create an InetAddress object into which would be placed the address information of the sender of the datagram packet. This object would then be attached to the DatagramPacket object that was passed down on the receive( ) call. Unfortunately, the cost of creating a Java object *while in Jvm C code* is very high. It requires a call to execute_java_constructor( ) that is very expensive. A fix was devised where an InetAddress object was only created if one did not already exist. This overhead is eliminated by assuming that the application reused its DatagramPacket object for receive. The effect of this improvement is shown by the Proto 3 bar in Figure 9.

Figure 10 VolanoMark performance history on OS/2



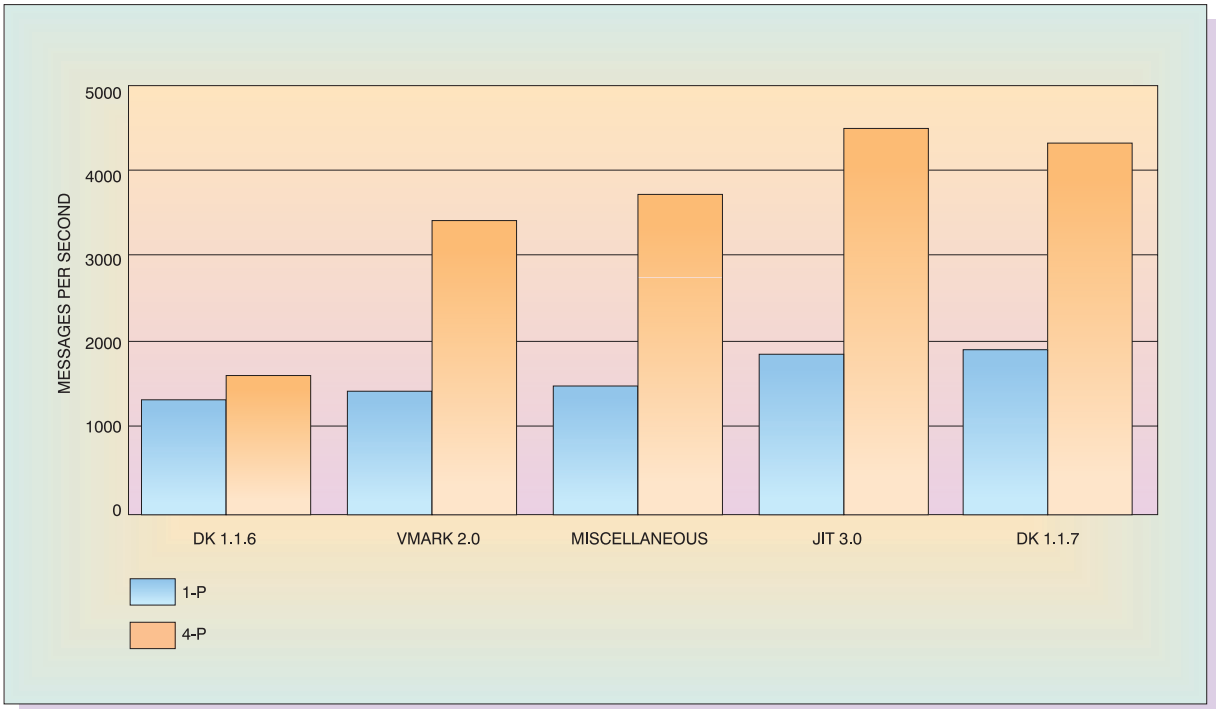
The above improvements were achieved by use of the SockPerf microbenchmark. At about the time these improvements had been completed, the VolanoMark benchmark was emerging as an industry Java server benchmark. Although the changes effected via SockPerf certainly improved VolanoMark throughput, the latter was not available during this improvement cycle and so did not directly drive these changes. Instead, VolanoMark was used to make additional performance enhancements.

VolanoMark employs hundreds, and even thousands, of threads and sockets. As such, it produces a lot of stress on the system. In fact, it was instrumental in detecting several functional problems in the Jvm and even the underlying operating system and TCP stack. These problems are not described in the paper, but serve to highlight the fact that a complex real-world benchmark such as this one tests not just the Jvm but how well it maps to the operating system.

Figure 10 shows the history of VolanoMark loopback throughput on OS/2. The test was run in loopback mode (i.e., the client and server run on the same system) on an IBM PC Server 704 system with a 200-MHz Pentium Pro processor and 1-GB RAM, running OS/2 Warp Server SMP with TCP 4.1. The initial bar shows the throughput of VolanoMark 1.0 on IBM DK, v 1.1.4. Very early, a bug was detected in the OS/2 loopback implementation that was resulting in dropped Internet Protocol datagrams. By fixing this bug, the score in the "loopback fix" bar was achieved. The next bar simply shows the cumulative effect of the various performance improvements that went into DK, v 1.1.6.

VolanoMark had a flaw whereby the benchmark was unable to saturate the CPU of the system even while running in loopback mode. On the basis of a change proposed by one of the authors to set the TCP_NODELAY flag, Volano LLC changed their benchmark and released VolanoMark 2.0. The effect of the benchmark change is shown in the "VMark 2.0" bar in Figure 10. The benchmark now saturates the CPU in both uniprocessor and SMP tests. All subsequent measurements mentioned in this paper will be assumed to refer to version 2.0.

Figure 11 VolanoMark performance history on Windows NT



One of the primary performance enhancements to the Jvms in the DKs between versions 1.1.6 and 1.1.7 was version 3.0 of the IBM JIT compiler, referred to here as JIT 3.0. JIT 3.0 had a profound impact on VolanoMark throughput. In addition to better code generation, JIT 3.0 drastically reduced the overhead of method invocations. The "JIT 3.0" bar in Figure 10 shows the improvement that resulted.

The "timeout fix" bar in the same figure reflects another improvement to the Jvm network code. Java provides a mechanism for the programmer to specify a timeout on a socket read via the Socket.setTimeout( ) API. The Jvm was coded to conservatively assume that the underlying native sockets implementation did not support a timeout feature. In fact, the OS/2 TCP 4.1 stack *does* support socket timeout via the SO_RCVTIMEO socket option. The conservative Jvm behavior was to issue a select( ) prior to a recv( ), using the timeout flag on the select( ) call. The rationale for this is easy to understand: most TCP implementations support select( ) but few support SO_RCVTIMEO. However, on OS/2 a significant gain was realized by short-circuiting the select( ) call.

Some of the improvements just listed also applied to the Jvm in the IBM DK for Windows. Figure 11 shows a similar history of improvements on the Windows platform. In these measurements, the same computer hardware was used running Windows NT 4.0 Server, Service Pack 3.

Connection scalability improvements. As the scope of the Java platform on the server grows, it is being used to develop highly scalable server applications. These applications require the ability to maintain thousands of concurrent long-lived connections. Java applications must typically devote a thread to each connection on which they receive data. Thus applications that require thousands of concurrent connections require thousands of concurrent threads and sockets from their Jvm. Threads and sockets are system resources and, as such, are finite commodities. Moreover, they consume other system resources such as virtual memory. Once the thread, socket, or memory limit is reached, no further connections can be established. VolanoMark is typical of this class of application. The design of the Volano server is such that it uses two threads per client connection.

Figure 12 VolanoMark connection scalability history

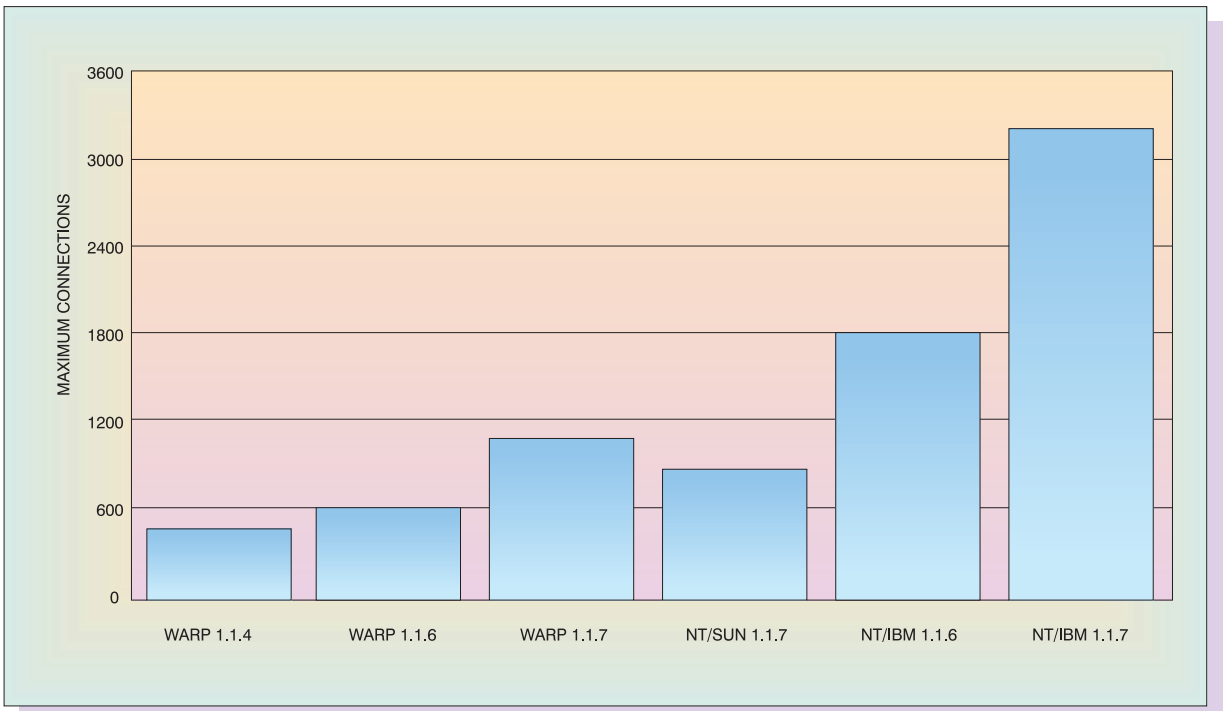


Figure 12 shows the history of connection-scaling improvements in the OS/2 Warp and Windows NT environment. In both environments, the critical resource is virtual memory. The pressure on this resource comes from the combination of the Java heap as well as the native thread stacks of the threads. On OS/2, each process has a private memory space of 512 MB. On Windows NT, this space is 2048 MB.

On OS/2, the connection limit on DK, v 1.1.4 was only 460 connections. By eliminating the use of select( ) for socket timeout, the limit rose to 600 in DK, v 1.1.6. The reason was that select( ) uses a bitmap per socket that consumes memory. In DK, v 1.1.7, this limit rose further to 1100 by moving the Java heap to a region above the 512-MB boundary, called *high memory*. Ultimately though, the connection limit on OS/2 remains modest because thread stacks must reside in the 512-MB region and cannot be moved to high memory. Moreover, thread stacks consume memory in segments of 64 KB. Thus, even though the user can specify the desired size of thread stack via the -ss command line option, the minimum thread stack consumption is still 64 KB.

On Windows NT, the amount specified via the -ss command defines the amount of virtual memory actually allocated for each of the thread stacks of the Jvm. However, unless otherwise specified, the operating system still reserves 1 MB of virtual memory by default for each thread. Since Windows NT allows a total 2 GB of virtual memory for the private address space of an application, the 1-MB stack reservation per thread effectively limits the Jvm to approximately 2048 threads (2048 MB total virtual space divided by 1 MB per thread equals 2048 threads). The Jvm in the IBM DK for Windows achieves a much higher effective thread limit by reducing the virtual memory reservation per thread from its 1-MB default.

It is easy to see why the Sun Java Development Kit (JDK**) 1.1.7 tops out at 920 connections. Given the default Windows NT thread stack size reservation of 1 MB, and the fact that the Volano server uses two threads per connection, an upper bound on the VolanoMark connection limit is set to 1024. Consumption of other memory by the process accounts for the gap between 920 and 1024 connections.

Table 1 System.arraycopy( ) performance improvement for character arrays

Array size	1	5	10	100	1000
Function call (us)	0.51	0.55	0.65	1.63	6.35
Inline/align arraycopy (us)	0.15	0.19	0.24	1.21	4.75
Percent reduction	71%	65%	63%	26%	25%

On IBM DK, v 1.1.6, the thread stack reservation was changed to 512 KB, resulting in a doubling of the connection limit. In IBM DK, v 1.1.7, this reservation was reduced further to 256 KB, and the connection limit rose to 3260.

JIT compiler enhancements

The relative importance of Java compiled code is different for client and server workloads. Client workloads spend the majority of their time executing Java bytecode or in the Abstract Window Toolkit (AWT), and therefore, their overall performance is highly sensitive to the quality of the "JITed" (JIT compiled) bytecode. In addition, the application or applet startup time, which is often dominated by the time spent "JITing" (JIT compiling) the classes, is crucial on a client system. In contrast, a server workload generally "JITs" bytecode infrequently and spends a relatively smaller percentage of the total CPU time executing the JITed code. For instance, a three-tier database application that had its second tier written completely in the Java language spent less than 10 percent of that tier executing JITed code. A representative workload based on the SanFrancisco framework was measured to spend less than 20 percent of its time in JITed code, and WebSphere* servlet workloads have been seen to only consume 13 percent of the CPU time with JITed code.

However, this is not to say that the JIT compiler is unimportant to server workloads. The three-tier application mentioned above shows a 35 percent slowdown when the JIT compiler is not used, and a San-Francisco-based workload degraded about 32 percent without the JIT compiler. It is the case, though, that server workloads place different stresses on the JIT compiler than client workloads. Significant JIT compiler improvements have been made by the IBM Tokyo Research Lab for both client and server workloads.[30] This section summarizes some of the server-specific changes.

One family of functions common to server workloads that has been hand-tuned by the Tokyo Research Lab consists of memory movement operations (e.g., functions mimicking C library memset, memcpy, and memmove). Specifically, the Java language method System.arraycopy, which copies arrays of data, was tuned by first "inlining" a check to see whether the size of the copy is small. If it is, then it is done immediately without a function call. Otherwise, a call is made to code that guarantees data alignment on an eight-byte boundary. The eight-byte boundary is important for optimum performance on a Pentium** II system. Table 1 summarizes the results showing greater than three times improvement on small copies that make calls out to a memmove type function.

There is no corresponding memset operation in the Java language specification, but it is still necessary and common to set an entire array to a single value. This shortcoming has been a sore spot for Java application writers, and a number of clever techniques have been introduced to improve the efficiency of initializing a primitive array to a common value. The most common technique used is to initialize a smaller piece of the array and use the System.arraycopy call to fill in the rest of the array in an expanding binary fashion. The IBM implementation recognizes this shortcoming and remedies it through JITed code. If an application is initializing or setting a primitive array to a common value, the JIT compiler recognizes the code (Figure 13) and generates a call to an optimized C-library memset call. In this way, byte arrays can be initialized as much as 16 times faster. Table 2 illustrates the performance improvement obtained through this enhancement.

Finally, a number of Java library functions have been identified as crucial to server performance. These functions have been specifically tuned using either built-in or inline technology. An especially interesting case is the Random.next( ) method. The implementation of this function requires all methods that access the seed of the Multiplicative Linear Congru-

Figure 13 Sample Java code required to initialize an array

```
Int[] ARR = new ARR[SIZE];
for (int i=0; i<ARR.length; i++)
        ARR[i] = value;
```

Table 2 Improvement in array initialization with JIT recognition

Byte array size	10	100	1920	10000
X-times speedup "memset"	1.6X	4.8X	16X	12X

ent Generator to be synchronized. This over-synchronization can result in poor performance on multithreaded applications running on multiprocessor systems. This limitation was avoided by building a special Random.next( ) into the JIT compiler that uses efficient machine-level primitives (cmpxchg on Intel processors) to access the seed. In this way, the integrity of the seed is maintained without synchronizing the full method.

Conclusions

As the Java platform has matured, its importance has grown in the server space. We have described the enhancements to the IBM Jvms on the Intel-based platforms that have enabled them to be the highest-performing Jvms in the industry on server workloads.

Although Java server applications spend a significant amount of time executing non-Java code, e.g., in the operating system or the network or file system, it is nonetheless true that the quality of the Jvm run time and the JIT compiler can and do have a profound impact on the performance of these applications. This statement is consistent since much of the performance of the Java run time is affected by the efficiency of the mapping of the Java abstraction of system services to the underlying operating system services. The enhancements to the IBM Jvm reinforce this theme. Both the monitor improvements and the network I/O improvements described here relate to the notion of exploiting the underlying operating system to the fullest.

Garbage collection and the underlying memory management are possibly the most important subsystems of the Jvm. We have described improved algorithms and heuristics to reduce both the frequency and duration of garbage collection. Using better schemes for memory management, the IBM Jvm allocates memory fast and in a manner that expedites its collection.

Similarly, the JIT compiler has a profound impact on application performance. Even for Java server workloads, where a relatively smaller fraction of CPU time is spent in JITed code, the quality of the JIT compiler is a key factor in application performance. We have described the impact of improvements in the IBM JIT compiler on key server workloads.

Looking to the future, we are exploring improvements to compaction in cases where it cannot be avoided. To improve the scalability of the collector, parallel versions of mark and sweep are under consideration. In the longer term, support for type accuracy will allow the removal of conservative assumptions from the garbage collector,[31] thus enhancing our ability to exploit generational collection techniques.

Other areas of exploration are: heap shrinkage to track working set reduction, performance of inflated monitors, and further increases in connection scaling, especially on OS/2.

Acknowledgments

The work presented in this paper is the culmination of the labor of many persons. The authors would especially like to thank Ben Hoflich, Walter Fraser, MengWu Wang, and Robert Reynolds. Thanks also go to the DB2 database group for their informed discussion of locking.

Cited references and notes

1. 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).
2. J. Neffenger, "The Volano Report: Which Java Platform Is the Fastest, Most Scalable?" *JavaWorld* (March 1999).
3. T. Young, "No More Mr. Slow for Java," *InfoWorld* (September 1998).
4. "SPEC JVM98 Results," Standard Performance Evaluation Corporation (SPEC), Manassas, VA, http://www.spec.org/osg/jvm98/results/index.html.
5. "IBM Java Performance Update," IBM Corporation, http: / / www.software.ibm.com / os / warp / performance / javaperf_1298_update.htm.
6. W. Gu, N. A. Burns, M. 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).
7. E. Armstrong, "HotSpot: A New Breed of Virtual Machine," *JavaWorld* (March 1998).
8. D. Bacon, R. Konuru, C. Murthy, and M. Serrano, "Thin Locks: Featherweight Synchronization for Java," *ACM Conference on Programming Language Design and Implementation* (June 1998), pp. 258–268.
9. J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Transactions on Computer Systems* **9,** No. 1, 1–20 (February 1991).
10. All IBM Jvms are based on a handleless object model. The handleless design, along with other optimizations, freed enough space per object to provide room in the header for the 24-bit lock.
11. W. Alexander, R. Dimpsey, and B. Olszewski, "AIX Operating System SMP Performance," *AIXpert*, 35–42 (November 1994).
12. B. Mukherjee and K. Schwan, "Experiments with Configurable Locks for Multiprocessors," *International Conference on Parallel Processing* (1993).
13. B. Mukherjee and K. Schwan, "Improving Performance by Use of Adaptive Objects: Experimentation with a Configurable Multiprocessor Thread Package," *Proceedings of High Performance and Distributed Computing* (July 1993).
14. J. Zahorjan, E. Lazowska, and D. Eager, *Spinning Versus Blocking in Parallel Systems with Uncertainty*, University of Washington Technical Report 88-03-01 (March 1988).
15. A. Karlin, K. Li, M. Manasse, and S. Owicki, "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor," *Proceedings of ACM Symposium on Operating Systems Principles* (1991), pp. 41–55.
16. T. E. Anderson, "The Performance of Spin Lock Alternatives for Shared Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems* (January 1990), pp. 6–16.
17. This characteristic is most likely a result of both the implementation of the Java classes and the propensity of Java programmers to over-synchronize access to data because of the coarseness of locking granularity provided by the Java language.
18. M. Campbell et al., "The Parallelization of UNIX System V Release 4.0," *USENIX* (Winter 1991).
19. As of the writing of this paper, the Jvm in the IBM DK, v 1.1.8 was to have a third quarter 1999 release date.
20. VolanoMark 2.1.2 Benchmark, Volano LLC, San Francisco, http://www.volano.com/benchmarks.html.
21. R. Jones and R. Lins, *Garbage Collection*, John Wiley & Sons Ltd., West Sussex, England (1996).
22. It should be noted that not all objects are eligible for movement during the compaction phase. For instance, objects referenced from the stack, class objects, and objects referenced by native code are all prohibited from being moved. These objects are referred to as being *pinned* in the Java heap.
23. Chet Murthy, IBM Corporation, private correspondence.
24. C. J. Cheney, "A Non-Recursive List Compacting Algorithm," *Communications of the ACM* **13,** No. 11, 677–678 (November 1970).
25. 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, UK (September 1995).
26. M. S. Johnstone, *Non-Compacting Memory Allocation and Real Time Garbage Collection*, Ph.D. thesis, University of Texas at Austin, Austin, TX (1997).
27. D. G. Korn and K.-P. Vo, "In Search of a Better Malloc," *Proceedings of USENIX* (Summer 1995), pp. 489–506.
28. "SockPerf: A Peer-to-Peer Socket Benchmark Used for Comparing and Measuring Java Socket Performance," IBM Corporation, http://www.alphaWorks.ibm.com/formula/sockperf.
29. The common case referred to here is when running on a recent version of the operating system (Fixpack 3 or greater for OS/2 Warp and Fixpack 33 or greater for Warp Server). In this case, the Jvm uses a mechanism called "hard suspend" to stop all threads for garbage collection. Under this scheme, preBlock( ) does not need to obtain the thread context. A more detailed description of this scheme is beyond the scope of this paper.
30. 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).
31. O. Agesen, D. Detlefs, and J. E. B. Moss, "Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Quebec, Canada (June 1998), pp. 269–279.

Accepted for publication August 30, 1999.

Robert Dimpsey *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: dimpsey@us.ibm.com).* Dr. Dimpsey received his Ph.D. degree in 1992 from the University of Illinois where he studied performance measurement, modeling, and analysis of large shared memory multiprocessors. In 1992 he joined IBM to work on symmetric multiprocessing performance of the AIX operating system. In 1994 he began work on the IBM cross-operating system microkernel project. This was followed by work on kernel-level multiprocessor performance for WARPSMP and scalable, journaled file systems. Currently, he is working on server-focused, core Jvm performance for IBM Jvms on Intel-based platforms.

Rajiv Arora *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: rarora@us.ibm.com).* Dr. Arora is a performance engineer whose current interest is the performance of the Java language on the server. He is working on core performance of the IBM Jvms on the Intel platform. He also represents IBM on the SPEC server-side Java benchmark working group. Prior to his Java work, he has worked on Web server performance and the performance of TCP/IP on AIX, the IBM microkernel, and OS/2. Dr. Arora joined IBM in 1992. He holds M.S. and Ph.D. degrees in electrical engineering from the University of Rochester in the area of network protocols.

Kean Kuiper *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: kuiper@us.ibm.com).* Mr. Kuiper is a senior engineer working primarily on Java memory management. He is also actively involved in enhancing code generation for Intel IA32 processors. Mr. Kuiper has extensive experience working in multiprocessor environments, including OS/2, the S/390® I/O subsystem, and earlier midrange S/370™ processors.

Reprint Order No. G321-5721.