Thin Locks: Featherweight Synchronization for Java

David F. Bacon Ravi Konuru Chet Murthy Mauricio Serrano
IBM T.J. Watson Research Center

Abstract

Language-supported synchronization is a source of serious performance problems in many Java programs. Even single-threaded applications may spend up to half their time performing useless synchronization due to the thread-safe nature of the Java libraries. We solve this performance problem with a new algorithm that allows lock and unlock operations to be performed with only a few machine instructions in the most common cases. Our locks only require a partial word per object, and were implemented without increasing object size. We present measurements from our implementation in the JDK 1.1.2 for AIX, demonstrating speedups of up to a factor of 5 in micro-benchmarks and up to a factor of 1.7 in real programs.

1 Introduction

Monitors [5] are a language-level construct for providing mutually exclusive access to shared data structures in a multi-threaded environment. However, the overhead required by the necessary locking has generally restricted their use to relatively "heavy-weight" objects.

Recently, their incorporation in Java [3] has led to renewed interest in monitors, because of both their prevalence and their associated performance problems. Java uses monitor semantics derived from Mesa [11]. In Java, the methods of an object may be declared synchronized meaning that the object must be locked for the duration of the method's execution.

Since Java is an explicitly multi-threaded language, designers of general-purpose class libraries must make their classes thread-safe. For instance, the most commonly used public methods of standard utility classes like Vector and Hashtable are synchronized. When these classes are used by single-threaded programs, or used locally within a thread, there is substantial performance degradation *in the absence of any true concurrency*. We have measured slowdowns due to synchronization of up to a factor of two in both compiled and interpreted Java programs.

One way to speed up synchronization is to dedicate a portion of each object as a lock. Unfortunately, Java's design inherently allows *any object* to be synchronizable, even those without synchronized methods. Therefore, adding one or more synchronization words to each object is an unacceptable space-time tradeoff.

The current Sun JDK favors space over time. Monitors are kept outside of the objects to avoid the space cost, and are looked up in a monitor cache. Unfortunately this is not only inefficient, it does not scale because the monitor cache itself must be locked during lookups to prevent race conditions with concurrent modifiers. In addition, if large numbers of synchronized objects are created, the space overhead of the monitor structures may be considerable.

In this paper we describe *thin locks*, an implementation of monitors in IBM's version of the JDK 1.1.2 for AIX. Our implementation has the following desirable characteristics:

Speed In the absence of contention both initial locking and nested locking are very fast – only a few machine instructions. In the presence of contention performance is still better than in the JDK.

Compactness Only 24 bits in each object are used for locking, but object size is not increased due to other space compression techniques.

Scalability Use of global locks and of synchronization instructions that must be broadcast over the global bus are kept to an absolute minimum, allowing efficient execution on large multiprocessors.

Simplicity The scope of changes required in the JVM is small – thin locks are implemented as a veneer over the existing heavy-weight locking facilities.

Maintainability The thin lock code is fully portable, assuming only the existence of a compare-and-swap operation. While hand-coded assembly language routines are required for maximum performance, the amount of platform-specific assembly language code is small and localized into two functions in a single file.

The goal was a locking algorithm with very low overhead for single-threaded programs, but also with excellent performance in the presence of multithreading and contention. These parameters are appropriate to a Java server or to a client that is running windowing or network code that is likely to involve multiple threads of control.

Appears in the Proceedings of the ACM Conference on Programming Language Design and Implementation (Montreal, Canada), SIGPLAN Notices volume 33, number 6, June 1998. Author contact: D. Bacon, (914) 784-7811, dfb@watson.ibm.com.

1.1 Outline

Section 2 describes the locking algorithm in detail. Section 3 presents measurements from our implementation and compares them against two other JDK-based implementations. Section 4 discusses related work, and Section 5 presents conclusions and directions for future work.

2 Locking Algorithm

In order to properly optimize Java's locking performance, one must know what the most common cases are. Implicit in our design is the assumption that the order of frequency of different locking scenarios is as follows, with each scenario about an order of magnitude less common than the one preceding it:

1. locking an unlocked object.
2. locking an object already locked by the current thread a small number of times (shallowly nested locking).
3. locking an object already locked by the current thread many times (deeply nested locking).
4. attempting to lock an object already locked by another thread, for which no other threads are waiting.
5. attempting to lock an object already locked by another thread, for which other threads are already waiting.

We provide detailed measurements supporting our assumptions in Section 3.2. The measurements show that for our benchmarks, a median of 80% of all lock operations are on unlocked objects, and that nesting is very shallow.

2.1 Software Environment

We assume that there is a pre-existing heavy-weight system in place to support the full range of Java synchronization semantics, including queuing of unsatisfied lock requests, and the wait, notify, and notifyAll operations. Such a system will represent a monitor as a multi-word structure which includes space for a thread pointer, a nested lock count, and the necessary queues.

We refer to such multi-word lock objects as *fat locks*.

2.2 Hardware Support

Almost all modern high-performance microprocessors are designed so that they can be used in multi-processor systems. Therefore, they provide user-mode instructions for performing synchronization – either compare-and-swap (introduced in the IBM 370 and provided on Intel 80486 and higher processors) or load-and-reserve and store-conditional instructions (as on the PowerPC, Alpha, and Sparc) which allow various atomic primitives to be synthesized from short instruction sequences.

We will assume only the existence of a compare-and-swap operation (either as a primitive instruction or as a synthesized operation). On older systems without user-level atomic primitives, some other mechanism for achieving atomicity will be required. Our implementation is binary-compatible with both PowerPC machines and with older POWER architecture machines that do not have load-and-reserve. In the latter case, we use a system call to a compare-and-swap operation that is implemented by the kernel.

The compare-and-swap operation takes three inputs: an address, an old value, and a new value. If the contents of the address is equal to the old value, the new value is stored at the address and the operation returns true; if the contents of the address are not equal to the old value, storage remains unchanged and the operation returns false. The compare-and-swap operation is atomic.

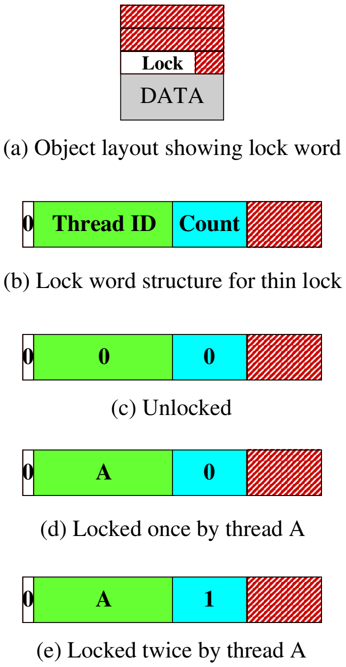


Figure 1: Thin Locks

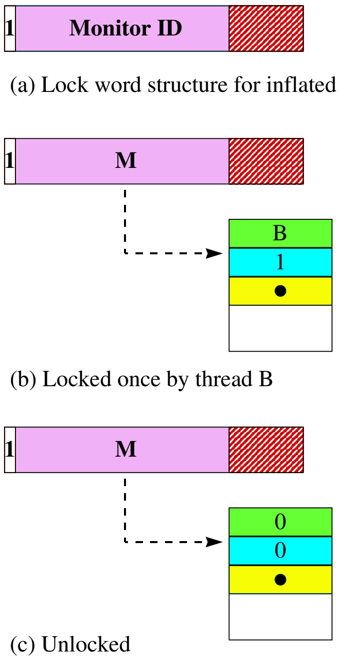
2.3 Monitor Implementation

In the Java run-time system upon which we based our implementation (IBM's AIX port of the 1.1.2 JDK), each object consists of a three-word header followed by data. In order to implement thin locks, we reserve 24 bits in the header of each object as shown in Figure 1(a). We were able to obtain 24 free bits using various encoding techniques for the other values that are typically stored in the header. Allocating an extra word per object was deemed unacceptable both because of the additional space overhead and because there is already a substantial body of native code with dependencies on the object size.

The 8 bits that share the word with the lock field are either constant or subject to change only when an object is moved, and since the garbage collector is not concurrent we can treat those 8 bits as constant values.

The structure of the 24-bit lock field has been very carefully engineered to allow the most common locking and unlocking operations to be performed with the minimum number of machine instructions. The lock field represents either a thin lock or a reference to a fat lock: the first bit (the *monitor shape bit*) is 0 if the lock is thin and 1 if it is fat.

Thin locks are used for objects that are not subject to contention, do not have wait, notify, or notifyAll operations performed upon them, and are not locked to an excessive nesting depth (in our implementation, we define "excessive" as 257). The vast majority of all objects meet this criterion; those that do not have their locks implemented as fat locks.



(a) Lock word structure for inflated lock

(b) Locked once by thread B

(c) Unlocked

Figure 2: Inflated Locks

Once an object's lock is inflated, it remains inflated for the lifetime of the object. This discipline prevents thrashing between the thin and fat states. It also considerably simplifies the implementation.

The structure of a thin lock is shown in Figure 1(b). The monitor shape bit is 0. The remaining 23 bits are divided between a thread identifier (15 bits) and a nested lock count (8 bits). If the thread identifier is 0, the object is unlocked (and the count field must also be 0). If the thread identifier is non-zero, it is an index into a table we maintain which maps thread indices to thread pointers.

When the object is locked, the count field represents the number of locks minus one.

The structure of an inflated lock is shown in Figure 2(a). The monitor shape bit is one, and the remaining 23 bits of the lock field contain the index of a fat lock. We maintain the table which maps inflated monitor indices to fat locks.

Figure 2(b) shows an inflated lock with the fat lock to which the lock index M refers. The fat lock contains a thread identifier for the lock owner (in this case, thread B), a count of the number of locks (*not* the number of locks minus one, as in a thin lock), the necessary queues, and other fields.

The locking algorithm is greatly simplified by adopting a discipline in which the lock field of an object, while it is owned by a particular thread, is never modified by any other thread. This has major performance implications: once a thread has locked an object, all subsequent operations (including the unlock) can be performed with loads and stores, instead of with atomic primitives.

2.3.1 Locking without Contention

We will now explain the operation of the locking algorithm by taking an object through a series of locking operations under varying conditions. Initially, the object is unlocked, so the entire lock field is 0, as shown in Figure 1(c). Thread **A** wishes to lock the object.

Assuming that the object is unlocked (since this is the most common case), thread **A** performs a compare-and-swap operation on the word containing the lock field. The "old" value supplied to the compare-and-swap is that shown in Figure 1(c), which is constructed by loading the lock word and masking out the high 24 bits. The "new" value is a lock field containing a monitor shape bit of 0, a thread index corresponding to thread **A**, and a count of 0, as shown in Figure 1(d).

The "new" value is constructed by taking the bitwise **or** of the "old" value and the thread index, shifted 16 bits to the left. The thread index of the currently running thread is stored in the "execution environment" structure, and can be accessed with a single load instruction. The thread index is stored pre-shifted by 16 bits, so that the locking code does not have to perform an extra ALU operation.

If the compare-and-swap succeeds, then the object was not already locked by another thread (and no concurrently locking thread obtained the lock). By convention, the count field is the number of locks minus one, so the compare-and-swap operation has already properly set the count, and the lock operation is complete.

2.3.2 Unlocking without Contention

At some later time, thread **A** unlocks the object. Since the most common case for unlocking is that the current thread owns the lock and has locked the object once, we construct an "old" value as in Figure 1(d) and a "new" value as in Figure 1(c). However, instead of performing a compare-and-swap, we simply check that the value of the lock word is equal to our "old" value, and if so, store our "new" value into the lock word.

Unlocking does not require a compare-and-swap because of our discipline that once a thread owns a lock, no other thread may modify the lock word. Locking is a stable property: if thread **A** owns the lock, the value will not be stale; if thread **A** does not own the lock, it does not matter if the value read is stale, because any possible stale value will show that thread **A** does not own the lock.

2.3.3 Nested Locking and Unlocking

Now assume that thread **A** once again locks the object, and then attempts to lock it a second time. It will begin by performing the compare-and-swap operation, which will fail because the object is already locked (by thread **A** itself). The locking routine will then check for the next most likely case: nested locking by the owning thread.

In particular, the locking routine will check that the monitor shape bit is 0, that the thread index is **A**'s thread index, and that the count field is less than 255 (8 bits, all ones). The lock word layout has been designed so that this check can be implemented by taking the pre-shifted thread index of **A**, taking its bitwise exclusive-or with the contents of the lock word, and checking that the resulting quantity is less than 255 shifted left by 8 bits (which happens to fit into a 16-bit unsigned immediate field on most RISC architectures).

If the check succeeds, the count field is incremented by adding 256 to the lock word. The updated value, shown in Figure 1(e), is written to memory using a simple store instruction, following the same argument that we applied to using store instructions for unlocking.

When thread **A** unlocks the object, an analogous procedure is followed for decrementing the lock count.

In the event that the nested lock count overflows, we inflate the lock. Lock inflation is described more fully below.

2.3.4 Locking with Contention

Assume now that thread **A** has the object locked once, and thread **B** attempts to lock the object. Thread **B** will first attempt to lock the object with a compare-and-swap, which will fail. It will then check whether it has already locked the object, and this test will fail. Therefore, the object is locked by another thread (thread **A**, in fact).

Thread **B** therefore needs to force a transition on the object from a thin lock (monitor shape bit equal to 0) to an inflated lock (monitor shape bit equal to 1). However, our locking discipline is that the lock field is only modified by the owning thread.

Therefore, thread **B** enters a spin-locking loop on the object. Once thread **A** unlocks the object, thread **B** obtains the lock. Thread **B** creates a fat lock, assigns a monitor index to the newly created monitor object, and changes the lock field to contain a monitor shape bit of 1 and the new monitor index. Figure 2(b) shows the resulting lock structure. The monitor index M is an indirect reference to the fat lock, via the vector that maps monitor indices to monitor pointers.

Finally, when thread **B** unlocks the object, it remains in the inflated state, as shown in Figure 2(c). Subsequent attempts to lock the object will use the fat lock, and if there is contention the fat lock discipline will handle the necessary queuing.

While spin-locking in general is undesirable, we deem it to be acceptable because we are assuming a "locality of contention" principle: if there is contention for an object once, there is likely to be contention for it again in the future. Therefore, we will only pay spin-locking costs once and those costs will be amortized over the lifetime of the object. In general, this works well. The only pathological case occurs when an object is locked by one thread and not released for a long time, during which time other threads are spinning on the object. Standard back-off techniques [1] for reducing the cost of spin-locking can be applied to solve this problem.

3 Measurements

In this section we evaluate our implementation of thin locks in the JDK 1.1.2 for IBM's AIX operating system on the PowerPC. We compare our implementation to a straightforward port of Sun's JDK 1.1.1 to AIX (using the POSIX threads package to support locking) and to IBM's 1.1.2 version of the JDK for AIX, which contains significant monitor optimizations. We refer to the three versions as "ThinLock", "JDK111", and "IBM112", respectively.

The IBM112 implementation assumes that most applications will have a small number of heavily used locks. It therefore pre-allocates a small number (32) of "hot locks". The system begins by using the default fat locks, slightly modified to record locking frequency. When a fat lock is detected to be "hot", a pointer to the hot lock is placed in the header of the object. Because a full 32-bit pointer is used, the displaced header information is moved into the hot lock structure. One bit in the header word indicates whether the word is a hot lock pointer or regular header data.

The hot lock scheme allows the overhead of the monitor cache to be by-passed in the most common cases. However, as we will see, it suffers when large numbers of locks are used, which happens more often than one might expect.

All performance measurements represent the median of 10 sample runs. Time measured is elapsed time on an unloaded IBM RS/6000 43T workstation, containing a 120 MHz PowerPC 604 microprocessor with 128 MB of RAM memory. The 604 has 16KB 4-way associative split caches, 64 entry 2-way associative split TLB's, a 512KB direct-mapped physically addressed level two cache, and a 32 byte reservation grain size for the load-and-reserve instructions.

The 604 is a fairly aggressive superscalar processor for its generation. It is capable of dispatching up to four instructions per cycle, including two ALU operations. It also has a 512 entry branch history table and performs speculative execution of instructions beyond unresolved branches.

These processor characteristics play a significant part in our low-level design and affect the trade-offs we made in the hand-tuned assembly language code, as we will describe in more detail below.

3.1 Macro-Benchmarks

Table 1 summarizes the macro-benchmarks we used for our performance measurements. These macro-benchmarks are real programs, and can therefore be expected to give some indication of the type of speed-ups that could be obtained in practice. To give a sense of the scale of the benchmarks, the size in bytes of both the application and the library bytecode files is given. Library bytecode size is for all classes transitively reachable from the application bytecodes; all code in the `java` and `sun` hierarchy is considered library code.

To give an overall characterization of synchronization behavior, we measured the total number of objects created, the number of objects that were synchronized, and the total number of synchronization operations. The number of synchronized objects is generally less than a tenth of the total number of objects created. The average number of synchronizations per synchronized object shows that re-synchronization is quite common; the median number of synchronizations per synchronized object is 22.7.

The benchmarks suffer from two disadvantages: they are all single-threaded programs, and they are predominantly language processing tools.

While it may initially seem nonsensical to use single-threaded benchmarks to measure speed-ups gained from a locking implementation, such benchmarks actually illustrate the point of our work. Thin locks are designed to be highly efficient when there is no sharing or when despite sharing there is no actual contention. By evaluating thin locks on single-threaded benchmarks, we demonstrate that they are able to remove the performance tax that Java levies on single-threaded applications as the price of using a multi-threaded language.

3.2 Characterization of Locking

In Section 2 we made some assumptions about the relative frequency of various types of locking operations, upon which key aspects of the thin lock design were based. However, these assumptions should also be validated experimentally.

Figure 3 shows the frequency of locking operations by nesting depth. Because the benchmark programs were single-threaded, the contention scenarios were not measured.

The measurements do show that locking unlocked objects is indeed far more common than any other case: at least 45% of locks obtained by any of the benchmark applications were for unlocked objects; the median is 80%.

Program	Description (source)	App Size	Lib Size	Objects	Sync'd Objects	Syncs	Syncs/ S.Obj.
trans	High Performance Java Compiler (IBM)	124751	159747	486215	49313	873911	17.7
javac	Java source to bytecode compiler (Sun)	0	298436	345687	24735	856666	34.6
jgl	Java Generic Library 1.0 (ObjecSpace)	12182	159747	4258177	150175	12975639	86.4
jacorb	Java Object Request Broker 0.5 (Freie U.)	59431	159747	433592	39138	888390	22.7
javaparser	Java grammar parser (Sun)	52961	159747	127593	31	621	20.0
jolt	Java to C translator (K.B. Sriram)	23743	166472	594891	70796	1611558	22.8
jobe	Java Obfuscator 1.0 (E. Jokipii)	10105	159758	323792	12243	90573	7.4
toba	Java to C translator (U. Arizona)	24154	161229	625039	119179	1651763	13.9
javalex	Lexical Analyzer for Java (E. Berk)	25058	159747	43392	10333	1975481	191.2
jax	Java Scanner Generator (K.B. Sriram)	19182	160963	24615	4629	19960283	4312.0
javacup	Java Constructor of Parsers (S. Hudson)	30569	160963	221093	23676	330100	13.94
NetRexx	NetRexx to Java translator 1.0 (IBM)	136535	298436	2258960	139253	1918352	13.8
Espresso	Java source to bytecode compiler (IPD)	72737	161082	188608	408	12305	30.2
HashJava	Java Obfuscator (K.B. Sriram)	16821	160827	247723	7281	212148	29.1
crema	Java Obfuscator (H.P. van Vliet)	26008	161071	84532	10228	275155	26.9
jaNet	Java Neural Network ToolKit (W. Gander)	8825	160827	1083688	234	23369	99.9
javadoc	Java document generator (Sun)	0	305285	879254	107510	2175567	20.2
javap	Java disassembler (Sun)	0	266198	824681	61951	917038	14.8
mocha	Java decompiler (H.P. van Vliet)	—	—	437793	61064	807000	13.2
pizza	Java source to bytecode compiler (M. Odersky)	139800	161096	334824	448	12030	26.9
wingdis	Java decompiler, demo version (WingSoft)	79260	162650	2577899	633145	3647296	5.8

Table 1: Macro-Benchmarks

Nesting of locks in general is very shallow: none of the benchmarks obtained any locks nested more than four deep. These measurements tell us is that in most cases only a few bits need to be allocated for the lock nesting count. Our use of 8 bits for the lock count is highly conservative; 2 or 3 bits is probably sufficient.

Note that locking behavior can vary between different releases of Java. While the overall pattern remained the same, we saw significant individual differences between the 1.0 and 1.1 releases of the JDK.

3.3 Micro-Benchmark Results

While micro-benchmarks do not give an accurate portrayal of the types of performance improvements that can be expected in practice, they are very useful for gaining insight into how different implementations behave in various parts of the design space.

Table 2 summarizes our micro-benchmarks. Each benchmark runs a tight loop for a specified number of iterations; inside the loop an integer variable is incremented. The benchmarks differ in what occurs between the outer loop and the inner variable update. For instance, the NoSync benchmark does nothing at all between the loop and the update. It therefore measures the cost of bytecode interpretation of the loop.

The Sync benchmark is a loop containing a synchronized() block, which in turn contains an integer increment statement. The object that is the argument of the synchronized() block is unlocked, so the Sync benchmark measures the cost of initial locking using the **monitorenter** and **monitorexit** bytecodes. NestedSync is like Sync, except that the object is locked outside of the loop, so that it measures the cost of nested locking (at level 1).

MultiSync is like Sync, but it synchronizes n objects on every iteration. It is designed to simulate the effects of various "working sets" of locks, where n is the size of the working set.

The results are shown in Figure 4. For initial locking (Sync), our thin lock implementation is 3.7 times faster than the Sun JDK 1.1.1 ported to AIX (JDK111), and 1.8 times faster than the IBM 1.1.2 version of the JDK with hot locks (IBM112). This is unsurprising, since the locking overhead for thin locks in the most common case is only 17 instructions, whereas the other implementations are following several levels of indirection into the fat lock structure, and are performing a system call to acquire the lock. In addition, the JDK111 implementation is looking up the fat lock in the monitor cache, which must itself be locked.

For nested locking (NestedSync), the performance advantage of thin locks is significantly reduced compared to the IBM112 hot lock implementation, since in that implementation nested locking of a hot lock essentially involves following a pointer, comparing a thread identifier, and incrementing a memory location. The vanilla JDK111 is still much slower, because it must perform a lookup in the monitor cache and then execute a system call to obtain the thread identifier.

The MultiSync benchmark demonstrates the Achilles heel of the hot lock approach: when the number of hot locks exceeds 32, the IBM112 implementation slows down considerably. Surprisingly, the JDK111 implementation also slows down as the number of locked objects increases. This is due to the fact that the monitor cache thrashes its free list when the working set of monitors exceeds the size of the monitor cache.

The Call, CallSync, and NestedCallSync benchmarks are the analogues of the first three micro-benchmarks, except that they call synchronized methods instead of executing a synchronized() block. Speedups achieved by thin locks are still large, but slightly lower because of the extra overhead involved in performing method invocations.

Finally, the Threads benchmark spawns n threads, each of which runs a tight loop of synchronized() blocks on the same object. Unlike the other micro-benchmarks, which

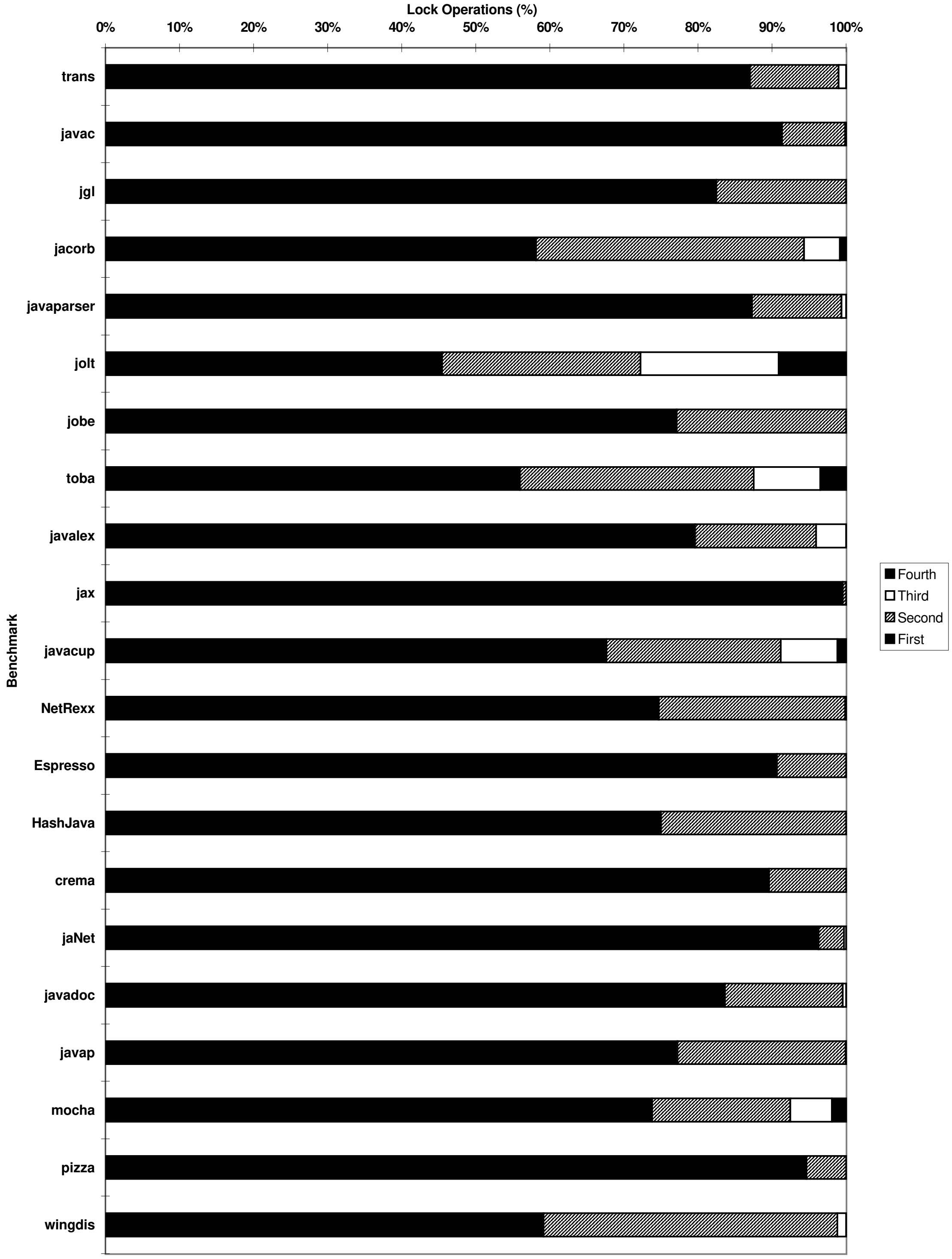


Figure 3: Depth of lock nesting by benchmark. Most lock operations are performed on objects that are not locked (they are the "First" lock on the object). Of the remaining lock operations, the vast majority are "Second" locks.

Program	Description
NoSync	No locking – reference benchmark
Sync	Initial lock with a `synchronized()` statement
NestedSync	Nested lock with a `synchronized()` statement
MultiSync n	Like Sync, but synchronizes n objects every iteration
Call	Calls a non-synchronized method – reference benchmark
CallSync	Calls a synchronized method to obtain an initial lock
NestedCallSync	Calls a synchronized method to obtain a nested lock
Threads n	Initial locking performed concurrently by n competing threads

Table 2: Micro-Benchmarks

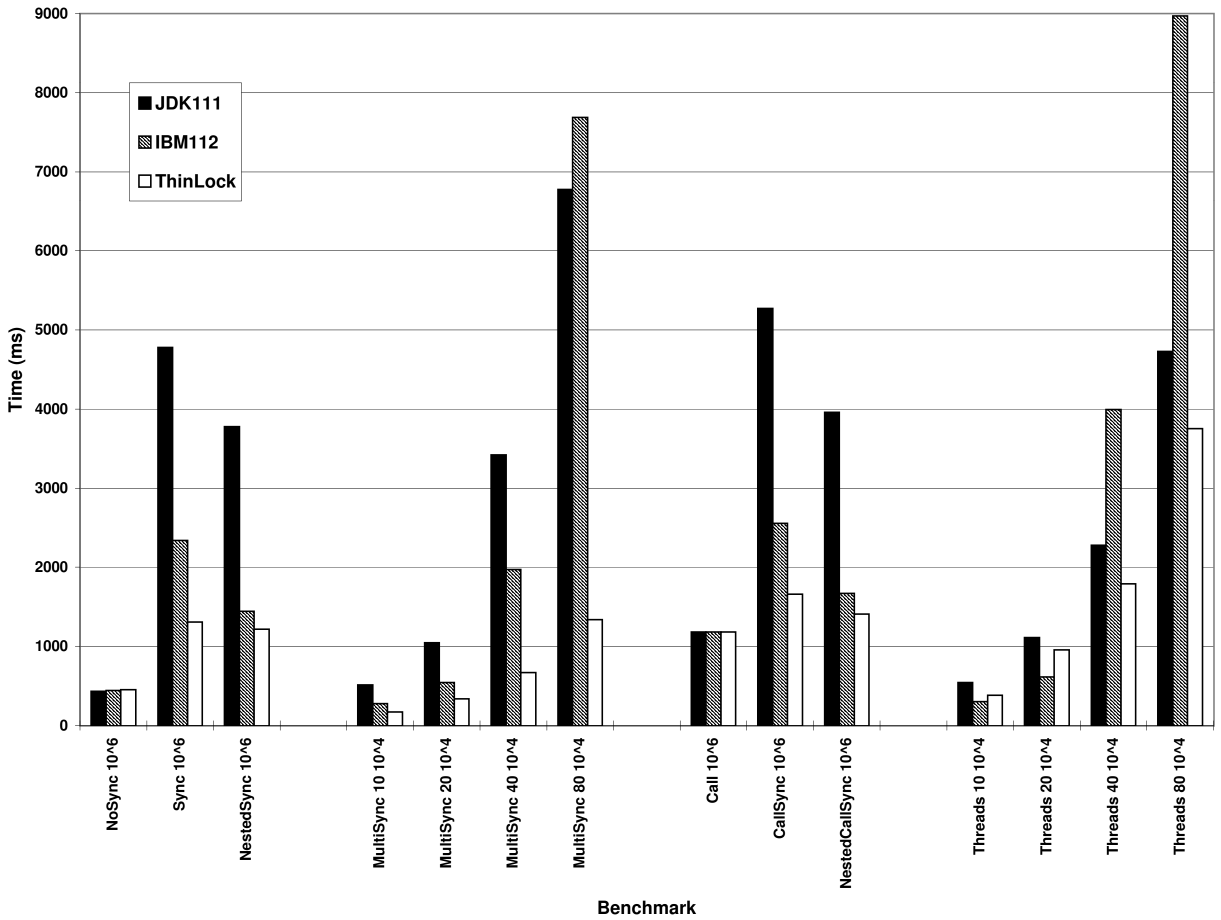


Figure 4: Performance of locking mechanisms on various micro-benchmark tests.

measured performance when the lock was in the uninflated "thin" state, the Threads benchmark will cause the locks in question to be inflated when run with our thin lock implementation.

The Threads benchmark shows the real advantage of the hot lock approach of the IBM112 implementation: when there is significant contention for a small number of objects, hot locks are almost twice as fast as the JDK111. However, as before, the performance of hot locks suffers significantly when the working set size increases.

Thin locks achieve some performance improvement over the monitor cache approach of the JDK111, since instead of locking the monitor cache and performing a table lookup, the fat lock pointer is simply obtained by shifting the monitor index to the right and indexing into the vector that maps indices to pointers.

Note that the thin lock implementation is the only one that scales linearly for both the MultiSync and the Threads benchmarks.

3.4 Macro-Benchmark Results

While we have obtained speedups of more than a factor of five on some micro-benchmarks, real applications do not usually consist of tight loops performing synchronization operations. Figure 5 shows the results of running the macro-benchmarks of Table 1. Thin locks sped up the benchmark programs by a median of 1.22 and a maximum of 1.7 over the JDK111 implementation.

The IBM112 implementation only achieved a median speedup of 1.04, due to the fact that a significant number of applications were actually slowed down. We believe this to be due to the frequent use of a "large" (more than 32) working set of synchronized objects.

In fact, some of the benchmarks *are* in effect tight loops performing synchronized operations. The `javalex` benchmark performs 3.4 million method calls, of which 2.4 million are synchronized. Almost one million calls are to the synchronized **elementAt** method of the Vector class. The `javalex` benchmark was sped up by 6.6 seconds. From Figure 4 we can predict 2.7 seconds of speedup per 1 million synchronized method invocations, or 6.5 seconds of speedup for 2.4 million synchronized method calls.

Another interesting example is `jax`, which was sped up by 66 seconds. `Jax` made almost 19 million calls to the get method of BitSet (two orders of magnitude more than for any other method). The get method is *not* synchronized; however, it executes a synchronized() block after checking for some error conditions. From Figure 4 we predict 3.5 seconds of speedup for every 1 million synchronized block executions, or 66.5 seconds.

3.5 Tradeoffs

At the beginning of this section, we mentioned that low-level hardware characteristics significantly influenced the implementation of the hand-tuned assembly language code that implements locking and unlocking. We will now explore these issues in detail.

Figure 6 shows the performance of a number of variations of our thin lock implementation on selected micro-benchmarks, using the IBM112 as a reference for comparison purposes. The MixedSync benchmark is a cross between Sync and NestedSync – it performs three nested locks of the same object on every iteration.

The "NOP" case represents the "speed of light" – the very best that any implementation could achieve within the framework of the existing system. These measurements were obtained by removing all instructions related to synchronization from the assembly language version of the interpreter loop.

The only overhead for synchronization in the NOP case is the extra bytecodes that are executed (this overhead is significant – it amounts to more than a factor of two!). NOP results could not be collected for the CallSync and Threads cases because the Java VM was unable to initialize itself properly.

The "Inline" case represents our best implementation of thin locks, regardless of portability and maintenance concerns. The assembly language code for locking and unlocking is inlined into each relevant bytecode implementation, and specialized if possible. For the Sync benchmark, the time increases by 20ms relative to the "NOP" case. In fact, three quarters of that time is due to the synthesized compare-and-swap operation (using the load-and-reserve and store-conditional instructions `lwarx` and `stwcx`).

Inlined, specialized assembly code is not well suited to long-term code maintenance. We therefore experimented with using a single lock and unlock routine (the "FnCall" case), and calling the routines from the bytecode implementations. This change resulted in a surprisingly small degradation in performance, presumably due to pre-fetching.

3.5.1 Architectural Variations

The next problem we faced was more significant. AIX runs on PowerPC uniprocessors, PowerPC multiprocessors, and old IBM POWER and POWER2 uniprocessor machines. The POWER and POWER2 architectures do not have user-level synchronization instructions, so the compare-and-swap must be performed by calling a kernel routine.

On the other hand, on a PowerPC multiprocessor locking and unlocking must be followed by the issuance of `isync` and `sync` instructions, respectively, which ensure that if another processor locks the object that it will observe a consistent state. These instructions are not needed on a uniprocessor, but as the measurements of the "MP Sync" case in Figure 6 show, adding those instructions resulted in a 25ms slowdown of the Sync benchmark.

These architectural variations raise a problem: to gain maximum performance, we need to perform different lock and unlock operations depending on the type of hardware. Unfortunately, building those differences into the Java virtual machine would have resulted in an unacceptably complex change to the JDK source code. We also considered using a dynamically linked library, but the cost of calling an AIX dynamically linked library function is significantly higher than calling a local function.

However, because of the availability of surplus superscalar parallelism, we were able to solve this problem by dynamicaly testing the architecture type on *every lock and unlock operation*, as shown in the case labeled "ThinLock" – our final implementation, which we used for the other benchmark results in this paper. Dynamic testing of the CPU type only slowed down the Sync benchmark by an additional 3ms, about the same slowdown as from adding the function call.

Finally, to demonstrate the advantages of our discipline in which only the owner of the lock is allowed to modify the lock field, we modified the code to perform the unlock operation using a compare-and-swap instead of a load followed by a store. As seen in the "UnlkC&S" case, the cost of the additional atomic operation is significant.

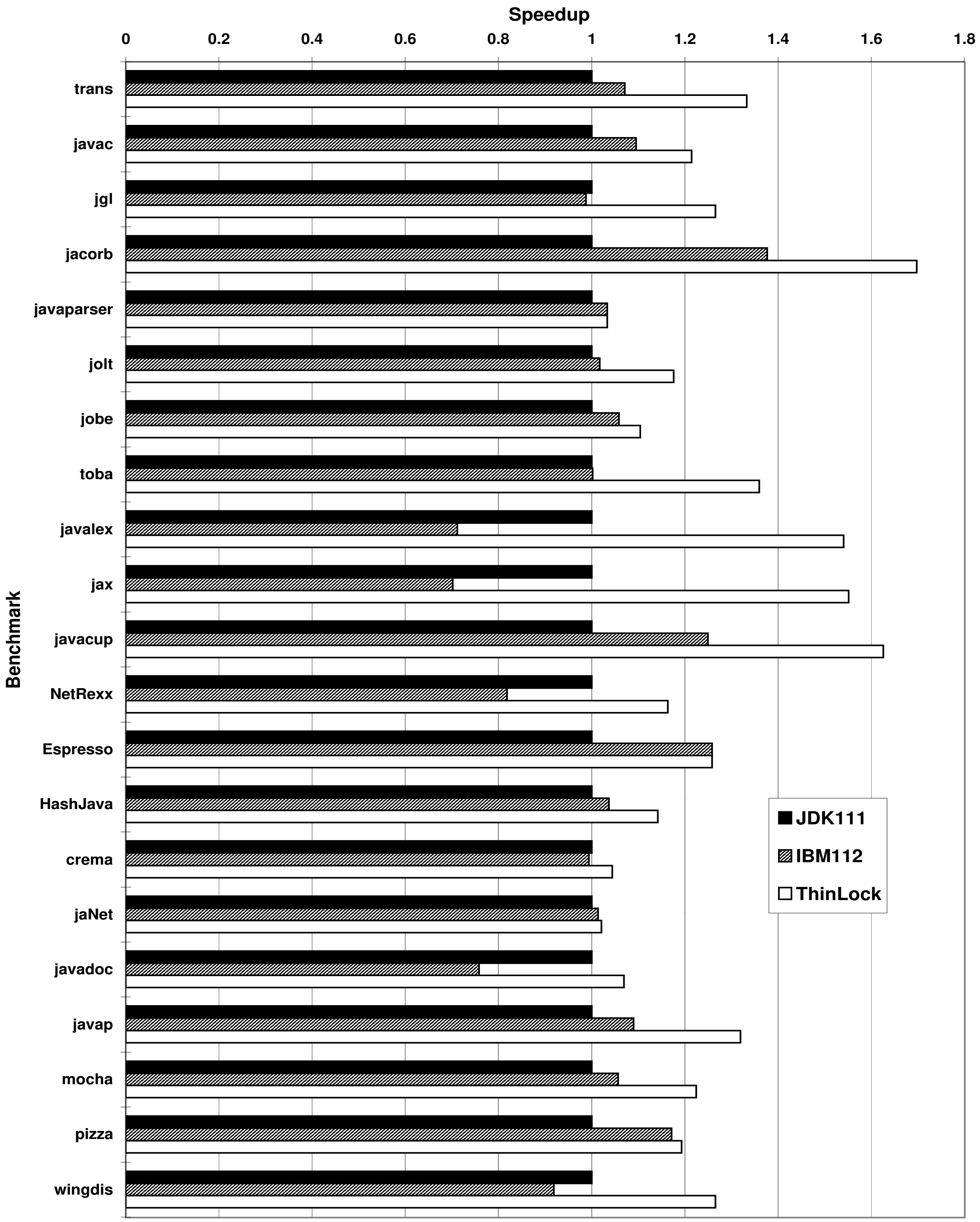


Figure 5: Relative performance of locking mechanisms on various macro-benchmarks.

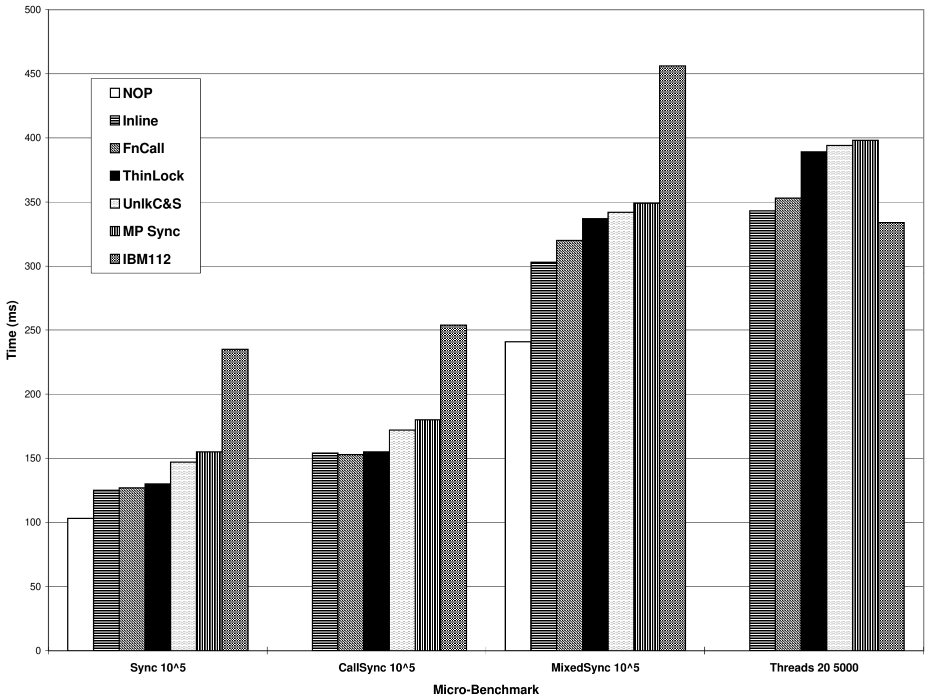


Figure 6: Effect of various performance tradeoffs on selected micro-benchmarks.

4 Related Work

Krall and Probst [7] implemented monitors for the CACAO Java JIT compiler. They argue that because object size is at a premium, monitors should be kept externally in a hash table. However, our 24-bit thin locks have been used in a Java implementation with only two words of overhead per object. The overhead can not be further reduced without converting the class pointer to a class index, which would have unacceptable performance implications in most environments. Therefore, our 24-bit dedicated monitors do not increase object size.

On the other hand, the dedicated monitors greatly reduce the number of instructions required to obtain a lock as well eliminating the needs to synchronize the monitor cache. CACAO is implemented with user-level threads, so mutual exclusion on the monitor cache is obtained by setting a flag to disable pre-emption. However, our thin lock approach could be adapted to user-level threading and would still require substantially fewer instructions and fewer memory accesses than a monitor cache approach.

Krall and Probst also rely on the assumption that consecutive accesses of the same bucket in the monitor cache hash table will usually be for the same monitor. Therefore, when an object is unlocked they leave the monitor installed in the cache with a count of zero. Their fast path for monitor entry assumes that the monitor is already installed in the cache. This approach is similar to that used by the IBM112 "hot locks" implementation; as our measurements showed, some applications have a sufficiently large "working set" of locks that they will thrash such a cache, and will pay a substantial performance penalty.

4.1 General Locking Research

The MCS locks of [12] are similar to thin locks in that they only require a single atomic operation to lock an object in the most common case. However, MCS locks also require an atomic operation to release a lock, whereas we can release a lock with a much less expensive load-store sequence.

MCS locks are designed for maximal efficiency on multiprocessor systems with significant amounts of contention, whereas thin locks are designed for maximal efficiency on uniprocessor systems or on multiprovessor systems with relatively small amounts of contention.

There is a significant body of work on how to achieve mutual exclusion with only atomic read and write operations [2, 9, 10, 13, 14]. These solutions were rendered obsolete by the introduction of instructions that performed compound atomic operations such as exchange, test-and-set, and compare-and-swap [6]. Such operations were later generalized to the Fetch-and-Φ primitive for multiprocessors [8], and were particularly popular on machines with butterfly-type interconnects because concurrent Fetch-and-Φ operations to the same location could be combined in the network [4].

Many microprocessors have not had compound atomic operations until relatively recently because mutual exclusion was generally considered to be the province of operating systems and parallel processors. In 1987, Lamport stated that "if the concurrent processes are being time-shared on a single processor, then mutual exclusion is easily achieved by inhibiting hardware interrupts at crucial times" [10]. He is implicitly assuming that mutual exclusion is only being used in the operating system, or that the overhead of an operating system trap is acceptable on every lock and unlock operation from user-level code.

Both Anderson [1] and Mellor-Crummey and Scott [12] provide thorough discussions of synchronization algorithms for multiprocessors and include comparative performance measurements.

5 Conclusions

We have presented *thin locks*, a method for implementing monitors in the Java programming language using a partial word of storage per object. Thin locks are implemented as a veneer on top of an existing heavy-weight locking subsystem.

We have implemented our technique in the Sun JDK, and have shown that it yields significant speedups. For micro-benchmarks, thin locks are as much as five times faster than the original JDK implementation.

For real programs, thin locks achieve a median speedup of 1.22, and a maximum speedup of 1.7.

Thin locks do not increase the size of an object, and because fat locks are only created under contention, thin locks also result in a significant savings in space when there are large numbers of synchronized objects.

The efficiency of our technique is due to careful engineering to allow the most common cases to be executed with a minimal number of machine instructions, and a design which obviates the need for atomic operations when unlocking or when acquiring nested locks.

Acknowledgements

Thanks to Tamiya Onodera for his advice on the implementation, Rob Strom for his help in validating the algorithm, Mark Wegman for suggesting a key improvement, Kevin Stoodley for his help with optimizations for the Pentium, and Alex Dupuy and Peter Sweeney for their comments on earlier drafts of this paper.

References

[1] Anderson, T. E. The performance of spin lock alternatives for shared memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 1*, 1 (Jan. 1990), 6–16.

[2] Dijkstra, E. W. Solution of a problem in concurrent programming and control. *Commun. ACM 8*, 9 (Sept. 1965), 569.

[3] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification.* Addison-Wesley, Reading, Massachussetts, 1996.

[4] Gottlieb, A., Lubachevsky, B. D., and Rudolph, L. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst. 5*, 2 (Apr. 1983), 164–189.

[5] Hoare, C. A. R. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct. 1974), 549–557.

[6] IBM Corporation. *IBM 370 Principles of Operation.*

[7] Krall, A., and Probst, M. Monitors and exceptions: How to implement Java efficiently. In *ACM Workshop on Java for High-Performance Network Computing* (1998).

[8] Kruskal, C. P., Rudolph, L., and Snir, M. Efficient synchronization on multiprocessors with shared memory. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing* (1986), pp. 218–228.

[9] Lamport, L. The mutual exclusion problem. *J. ACM 33*, 2 (Apr. 1986), 313–348.

[10] Lamport, L. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst. 5*, 1 (Feb. 1987), 1–11.

[11] Lampson, B. W., and Redell, D. D. Experience with processes and monitors in Mesa. *Commun. ACM 23*, 2 (1980), 105–117.

[12] Mellor-Crummey, J. M., and Scott, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst. 9*, 1 (Feb. 1991), 1–20.

[13] Peterson, G. L. A new solution to Lamport's concurrent programming problem using small shared variables. *ACM Trans. Program. Lang. Syst. 5*, 1 (Jan. 1983), 56–65.

[14] Raynal, M. *Algorithms for Mutual Exclusion.* MIT Press Series in Scientific Computation. MIT Press, Cambridge, Massachussetts, 1986. Translated from the French by D. Beeson.