

# Thin Locks

David F. Bacon

Ravi Konuru  
IBM T.J. Watson Research Center  
P.O. Box 704  
Yorktown Heights, NY 10598

Chet Murthy

Mauricio J. Serrano  
Intel Microprocessor Research Labs  
2200 Mission College Blvd.  
Santa Clara, CA 95052

## 1. INTRODUCTION

This work is only five years old. It owes its influence to the explosive growth of the internet and the Java programming language over that period.

In 1997 the adoption of Java for use in production commercial applications was widely viewed as risky, and one of the primary barriers to acceptance was its poor performance. Aside from being interpreted, Java also incorporated a number of features not previously seen in a mainstream programming language: garbage collection, dynamic loading, and concurrency. These features created various sorts of performance challenges.

We could tell that Java was spending about half its time in synchronization, even for single-threaded benchmarks, which would surely form a large portion of Java programs. This led us to begin thinking about how to make a single-threaded Java program fast, even if it used thread-safe libraries that used synchronization extensively.

Previous experience with Hermes [12], a programming language developed at IBM 10 years earlier, led us to the conviction that when scheduling is performed at user-level and optimized for the most common case, concurrency can be incorporated into a language without the order-of-magnitude slowdowns that were commonly accepted.

Thin locks essentially reduce the most common case of locking to a single compare-and-swap operation. While compare-and-swap had long been a feature of mainframe architectures like the IBM 370, at the time we began the work on thin locks compare-and-swap was only available on the newer generations of microprocessors. It was notably absent on the original IBM POWER architecture and on the Intel 80386, both of which were still in use and had to be supported with alternate (kernel-based) locking.

Thin locks were highly successful, providing large speedups to Java applications. In general, the design met the goals that we set out at the beginning of the work: *speed*, *compactness*, *scalability*, *simplicity*, and *maintainability*.

The importance of the last two aspects of the work is perhaps overlooked. Thin locks provide a fast-path layer, but when the fast-path fails, a pre-existing, heavier-weight locking mechanism can be used. This ability to layer the locking protocol greatly simplifies the implementation and improves portability.

## 2. VARIATIONS AND IMPROVEMENTS

Some aspects of thin locks have been improved in various ways, albeit often at the expense of some of the original desiderata:

- the use of busy-waiting when a lock is first inflated, which could be of arbitrary duration;
- the lack of deflation of contended locks, potentially leading to high resource overheads in certain pathological cases;
- space overhead of 24 bits per object for the lock, even though most objects are never locked;
- use of atomic operations for locking, even though objects are often locked many times in a row by the same thread.

The last two problems can be seen as implications of further locality principles of locking that were not exploited by thin locks.

### 2.1 Busy-Waiting

The problem of busy-waiting was first dealt with by Onodera and Kawachiya [10]. They suggested using a special “contention bit”, which had to reside in a separate word in the object header, to signal contention, thereby allowing a variant of thin locks that did not busy-wait.

However, header bits are scarce, especially when they have to occupy multiple words. Gagnon [7] proposed a variant of Onodera and Kawachiya’s algorithm which placed the contention flag in the thread structure of the lock owner instead of in the object.

Neither of these solutions is completely satisfactory, since they both complicate the algorithm and slow down the fast path of the lock release code.

### 2.2 Deflation

The thin locks algorithm never deflates a lock once it is expanded. If there is contention for a large number of objects, then it is possible that finite lock resources will be consumed causing excessive space consumption or even resource failure.

While “inflation overflow” is unlikely to be a problem in practice, it would be satisfying to solve the problem. Agesen et al. [1] present an alternative approach called *meta-locking*, in which a word in the header contains a two-bit meta-lock and a 30-bit data field, whose meaning is determined by the meta-lock bits. Once the meta-lock is obtained, the data field can be updated to point to a lock structure, and the information that normally resides in the data field (the hash code and GC state) is temporarily moved into the lock structure. When an object is unlocked, the reverse operation is performed (assuming no queued threads), so deflation is part of the algorithm.

While meta-locking always deflates locks, it is also true that it always inflates as well, requiring allocation of a lock structure and copying of the data field even on the fast path. Since the lock data is in a separate structure, an additional cache line is wasted as well.

A simple approach to deflation that we have not seen in the published literature is simply to deflate locks during garbage collection

(assuming that garbage collection is stop-the-world). This could be applied to thin locks without requiring any additional synchronization in the locking code.

## 2.3 Space Consumption

Besides the speedup due to thin locks, a secondary benefit was that the default object size was reduced from three words to two, with the lock consuming only 24 bits. Similarly, meta-locks [1] consume only two bits in the common case (but use a full 32-bit word when the lock is inflated).

Fundamentally, the only substantive improvement is to reduce the size of the object header down to a single word. Bacon et al. [3] showed that this was practical by combining a number of heuristics: synchronized blocks are far less common than synchronized methods, and most objects that are locked have at least one synchronized method. In their modification to thin locks, the lock is treated like an implicit data field that is declared by the first synchronized method in the object hierarchy. This then allows a full 32-bit word to be used for the thin lock, and while the offset varies from object to object, it is always a compile-time constant for synchronized method invocations.

## 2.4 Atomic Operations

In many cases, a thread will repeatedly lock the same object without any locks by an intervening thread. The degenerate case is an object that is synchronized but never actually shared, which is quite common in practice. In such situations, the atomic operations that make up the most expensive portion of the operation (and the portion whose cost is rising with architectural trends) are not required on uniprocessors and can be significantly optimized on multiprocessors (they can not be eliminated entirely in the latter case because Java's synchronization semantics have non-local cache effects).

Bacon and Fink [2] proposed a variant of thin locks in which objects were initially owned by their creator, and lock and unlock operations on an object by its owning thread could be performed using simple load and store operations instead of compare-and-swap, because non-owners must use a heavy-weight protocol to get the lock, effectively sending a message to the owning thread asking it to change the state of the lock.

Kawachiya et al. [9] independently developed a similar scheme with a better locality heuristic: the first locker becomes the owner. They also implemented the algorithm and showed that it could provide significant speedups in some cases.

## 3. WHITHER SHARED MEMORY?

Thin locks and related work have shown that it is possible to implement Java's ubiquitous synchronization, in which every object is potentially a lock, in a manner that is both space- and time-efficient. However, experience with Java's concurrency semantics also has some broader implications.

To our knowledge Java is the first language to attempt to define a precise semantics for shared-memory concurrency, and furthermore, the wide availability of shared-memory multiprocessors running Java makes those semantics of more than academic interest.

As it turns out, the original definition of the shared-memory semantics of Java [8] was flawed, and attempts are underway to correct and revise them [11]. However, the resulting definition is quite complex and is beyond the scope of the majority of programmers — in fact the designers themselves have had considerable difficulty in defining the semantics and their interaction with hardware and software reordering optimizations. This begs the question whether a language definition should be promulgated that is so complex that

it is not understood by its users.

Given the difficulties of defining a clean and simple semantics for shared-memory parallelism at the language level, and the potential performance problems of implementing such a definition safely, we believe it is worth questioning whether this is an appropriate programming model for a general-purpose high-level programming language.

We have explored the possibility of defining a Java-like language in which all access to shared objects is mediated by true monitors, which encapsulate sharable data and thereby prevent data races [4]. Related work [5, 6] has taken a different approach, retaining a shared memory model but requiring annotations that allow a program to be provably race-free at compile-time.

## 4. CONCLUSIONS

Java's surprisingly rapid acceptance, fueled by the explosive growth of the internet, introduced a number of technologies into the mainstream of programming languages: garbage collection, dynamic loading, and concurrent programming. This in turn stimulated an enormous amount of research in these areas which was rapidly moved from the laboratory into the marketplace. Our work on thin locks is but one example of this trend.

## REFERENCES

- [1] AGESEN, O., DETLEFS, D., GARTHWAITE, A., KNIPPEL, R., RAMAKRISHNA, Y. S., AND WHITE, D. An efficient meta-lock for implementing ubiquitous synchronization. In *OOPSLA'99 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications* (Oct. 1999). *SIGPLAN Notices*, 34, 10.
- [2] BACON, D. F., AND FINK, S. J. Method to provide concurrency control over objects without atomic operations on non-shared objects. U.S. patent filed August 7, 2000.
- [3] BACON, D. F., FINK, S. J., AND GROVE, D. Space- and time-efficient implementation of the Java object model. In *Proceedings of the Sixteenth European Conference on Object-Oriented Programming* (June 2002), B. Magnusson, Ed., vol. 2374 of *Lecture Notes in Computer Science*, pp. 111–132.
- [4] BACON, D. F., STROM, R. E., AND TARAFDAR, A. Guava: A dialect of Java without data races. In *ACM Conference on Object-Oriented Systems, Languages, and Applications* (Oct. 2000). *SIGPLAN Notices*, 35, 10, 382–400.
- [5] BOYAPATI, C., AND RINARD, M. A parameterized type system for race-free Java programs. In *OOPSLA 2001 Conference Proceedings: Object-Oriented Programming, Systems, Languages, and Applications* (Oct. 2001).
- [6] FLANAGAN, C., AND FREUND, S. N. Type-based race detection for Java. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 2000), pp. 219–232.
- [7] GAGNON, E., AND HENDREN, L. SableVM: A research framework for the efficient execution of Java bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium* (Apr. 2001), pp. 27–40.
- [8] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.
- [9] KAWACHIYA, K., KOSEKI, A., AND ONODERA, T. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA'2002 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications* (Nov. 2002).
- [10] ONODERA, T., AND KAWACHIYA, K. A study of locking objects with bimodal fields. In *OOPSLA'99 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications* (Oct. 1999). *SIGPLAN Notices*, 34, 10, 223–237.
- [11] PUGH, W. Fixing the Java memory model. In *Proceedings of the ACM Java Grande Conference* (San Francisco, 1999).
- [12] STROM, R. E., BACON, D. F., GOLDBERG, A., LOWRY, A., YELLIN, D., AND YEMINI, S. A. *Hermes: A Language for Distributed Computing*. Prentice-Hall, 1991.