Fast Multi-Level Locks for Java

A Preliminary Performance Evaluation

Khilan Gudka   Susan Eisenbach

Imperial College London

{khilan, susan}@imperial.ac.uk

Abstract

Atomic sections guarantee atomic and isolated execution of a block of code. Transactional Memory can be used to implement them but suffers from the inability to support system calls and has high overhead. Lock inference is a pessimistic alternative that infers the locks necessary to prevent thread interference. Our research looks at lock inference techniques for Java programs.

An important aspect of the performance of a lock inference approach is the efficiency of its runtime locks. In this paper, we describe an implementation of the multi-level locks of Gray et al [7] using Java's Synchronizer framework [14] and present some preliminary performance results for a number of workloads that perform a varying proportion of fine-grained and coarse-grained operations. We compare our lock implementation against Java's `ReentrantReadWriteLock` and the STM algorithms TL2 [6] and LSA [17].

For fine-grained workloads, we show that multi-level locks perform similarly to `ReentrantReadWriteLock` but in workloads that mix fine-grained and coarse-grained data operations, they achieve better performance (upto 11x against `ReentrantReadWriteLock` and 3x against STM).

1. Introduction

Atomicity [15] is an important safety property for concurrent programs that provides strong guarantees against errors caused by unanticipated thread interactions. Transactional Memory [13] is a popular approach for providing atomicity but due to its inherent dependence on rolling back execution, it restricts the language features that can be used and additionally, imposes overheads due to buffering and wasted computation.

Lock inference is a pessimistic alternative that statically infers and inserts the locks necessary to prevent interference without causing deadlock. Due to its compile-time nature, lock inference must consider all possible execution paths and thus potentially introduce more locks than necessary. Although this could result in less concurrency, it has the advantages that irreversible operations are allowed and performance overhead is significantly lower. An important aspect of lock inference's performance is the granularity of its locks and how efficient lock acquisitions/releases are.

Our research [3, 4, 8] looks at producing a lock inference implementation for Java. The general approach is to use the Soot framework [20] to analyse Java classes annotated with atomic sections and replace these annotations with suitable locks. This entire process consists of three stages: (i) infer object accesses, (ii) map object accesses to their corresponding locks and (iii) instrument locking code. The first two have been described in previous work [4, 8]. A simple example of our lock inference approach is shown in Figure 1.

```
atomic {                            lock(x);
  x.f = 1;    --converted to-->     x.f = 1;
}                                   unlock(x);
```

Figure 1. Simple lock inference example

```
                          Account a = bank.head;
class Account {           atomic {
  int balance;              while (a != null) {
  Account next;               a = a.next;
}                           }
                          }
       (a)                         (b)
```

Figure 2. We don't know the set of objects accessed in (b)

To achieve high-levels of concurrency, we acquire individual instance locks. However, there are times when the set of objects being accessed is unknown at compile-time. Consider the example in Figure 2, which is a traversal through a linked list of bank `Account` objects.

In general, we do not know how many accounts are in the list and so can only assume that the while loop will iterate any number of times, resulting in the infinite set: {`a`, `a.next`, `a.next.next`, ...}. This creates a problem because we can only acquire a finite number of locks! One option is to infer some lock `L` and then at runtime to interpret this as meaning "lock all instances of type `Account`." However, this will incur tremendous locking overhead if there is a large number of accounts.

We solve this with two types of locks: (i) *instance* locks and (ii) *type* locks. Instance locks protect individual instances whereas type locks protect all instances of a particular type. We use multi-granularity locking [7] to support both varieties of locks simultaneously. That is, an instance lock can only be acquired if the corresponding type lock has not already been acquired and similarly, a type lock can only be acquired if one or more instance locks have not already been acquired. The multi-level lock implementation takes care of the orchestration. For the example in Figure 2, we therefore lock `Account`.

The key advantage of multi-level locks is that they reduce locking overhead when a large number of locks need to be acquired. We produced an implementation of multi-level locks but found it to be 50x slower than `synchronized` (using the counter microbenchmark of Section 4). We therefore looked to achieve a much more efficient implementation using the Synchronizer framework [14], which is the focus of this paper. In particular:

- We implement the multi-level locks of Gray et al [7] in Java using Doug Lea's Synchronizer framework [14] (Sections 2-3).

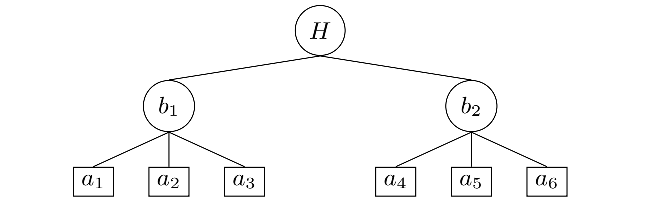


Figure 3. Bank account example structured into multiple branches

- We evaluate performance using two microbenchmarks (Section 4) and show that for fine-grained workloads, it provides performance comparable to Java's `ReentrantReadWriteLock` and for workloads containing a mixture of fine-grained and coarse-grained operations, it outperforms both `ReentrantReadWriteLock` (upto 11x) and STM (upto 3x).

2. Multi-Level Locks

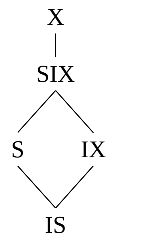
Multi-level locks [7] are useful when data is hierarchically structured and accesses of this data vary in granularity. They provide low locking overhead when accessing a large amount of data and high-concurrency when accessing small amounts of data.

To illustrate this, we structure the famous bank account example so that a bank has a number of branches, which in turn have a number of accounts. In Figure 3, there are two branches each having three accounts. We assume that each node in the graph has a lock associated with it. If we were using normal single-level locks and wished to sum the balances of all accounts in branch b_2, we would first acquire a read lock on H, followed by branch b_2 and finally on all account objects in b_2 (a_4, a_5 and a_6). The summation operation should be atomic and so all accounts must be locked to prevent concurrent modifications, however if the number of accounts is large this will result in a lot of lock acquisitions.

What is actually occurring here is that data are being accessed at the granularity of a branch. Multi-level locks allow the entire branch including all its accounts to be locked by acquiring only b_2's lock. Note, acquiring the multi-level lock on an account has the same behaviour as in the single-level lock case (as there are no child nodes). In general, multi-level locks can be acquired in either *shared* (S) or *exclusive* (X) mode, each of which implicitly locks all child nodes in the same mode.

Given the hierarchical nature of multi-level locks, care has to be taken to ensure that an ancestor node hasn't already been locked in a mode that is incompatible (e.g. trying to acquire the S lock on b_1 when H's X lock has already been acquired by another thread). To prevent this, two additional modes are used: *intention shared* (IS) and *intention exclusive* (IX), which indicate that S or X locking is to be performed respectively further down the graph. For example, before acquiring the S lock on b_2, the IS lock has to be acquired on H. As another example, suppose we wished to perform a deposit on account a_5 and thus required acquiring a_5's X lock. In this case, we would first acquire the IX lock on H, then the IX lock on b_2 and then the X lock on a_5. Figure 4(a) gives the partial ordering of the different lock modes and Figure 4(b) shows which modes can be simultaneously granted to distinct threads.

Note, an additional mode called *Shared Intention Exclusive* (SIX) is also used to achieve more concurrency in the common case where a thread may read many nodes in a sub-tree but only write to a few. Normally, the thread would need to acquire the X lock on the sub-tree but this is overly conservative, as it prevents concurrent threads from performing reads lower down. Please refer to [7] for the full details. In the next section, we describe our implementation.



(a)

	IS	**IX**	**S**	**SIX**	**X**
IS	Y	Y	Y	Y	N
IX	Y	Y	N	N	N
S	Y	N	Y	N	N
SIX	Y	N	N	N	N
X	N	N	N	N	N

(b)

Figure 4. (a) Mode lattice and (b) compatibility matrix (from [7])

3. Implementation

The Synchronizer framework [14] provides common mechanics for atomically managing synchronisation state, blocking and unblocking threads, and queuing. Queues are non-blocking and all state updates are performed using CAS. All these behaviours are encapsulated in the base class `AbstractQueuedSynchronizer`. To implement a custom synchronizer, AQS is extended and the `tryAcquire`, `tryRelease`, `tryAcquireShared` and `tryReleaseShared` methods are overriden. AQS internally supports the two modes *exclusive* and *shared*, however the framework is flexible as to how a synchroniser's modes map to them.

The multi-level locks have five modes they can be acquired in: *exclusive* (X), *shared* (S), *intention shared* (IS), *intention exclusive* (IX) and *shared intention exclusive* (SIX) [7]. Of these, SIX can be implicitly represented by non-zero counts for both S and IX, hence we only explicitly represent the four modes X, S, IS, IX allocating 16 bits for each of their counts (we use the `Long` version of AQS). This allows up to 65535 reentrant acquires in each mode. We map X to exclusive and the remaining three modes (S, IS, IX) to shared. Note, the disambiguation of the latter three modes is made in the `tryAcquireShared` and `tryReleaseShared` methods.

Per-thread counts are stored but thread-local lookups are expensive so they are only queried if absolutely necessary (i.e. if querying the global state is insufficient to determine if the request can be satisfied).

4. Evaluation

We now evaluate the performance of our implementation. Our experimental machine is an SGI Altix 350 containing 32 Itanium 2 CPUs (each running at 1.6GHz) of which we have access to 16. The machine runs SUSE Linux Enterprise Server 10 64-bit and we use Sun Java 1.6 (build 1.6.0_18-b0701) 64-bit.

We run the following two microbenchmarks:

- N threads increment a counter. The purpose of this is to test scalability when there is very high contention. We show that our implementation achieves performance similar to `ReentrantReadWriteLock`.
- N threads perform various operations on a hierarchical bank account model (see Figure 2) containing 10 branches each with 10 accounts. Here we test performance when performing fine-grained or varying mixtures of fine-grained and coarse-grained operations. We show that for workloads that contain a mixture of fine-grained and coarse-grained data accesses, multi-level locks achieve better performance.

We run each experiment for 1 to 16 threads and each thread performs 1,000,000 operations. We report throughput as 1000's of operations per second and make comparisons with Java's `ReentrantReadWriteLock` as well as two STM algorithms: TL2 [6] and LSA [17] (as implemented in Deuce STM v1.3 [11]). For a

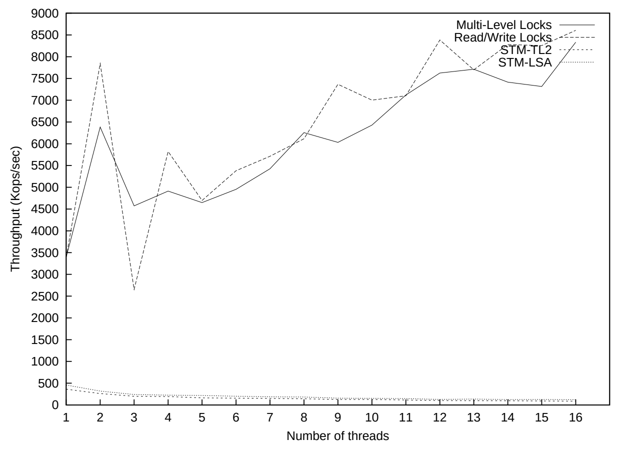


Figure 5. Results for counter microbenchmark

fair comparison, we instruct Deuce not to instrument classes in the `java` and `sun` packages.

4.1 Counter

The counter increment test is commonly used to measure the scalability of a lock when under very high contention. Figure 5 contains the results. Given that we use the same framework as `ReentrantReadWriteLock`, we get similar performance. Furthermore, the STMs suffer because of the large number of conflicts and subsequent rollbacks that result. Note that although the `ReentrantReadWriteLock` generally has slightly better throughput, the trend is very similar and if averaged over more runs may give a much closer line.

4.2 Bank

Our bank microbenchmark has 10 branches each with 10 accounts. We perform the following four operations:

1. Withdraw money from a randomly chosen account.
2. Deposit money into a randomly chosen account.
3. Sum balances of all accounts in a randomly chosen branch.
4. Sum balances of all accounts across all branches.

These operations vary in the granularity of the data they access (individual account, branch, whole bank). To get a feel for how multi-level locks perform when operations of different granularities are carried out concurrently and what types of workloads they perform better for, we carry out three experiments where we vary the balance of these four operations:

Experiment 1 (fine-grained) We only perform withdrawals and deposits, and do so equally often (i.e. 50% each). This experiment aims to gauge performance when only fine-grained operations are being performed. The results are shown in Figure 6. Again, as there are no coarse-grained accesses, we expect performance similar to `ReentrantReadWriteLock`. We are not entirely sure why the STMs perform better but one reason might be because the withdrawal operation only updates the balance if the new balance is non-negative. Consequently, if the balance is not updated, the withdrawal operation is read-only and does not require updating versions.

Experiment 2 (medium-grained) We introduce the coarser-grained summing operations into the mix but the majority are still fine-grained operations. The balance is 40% withdrawals, 40% deposits,

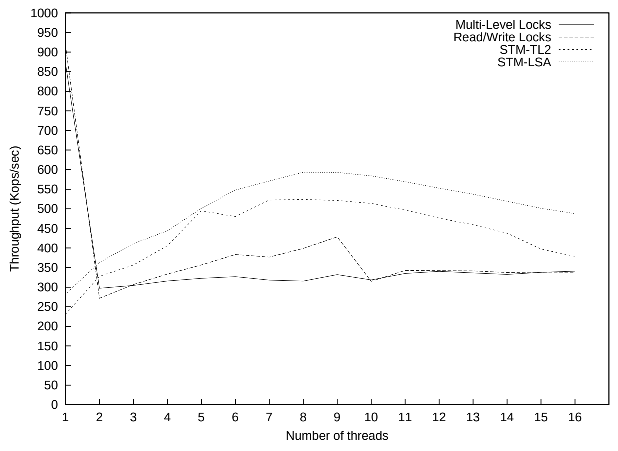


Figure 6. Results for bank experiment 1

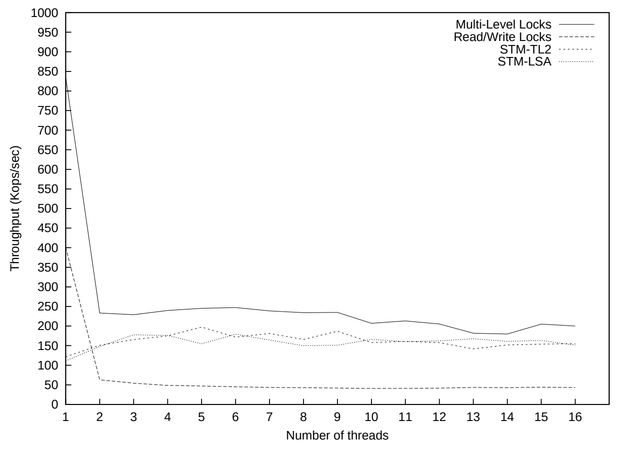


Figure 7. Results for bank experiment 2

10% summing over a random branch and 10% summing over all branches. Figure 7 contains the results.

In the case of `ReentrantReadWriteLock`, the summations require locking a large number of accounts resulting in the 5x less throughput. STM throughput varies between 0.6x and 0.9x that of multi-level. This might be because the coarse-grained operations are relatively large transactions thus increasing the possibility of conflicts. However, the STM results are not as bad as for `ReentrantReadWriteLock`, probably because TL2 and LSA acquires locks late and so potentially more operations can proceed in parallel.

Experiment 3 (coarse-grained) We test performance for when the coarse-grained operations outnumber the fine-grained operations. The balance is 20% withdrawals, 20% deposits, 30% summing over a random branch and 30% summing over all branches. The results are shown in Figure 7. In the case of `ReentrantReadWriteLock`, account objects are now locked far more frequently leading to significant overhead and multi-level locks performing upto 11x better. Furthermore, the majority of transactions now touch a large number of accounts thus inducing many conflicts. Consequently, multi-level locks perform upto 3x better than STM. Unfortunately, we were only able to take results from a single run, hence averaging over more runs may yield a smoother graph.

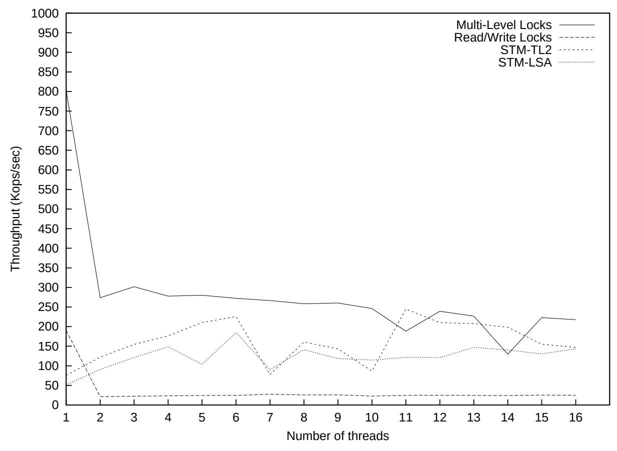


Figure 8. Results for bank experiment 3

5. Related Work

While software transactional memory [13] remains the popular approach for implementing atomic sections, recent work [2, 4, 8–10, 16, 21] has also looked at statically inferring locks sufficient for atomic and deadlock-free execution. There is also a large body of work on efficient lock implementations [1, 5, 12, 18, 19]. However, the locks described in this paper are different because they assume data is structured in a hierarchy and subsequently support locking at different granularities.

6. Conclusion

Our research looks at lock inference techniques for Java programs. Our particular approach uses type locks for when the set of object accesses is unknown and per-instance locks otherwise. To support both locking granularities simultaneously, we use the multi-granularity locking protocol as described by Gray et al [7].

An important factor in the performance of a lock inference approach is the performance of its locks. To achieve an efficient multi-level lock implementation, we use Doug Lea's Synchronizer framework [14]. The main contributions of this paper are this implementation as well as a preliminary evaluation of its performance. We have shown that for fine-grained operations, it performs similarly to `ReentrantReadWriteLock`. However, when coarse-grained operations are concurrently performed, multi-level locks reduce locking overhead and perform better.

We are aware that our type locks are very coarse and as future work would like to explore more fine-grained alternatives, such as using ownership types. Nevertheless, a hierarchy will still exist as one or more objects will need to represent a set of objects, and so an efficient multi-level lock implementation will still be necessary.

Acknowledgments

We are very grateful to Microsoft for funding this work. We would also like to express thanks to Sophia Drossopoulou for very helpful discussions and for proof reading drafts at short notice. A further thanks to Tim Harris for discussions at the early stages of this paper. Finally, we are indebted to members of the `concurrency-interest` list for their insights and helpful advice, namely Doug Lea, David Holmes, Bryan Thompson and Tim Peierls.

References

[1] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for java. *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 258–268, 1998.

[2] S. Cherem, T. M. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In R. Gupta and S. P. Amarasinghe, editors, *PLDI*, pages 304–315. ACM, 2008.

[3] D. Cunningham, S. Drossopoulou, and S. Eisenbach. Lock Inference Proven Correct. In *FTfJP*, June 2008. URL `http://pubs.doc.ic.ac.uk/lock-inference-proven/`.

[4] D. Cunningham, K. Gudka, and S. Eisenbach. Keep off the grass: Locking the right path for atomicity. In L. J. Hendren, editor, *CC*, volume 4959 of *Lecture Notes in Computer Science*, pages 276–290. Springer, 2008.

[5] D. Dice and N. Shavit. TLRW: return of the read-write lock. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*. Citeseer, 2009.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. *Proc. International Symposium on Distributed Computing*, 2006.

[7] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In *VLDB '75: Proceedings of the 1st International Conference on Very Large Data Bases*, pages 428–451, New York, NY, USA, 1975. ACM.

[8] K. Gudka, S. Eisenbach, and T. Harris. "Hello World!": Lock Inference in the Presence of Large Libraries. Technical report, March 2010. URL `http://pubs.doc.ic.ac.uk/lock-inference-large-libraries/`.

[9] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge. Component-based lock allocation. In *PACT*, pages 353–364. IEEE Computer Society, 2007.

[10] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[11] G. Korland, N. Shavit, and P. Felber. Noninvasive Java concurrency with Deuce STM (poster). In *SYSTOR '09: The Israeli Experimental Systems Conference*, may 2009. Further details at `http://www.deucestm.org/`.

[12] E. Koskinen and M. Herlihy. Dreadlocks: efficient deadlock detection. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 297–303, New York, NY, USA, 2008. ACM.

[13] J. Larus and R. Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

[14] D. Lea. The java.util.concurrent synchronizer framework. *Sci. Comput. Program.*, 58(3):293–309, 2005.

[15] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGPLAN Not.*, 12(3):128–137, 1977.

[16] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. *ACM SIGPLAN Notices*, 41(1): 346–358, 2006.

[17] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC06*, pages 284–298. Citeseer, 2006.

[18] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *ACM SIGPLAN Notices*, 41(10):272, 2006.

[19] M. Spear, A. Shriraman, L. Dalessandro, and M. Scott. Transactional Mutex Locks. In *SIGPLAN Workshop on Transactional Computing*, 2009.

[20] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In S. A. MacKay and J. H. Johnson, editors, *CASCON*, page 13. IBM, 1999.

[21] Y. Zhang, V. C. Sreedhar, W. Zhu, V. Sarkar, and G. R. Gao. Minimum lock assignment: A method for exploiting concurrency among critical sections. In J. N. Amaral, editor, *LCPC*, volume 5335 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2008.