Lock Prediction

Brandon Lucia Joseph Devietti Tom Bergan Luis Ceze Dan Grossman

University of Washington, Computer Science and Engineering

http://sampa.cs.washington.edu

Abstract

This paper makes the case for studying and exploiting the predictability of lock operations in multithreaded programs. We discuss several uses of *lock prediction* and identify the two key events we believe are useful to predict: (1) which thread will be the next one to acquire a given lock and (2) which lock a given thread will acquire next. We describe multiple prediction algorithms and show that while their accuracy varies widely, they are likely accurate enough for many uses.

1. Introduction

Prediction is a fundamental technique for improving the performance of computer systems. It is at the heart of branch prediction, caching, value prediction, and various forms of software speculation. There is a natural evolution of prediction schemes. They are hatched based on empirical properties of software (*e.g.* dynamically there are more backward branches than forward branches), become more sophisticated over time, and eventually become self-fulfilling prophecies as higher levels of the system stack are engineered so that the prediction schemes will perform well.

With the move to multicore, Amdahl's Law ensures that performance will become more about scalability and less about single-threaded performance. This trend is an opportunity to rethink what is worth predicting and why. Certainly, there have been prediction/speculation schemes related to parallelism, such as TLS [10], SLE [7] and CMO [12], but we suggest a complementary focus on fundamental questions related to shared-memory synchronization. In this paper we make the case that predicting *synchronization events* in multithreaded programs is worth studying, can have good accuracy, and has many potential uses.

Two empirical questions are worth asking and, to our knowledge, they have not been asked directly:

- When thread T releases lock L, is it predictable what lock L' the thread T will next acquire? This is the *thread's-next-lock* question.
- Conversely, when thread T releases lock L, is it predictable what thread T' will next acquire L? This is the *lock's-next-thread* question.

We have performed a preliminary study of these questions and found that while predictability varies significantly, it is often reasonably accurate and sometimes very accurate. Whether a given prediction accuracy is sufficient or not depends on how it will be used. For some uses the benefit of a correct prediction is far greater than the cost of a misprediction; other uses have the opposite behavior. We have identified several compelling uses of lock prediction (Section 2). We have explored several predictors of varying sophistication (Sections 3, 4, and 5). We conclude (Section 6) by challenging the community to explore lock prediction by developing new predictors as well as new uses for them.

2. Uses of Lock Prediction

The most natural use-case for lock prediction is to accelerate the performance of synchronization primitives. For example, if a lock's next owner is the same as its current owner, no expensive memory ordering or atomic instructions need to be used to acquire it. Several pieces of related work (which we discuss in Section 6) explore simple forms of lock predictability to improve lock implementations. Given these existing success stories for lock prediction, we are optimistic about applying lock prediction techniques to other domains.

Temporal Scheduling Lock prediction could enable more efficient OS-level scheduling, as the predicted next acquirer of a lock is more likely to be on the critical path of an execution than a thread not accessing shared resources. An OS scheduler can boost the priorities of threads on the estimated critical path, accelerating overall application performance.

Prefetching and Spatial Scheduling Knowing the next acquirer of a lock L can enable smarter prefetching: L's releaser can preemptively push the cache line containing L (and potentially data guarded by L as well) into the cache of the processor likely to next acquire L. This saves cache misses for the next acquirer. Alternatively, the thread predicted to next acquire L could be migrated to the processor where L and its associated data are locally cached.

Anomaly detection With a sufficiently accurate predictor mispredictions can be viewed as anomalies, indicative of unexpected behavior, security breaches, or bugs.

Determinism Deterministic execution for data-race free programs can be implemented using just a library of deterministic synchronization primitives [5]. Lock prediction can help avoid situations where a program must stall to ensure a deterministic outcome. Given a deterministic prediction mechanism, a superior schedule with fewer stalls can be chosen based on the predicted behavior of the program.

3. Lock Prediction Design Space

The basic prediction problem is the following: given a history of lock operations up to the current time, what will be the next operation? In this paper we lay out the design space of solutions to this problem along four dimensions:

What to predict We explore *lock's-next-thread* prediction in Section 4 and *thread's-next-lock* prediction in Section 5.

How to predict A variety of statistical methods can be applied to the lock prediction problem. We explore a few such methods in Section 4.1.

Aliasing Locks need not be named uniquely; they may *alias*. For example, when two locks are aliased, the prediction information for both locks is effectively merged. Similarly, different threads need not be named uniquely. We explore the consequences of and motivation for aliasing in Section 4.2.

Context A history of lock operations can be enhanced by adding *context* to each operation, for example the call stack at the operation. We explore this possibility in Section 5.1.

In the following sections we explore this design space qualitatively and quantitatively. We start with lock's-next-thread and discuss aliasing, and then describe thread's-next-lock and explore the effect of adding context.

We have evaluated the prediction accuracy of several techniques in this design space using the PARSEC benchmark suite of multithreaded C/C++ programs [1], configured to run with 8 threads. We omit applications that perform fewer than 200 dynamic lock acquires when run with the `native` input set. Using a trace generator written as a wrapper around pthreads, we collected traces of calls to lock acquisition functions, recording the calling thread and the dynamic address of the lock being acquired. We fed these traces to offline implementations of our prediction techniques. We used a trace-based approach for an accurate comparison across prediction techniques. For all applications we used the `native` input size, except `fluidanimate`, for which we used the `simlarge` input due to the prohibitive size of the trace produced using `native`. All results presented are the average of values from 30 different executions. We computed 95% confidence intervals for values shown, and none was larger than ±0.3%.

4. Predicting Lock's-next-thread

This section presents a series of algorithms from simple to more sophisticated. We describe these algorithms in terms of lock's-next-thread prediction, which is the problem of predicting the next thread to acquire a given lock. Figure 1(a) demonstrates these algorithms with a simple example. Table 1 summarizes the accuracy along with the number of dynamic lock acquires (Column 2) and distinct locks that are acquired at least once (Column 3).

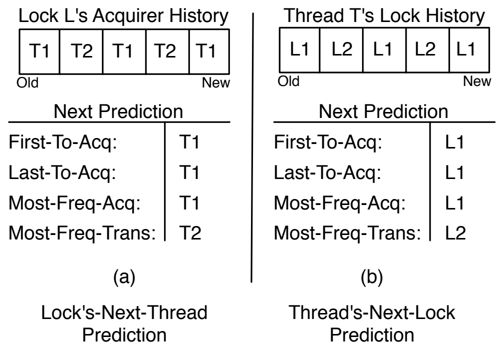

Figure 1. lock's-next-thread and thread's-next-lock prediction

4.1 Algorithms

4.1.1 First-to-acquire

The simplest scheme we consider is to track the first thread to acquire a lock and always predict that thread. This exploits the observation that many locks are local to one thread for the whole execution. It has been shown that this simple predictor works well for many Java programs because many locks are predominantly acquired by only one thread [3].

4.1.2 Last-to-acquire

Another simple scheme is to predict that the last thread to acquire a lock will acquire it next. Like first-to-acquire, this technique exploits the fact that locks tend to be acquired by the same thread, but accounts for the fact that some data is local to different threads at different points in an execution.

4.1.3 Most-frequent-acquirer

A third scheme is to monitor how many times each thread has acquired a given lock, then predict that the next thread to acquire a lock is the one that has most frequently acquired it in the past. This predictor works well when a lock shows affinity for a particular thread, *i.e.* when it is mostly thread-local. Unlike the above predictors, this technique takes the entire history of a lock into account, making it more robust to temporary deviations from a global pattern.

Accuracy Last-to-acquire (Column 5) outperforms first-to-acquire (Column 4) in most cases, by as much as 66.5% (`ferret`). It predicts significantly less accurately than first-to-acquire in the cases of `vips`, `x264`, and `bodytrack`. Most-frequent-acquirer (Column 6) and last-to-acquire are accurate to within 15% of each other in all but two cases: `dedup` and `vips`.

All three predictors perform very poorly (less than 15% accuracy) on `bodytrack` and `x264`. For `bodytrack`, this is explained by the fact that every lock in the program is acquired by every thread at approximately the same rate for the entire course of execution; thus the locks do not exhibit any temporal locality or any affinity for particular threads.

App	App Characteristics		% Correct Predictions			
	# Dyn. Acquires	# Distinct Locks	First-to-acq	Last-to-acq	Most-freq-acq	Most-freq-trans
bodytrack	6.8M	3	12.7	0.9	12.8	15.3
dedup	994K	98K	5.6	61.8	38.9	63.3
facesim	1.8M	12	32.5	44.1	50.4	59.1
ferret	3.3K	33	32.6	99.1	87.8	99.9
fluidanimate	9.3M	477K	76.6	94.7	90.9	95.2
vips	644K	67	49.7	4.0	49.5	63.9
x264	207K	27	12.7	5.2	14.9	25.3

Table 1. Accuracy of lock's-next-thread prediction.

4.1.4 Most-frequent-transition

During an execution there will likely be repeating patterns of lock handoff from one thread to another. This suggests we can use the frequency of handoff events to make predictions. For each lock, we maintain a weighted, directed graph in which nodes represent threads and edges represent handoffs. The edge connecting the nodes for threads T_1 and T_2 represents the situation where T_2 acquired the lock when T_1 was the last thread to have acquired the lock, *i.e.*, the edge represents the lock's handoff from T_1 to T_2. The weight of the edge is the frequency of the handoff.

For a lock last held by thread T, we predict that the next thread to acquire the lock is the thread targeted by the edge of greatest weight originating at T's node.

Accuracy Most-frequent-transition's prediction accuracy is shown in Column 7 of Table 1. In our experiments, it outperformed the other predictors across the board. It works well because it takes advantage of both the frequency of events, like most-frequent-acquire, and temporal locality, like last-to-acquire. This advantage is made clear by `dedup` and `vips`: `dedup` performs poorly with most-frequent-acquire but well with last-to-acquire, while the opposite is true for `vips`. However, both perform well with most-frequent-transition, showing that most-frequent-transition has superior accuracy compared to the other prediction techniques we have seen so far.

4.1.5 Most-frequent-sequence

We generalize most-frequent-transition by using *arbitrary length sequences* of handoffs, instead of single handoffs. In this scheme, each lock keeps an ordered history of the last n threads to acquire it. This is the "current sequence" for the lock. In addition to this, for each lock, we maintain a weighted, directed, bipartite graph. The two disjoint sets of nodes in the graph are *sequence nodes* and *next-in-sequence nodes*. There is a sequence node for every length-n sequence of threads to acquire the lock during an execution. There is a next-in-sequence node for every thread. An edge connects the sequence node for a sequence, S, to the next-in-sequence node for a thread, T, if when the lock's current sequence was S, the next thread to acquire the lock was T. The weight of the edge is the frequency of this event.

Given a lock with current sequence S, we predict that the next acquirer is the thread targeted by the edge with the greatest weight originating at the sequence-node for S.

Using this prediction technique we saw very little or no improvement in accuracy over most-frequent-transition using sequences ranging from 1 to 10 acquisitions in length. We omit the complete results for the sake of brevity.

Quantifying "Cold-Start" Effects Our predictors need to amass information on which to base predictions before they can predict. The initial collection period is the "cold-start" period. During the cold-start period, no prediction is made. In general, the fraction of predictions not made is low, often less than 0.01%, and never more than 3.26% in our experiments. There is a useful distinction between cold-starts and mispredictions: in some situations there may be a high misprediction cost but if instead, no prediction is made, it may be possible to avoid incurring the cost.

Quantifying Storage Overheads The per lock space overhead of lock prediction depends on the prediction strategy. For first-to-acquire and last-to-acquire, the overhead is a single thread id. This small amount of information could potentially be embedded in the lock word itself. In most-frequent-acquirer, the per-lock overhead is a counter per thread. For most-frequent-transition, the per-lock overhead increases to one counter for each possible handoff – a number of counters that is the square of the number of threads. While the latter techniques maintain more information, because the number of threads is typically small, the storage overhead is not likely to be problematic.

4.1.6 Adding Bias

All of the above techniques are specific instances of a general prediction technique. Each technique defines a predictor as a function of the per-lock frequency distribution of acquisition events (or handoff events, in most-frequent-transition and most-frequent-sequence). The techniques vary in the importance, or *weight*, placed on each entry in the distribution. In first-to-acquire, for instance, all entries have a weight of 0 except the entry for the first acquirer of the lock, which has a weight of 1. Similarly, in last-to-acquire, all entries have a weight of 0 except the thread that last acquired the lock. In the other techniques, all entries are equally weighted.

More generally, we can adjust the individual weight of each entry in the per-lock distributions. For instance, we could bias the most-frequent-acquirer predictor toward the recent past by using heavy weights for recent events and smaller weights for less recent events. Periodically reducing accumulated frequencies efficiently achieves this effect.

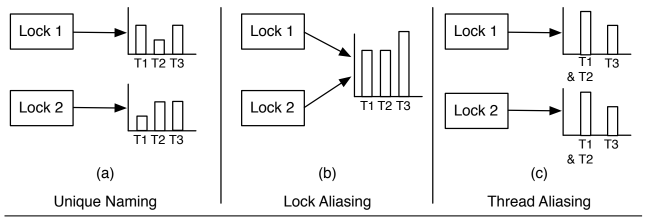

Figure 2. Aliasing for Lock's-Next-Thread prediction

4.2 Aliasing

We have thus far assumed that each lock is given a unique name at runtime. One simple scheme is to name locks by their address in memory. However, lock names need not be unique. In fact, it can be beneficial to group many locks into "buckets", for a few reasons. First, when groups of locks behave similarly, it is more space efficient to maintain a single prediction data structure for an entire bucket, rather than a separate structure for each individual element. Second, bucketing makes prediction less susceptible to cold-start problems because buckets aggregate the histories of many elements. Finally, for some uses it is beneficial to use data from prior runs as the basis for prediction. However, unique names are not usually stable across multiple runs of a program; across runs, there may be different numbers of threads and locks allocated in different orders at different addresses. Thus, when using data from prior runs for prediction we must use a stable bucketing scheme.

When locks are placed in the same bucket, we say they *alias*. The prediction data structures for all locks in the same bucket are merged. For example, under most-frequently-acquired prediction, all locks in the same bucket will share a set of acquire frequencies.

There are many ways to bucket locks. Three interesting ones are by *class*, by *initialization site*, and by *one-bucket*. Bucketing locks by static type or *class* may be useful in languages like Java. Bucketing locks by *initialization site* can partition locks based on behavior, since locks initialized by the same piece of code tend to be used similarly. Finally, bucketing all locks together is equivalent to predicting which thread will be next to acquire *any* lock.

Like locks, threads may also alias. Instead of naming threads by unique dynamic threads IDs, we can merge threads into groups, for example by merging all threads spawned from the the same *creation site* into the same group. Thread aliasing has benefits in space efficiency, cold-starts, and cross-run stability similar to lock aliasing.

Figure 2 illustrates the effects of aliasing, using most-frequent-acquirer prediction as an example. Figure 2(a) shows prediction with no aliasing. Figure 2(b) shows lock aliasing: two aliased locks, L1 and L2, share the same prediction information, in effect, merging L1's and L2's individual frequency distributions. Figure 2(c) shows thread aliasing: the frequencies of two aliased threads, T1 and T2, are merged in the frequency distributions of both locks.

App.	# Init Sites	% Correct Predictions		
		Unique	Init-Site	One-Bucket
bodytrack	1	15.3	15.3	15.3
dedup	14	63.3	66.0	65.4
facesim	2	59.1	43.5	56.0
ferret	8	99.9	99.9	99.9
fluidanimate	1	95.2	31.9	31.9
vips	3	63.9	71.4	71.4
x264	1	25.3	75.8	75.8

Table 2. Accuracy for Most-Frequent-Transition Lock's-Next-Thread Prediction with lock aliasing

Accuracy We evaluate the impact of lock aliasing on the accuracy of most-frequent-transition prediction. We bucketed locks based on initialization site (*init-site*) and by putting all locks into one bucket (*one-bucket*). Table 2 shows the accuracy of both, as well as with no aliasing (*unique*).

For `facesim` and `fluidanimate` accuracy decreases when locks are bucketed. This follows the intuition that valuable per-lock information can be lost when locks alias.

In `vips` and `x264`, putting all locks in one bucket is beneficial — In the case of `x264`, accuracy increases by about 50%. This suggests that for these applications, it is easier to decide which thread will be acquiring *any* lock next, but more difficult to determine which thread will next acquire a given lock.

Three benchmarks (`fluidanimate`, `vips`, `x264`) saw exactly the same accuracy for *init-site* bucketing as for *one-bucket*. In `x264` and `fluidanimate`, this is because there is only one initialization point, and hence, only one bucket. Interestingly, in `vips`, there are three buckets when using *init-site* bucketing and accuracy is the same as with *one-bucket*. This suggests that there are some locks that, when grouped, are more predictable. Using either *init-site* bucketing, or *one-bucket*, these locks alias and accuracy improves.

Locks share prediction information when they alias, so cold starts are less frequent than with unique naming. In all cases, cold starts occurred less than 1.2% of the time.

Overall, the disparity in prediction accuracy in the presence of aliasing suggests that its impact is largely dependent on the application.

5. Predicting Thread's-next-lock

Thread's-next-lock prediction is symmetric to lock's-next-thread prediction. In fact, the prediction techniques described in the previous section can be applied to thread's-next-lock by simply swapping the roles of threads and locks. Figure 1 demonstrates how this mapping works.

Accuracy Table 3 shows the accuracy of our last-to-acquire (Column 2), most-frequently-acquired (Column 3), and most-frequent-transition (Column 4) thread's-next-lock prediction schemes. For all applications, most-frequent-transition has the highest accuracy. For most applications, last-to-acquire is outperformed by most-frequently-acquired.

Most-frequent-transition achieves around or greater than 50% accuracy for all applications except `fluidanimate` and `x264`. This is because in these applications there is a

App.	% Correct Predictions		
	Last-to-acq	Most-freq-acq	Most-freq-trans
bodytrack	98.9	77.0	99.3
dedup	39.6	42.0	46.0
facesim	43.7	39.5	59.1
ferret	0.8	50.4	76.6
fluidanimate	9.2	0.0	17.5
vips	0.5	24.7	76.2
x264	0.2	6.6	31.3

Table 3. Accuracy for Thread's-Next-Lock Prediction

App	% Correct Predictions	
	No Context	Last Acquire Context
bodytrack	99.3	99.3
dedup	46.0	46.0
facesim	59.1	80.7
ferret	76.6	77.4
fluidanimate	17.5	27.5
vips	76.2	76.3
x264	31.3	35.5

Table 4. Accuracy for Most-Frequent-Transition Thread's-Next-Lock Prediction with context

great deal of data dependent control-flow making the next acquire hard to predict.

5.1 Adding Context to Lock Operations

The most-frequent-transition and most-frequent-sequence predictors can be more precise when lock operations are labeled by the context in which they occur. One example of context is a call stack; lock operations can be tagged by some or all of the addresses on the call stack at the operation.

Conceptually, this is the dual to bucketing (Section 4.2). Adding context effectively reduces locks' aliasing *with themselves*, by distinguishing them in the history based on acquire context. Conversely, bucketing increases aliasing.

Accuracy We implemented a lightweight version of this in which just the top-most return address on the call stack is included. Table 4 reports the accuracy for the most-frequent-transition scheme performing thread's-next-lock prediction. Column 2 shows accuracy with no context (reproduced from Table 3 for comparison), and Column 3 shows the accuracy when the top-most return address is added as context. In many cases, context has little or no effect on accuracy. In our experiments, adding context never decreased accuracy.

Two benchmarks see a substantial improvement in accuracy – as much as 21% – with the addition of context: `facesim` and `fluidanimate`. With context, each thread maintains a frequency distribution per lock for each context in which it was acquired, instead of just a single distribution per lock. This adds a notion of *control-dependence*. It is often the case that different phases of a program have distinct locking behavior which are potentially easily predictable in isolation. When lock operations are not labeled by context, these differences are lost, and accuracy suffers.

With context, the number of cold-starts increases because information must be accumulated for each context, instead of just for each lock. Still, cold starts occurred for no more than 4.1% of acquires. The largest increase was in `fluidanimate`, which saw 3.3% more cold start scenarios with context than without.

6. Related Work and Conclusions

Limited forms of lock prediction have proved useful in the context of accelerating the performance of synchronization primitives. Work by Kawachiya et al. first proposed "lock reservations", the ability to bias a lock towards a particular thread, said to "own the bias" on that lock [3]. A lock can be acquired by the bias owner without the use of any atomic or memory ordering instructions. As soon as a non-bias-owner thread tries to acquire a biased lock, however, the lock "inflates" back to normal (e.g., a CAS-based implementation). The prediction used by lock reservations is equivalent to our first-to-acquire scheme, with the additional restriction that the first misprediction disables the optimization.

Later work generalized lock reservations to handle intermittent lock acquisition by non-bias-owning threads [6], making lock reservation's prediction fully equivalent to our first-to-acquire scheme. Subsequently, it was also shown how to support switching the bias among multiple threads [2, 4, 9]. Bias typically gets transferred based on the lock's most recent acquirer, just like our last-to-acquire scheme. These biased locking techniques have been implemented in several production Java Virtual Machines, and provide noticeable speedups (5-10%) for single- and multithreaded Java applications running on commodity hardware.

There have also been several hardware techniques to reduce the overhead of synchronization, via speculation [7] and identifying dynamically unnecessary memory ordering instructions [12]. These optimizations leverage the same locality properties of lock acquisition that help our predictors to perform well. Work on cache prefetching for multiprocessor workloads [11] exploits similar locality properties, though not for synchronization objects per se.

Rajwar, et al [8] proposed an optimization leveraging temporal locality of lock and protected data accesses. The key is "speculative push" of data to processors waiting to acquire the lock protecting the data at lock release time. Lock prediction may enable more aggressively pushing data.

We find it encouraging that the straightforward prediction mechanisms already proposed can accelerate shared-memory synchronization primitives. We have shown that a generalization of these techniques (i.e., our most-frequent-transition predictor) can provide a substantial boost in accuracy. Given the improvement, coupled with the fact that many of our proposed uses of lock prediction – such as scheduling and anomaly detection – are less latency-sensitive than low-level mutex implementations, we are optimistic that lock prediction can be successfully applied to new domains. We have presented a first step toward this goal; we encourage the community to join us in investigating more accurate prediction strategies and the new uses of lock prediction that greater accuracy will enable.

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[2] D. Dice, M. Moir, and W. Scherer III. Quickly Reacquirable Locks. http://home.comcast.net/∼pjbishop/Dave/QRL-OpLocks-BiasedLocking.pdf.

[3] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA*, 2002.

[4] T. Ogasawara, H. Komatsu, and T. Nakatani. To-lock: Removing lock overhead using the owners' temporal locality. In *PACT*, 2004.

[5] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.

[6] T. Onodera, K. Kawachiya, and A. Koseki. Lock Reservation for Java Reconsidered. In *ECOOP*, 2004.

[7] R. Rajwar and J. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO*, 2001.

[8] R. Rajwar, A. Kägi, and J. R. Goodman. Inferential queueing and speculative push for reducing critical communication latencies. In *ICS*, 2003.

[9] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *OOPSLA*, 2006.

[10] G. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA*, 1995.

[11] S. Somogyi, T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *ISCA*, 2006.

[12] C. von Praun, H. Cain, J.-D. Choi, and K. D. Ryu. Conditional memory ordering. In *ISCA*, 2006.