Adaptive Integration of Hardware and Software Lock Elision Techniques

Dave Dice
Oracle Labs
dave.dice@oracle.com

Alex Kogan
Oracle Labs
alex.kogan@oracle.com

Yossi Lev
Oracle Labs
yossi.lev@oracle.com

Timothy Merrifield
University of Illinois at Chicago
tmerri4@uic.edu

Mark Moir
Oracle Labs
mark.moir@oracle.com

ABSTRACT

Transactional Lock Elision (TLE) and optimistic software execution can both improve scalability of lock-based programs. The former uses hardware transactional memory (HTM) without requiring code changes; the latter involves modest code changes but does not require special hardware support. Numerous factors affect the choice of technique, including: critical section code, calling context, workload characteristics, and hardware support for synchronization.

The `ALE` library integrates these techniques, and collects detailed, fine-grained performance data, enabling policies that decide between them at runtime for each critical section execution. We describe an adaptive policy and present experiments on three platforms, two of which support HTM, showing that—without tuning for specific platforms or workload—the adaptive policy is competitive with and often significantly better than hand-tuned static policies.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; E.1 [**Data Structures**]

Keywords

Lock elision; transactional memory; sequence locks

1. INTRODUCTION

Effectiveness of techniques for improving concurrent programs' scalability depends on factors such as the workload, hardware platform, and synchronization support. Some techniques that are well-suited to some use cases are ineffective—sometimes even harmful—for others; some techniques depend on hardware support not available on all systems, and others use software techniques that are difficult or impossible to apply in some cases. Selecting (combinations of) techniques and tuning them for a particular context is often impractical because these choices depend on numerous factors including the workload, which may change over time, even for a given application on a given platform.

Adaptive Lock Elision (`ALE`) provides pragmatic support for improving scalability of legacy lock-based applications. *Lock elision* techniques can improve scalability without using finer-grained locking or non-blocking synchronization, both of which significantly complicate applications and may harm performance for some platforms and workloads.

Transactional Lock Elision (TLE) [3, 11] allows critical sections protected by the same lock to execute concurrently, exploiting hardware transactional memory (HTM) to detect conflicts between them and retry if necessary. Optimistic software techniques behave similarly, but use software techniques (e.g., *seqlocks* [1]) to detect conflicts. The required changes to critical sections can be achieved via compiler support in some cases [13], but are manual in this paper.

Our `ALE` library can execute a given critical section in one of three *modes*: `HTM` (i.e., TLE), `SWOpt` (i.e., software optimistic execution), and `Lock` (i.e., acquiring the lock). Integrating these techniques enables the choice of mode for a given critical section to be made heuristically at runtime.

The library collects statistics and profiling information that provide guidance for programmers about which modes should be enabled for which critical sections, and furthermore are used by pluggable *policies* that determine at runtime how to execute each critical section. This approach is preferable to programmers determining execution modes in advance, as the choices can be made based on factors such as the platform, workload, and calling context.

Section 3 describes in detail a HashMap implemented using the proposed approach, which enables all three execution modes, and uses the `ALE` library to decide between them at runtime. Section 4 describes the implementation of the library and two simple policies: a static one based on fixed parameters, and an adaptive one that chooses execution modes and retry parameters based on observed runtime behavior.

Section 5 describes experiments on three platforms, two of which support HTM. (Experiments on additional platforms are not shown due to lack of space.) We use a simple HashMap microbenchmark, as well as a more complex Kyoto Cabinet [8] benchmark that demonstrates use of `ALE` with a readers-writer lock and nesting. Experiments with simple static policies demonstrate the importance of choosing execution modes based on observed runtime behavior. Our adaptive policy is almost always competitive with the best static policy, without tuning for the platform and workload.

SPAA'14, June 23–25, 2014, Prague, Czech Republic.

http://dx.doi.org/10.1145/2612669.2612696.

2. BACKGROUND

In the *Transactional Lock Elision* (TLE) technique [3, 11], critical sections are executed atomically using HTM, while confirming that the lock is not held. If no data conflicts between two critical sections occur, then they can execute in parallel, even if using the same lock. If hardware transactions fail often (e.g., due to conflicts or HTM limitations), performance can degrade. Reasons for transactions failing and the best retry strategy both depend on numerous factors, including the HTM implementation, the workload, etc.

With *optimistic execution*, critical sections can be executed concurrently provided they do not conflict; if a critical section experiences a conflict, no harm is done, and it can be retried. (The lock can be acquired if this occurs repeatedly.) In this paper, conflict detection is achieved using a variant on sequence locks (seqlocks). A *seqlock* [1, 9] is a lock that has an associated sequence number that is initially zero and is incremented on each acquire and release. Data protected by a seqlock can be read without acquiring the lock, provided the sequence has the same even value—implying that the lock is never held—during reading.

Optimistic execution and TLE can both be viewed as forms of lock elision. For both techniques, operations may be attempted concurrently, and may fail and need to be retried depending on a number of factors, including aspects of the workload that may change over time. Despite these similarities, different situations favor different techniques. Optimistic execution can be highly scalable in read-heavy workloads but less effective with more frequent mutating operations. TLE can greatly improve scalability even in the face of frequent mutating operations, but depends on HTM and its effectiveness for the given workload.

Together, TLE and optimistic execution can benefit a wide variety of workloads. However, they do not interoperate effectively if combined naively: mutating operations can succeed concurrently using TLE, but incrementing the sequence number as required for optimistic software execution causes concurrent operations using TLE to conflict with each other, defeating TLE's benefit. We demonstrate that these conflicts can be reduced, and even eliminated in some cases, significantly improving performance and scalability.

We combine TLE and optimistic execution so that either technique can be used effectively, and the choice of which one to use can be made dynamically, depending on the availability and behavior of each option for the current platform and workload. Thus, we show that programmers can get the "best of both worlds" of TLE and optimistic execution.

The idea of dynamically selecting between multiple integrated synchronization techniques is not new. For example, Lim's reactive locking algorithm [10] can switch between fast and scalable alternatives for locking depending on the workload. It can adapt differently for different locks, but it does not support concurrent use of different techniques for different critical sections for the same lock, and does not exploit techniques such as TLE or optimistic execution.

Another example is Adaptive Transactional Memory [12], which is similar in spirit, using profiling and machine learning techniques to adapt between a number of software TM (STM) implementations. Our approach applies to an established lock-based programming model that does not require new compiler and language support, and is thus more pragmatic and more immediately practical.

Other work (e.g., [5, 7, 14]) explores adapting TLE retry parameters, but we are not aware of previous attempts to integrate it with software optimistic execution. Furthermore, to our knowledge, `ALE` provides more detailed and finer-grained reporting and adaptation than prior efforts.

3. USING THE ALE LIBRARY

Next, we describe the use of the `ALE` library with a simple HashMap example. The HashMap supports three operations. An `Insert` operation inserts a new key-value pair if the key is not already present and overwrites the value associated with the key otherwise; a `Remove` operation removes the specified key if it is present and has no effect otherwise; and a `Get` operation copies the value associated with the specified key to a specified memory area and returns true if the key is present, or returns false otherwise.

In the base implementation, the HashMap is protected by a single lock, `tblLock`, and every operation is executed in a single critical section that holds this lock. In Section 3.1, we describe how to integrate this implementation with `ALE`, and how to enable the use of `HTM` mode with it. In Section 3.2, we explain how to add a `SWOpt` execution alternative.

3.1 Basic use of ALE

Integrating a lock with `ALE` is achieved by macros via two simple changes. The first, in the same scope as the lock's declaration, associates a label with the lock and causes metadata to be declared for the lock. The second, in the same scope as the lock initialization code, causes the library to initialize this metadata. The type and name of the metadata are transparent to the programmer: All communication with the library for a given lock uses the lock's label.

Next, the `BEGIN_CS` and `END_CS` macros are used to replace locking and unlocking code for each critical section to be enabled for `ALE` (see example in Section 3.2).

The library can now collect statistics and profiling information for these critical sections. Even without using the `HTM` or `SWOpt` modes, `ALE`'s reports provide valuable insights to guide optimization efforts. This is particularly useful in larger and more complex examples. Enabling `HTM` mode for `ALE`-integrated critical sections is as simple as using appropriate compilation flags for the application and library.

3.2 Adding a simple optimistic alternative

Enabling `HTM` mode is simple because changes are required only at critical section boundaries, not to the critical section code itself. Adding a `SWOpt` alternative is more complicated. In particular, a critical section executed in `SWOpt` mode can be executed concurrently with other critical sections protected by the same lock, and these may cause it to observe inconsistent data (if they execute in `HTM` or `Lock` mode).

The programmer must ensure that the `SWOpt` path can detect such interference in order to retry in this case, and that unsuccessful attempts have no harmful side effects. The HashMap example illustrates one simple way to achieve this.

A critical section is prepared for `SWOpt` execution by using a variant of the `BEGIN_CS` macro and by adding `SWOpt` support code to the critical section that is executed only if the `GET_EXEC_MODE` macro (provided by the library) indicates that the critical section is running in `SWOpt` mode.

Programmers using `ALE` to integrate a `SWOpt` path into a critical section must follow some simple rules:

- Interference with the `SWOpt` path should be caused only

by concurrent execution of a critical section protected by the same lock, while executed not in SWOpt mode.

- Such executions must provide the means for the SWOpt path to detect any (potential) interference they cause.
- Critical sections executed in SWOpt mode must avoid any harmful side effects due to such interference. When interference is detected, the SWOpt execution can be explicitly retried (if desired), after notifying the library of the failed attempt.

Many possibilities exist for constructing SWOpt paths. We have enhanced the simple seqlock-based approach described earlier to overcome several disadvantages. First, as discussed in Section 2, the simple seqlock-based approach loses the benefit of HTM mode. Even if HTM is not available, the entire critical section is executed between the two increments of the sequence number when the lock is acquired, preventing successful execution of any SWOpt path associated with the same lock for this entire interval. This is unnecessary if conflicting actions rarely occur, or occur during only a small fraction of the critical section's execution. It is thus better to explicitly identify code regions that may perform conflicting actions than to conservatively assume that any part of the critical section may do so.

To illustrate these ideas with our simple HashMap implementation, we first add to the HashMap class a version number field tblVer, initialized to zero. We also add BeginConflictingAction and EndConflictingAction methods, both of which simply increment tblVer, and a GetVer method, which returns the value of tblVer, first waiting until it is even if a boolean argument indicates this is required. (Concurrency could be improved by using multiple version numbers, say one for each HashMap bucket. We have not yet experimented with this option.)

The first two methods are used to bracket part(s) of critical section code that may interfere with concurrent SWOpt execution of a critical section protected by the same lock. The GetVer method is used both for reading the version number at the beginning of a SWOpt execution of a critical section (first waiting until tblVer is even), and for checking whether the version number has since changed.

The obvious first SWOpt candidate is Get, as concurrent executions of it do not interfere with each other. Conflicting actions can be performed only by concurrent Insert or Remove operations. A Remove operation that explicitly identifies its conflicting region might look like this:

```
1  BEGIN_CS(&LockAPI, &tblLock, md_tblLock);
2  < search a node containing a given key>
3  if ( < node is found > ) {
4      BeginConflictingAction();
5      unlink( < node > );
6      EndConflictingAction();
7  }
8  END_CS(&LockAPI, &tblLock, md_tblLock);
```

(Here, md_tblLock is the label associated with tblLock, and LockAPI is a structure that identifies methods used to acquire and release this lock, as well as an is_locked method that is used to check and monitor a lock when an associated critical section is executed in HTM mode. This approach enables the ALE library to be used with any type of lock). Note that Remove conflicts with concurrent SWOpt executions only briefly and only if it actually removes a node.

```
1  template <bool SWOptMode>
2  int32_t HashMap::GetImp(const& key_t key,
3            uint32_t hashedKey, value_t& retVal) {
4    if (SWOptMode) v = GetVer(true);
5    idx = hashedKey % this->numBuckets;
6    Bucket* barr = this->buckets;
7    if (SWOptMode && (v != GetVer(false))) return -1;
8    BucketNode* bP = barr[idx].firstNodeP;
9    if (SWOptMode && (v != GetVer(false))) return -1;
10   while (bP && !keyEqual(bP->key, key)) {
11     bP = bP->next;
12     if (SWOptMode && (v != GetVer(false))) return -1;
13   }
14   if (bp != NULL) {
15     retVal = bp->val;
16     if (SWOptMode && (v != GetVer(false))) return -1;
17     return 1;
18   }
19   return 0;
20 }
```

Figure 1: Auxiliary method used by the Get method.

To integrate a SWOpt path into the Get method, we begin by creating a utility method GetImp, shown in Figure 1, which factors out the main functionality of the Get method.

We use a boolean template argument SWOptMode to create two versions of the GetImp method—one for SWOpt mode and one for other modes—that differ only as shown in orange.

The main difference between the two versions is validation when executing in SWOpt mode; in the absence of compiler support for this purpose, the programmer adds this validation manually. The general rule of thumb is to validate before using any value that was read since the last validation. However, some validations can be omitted if the use of the value cannot cause any error even if it is invalid. (In some systems, optimistically reading could cause an error if the data has already been deallocated, and the page in which it resides has been returned to the operating system and removed from the address space. This issue does not apply in many cases, including the commercial applications motivating our work, because the application does not deallocate memory during its lifetime, or the operating system does not deallocate freed pages. In some other contexts, the issue may be addressed by using non-faulting loads.)

The Get method itself is implemented as a simple wrapper method that begins a critical section (using a variant of the BEGIN_CS macro that specifies that a SWOpt path exists), and uses GET_EXEC_MODE to determine whether it is in SWOpt mode and should call GetImp<true>, or otherwise call GetImp<fasle>. Because GetImp<true> may return -1, indicating that it did not complete successfully due to a conflict, the Get method is executed inside a loop that only terminates when an execution completes successfully.

3.3 Advanced usage and optimizations

Identifying conflicting regions allows SWOpt executions of critical sections using the same lock to detect interference. The COULD_SWOPT_BE_RUNNING library macro returns a (possibly conservative) indication of whether such executions exist. This allows executions in HTM mode to elide the conflict indication when no SWOpt path is running, thus avoiding unnecessary aborts due to modifications of tblver.

Next, as mentioned above, some operations cause conflicts in some cases, but not in others. This suggests that it might sometimes be profitable to allow a critical section to be executed in SWOpt mode only in some cases, as determined at runtime. We describe two ways this can be achieved.

One approach is to use the "self abort" idiom: a critical section executes in SWOpt mode, and if a conflicting region is encountered, retries the critical section, indicating that the SWOpt path should *not* be used. Self abort is simple and convenient, but provides less benefit as conflicting operations become more common. We address this case next.

While *searching* for the specified key, Insert and Remove do not interfere with SWOpt paths. Therefore, we can provide a SWOpt path for the first parts of these methods too. However, when they perform an action that interferes with other SWOpt executions, this cannot be done in SWOpt mode, so a nested critical section that has no SWOpt path must be used to perform such actions. (The ALE library's support for nesting is described in Section 4.1.) Some care is required due to the possibility of the SWOpt execution being invalidated before or during the lock acquisition by the nested critical section. Therefore, the nested critical section must first check if a conflict has occurred, and if so, the critical section should be ended without performing the conflicting action, and the whole operation should be retried (after reporting the SWOpt failure to the library).

While additional code can be executed in the outer SWOpt critical section *after* the short critical section is completed, in general code that might be invalidated by concurrent operations should be avoided: the effects of the short critical section may already be seen by other threads, so retrying in SWOpt mode after this point is generally undesirable.

3.4 Statistics and profiling information

The ALE library collects statistics and profiling information for policies to use when choosing execution modes. Reports based on this information are useful in their own right. Even without using HTM or SWOpt modes, these reports provide insights into application behavior on a given platform or workload. These insights provide guidance about which critical sections might benefit from a SWOpt path, for example. The reports have also been invaluable in understanding and improving behavior of adaptive policies.

Different executions of a given source-level critical section may use different locks, and may be executed in different "contexts" (see below). These factors can affect the best choice of execution mode. The library therefore collects statistics and profiling information at the granularity of <lock, context> pairs.

Each critical section integrated with the ALE library (using a library macro such as BEGIN_CS) defines a *scope*. A thread's *context* is an initially-empty sequence of scopes. When a thread begins execution of a critical section, its scope is added to the thread's context; when the critical section is completed, the scope is removed.

As described so far, a thread's context is determined by the ALE-enabled critical sections within which it is executing. However, programmers can explicitly create additional scopes, allowing the library to collect statistics and profiling information at even finer granularity.

The importance of this ability is shown by the C++ "scoped locking" idiom. A scoped lock is a class that encapsulates a lock, and whose constructor and destructor are responsible for acquiring and releasing the lock, respectively. Enabling critical sections introduced in this way for ALE involves using the BEGIN_CS and END_CS macros (or variants thereof) in the constructor and destructor. Thus, there is only a single critical section at the source level, implying a single set of statistics for all acquisitions of each lock used this way, so policies cannot specialize behavior for (effectively) different critical sections. This issue can be addressed by declaring additional scopes explicitly, as follows:

```
1 void foo() {
2     ...
3     BEGIN_SCOPE("foo.CS1");
4     {
5         ScopedLock(&myLock);
6         // CS body
7     }
8     END_SCOPE();
9 }
```

Here, the BEGIN_SCOPE macro introduces a new scope, with label "foo.CS1". The result is that the critical section that begins in the constructor of ScopedLock will execute in different contexts depending on where the constructor is called from, allowing the library to distinguish the different cases.

For a given <lock, context> pair, the library may record: how often a critical section was executed using this lock in this context; how many times each mode (HTM, SWOpt, or Lock) was attempted/successful; how much time was spent in each mode, etc.

Another way to use different contexts for the same critical section allows for cases in which it is expected that different execution modes may be best for different cases, as in:

```
1  if (<condition>)
2    BEGIN_CS_NAMED(&LockAPI, &tblLock,
3              md_tblLock, "condition is true");
4  else
5    BEGIN_CS_NAMED(&LockAPI, &tblLock,
6              md_tblLock, "condition is false");
7  {
8    <CS body>
9  }
10 END_CS(&LockAPI, &tblLock, md_tblLock)
```

This results in associating a different scope, and thus a separate context and statistics, with each call to BEGIN_CS_NAMED, allowing the library to adapt differently depending on whether the condition holds. The label provided to BEGIN_CS_NAMED describes the scope, improving readability of reports.

4. LIBRARY IMPLEMENTATION

The ALE library separates common, policy-independent functionality from a pluggable policy. The policy-independent code provides two interfaces: one for programs that use the library (e.g., via the ALE macros described in Section 3), and one for policy code, which the library uses to determine the mode for each critical section execution attempt.

Each ALE-enabled lock has associated metadata, which is allocated and initialized once. In addition, the library associates *granule* metadata with each <lock,context> pair with which a critical section is executed, which is used to record information and statistics about these executions. The lock and granule metadata may be used by the policy code when

determining the mode for an execution attempt, and their structure may be policy-dependent. The library's API, however, abstracts away any dependency on the policy by providing the macros for defining lock labels and scope IDs that uniquely identify the associated metadata.

Each time a critical section is attempted, the library invokes the policy to determine the mode in which it should be executed (`HTM`, `SWOpt`, or `Lock`), and executes appropriate critical section preamble code accordingly. For `Lock` mode, it acquires the lock. For `HTM` mode, it first waits for the lock to be free, then begins a hardware transaction, and then checks that the lock is not held, aborting the transaction and retrying (possibly in a different mode) if the lock is held, and returning to user code to execute the critical section otherwise. For `SWOpt` execution, the library returns to user code without acquiring the lock. The library keeps track of the execution mode, which can be obtained via the `GET_EXEC_MODE` macro (see example in Section 3.2). We also note that using the `is_locked` function provided by the lock's API, the library estimates whether a hardware transaction has been aborted due to a concurrent lock acquisition by another thread. To reduce the chance for a cascade effect, the library accounts for such aborts in a much lighter way than for others.

4.1 Nesting

Nesting of `ALE`-enabled critical sections may be motivated by performance, as in Section 3.3, and is also likely to arise as a natural consequence of modularity. In this section, we describe the `ALE` library's support for nesting.

`ALE`-enabled critical sections must be properly nested: a lock released at the end of or within an `ALE`-enabled critical section must be the most recently acquired lock that has not since been released. This requirement is an outcome of our design choice, whereby per-thread stacks of *frame*s are used to record information associated with the critical section executed at each nesting level, including the lock accessed and its associated metadata, the relevant granule, and information about the current mode and execution attempt.

For each critical section execution, a frame is pushed onto the thread's stack before the first attempt, and removed after successful completion. (For reasons explained below, no frame is pushed for a critical section that is nested within another that is executing in `HTM` mode.)

If a critical section is not nested within another `ALE`-enabled critical section, then the eligible modes for executing the critical section are determined by the availability of HTM and of a `SWOpt` path (unless the programmer explicitly prohibits one or both). For a *nested* critical section attempt, the library further restricts the choice of modes in some cases.

First, if a critical section (nested or not) is executed using HTM, all critical sections nested within it are also executed using the same hardware transaction, while checking that their associated locks are not held. (If a nested critical section does not allow `HTM` mode, the hardware transaction is aborted.) This is because committing a transaction for an enclosing critical section in order to begin a nested critical section in another mode would likely violate the atomicity of the enclosing critical section. To minimize the duration of hardware transactions, and to reduce the amount of data written within them, a frame is pushed onto the stack only for the outermost critical section executed in `HTM` mode.

Next, if a thread already holds the lock accessed by a nested critical section, a `SWOpt` path is not used even if available. This is because there would be no benefit to doing so, and allowing this case would complicate the library. In this case, `HTM` mode may be chosen but, to avoid an unnecessary abort, the library does not check whether the lock is held.

Finally, `SWOpt` mode is not eligible if the thread is already executing in `SWOpt` mode for a critical section associated with a *different* lock. We believe that using the library correctly in this way would be too difficult and error prone. This does not imply that programmers cannot nest `SWOpt`-capable critical sections, only that the library will not choose `SWOpt` mode for the nested critical section in this case.

4.2 Policies

Below we describe two policies we have implemented. These policies use parameters X (respectively, Y) for the number of attempts to use in `HTM` (respectively, `SWOpt`) mode before switching to the next mode in the chosen progression. The *static* policy uses fixed parameters, while the *adaptive* one "learns" parameters based on observed behavior.

Static policy.

The static policy uses fixed values of X and Y for all critical section executions. It makes up to X attempts using HTM (if available). If unsuccessful it then makes up to Y attempts using the `SWOpt` path (if available). It resorts to acquiring the lock if these attempts are also unsuccessful.

Adaptive policy.

The adaptive policy is more flexible and more dynamic than the static policy, as it may choose a different number of attempts in each mode for each granule (a combination of a lock and a context), and it chooses these values based on a learning mechanism that takes into account the statistics collected by the library for the granule. This allows the policy to recognize, for example, that the fastest execution of one critical section in a given context is likely to be achieved by using `HTM`, while `SWOpt` is preferable for another, and determine the number of attempts to be used for each such context. Finally, adaptive policies can exploit `ALE`'s fine-grained statistics to improve performance by controlling concurrency; one example is described next.

Grouping mechanism. Recall that a `SWOpt` path is caused to retry only if a critical section protected by the same lock executes a conflicting region (implying that the latter is executed in either `HTM` or `Lock` mode). Thus, if such critical section executions for that lock are temporarily prevented from executing, then all `SWOpt` paths for critical sections associated with the same lock can execute in parallel without interference. Therefore, we employ a *grouping mechanism* that attempts to run executions of `SWOpt` paths associated with the same lock concurrently, while delaying the execution of critical sections that may conflict with them.

In most cases, this approach is sufficient to guarantee that any critical section for which a `SWOpt` path is available can always complete without acquiring the lock. However, this is not guaranteed in some complex nesting cases. Thus, the Adaptive policy still sets Y to a large value to ensure that (rare) livelocks do not persist indefinitely. However, in our experience so far, grouping is effective enough that `SWOpt` mode always succeeds with much fewer than Y attempts.

The grouping mechanism uses a scalable non-zero indicator (SNZI) [6] to track whether any threads executing `SWOpt`

are retrying. If so, executions that potentially conflict with SWOpt executions wait for the SNZI to indicate that all such SWOpt executions have completed. Though we have not yet done so, concurrency could be increased by respecting the SNZI probabilistically, which would still ensure that potentially conflicting executions will *eventually* defer to concurrent SWOpt executions. Grouping can improve performance significantly when SWOpt executions retry multiple times.

Learning mechanism. Each lock goes through one learning phase for each mode progression (Lock, SWOpt+Lock, HTM+Lock, HTM+SWOpt+Lock). Phase transitions for lock L occur when some context of L completes a certain number of executions. (We do not wait for this to occur for *all* contexts of L as some contexts may be used infrequently.) In each learning phase, statistics including the average time of successful and failed attempts in each mode are collected for each granule, and these are used to choose relevant X and Y parameters for the granule, as described below.

To choose X parameters, phases for combinations that include HTM mode comprise three sub-phases. In the first sub-phase, we start with X set to a large number, and then adjust its value to the maximal number of attempts so far required to complete executions of the critical section using HTM, plus a small constant. In the second sub-phase, using the value learned in the first, we create a histogram of the number of attempts required to succeed in HTM mode, and count the number of executions that did not complete using HTM. Using the histogram and the timing statistics, we estimate the expected execution time of the critical section for each possible number of HTM attempts, as explained below, and set the X parameter for use in the third (performance measurement) sub-phase to the number of attempts that yields the lowest estimate.

To estimate the expected execution time for a given number of HTM attempts, we must estimate the time to execute the critical section if HTM is unsuccessful. We use as an upper bound the time measured during learning when HTM was not attempted (i.e., in Lock or SWOpt+Lock learning phase), and as a lower bound the time taken after failing the maximum number of HTM attempts. For each number of HTM attempts, we interpolate between these bounds, i.e., we assume that the non-HTM execution time grows linearly from the lower bound to the upper bound as we reduce the number of HTM attempts from the maximum to zero. While this simple model ignores many practical factors, we have had good success with it (see Section 5).

Having measured execution times for each mode progression for each lock, we could choose for each granule the mode progression that yielded the lowest average execution time for that granule. However, as different granules associated with the same lock may choose different mode progressions, this may result in a combination of mode progressions that has not been measured. This may or may not result in performance that is better than any of the measured mode progressions, because interactions between concurrent use of different mode progressions for different granules may help performance or hurt it. For example, executing one granule with the HTM+Lock progression concurrently with another granule using SWOpt+Lock may result in many conflicts between HTM and SWOpt executions that never occurred during the learning phases. Conversely, some mixtures of mode progressions can interact beneficially, such as when some lock acquisitions make it easier for SWOpt executions to run without interference.

For this reason, we run one more "custom" measurement phase for each lock, based on the per-granule choices, and only use these local choices if they yield a lower average execution time than was measured during the learning phases; otherwise we choose for all granules of that lock the mode progression that achieved the lowest average execution time during the learning phases.

Finally, we note that the custom phase still does not necessarily find the configuration that maximizes the throughput for critical sections under a particular lock because the per-granule mode progression choices and the X values used are based on measurements taken when all granules used the same mode progression. Evaluating all possible configurations is impractical (exponential in the number of contexts). Nonetheless, finding more sophisticated learning mechanisms is part of our future work.

4.3 Statistics counters

Historical summary information recorded by the library includes counts of events, such as number of attempts and successes using a given method, and timing information, such as average lock acquisition time. We use two approaches to avoid excessive overhead and contention that naive methods for synchronizing this information would entail.

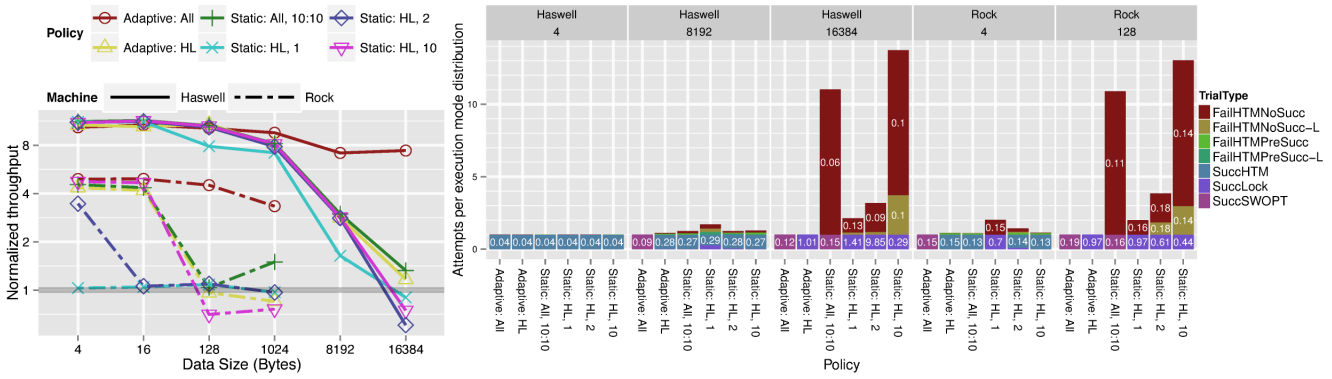
For time intervals, we measure the time period of interest for approximately 3% of events, and use CAS to update summary variables. Exponential backoff is employed to mitigate any remaining contention, which is typically low due to the sampling. While this approach is simple and reasonably effective, it does not provide a reliable level of accuracy until many hundreds of events have been measured.

For counting events, a higher level of accuracy, especially after a relatively smaller number of events, is desirable. For this purpose, we use a statistical counter algorithm (the BFP algorithm in [4]), which gradually reduces the probability of updating shared data, while maintaining high accuracy even after relatively small numbers of events. This algorithm supports counters that are incremented only by one, and thus cannot be used to record time-based statistics.

5. INITIAL EXPERIENCE WITH ALE

We have experimented with ALE using a HashMap microbenchmark and the wicked benchmark of the Kyoto Cabinet [8] on four platforms, including two that support best-effort HTM: Rock [5], a 1-socket 16-core SPARC system, and Haswell [14], a 1-socket hyper-threaded 4-core x86 system. Below, we present some of our experiments on these systems, and a 2-socket, 128-thread SPARC T2+ (T2-2).

In all figures, Instrumented denotes a version that is integrated with ALE, so that statistics and profiling information is available, but only the lock is used to execute critical sections. Uninstrumented denotes a baseline implementation that is not integrated with ALE. Other versions are named by the policy, the techniques used—HTM, SWOpt, or both (denoted as All)—and relevant parameters, if any. For example, Static-All-10:10 is the static policy that tries with HTM up to 10 times and then with SWOpt up to 10 times. Irrelevant parameters are omitted. For example, Static-HTMLock-2 denotes a version in which SWOpt is disabled and the static policy attempts HTM at most twice. For readability in figures, we abbreviate HTMLock as HL and SWOPTLock as



(a) Normalized throughput (b) Attempt/executions mode distribution (titles show machine & object size)

Figure 2: Evaluating policies on Haswell and Rock with `Get`-only HashMap workload, 4 threads, and different object sizes.

`SL`, and use more readable notation such as "`Static: HL, 2`" for `Static-HTMLock-2`. Each data point is the median over five runs. Most results exhibit negligible variance.

5.1 HashMap microbenchmark

In this microbenchmark, each of a fixed number of buckets (1,024 in our experiments) holds an unsorted list of key-value pairs whose keys hash to that bucket. `Insert`, `Remove` and `Get` operations are supported; `Get` copies the return value into a caller-provided memory location. In these experiments, only `Get` uses a `SWOpt` path. The table is protected by a single `pthread_mutex_t` lock.

Figure 2(a) shows a `Get`-only workload on the two HTM systems using HashMaps for various object sizes, chosen to reflect the systems' different capabilities in terms of how much data can be written in transactions. To focus on this difference, a small key range of 16 is used and the table is initially full, so the key sought by every `Get` operation is always present. Results are shown for 4-thread runs, and are normalized to `Instrumented` for a meaningful comparison between experiments with different object sizes. Note that we use log-scale for y-axis and discrete values for x-axis.

Focusing first on the `HTMLock` cases (i.e., those that do not use the `SWOpt` path), for "large" objects (which cannot be copied in a hardware transaction), the static policies all perform worse than the `Instrumented` (`Lock`-only) configuration because time spent on failed `HTM` attempts is wasted. On the other hand, for "small" objects, HTM can copy the objects, so throughput is improved if `HTM` mode is retried enough to succeed. Due to the different capabilities of the two platforms, objects that are "large" for Rock can be "small" for Haswell. Furthermore, the poor performance with one and two `HTM` trials on Rock clearly shows that the best choice for the number of trials is architecture-dependent, and thus the best policy choices depend on both the platform and the workload. Thus, while no static policy suffices for all cases, the adaptive policy achieves competitive performance with all static policies in all cases.

Adding `SWOpt` mode improves the performance for both the static and the adaptive polices; however, it has a much bigger impact with the adaptive policy, which avoids the failed `HTM` attempts prior to using `SWOpt`.

Figure 2(b) uses data collected by the library to shed more light on the results in Figure 2(a). The height of the bars in the figure shows, for each <policy, machine, object size> combination, the average number of attempts for each successful execution. Each bar breaks down these attempts by category, indicating their mode, whether they were successful, what happened subsequently, and for `HTM` failures, whether we conjecture that the failure is due to the lock being held (indicated by the "-L" suffix of the label). For example, `FailHTMNoSucc` represents attempts in `HTM` mode that failed in critical section executions that were not eventually completed in `HTM` mode, while `FailHTMPreSucc` represents failures in `HTM` mode for critical sections that eventually succeeded in `HTM` mode. Finally, the numbers presented in some of the categories' bars indicate the average *time* of a trial in that category, relative to the average time it took to execute the operation in the `Instrumented` (`Lock`-only) version.

The figure confirms that, on both machines, for "large" objects, the static policies pay a significant overhead for wasted `FailHTMNoSucc` and `FailHTMNoSucc-L` attempts, even when `SWOpt` is enabled. Still, when comparing `Static-All-10:10` and `Static-HL-10`, we can clearly see that the successful attempt in `SWOpt` is up to three times faster than that in `Lock` mode. The failed trials in `HTM` are also faster when `SWOpt` is enabled, because they do not need to wait for the lock to be free prior to starting a hardware transaction. Indeed, the static policy with `SWOpt` achieved the best results among all static polices in this experiment. The adaptive policy, on the other hand, does not make any attempts in `HTM`, which explains why it outperforms all static policies.

We also see that the adaptive policy chooses to use `SWOpt` even for 4B objects on Rock, while on Haswell it uses `HTM` for this case, and only uses `SWOpt` for larger objects. This choice is due to the *relatively* higher overhead of validation when the object is small, and the fact that on Haswell we almost always succeed using `HTM` on the first attempt with 4B objects. This is not the case on Rock, as demonstrated by the bar for the static policy with one attempt in `HTM`.

Another interesting phenomenon with large objects is how additional failed attempts in `HTM` reduce the contention on the lock, and thus the time it takes to execute the last trial of an execution. The figure shows that `SuccLock` trials take less time with `Static-HL-10` than with versions like `Instrumented` and `Adaptive-HL` that acquire the lock immediately. On the other hand, there are some cases where adding only a few failed attempts in `HTM` makes the average attempt using the lock significantly worse, like in the case for `Static-HL-2` on Haswell. We looked into this case using

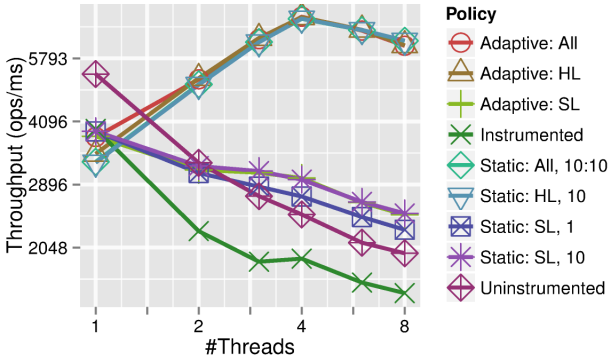


Figure 3: Mixed workload on Haswell (log-log scale)

additional stats (not shown) and noticed that, in these cases, there was always unfairness between the threads, where one thread acquired the lock much more frequently (and without contention), while the others were delayed for a long time. The number of factors and tradeoffs involved clearly makes it impractical to tune static policies manually for different platforms and workloads. This highlights the importance of adaptive policies that can make these decisions based on observed behavior.

These `Get`-only experiments may suggest that `SWOpt` alone is sufficient and `HTM` is not needed. However, the advantages of `HTM` over `SWOpt` are clearly demonstrated in the next experiment. Figure 3 shows results with a mixed workload (60% `Get`, 20% `Insert`, 20% `Remove`) on Haswell, for a HashMap with a key range of 4,096 and 1,024 buckets. The HashMap is initially half full, so there are about two pairs per bucket on average throughout the experiment. The object size is 1,024B, so `Get` can succeed in `HTM`.

First, `Instrumented` achieves 26% less throughput relative to `Uninstrumented` in single-threaded experiments. The gap reduces to less than 20% at 8 threads, as the instrumentation cost introduced by `ALE` is hidden by lock contention. However, all other `ALE`-enabled versions outperform `Uninstrumented` for two or more threads, demonstrating that the scalability and analytic benefits gained from using `ALE` far outweigh the cost of instrumentation.

Because the `Insert` and `Remove` methods acquire the lock, the `SWOpt` path for `Get` encounters significantly more interference in this case. Therefore, while `SWOpt` improves performance relative to `Uninstrumented`, it still exhibits negative scalability. Configurations with `HTM` fare significantly better for two or more threads by exploiting `HTM`'s ability to execute multiple `Insert` and/or `Remove` operations in parallel.

The `Adaptive-All` version performs comparably with or better than other configurations (except `Uninstrumented` at one thread). Data collected by `ALE` reveals that the adaptive policy sometimes avoids `HTM`, and uses `SWOpt` or `Lock` for `Get`. This allows `Adaptive-All` to outperform `Static-All-10:10` at small thread counts.

5.2 Kyoto Cache Benchmark

The CacheDB variant of the Kyoto Cabinet (KC) library implements an in-memory database containing records in a hash table [8]. We used the `wicked` benchmark provided with the KC library, modified to keep the key range and the total number of operations independent of the number of threads, for easier scalability evaluation. The benchmark applies a random series of database operations.

We first used `ALE` solely to collect statistics, without using any optimistic techniques. The statistics revealed that the most common method is `accept` and that contention between read acquisitions of a `pthread_rwlock_t` readers-writer (RW) lock [2] in this method caused *negative* scalability, even at low thread counts. (Using `ALE` with the RW-lock is achieved by using different `LockAPI` structures to express acquisitions for reading and writing.)

The statistics also revealed that one of two methods (in the original benchmark) that begins a database transaction has a performance bug. Only one percent of the benchmark's operations try to begin a transaction, and only half of these use the buggy method. We changed the benchmark to use the non-buggy method for all begin transaction operations.

The purpose of the RW-lock is to allow some operations—such as beginning and ending a database transaction—to gain exclusive database access; these operations acquire the RW-lock for writing, and all other database access operations acquire it for reading.

The `accept` method searches by key for a hash table entry and, if found, applies a caller-provided operation on it. The hash table has 16 *slots*, each comprising a number of *buckets*, implemented as search trees. A key is first hashed to identify the slot where its bucket resides, and then hashed again to identify its bucket in that slot.

Each slot has an associated `pthread_mutex_t` slot lock. The `accept` method acquires the slot lock for the slot containing the specified key; the critical section that acquires this lock is nested within another one that acquires the RW-lock in read mode, as discussed above.

Providing a `SWOpt` path for `accept`'s read acquisition of the RW-lock is more interesting than the simple cases discussed so far, because of the critical section nested within it that acquires one of the slot locks. For space reasons, below we sketch only the most important aspects of our fairly detailed exploration of using `ALE` to improve KC performance.

When no entry associated with the specified key is present, the `accept` method does not modify shared state other than the RW-lock and the relevant slot lock. These actions do not conflict with other operations, and therefore it is straightforward to provide a `SWOpt` path in which the outer critical section (which acquires the RW-lock) is executed in `SWOpt` mode, and the nested critical section (which acquires a slot lock) has no `SWOpt` path. (Some care is required after acquiring the slot lock, because another thread may acquire the RW-lock in write mode and expect that no other thread is holding any slot lock. Thus, for example, `SWOpt` execution must be validated while traversing the entries in a bucket (even though the slot lock is held by the traversing thread) because of a potential conflict with an operation that holds the RW-lock in write mode and may modify entries in the bucket without acquiring the slot lock.)

The case in which the key *is* present is less straightforward. The simplest option is to use the self abort idiom described earlier: in this case, the `SWOpt` attempt is abandoned, and the critical section is retried, acquiring the RW-lock. This approach, denoted as `nomutate`, still avoids acquiring the RW-lock when the specified key is not found.

A better solution is to instead acquire the RW-lock in read mode, and then validate the `SWOpt` execution to detect interference that may have occurred during the acquisition. If validation succeeds, it is as if the RW-lock were acquired as normal in the first place. However, reversing the order of

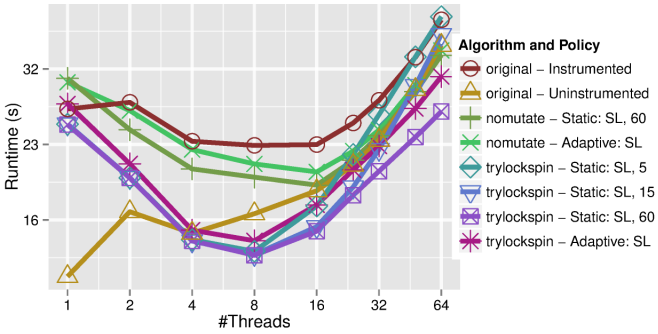


Figure 4: Execution time for KC (log-log scale) on T2-2

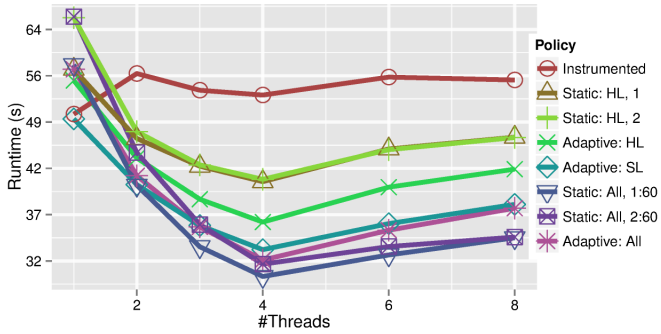


Figure 5: Execution time for KC on Haswell

acquiring the RW-lock and the slot lock introduced a potential deadlock. We therefore use `pthread_rwlock_tryrdlock` to *attempt* to acquire the RW-lock in read mode. If the attempt succeeds, all the locks that are required to complete the operation are held, and the `SWOpt` execution is validated one last time before completing the operation.

If the attempt to acquire the RW-lock for read fails, we could use the self-abort idiom as before, and retry the critical section after acquiring the RW-lock. However, because of the recent failure to acquire the RW-lock, it is likely that a thread has acquired or is waiting to acquire it for write and will be granted the lock first. We therefore implemented an optimization that performs a bounded wait for the version number to be modified and then retries, using another `SWOpt` attempt, avoiding additional contention on the RW-lock. To avoid starvation, the library will eventually acquire the RW-lock, but with the adaptive policy and the grouping mechanism this never happens in practice. We denote this algorithm as `trylockspin`.

Figure 4 presents results on T2-2, clearly showing that both of our algorithms, `nomutate` and `trylockspin`, outperform the version without `SWOpt` (`Instrumented`), and scale up to 16 and 8 threads, respectively. The figure also shows that `trylockspin` performs better than `nomutate`. Next we compare `trylockspin` with the adaptive policy to three variants of the static policy, which perform 5, 15, and 60 attempts in `SWOpt`, respectively. These choices reflect the manual tuning required to find that the best performing static policy for this algorithm is achieved with 60 trials in `SWOpt`. The adaptive policy, on the other hand, is generally competitive with this variant and does not require any manual tuning.

None of the solutions scale beyond 16 threads. The statistics showed high contention on the RW-lock due to *write* acquisitions, which in turn imposed a significant delay on the `SWOpt` attempts that do find the key, as these have to succeed in their attempt to acquire the RW-lock for reading in order to complete; we note that the `pthread_rwlock_t` implementation prevents readers from acquiring the lock not only when a writer is holding the lock, but also when a writer is waiting for the lock. Thus, a `SWOpt` execution has to wait for all the pending writers to complete their operations before it can succeed; this results in both additional `SWOpt` attempts as well as in longer waiting for the version number to change when a `SWOpt` attempt fails.

Next, we explore the use of `HTM` on Haswell to optimize `accept`. We first evaluate what can be achieved without requiring the code changes to implement a `SWOpt` path.

The simplest approach is to enable `HTM` mode for both the inner and outer critical sections, and fall back to the lock when `HTM` fails; our nesting mechanism allows both critical sections to be executed within a hardware transaction, checking that neither lock is held. We tested this approach by using the `HTMLock` configuration with both the adaptive policy, as well as with two static policies that use one and two attempts in `HTM` mode, respectively. The results are presented in Figure 5, with a log-scale `y` axis.

Focusing on the `HTMLock` results, we see that the static policy with a single trial in `HTM` outperforms the one with two trials in `HTM` with only one thread, but the two behave similarly at higher thread counts. The adaptive policy delivers better performance than both static variants for `HTMLock`, at all thread counts. This was also true with more `HTM` attempts with the static policy (not shown). The reason for that is because the static policy uses the same number of `HTM` attempts for both the internal and external critical sections. So, for example, with a single thread, because the external critical section has very little code that is executed outside of the internal critical section, it does not make sense to try the internal critical section with `HTM` given that the external already failed in `HTM`; still, the static policies attempt the internal critical section once or twice before acquiring the slot lock. The statistics showed that about 20% of `accept` executions failed to commit using `HTM`, even when running single threaded, and regardless of the number of trials. For static policies it means a total overhead of two or four failed hardware transactions for 20% of the executions. The adaptive policy, on the other hand, chooses to execute the internal critical section using the `Lock` method, and thus saves the failed `HTM` attempts for these 20% of the executions. Furthermore, the statistics showed that the remaining 80% of executions that do succeed on `HTM`, require just one attempt when the number of threads is low, and at least three attempts, on average, at higher thread counts. Indeed, the adaptive policy sets the limit on the number of `HTM` attempts to one when run single-threaded, and to five when run with eight threads.

We note that, in general, our learning mechanism is not sophisticated enough to take nesting into account; in particular, the learning of the internal critical section is done concurrently with that of the external critical section, and thus learning of the internal critical section is not limited to executions where the external critical section failed executing in `HTM`. However, for the `HTMLock` case, because we have only two learning phases (`LockOnly` and `HTMLock`), the policy was able to choose the right combination.

Despite the good results with the `HTMLock` configuration, the `SWOptLock` configuration achieves even better results (cf.

Figure 5). The statistics of the `nomutate` variant on T2-2, showed that 42% of the executions did not find the object they were seeking, and hence succeeded using `SWOpt`. The implication for the `trylockspin` algorithm is that in 42% of the cases the RW-lock is not acquired, and hence only the cost of a single acquisition of a slot lock is paid, while the remaining 58% of the cases incur an additional acquisition attempt of the RW-lock, which is usually successful when the number of threads is low. With `HTMLock` a relatively large hardware transaction is executed (and fails in 20% of the cases) and two lock acquisitions are needed: one of the RW-lock and the other of the slot lock.

These results led us to explore the case in which we enable both `HTM` and `SWOpt` for the external critical section, and only `HTM` for the internal critical section. The results are shown by the `{Static,Adaptive}:All` curves in Figure 5.

As the figure shows, except for the single thread case, the adaptive policy that uses both `HTM` and `SWOpt` matches or outperforms the adaptive policy that uses only `SWOpt`. The reason is that using `HTM` for the external critical section reduces the number of acquisition trials for the RW-Lock, which reduces contention at higher thread counts. In this case, however, both static policies outperform the adaptive policy at high thread counts. Based on the statistics, this is primarily because our learning mechanism is not sophisticated enough to deal with nesting. We hope to address this issue in future work.

These results reinforce two important conclusions from our work. First, the combination of software and hardware lock elision techniques achieves considerably better results than when only one of the techniques is available. Second, adaptive policies can perform comparably with the best static policy, without manual tuning.

6. CONCLUDING REMARKS

The `ALE` library integrates transactional lock elision using hardware transactional memory and optimistic execution of operations in software, so that either technique can be used to improve performance and scalability. Decisions regarding whether, when and which technique to use can be made by a pluggable policy, which can collect various profiling information and statistics, and can use this information to guide its decisions. This allows the choice of mechanism to be made at runtime based on the platform, availability of hardware features, and workload characteristics.

Using the library with a simple adaptive policy, we have achieved significant improvements in performance and scalability for a HashMap microbenchmark across a range of platforms and workload parameters, even though different techniques are better for different circumstances. This demonstrates the potential of our approach and establishes a foundation on which to explore more sophisticated policies, for example that can adapt to workloads that change over time.

We have further explored the use of our library in a real example. This effort again showed that we could achieve significant improvements using the adaptive policy, which is highly competitive with hand-tuned static policies, none of which is effective for all cases. Furthermore, the library has proved valuable simply for analyzing an application's behavior, and assessing which locks and critical sections are likely to be most profitable to optimize. Our work also revealed additional challenges that arise when using such techniques in realistic code bases, leading us to develop more sophisticated support for nesting, for example. In ongoing work, we are applying these techniques to a wider range of benchmarks and applications, and continuing to improve on our adaptive policy.

Acknowledgments: We are grateful to Tim Harris and Victor Luchangco for feedback on early drafts.

7. REFERENCES

[1] Corbet. Driver porting: mutual exclusion with seqlocks, February 2003. `lwn.net/Articles/22818/`.

[2] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[3] Dave Dice, Maurice Herlihy, Doug Lea, Yossi Lev, Victor Luchangco, Wayne Mesard, Mark Moir, Kevin Moore, and Dan Nussbaum. Applications of the adaptive transactional memory test platform. In *Workshop on Trans. Computing (Transact)*, 2008.

[4] Dave Dice, Yossi Lev, and Mark Moir. Scalable statistics counters. In *Proc. ACM SPAA*, 2013.

[5] Dave Dice, Yossi Lev, Mark Moir, Dan Nussbaum, and Marek Olszewski. Early experience with a commercial hardware transactional memory implementation. Technical Report TR-2009-180, Sun Microsystems Laboratories, 2009.

[6] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. SNZI: scalable nonzero indicators. In *Proc. ACM PODC*, pages 13–22, 2007.

[7] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. Improving in-memory database index performance with Intel TSX. In *Proc. IEEE HPCA*, 2014.

[8] FAL Labs. Kyoto Cabinet: A straightforward implementation of DBM. `http://fallabs.com/kyotocabinet`.

[9] Christoph Lameter. Effective synchronization on Linux/NUMA systems. In *Gelato Conference*, 2005. `www.lameter.com/gelato2005.pdf`.

[10] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proc. ASPLOS*, pages 25–35, 1994.

[11] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc. IEEE/ACM Micro*, pages 294–305, 2001.

[12] Qingping Wang, Sameer Kulkarni, John Cavazos, and Michael Spear. Towards applying machine learning to adaptive transactional memory. In *Workshop on Trans. Computing (Transact)*, 2011.

[13] Lingxiang Xiang and Michael L. Scott. Compiler aided manual speculation for high performance concurrent data structures. In *Proc. ACM PPOPP*, pages 47–56, 2013.

[14] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *Proc. SC*, 2013.