Predictive Log-Synchronization

Ori Shalev
School of Computer Science
Tel-Aviv University
Tel Aviv, Israel 69978
orish@post.tau.ac.il

Nir Shavit
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803-0903
shanir@sun.com

ABSTRACT

This paper proposes *predictive log-synchronization*, an alternative paradigm to the software transactional memory approach for simplifying the design of concurrent data structures. Predictive log-synchronization simplifies concurrent programming and program verification by requiring programmers to write only specialized *sequential code*. This sequential code is then automatically transformed into a *non-blocking* concurrent program in which threads coordinate all data structure operations via a shared lock-controlled *log*. The non-blocking progress property is achieved by having threads that fail to acquire the lock *predict* the outcome of their operations by reading the log and state and computing the effect of these operations without modifying the actual data structure.

Log-synchronization is founded on the belief (at this point unsubstantiated by statistical data) that in many concurrent data structures used in real-world applications, the ratio of high level operations that modify the structure to ones that simply read it, greatly favors read-only operations, and what's more, that many natural data structures have inherent sequential bottlenecks limiting the concurrency among operations that modify the structure. It follows that delegating all data structure modifications to a single lock-controlled thread at a time will not significantly harm the throughput of modifying operations. Moreover, as we show, it can boost read-only throughput by significantly reducing the overhead of coordination among concurrent operations, and provides a way to simplify concurrent data structures.

Initial experimental testing using a Java-based implementation of predictive log-synchronization showed that a log-synchronized concurrent red-black tree is up to five times faster than a simple lock-based one. This paper presents our current understanding of the advantages, drawbacks, and scope of predictive log-synchronization.

EuroSys'06, April 18–21, 2006, Leuven, Belgium.
Copyright 2006 ACM 1-59593-322-0/06/0004 ...$5.00.

Categories and Subject Descriptors

C.1.3 [**Concurrent Programming**]: Parallel Programming

General Terms

Algorithms

Keywords

Concurrent, Synchronization, Prediction, Monitor

1. INTRODUCTION

Designing lock-based concurrent data structures has long been recognized as a difficult task better left to experts. If concurrent programming and data structure design is to become ubiquitous, one must develop alternative paradigms that simplify code design and verification.

The *transactional memory* programming paradigm [7] is gaining momentum as an approach of choice for replacing locks in concurrent programming. Combining sequences of concurrent operations into atomic transactions seems to promise a great reduction in the complexity of both programming and verification, by making parts of the code appear to be sequential without the need to use locks. The "transactional manifesto" is that transactions will remove from the programmer the burden of figuring out the interaction among concurrent operations that happen to overlap or modify the same locations in memory. Transactions that do not overlap will run uninterrupted in parallel, and those that do will be aborted and retried without the programmer having to worry about issues such as deadlocks. The current implementations of transactional memory are purely software based (software transactional memories (STM)), but down the road, combinations of hardware and software will hopefully offer the simplified reasoning of transactions with a low computational overhead.

However, the transactional approach is not a panacea, and like locks, suffers from several notable drawbacks. Transactions simplify but do not eliminate the programmer's need to reason about concurrency, and the decision of which instructions to include in a transaction are explicitly in the hands of the programmer. This leaves the programmer with a trade-off similar to that of locks, between the size of the transactions used and the program's performance: any reduction of transaction size implies added complexity in code design and verification. Concurrently executing transactions also introduce numerous programming language issues such as nesting, I/O, exceptions [3], and early release [5], that offer to complicate transactional programming even if it is made efficient. Finally, though great progress is being made in this area, transactions implemented in software (i.e. STMs) are at this point less efficient, in some cases they can be even slower than monitor locks, because of the overhead of the conflict detection and copy-back mechanisms used in implementing them [4, 5, 10, 11, 13, 19, 21].

1.1 Predictive Log-Synchronization

This paper proposes the *Predictive Log-Synchronization* (PLS) framework, an alternative to software transactional memory for simplifying the design of concurrent data structures. PLS simplifies concurrent programming and program verification by requiring programmers to write only specialized *sequential code.* This code is then transformed into a *non-blocking* concurrent program in which threads coordinate all data structure operations via a shared log. Like with transactional memory, being non-blocking eliminates the possibility of deadlocks and mitigates some of the effects of processor delays.

In a nutshell, in PLS the shared data structure is duplicated, protected by a high level lock, and appended with a special *log* of high level operations. A thread owning the lock performs all data structure modifications logged by others on one copy, allowing all threads to concurrently read the other unmodified copy, and switches copies before releasing the lock. PLS is non-blocking since threads failing to acquire lock ownership make progress by inspecting the log and *predicting* the result of their own operation. Concurrently, all read-only operations can access the unmodified copy of the data structure in a non-blocking manner and with virtually no overhead (without processing the contents of the log.)

PLS is thus, in a sense, orthogonal to transactional memory. It is motivated by our belief that for many classes of applications, the number of calls to high level operations that modify a data structure is significantly smaller than that of ones that simply read it. What's more, many natural data structures (consider stacks, queues, heaps, search trees) have inherent sequential bottlenecks limiting the throughput of modifying high level operations. Thus, delegating all data structure modifications to a single lock-controlled thread at any given time does not significantly harm performance, yet, as we show, will allow high read-only throughput and simplified concurrent programming.

The idea of using a log to collect "write" operations and execute them in a batch mode has been proposed in the context of *log structured file systems* [16]. The idea of logging lists of operations on a lock-protected object to be executed by the thread owning the lock was proposed by Oyama et al [14] as a low overhead means of increasing locality in performing modifications to shared locations within a mutually exclusive section. In [21], Welc et al. use logs as a conflict detection mechanism for transactions, and in [20], they use logging to support rollback for delayed low-priority threads holding a resource. McKenney and Slingwine in [12] proposed the *read-copy update* (RCU) methodology, according to which modifying threads cooperatively schedule callbacks in such way that concurrent readers are not required to acquire locks in order to get a consistent view of the data structure. Writer thread operations are logged as callbacks, but unlike PLS, the log is not read by peer threads. RCU does not eliminate the need to use locks and handles synchronization based on integration with the operating system through the use of interrupts. PLS differs from the above approaches in a fundamental way: it uses the log to allow readers to predict the effect of write operations before they actually take effect, allowing them to proceed concurrently and in a non-blocking manner even if the actual modifications are performed sequentially by some process at a later time. The idea of switching among duplicate copies of memory locations to allow consistent concurrent reads appears in Riany et al [15] and was well known in various forms in the garbage collection and database literature long before.

In an earlier paper [18] we showed how to construct a word-level log-based synchronizer as a replacement for monitors. PLS differs significantly from that construction. Our former log synchronizer was a lower level mechanism, using the log to record word level operations, not method descriptions as in PLS. More importantly, our former log synchronizer did not use prediction and was thus blocking, in contrast to PLS which introduces prediction as a means of providing non-blocking progress even though the data structure is controlled by a lock.

1.2 Performance

We present a set of proof-of-concept micro-benchmarks intended to show that there is merit to the predictive log-synchronized approach, but are in no way intended to be a full and comprehensive exposition of PLS performance. We note that our benchmarks show high throughput but deteriorating PLS performance as concurrency increases. We believe that the results would look much better if we had some control over scheduling and/or could heuristically decide whether a thread should apply prediction or just backoff when failing to acquire the shared lock.

Though log-synchronization will not benefit every data structure under every workload, as an example of its potential, we chose a commonly benchmarked data structure: a red-black tree data structure representing a set (used in benchmarking various software transactional memory systems [4, 11, 6, 13]) and wrote it in a predictive log-synchronized manner. We compared our implementation to lock-based and an STM-based Java$^{\text{TM}}$ implementations of a red-black tree set provided in [10]. The STM we used was the implementation by Marathe et al. of Harris and Fraser's OSTM [1]. This is an "object-based" STM in which transactions are carefully designed to open and close objects for read and write to minimize STM calls. Our experiments show that PLS performs better than monitors and in write-dominated workloads, better than the object-based OSTM [10].

1.3 PLS versus STM: A Short Summary

The following is a short summary of our current understanding of the relative benefits and drawbacks of PLS relative to STM.

- PLS is directed at the implementation of concurrent data structures as shared objects. Its scope is thus not as general as STMs which can support transactions applied across objects. One could extend PLS in this direction but this would be outside the scope of this paper.

- The memory consistency condition provided by most STMs is Herlihy and Wing's linearizability [8], while the consistency provided by PLS is Lamport's sequential consistency [9]. Linearizability is a stronger property that guarantees better composability properties then sequential consistency.

- STMs offer concurrent transactional programming as an interface. Transactions simplify code by collecting operations into atomic transactions, but require the programmer to understand concurrency, granularity vs. performance tradeoffs, nesting, early-release, and various other semantic issues. The PLS Log-based programming may be simpler as it involves no concurrent reasoning since it is based on writing only sequential code. However, this sequential programming is not as straightforward as transactions: it requires an understanding of log-based computation. It is not clear which of these two approaches provides the better programming environment.
- Transactional code is concurrent (one must restrict granularity of atomicity to achieve good performance) and, though simpler than lock based code, is hard to verify correct, while log-synchronized code is sequential and thus significantly simpler to verify. This could prove to be a big benefit of PLS, as verification of concurrent programs is hard.
- The performance of PLS, on our tested benchmarks, is always superior to STMs, in most cases orders of magnitude better.
- There are various complex programming language issues with transactions that have yet to be solved, among them the handling of nesting and I/O. These issues are easily solved in PLS by requiring any operation containing nesting or I/O to be executed only when holding the lock.

2. PLS DESIGN

This section explains the programming and algorithmic design aspects of PLS.

2.1 Log-based Programming

How does the log-based programming of the PLS framework work? Consider the following log-based computational model: a data structure has an initial state and a log that contains the history of all high level operations (method calls) applied to the initial state. Clearly the application of the log to the initial state describes the current state of the structure at any point in time. To keep the log from growing without end, we can at any point apply in-order a subsequence of the operations at the head of the log to the state to derive a new state, and remove these operations from the log. This means that in order to compute the outcome of a given operation, one need only apply the shortened log to the state.

However, we can be even more efficient. Consider for example a log-based computation of a set object. In Figure 1, we add to the log an attempt to *delete*(6) which deletes the number 6 from the set and returns true if successful and false if 6 was not found in the set. Figuring out the result of the delete operation requires searching for 6 in the state representing the set, and then searching the log for the possible effects of preceding operations on this state. In the example, the element 6 was not found in the set represented by the state, and in order to return the result one need search the log *only* for *insert*(6) and *delete*(6) operations and apply them to the state. There is no need to apply any other operations in the log. Log synchronized programming thus has two main components:

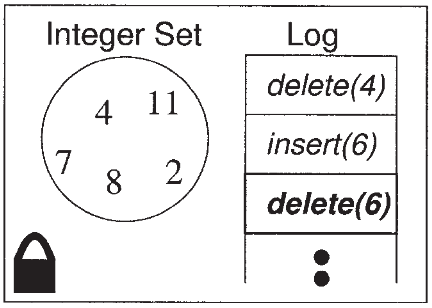

Figure 1: PLS Components of an Example Integer Set

State and Operation Design Design the state representation (the data structure) and the methods that modify it, all of which consist of standard sequential code (in our example this could be the set represented as a linked-list or a red-black tree, and the corresponding implementation of sequential code for insert, delete, and find on them. One could use the existing sequential code base for this part).

Efficient Prediction Design the log application pattern, that is, the way operations need to be applied from the log to the state in order to compute the outcome of a given operation. Programmers must specify what data is extracted from the state, and how operations in the log affect the result of another operation (in our example, a delete on a set requires applying the sequence of inserts and deletes only for the same value, independently of how the state and operations are implemented). This part of the programming is done once and for all for each data type.

There is a third *adjustments* component which we will discuss later, essentially providing hint pointers into the state data structure to speed-up modifications to it. The adjustments part requires an understanding of the internal structure of the sequential implementation of the data structure operations. The adjustments part is not really an unknown style of programming, rather, the adjustments are the same as one would create an efficient reconstruction sequence for a data structure as is done for backups.

Thus, all of the above log-based programming elements are free from any concurrency considerations. The concurrency is all hidden within the PLS implementation. Given a log-based description (program) for a given data-structure, we apply an automatic transformation that generates a concurrent PLS implementation that is a *sequentially consistent* [9] implementation of the data structure specified by its state and operations. This means that the result of any concurrent program execution is equivalent to that of some sequential ordering of all its operations in which each thread's operations are executed according to its own program order.

Though data structures for which there are no efficient prediction solutions may exist, looking through many commonly used structures, we could not find such structures. One should also remember that there are complex data structures, take Fibonacci heaps [2] as an example, for which even the transactional memory interface is non-trivial to use in order to compose an efficient implementation. We conducted an ad-hoc "programming experiment" [1] in which we timed the marginal programming effort of applying the PLS framework to a Fibonacci heap algorithm, assuming existing log-based code of a vanilla heap. Our experience was that the PLS based concurrent Fibonacci heap code required one hour of programming.

2.2 The PLS Algorithm

As we explained earlier, the log synchronized approach to introducing parallelism into a data structure is to allow threads to make progress by reading a consistent copy of the data and, if needed, using a log of high level operations to deduce the outcome of their operations. A modifying thread that does manage to get hold of the lock controlling the state, applies the operations in the log to the data structure representing the state, as a service to the other threads.

Let us return to our earlier example of a log-based implementation of a set of integers. Assume now that we wish to provide a PLS implementation of this algorithm, one in which concurrent threads perform inserts, deletes, and lookups. Each thread performing an insert or delete starts by appending its current operation to the shared log, and a single thread successfully acquires the lock. While this thread executes the set of operations in the log, other threads make progress. Recall the example of Figure 1: a thread deleting the number 6 from the set searches for 6 in the set, and analyzes the effect of preceding operations in the log on its result. Even if the element 6 was not found in the shared set, the thread searches the log for *insert*(6) and *delete*(6) operations appearing before its own. Only then it can decide on the result of the requested *delete*(6) operation.

Unfortunately, it cannot be assumed that during the modification of the data structure by the modifying thread, other threads would read consistent data. Our solution is to have two copies of the data structure, one used for writing and the other for reading. The two copies are switched at the end of every modifying session, a process that is coordinated via a global version count. When the version count is even, the first copy is used for reads and the second for writes, and when odd, they switch roles. That way, the "logical" application of all the changes of the last session is done by merely incrementing the version. We expand later in this section on how we handle version changes during reads.

There are two major implications of the need to maintain two data structures instead of one. One is the greater amount of space consumed, and the other is the cost of applying the changes to the other copy as well, a process we call *adjustment.* Note that for many data structures, one can optimize both space and computation to significantly less than double the cost of a single structure. Space consumption can be minimized by not duplicating data fields that are not accessed concurrently, and the cost of the second modification (the adjustment) can be minimized by using information recorded while modifying the first structure.

High level operations performed on shared data structures can be divided into three groups: *read-only*, *write-only*, and *read-modify-write.* Read-only operations do not mutate the data, write-only operations modify the data but do not return a result, and read-modify-write operations change shared data and return a result that is dependent on the data read. The three types imply different constraints on the structure's consistency, and are thus treated differently by the predicting threads. Threads performing write-only operations do not need to predict a result, they may complete by simply appending their operation to the operation log, to be executed at a later time by the modifying thread. This will still be sequentially consistent. We describe the implementation of read-only operations in the sequel.

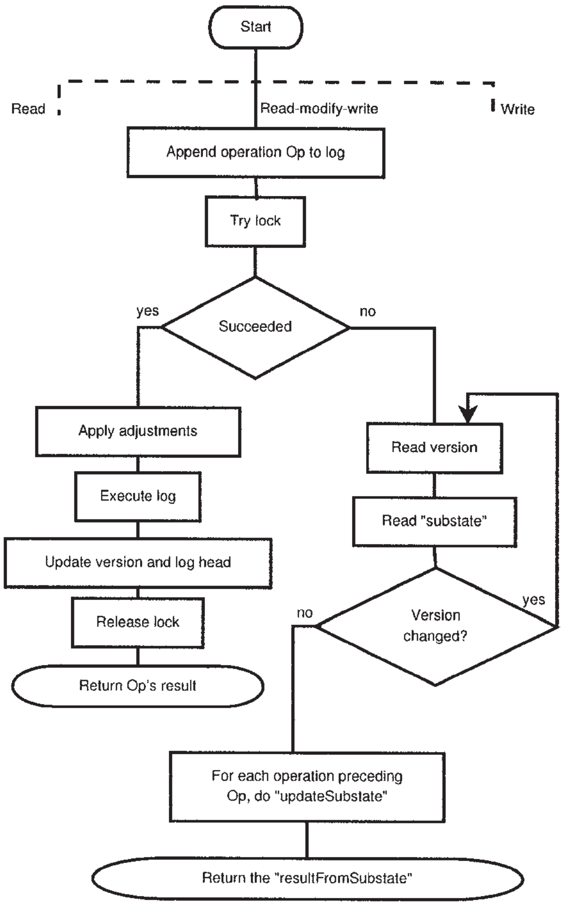

Figure 2: Flowchart of Read-Modify-Write PLS Execution

Consider now the implementation of read-modify-write operations as summarized in the flowchart in Figure 2. After appending the operation to the log, the modifying thread tries to acquire the object lock. Upon success, following the left-hand side of the chart, that *modifying* thread re-applies the modifications of the previous session (the adjustments) to the writable copy of the data structure. Then, it executes each one of the operations in the log, including its own, according to the order they were queued. Before releasing the lock, the version count is incremented and the log head is updated to point to an empty log or a log containing the very recently appended operations, if such exist. The version change immediately switches the roles of the two data structure copies, making the modifications appear to take

[1] Not in any way a benchmark.

place instantaneously.

The execution of threads that have not successfully acquired the lock, the *predicting* threads, is depicted on the right-hand side of the flowchart. This execution is dependent on complementary sequential code supplied by the programmer, consisting of three parts: the first, called makeSubstate is responsible for extracting the needed information from the current state of the data structure's readable copy. We refer to this information as the *substate*. The second, updateSubstate is for applying the effect of a preceding operation in the log on the substate. The third, resultFromSubstate is for extracting the operation's final result from the substate.

In the integer set example, when predicting the result of *delete*(6), makeSubstate returns the substate, a boolean denoting whether 6 is in the readable copy of the set structure. The function updateSubstate applies the effects of other operations on the substate: for *insert*(6) operations, it changes the substate to *true*, and for *delete*(6), the substate is changed to *false*. After inspecting the entire log, the predicting thread calls resultFromSubstate (which in this case is the identity function). The result of *delete*(6) should be *true* if and only if the element 6 was in the set.

Note that the prediction function's implementation is derived only from the shared data structure's semantics, and not from its implementation. For example, the prediction on a list-based set and on a tree-based set would use the same code. It is an interesting question whether the log synchronization framework fits every data structure ever invented, and the answer is probably not. We have found that many of the classic data structures in the literature (sets, maps, heaps, queues) can be supported.

Back to the right-hand side of Figure 2, the predicting threads call makeSubstate to extract the relevant information from the data structure. The substate has to be sampled carefully, as concurrent version changes can retrieve inconsistent data. Therefore, the makeSubstate function is called repeatedly until the version stabilizes. Then, for each operation in the log preceding their own, the predicting threads call updateSubstate on the substate object. Finally, they return the result obtained by a call to resultFromSubstate.

We now turn to read-only operations. Read-only operations do not need to queue themselves in the log, yet they must read from the readable copy of the data structure, that is, they need to call makeSubstate and resultFromSubstate. However, this is not enough to enforce sequential consistency. To provide sequential consistency a reading thread must consider the effect of all of its own modifying operations (write-only and/or read-modify-write) performed before the current read-only operation. We demonstrate this subtlety in Figure 3, illustrating the timelines of a modifying thread and a concurrent predicting thread. The right-hand side of the diagram depicts the current state of the operation log and the readable copy of the set. The predicting thread executes *delete*(6) and then *lookup*(6). Its delete operation (emphasized) is queued at the third slot in the log, following two other operations. Naturally, the *lookup*(6) call should return *false*, but since the readable copy of the set does not reflect the latest operations, the predicting thread would find 6 in the readable copy of the set and return *true*.

As the above example shows, in cases where sequential consistency is a requirement, a read-only thread must traverse the log up to its last queued modifying operation and predict by calling updateSubstate for each of these operations. Note that the log synchronization mechanism is designed in such way that read-only operations involve minimal overhead, as it is the premise of log-synchronization that these operations are the most common in many real-life applications.

2.3 Progress and Consistency Issues

Predictive log synchronization theoretically provides lock-free execution of read and write operations on the data structure. However, this lock-freedom property is somewhat weak since in some scenarios the number of steps a threads performs per operation increases unboundedly, although it is bounded for each individual operation. For example, when the modifying thread is delayed, the operation log length increases, while predicting operations are forced to traverse an increasingly long chains of operations.

The read-only operations' implementation as described above does not lead to linearizability, as reader threads scan the log only to the point of their latest modifying operation. In a scenario where a writer thread executes an operation entirely preceding the read operation, the effect of the earlier write may be ignored by the reader. In order to achieve linearizability, read-only operations must also be totally ordered and thus have to be logged as well. We avoid this at the cost of meeting weaker sequential consistency condition [9]. This is because one of our main goals is to provide low-overhead reads, and appending every read operation to the log and processing it would significantly degrade performance. Sequential consistency is what concurrent programmers and hardware designers usually think of when they specify something as atomic. However, the drawback of sequential consistency versus linearizability is that composed linearizable objects are linearizable by definition, while composed sequentially consistent objects are not. This implies that when composing sequentially consistent objects on must go through the process of proving that the new algorithm is sequentially consistent, a drawback for the software designer.

In some data structures, there may be an additional data consistency issue. A predicting thread working on an old version would eventually discover that fact, but it could be too late if the thread was reading from the currently writable copy. In this case, reading partially modified data can lead to memory access violations and infinite loops. The simple algorithmic solution is to have the lock owner thread perform prediction instead of mutation until all readers still reading the older version are no longer active. Implementing this solution without introducing an overhead is not straightforward because of the need to minimize the amount of inter-thread communication. To do so, our algorithm has reader threads share their status by attaching a locally updated record to a global linked list, which is traversed by the modifying thread when trying to initiate a new session. Aged elements are removed from the list so as to keep the list length proportional to the number of active threads.

2.4 The Programmer's Interface

To perform a log-based operation on the data structure, the programmer instantiates an object of a subclass of Operation, shown in Figure 4. The Operation derivatives contain a field for the result, a boolean field signaled by the modifying thread when its work is completed, and a next pointer to

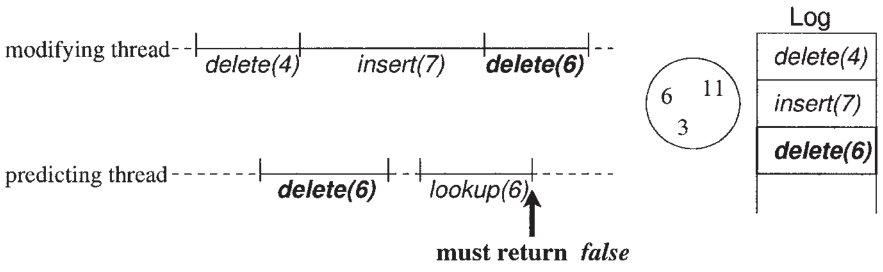

Figure 3: Enforcing Sequential Consistency

the next operation in the log. The programmer must implement four abstract methods, three of which were discussed in the previous subsection, and the fourth, executeOn is the traditional sequential implementation of that operation on the given data structure.

```
public abstract class Operation {
    Object result;
    boolean finished;
    AtomicReference<Operation> next;

    public abstract Object makeSubstate(
        Operation head, Object ds);
    public abstract void updateSubstate(
        Object substate);
    public abstract Object resultFromSubstate(
        Object substate);
    public abstract Object executeOn(
        Object ds);
}
```

Figure 4: The Operation Class

In Figure 5, we demonstrate how the integer set would be implemented in the framework. The example consists of the IntSetSubstate class representing the substate generated during the prediction process, and the InsertOperation class which is instantiated when performing inserts (we left out trivial details such as constructors). The executeOn method simply calls the sequential implementation of insert. The method makeSubstate instantiates a new substate containing the parameter to insert, determining whether it is present in the data structure by calling the sequential find. The role of updateSubstate is to apply the effect of an insert operation on a given substate: if the parameter associated with the substate equal to the insert operation's parameter, the isFound field is set to *true*. The resultFromSubstate method returns the negation of isFound, as insert operations succeed if and only if the element was *not* in the set before.

3. IMPLEMENTATION DETAILS

In this section we give some of the details corresponding to the system design presented in the previous section. To simplify the presentation, we do not include some of the mechanisms, keeping the focus on the ones at the core of the PLS framework. The syntax used to present the code is of the Java^{TM} programming language.

```
class IntSetSubstate {
    public int value;
    public boolean isFound;
}

class InsertOperation extends Operation {
    int parameter;
    public Object executeOn(IntSet ds) {
    this.result = ds.insert(parameter);
    return this.result; // a boolean
    }
    public Object makeSubstate(
        Operation head, IntSet ds) {
    return new IntSetSubstate(
            parameter, ds.find(parameter));
    }
    public void updateSubstate(
        IntSetSubstate substate) {
    if (parameter == substate.value)
        substate.isFound = true;
    }
    public Object resultFromSubstate(
        IntSetSubstate substate) {
    return !substate.isFound;
    }
}
```

Figure 5: Example: Log-based Integer Set Insert

When a predictive log-synchronized data structure is created, an identical secondary structure is allocated and initialized as well, and both copies point to a mutual object of class Log. The bulk of the mechanism is implemented as part of the Log class.

The Log class, presented in Figure 6, consists of the following fields: version – the global version count, structures – pointers to the two copies of the data structures, mutex – the lock, and headPointers – pointers to the head of the log (One used by the readers and one to be used when the version is incremented. The two head pointers are initialized to point at a dummy operation node). Additionally, we keep a list of adjustments. The list records the changes to the writable copy to be applied to the other copy at the beginning of the next session.

After instantiating the operation, the programmer calls one of the methods readModifyWrite, read, and write, based on the type of operation at hand. The methods write and

```
public class Log {
    int version;
    Object structures [2];
    ReentrantLock mutex = new ReentrantLock();
    Operation headPointers[2];
    ArrayList<Adjustment> adjustmentList;
    ...
    public Object readModifyWrite(Operation op) {
        Object result;
        appendToLog(op);
        if (tryLock()) {
            result = mutate(op);
            release ();
            return result;
        }
        return predict(op, false);
    }

    public Object read(Operation op) {
        return predict(op, true);
    }

    public void write(Operation op) {
        appendToLog(op);
        if (tryLock()) {
            mutate(op);
            version += 1;
            release ();
        }
    }
    ...
}
```

Figure 6: The Log Class

readModifyWrite append the operation to the log[2] and try to acquire the mutex in order to execute on the writable copy. If the mutex is taken, write simply returns where readModifyWrite calls predict. The read method simply calls predict.

The code for mutate, depicted in Figure 7, first adjusts the writable copy based on the latest changes applied to the other copy. Then, it traverses the log and executes each of the logged operations on the writable copy. Finally, it sets the new head pointer to point to the position in which it stopped traversing the log.

In Figure 8 we present the code for the predict method, which is less trivial. When executing predict, the thread must complete reading the substate from the readable copy before the session ends and the version is incremented. If it fails to do so and the operation has not been completed by the modifying thread (for read-modify-write operations) it must retry. Upon success, in lines 11-16, the thread decides how far back in the log it has to traverse. When the operation is a read-modify-write one, the log is traversed up to the position where the operation was enqueued. For read operations, if there is a pending modifying operation requested by this thread (local is a thread-local object), the prediction should proceed up to that operation. If the last operation has completed, or in the case where sequential consistency is not a requirement, there is no need to inspect the log at all.

```
private Object mutate(Operation op) {
    // ds is assigned to the writable copy
    Object ds = structures[1 - (version % 2)];
    for (adj : adjustmentList)
        adj.adjust(ds);
    adjustmentList.clear ();
    Operation prev = headPointers[version % 2];
    Operation e = prev.next.get();
    while (e != null) {
        e.executeOn(ds);
        e.finished = true;
        prev = e;
        e = e.next.get ();
    }
    headPointers[1 - (version % 2)] = prev;
    return op.result;
}
```

Figure 7: Code for mutate

```
   private Object predict(Operation op,
                                     boolean isRead) {
       do {
           oldver = this.version;
 5         savedHead =
               headPointers[oldver % 2].next.get ();
           savedLastOpFinished =
               local .lastModifyingOp.finished;
           substate = op.makeSubstate(
10                      savedHead, structures[oldver % 2]);
            if (op.finished)
               return op.result;
       } while (oldver != version);
       if (isRead)
15         upto = (savedLastOpFinished
                   || noSequentialConsistency)
                  ? savedHead
                  : local .lastModifyingOp;
       else
20         upto = op;
       for (Operation e = savedHead; e != upto;
                                   e = e.next.get ())
           e.updateSubstate(substate);
       return op.resultFromSubstate(substate);
25 }
```

Figure 8: Code for predict

3.1 A Log-based Red-Black Tree Integer Set

An integer set implemented as a sequential red-black tree can be made concurrent using the PLS framework by creating appropriate InsertOperation, DeleteOperation, and LookupOperation classes. The InsertOperation class is shown in Figure 5, the other two are as simple. The executeOn method of these three classes would consist of sequential red-black tree code. At the beginning of every session, the writable copy of the data structure has to be updated with the same

[2] Appending to the log is done using the atomic primitive *Compare&Swap* on the last element in the log

modifications made to the other copy during the previous session. This list of modifications, the *adjustments*, is easily constructed during their original execution and used by the modifying thread of the new session. Hence, when applying PLS on an existing sequential red-black tree, the only non-trivial programming task is the implementation of the operation classes.

In the red-black tree case, one can optimize the adjusting process by saving information from the first execution, such as pointers to tree nodes associated with the operation. All of the three set operations begin with a tree traversal to destination nodes that can be recorded. Each node in the tree can contain a pointer to its "twin" node in the other copy, which can be used as a shortcut when applying the adjustment instead of repeating the entire tree traversal.

4. PERFORMANCE

We present here a set of proof-of-concept microbenchmarks intended to show that there is merit to the predictive log-synchronized approach, but are in no way intended to be a full and comprehensive exposition of PLS performance. We chose a single specific data structure: a red-black tree data structure representing a set (used in benchmarking various transactional memory systems [4, 11, 6, 13]) and wrote it in a predictive log-synchronized manner.

We tested the red-black tree data structure in various configurations considered to be common application usage patterns. One of the parameters was the tree size, derived from the range from which keys are selected at random. In the "large" tree, we used a range of $0..10^6$ and initialized the tree by executing 10^5 insertions, while in the "small" tree we used keys taken from the range 0..200, initialized by executing 100 insertions. After initialization, a varying number of threads execute randomly selected operations among *insert*, *delete*, and *lookup* of a random integer within the range. Each test was repeated five times; the data presented in this section is the average of those runs. For our experiments, we used a 16-processor SunFire™ 6800, which is a cache coherent multiprocessor with 1.2GHz UltraSPARC® III processors, running the Solaris™ 9 operating system.

In our first set of microbenchmarks we compared the performance of three techniques that provide simple programming of concurrent red-black trees.

Locks Coarse grained locking via the Java *synchronized* monitor.

OSTM A Java based implementation by Marathe et al. [11, 10] of the OSTM by Harris and Fraser [1], which is a representative state-of-the-art software transactional memory systems. Unlike the C-based implementation in [1], this implementation does not use a specially tailored closed memory system, rather, it uses Java's built-in GC. In the red-black tree code, it uses a different mechanism than the early release used by Fraser [1] to eliminate various tree hotspots. A comparison of OSTM to other software transactional memory systems can be found in [10].

PLS A Java based implementation using PLS.

In addition, in order to analyze the cost of prediction, we benchmarked a variation of the PLS algorithm in which threads only read the substate without predicting (the resulted implementation is not sequentially consistent.) We did not use a contention manager in the either the OSTM or PLS implementation. A contention manager is a heuristic software mechanism [17] that decides when to back-off in the face of access contention on the data structure.

Graphs (a) and (b) of Figure 9 illustrate the throughput in accessing a large tree. Graph (a) shows the results of an experiment in which operations were evenly distributed between inserts and deletes (no lookups), on a large red-black tree. When all of the executed operations modify the data structure, PLS has higher throughput than OSTM on low concurrency levels due to lower overhead, but from 16 threads and up, the limited parallelism of PLS degrades its performance. On the large tree, where keys are often located as deep as 16 levels below the root, the OSTM algorithm involves relatively large transactions and thus suffers from significant overhead, most likely due to memory management. In Graph (b), where the execution is dominated by lookup operations, PLS performs significantly better due to the fact that non-modifying operations involve very low overhead.

Graphs (c) and (d) depict the experimental results on a small set containing 100 elements on average. In Graph (c), we see both PLS and OSTM utilize machine parallelism, where OSTM does it successfully due to small transaction size, and PLS due to a significant portion of insert (or delete) operations that do not modify the set as their parameter exists (or is absent, respectively) in the tree. In the experiment whose results are shown in Graph (d), although the OSTM algorithm outperformed the lock-based one, PLS was faster by a factor of 2 to 3.

We note that our results for the OSTM implementation agree with those of [10] but are dramatically worse than those of the C-based implementation of Harris and Fraser in [1]. There are probably various factors responsible for the difference. Harris and Fraser used a different mechanism than [10] to perform early release of hotspots like the tree's sentinel nodes. They used their own specialized closed memory allocation mechanism, and their C-based implementation naturally has a lower level of indirection than Java code.

As another example of data structure design using PLS, we conducted experiments on a red-black tree supporting fixed size range queries[3]. We implemented range queries in PLS by maintaining the substate as a subset of elements which we update during the prediction process. The graphs in Figure 10 illustrate the the results of experiments where 90% of the operations were range queries, 8% inserts, and 2% deletes. On the small tree, we picked a range size of 10 (Graph 10(a)) and 50 (10(b)), and on the large tree we used 100 (10(c)) and 1000 (10(d)). The performance of both PLS and OSTM degraded with respect to the lock-based algorithm when the range size was increased, but for different reasons. In PLS, operations always succeed, but a source of extra overhead is the substate maintenance, while in OSTM the increased transaction size increases the chances of a transaction encountering a modification within its range, which causes more transaction failures.

The preliminary benchmarks for PLS performance on red-black trees suggests PLS as a viable approach as a method for utilizing concurrency. However, one must remember that red-black trees are the classical example of a data-structure

[3] A range query is an operation that extracts a subset of keys within a given key range

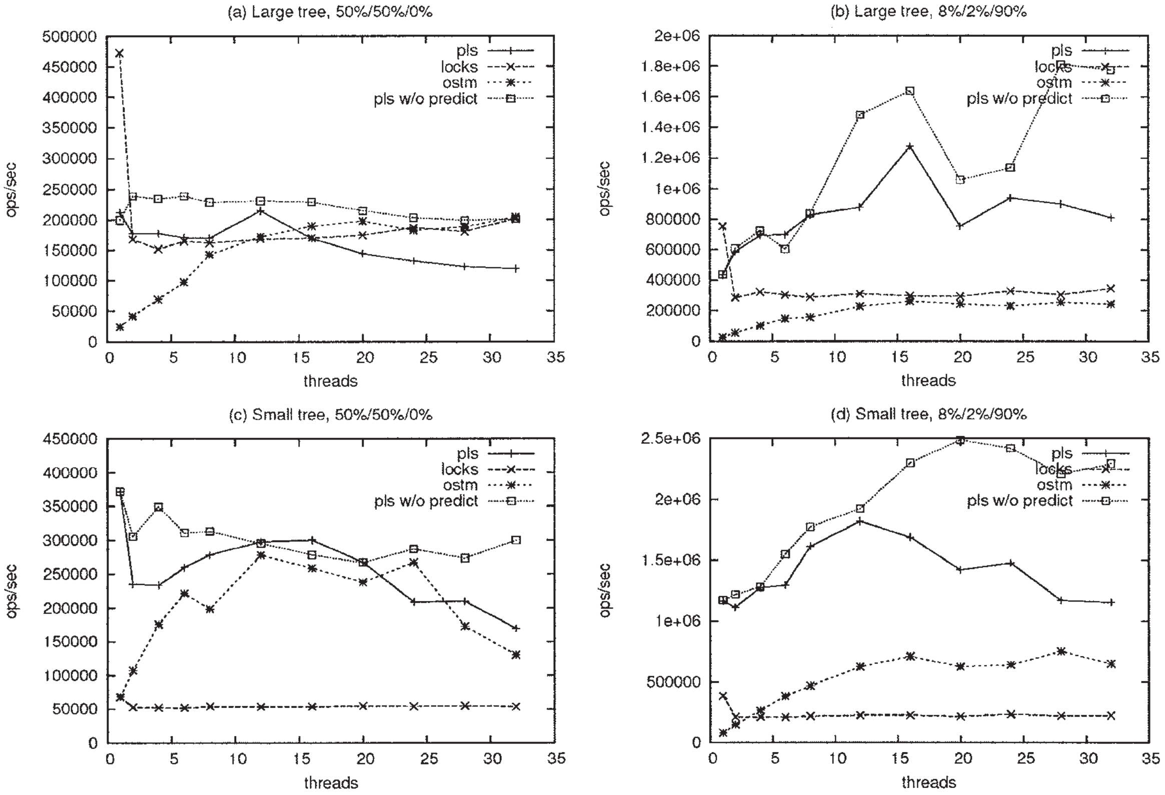

Figure 9: Throughput of Red-Black Trees Implemented Using PLS, OSTM, and Java Monitor Locks

that benefits from the predictive approach since it is inexpensive to compute the results of operations from the log. For example, we implemented a *heap* data structure on the PLS framework, supporting *insert*, *deleteMin*, and *getMin*. The prediction process of *deleteMin* and *getMin* operations required maintaining the set of keys inserted and removed during the current session. The amount of computation overhead in that case was unacceptable and the overall throughput measured was significantly lower than the one of a simple coarse-grained locking approach. A heap data structure is an example of a data structure for which the PLS technique is a bad fit, mostly because the effect of *deleteMin* is difficult to determine by inspecting its arguments. One cannot predict the results of operations succeeding a *deleteMin* without performing some sort of non-trivial emulation.

Clearly more work is needed to adapt contention management mechanisms, and further evaluation is necessary to understand the benefits and drawbacks of PLS for the design of various data structure classes.

5. CONCLUSION

In this paper we introduced predictive log-synchronization, a new concurrent programming paradigm that may be viewed as conceptually contradicting the accepted approaches to parallelization such as fine grained locking and transactional memory. These approaches aim to optimize parallelism when accessing sets of disjoint locations in memory, while PLS optimizes parallelism in reading from memory, at the cost of not providing parallelism among modifications to disjoint locations. Moreover, PLS is based on using a single coarse-grained mutual exclusion lock, a construct traditionally held accountable for limited scalability.

The proposed solution is not claimed to be asymptotically scalable: the throughput of modifying operations is technically bounded by the throughput of a single processor. However, one of our principal conjectures is that the distribution of high level operations in real-world applications is read-dominated. Unlike other paradigms, the PLS framework does not require parallel reasoning abilities from the programmer, but does oblige the introduction of supplementary code for the semantic specification of each type of data structure.

We believe that the PLS framework, presented here in its initial form, has the potential of being an interesting alternative to lock monitors and software transactional memory mechanisms in a variety of circumstances.

6. ACKNOWLEDGEMENTS

We thank Virendra Marathe, Bill Scherer, and Michael Scott for kindly providing us with the OSTM source code.

7. REFERENCES

[1] FRASER, K. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

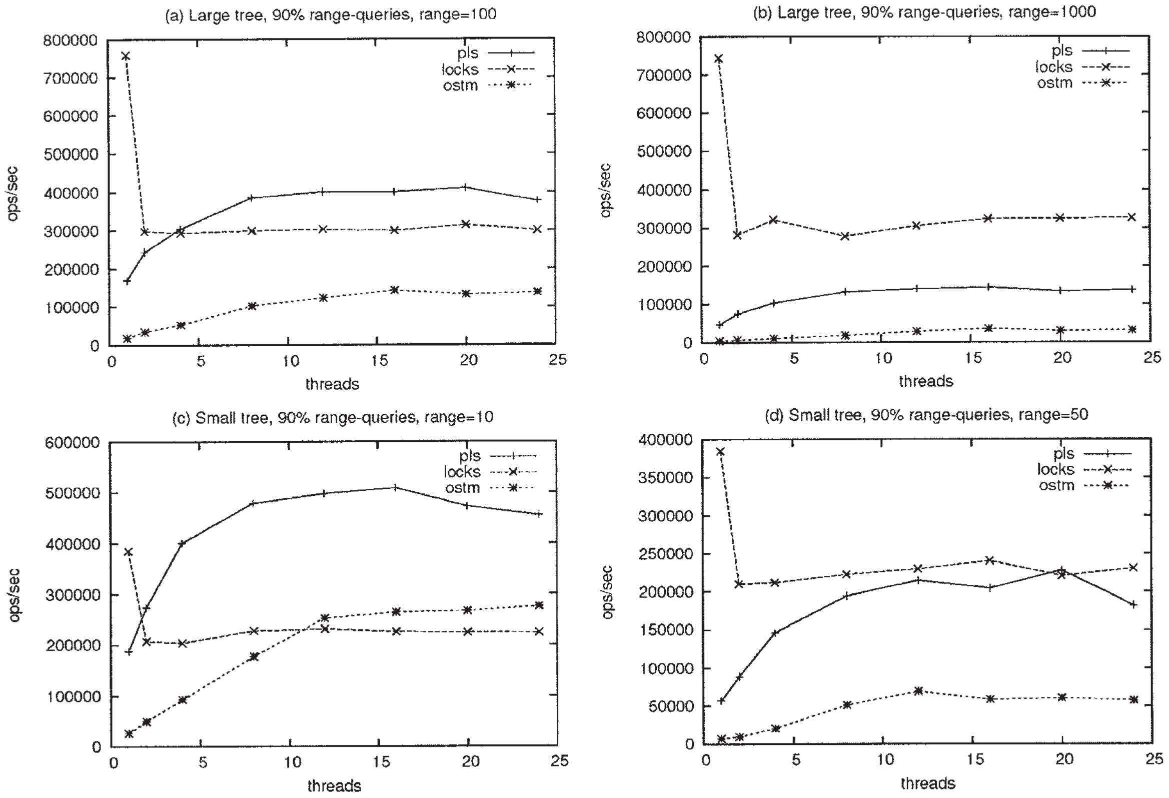

Figure 10: Red-Black Trees, 90% Range Queries, 8% Inserts, 2% Deletes

[2] Fredman, M. L., and Tarjan, R. E. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM 34*, 3 (1987), 596–615.

[3] Harris, T. Exceptions and side-effects in atomic blocks. In *Proceedings of the 2004 Workshop on Concurrency andSynchronization in Java programs* (Jul 2004), pp. 46–53. Proceedings published as Memorial University ofNewfoundland CS Technical Report 2004-01.

[4] Harris, T., Fraser, K., and Pratt, I. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on DIStributed Computing* (2002), pp. 265–279.

[5] Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. Software transactional memory for dynamic data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing* (2003).

[6] Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. N. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing* (New York, NY, USA, 2003), ACM Press, pp. 92–101.

[7] Herlihy, M., and Moss, J. E. B. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture* (1993), ACM Press, pp. 289–300.

[8] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (1990), 463–492.

[9] Lamport, L. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput. 46*, 7 (1997), 779–782.

[10] Marathe, V. J., Scherer, W. N., and Scott, M. L. Design tradeoffs in modern software transactional memory systems. In *In Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR'04)* (2004).

[11] Marathe, V. J., Scherer, W. N., and Scott, M. L. Adaptive software transactional memory. In *To Appear in the Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)* (2005).

[12] McKenney, P. E., and Slingwine, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, October 1998), pp. 509–518.

[13] Moir, M. HybridTM: Integrating hardware and software transactional memory. Tech. Rep. Archivist 2004-0661, Sun Microsystems Research, August 2004.

[14] Oyama, Y., Taura, K., and Yonezawa, A. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA)* (July

1999), pp. 182–204.

[15] Riany, Y., Shavit, N., and Touitou, D. Towards a practical snapshot algorithm. *Theoretical Computer Science 269*, 1-2 (2001), 163–201.

[16] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (1992), 26–52.

[17] Scherer, W. N., and Scott, M. L. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing* (New York, NY, USA, 2005), ACM Press, pp. 240–248.

[18] Shalev, O., and Shavit, N. The log-synchronizer: an alternative to monitors and software transactional memory. Unpublished manuscript, September 2005.

[19] Shavit, N., and Touitou, D. Software transactional memory. *Distributed Computing 10*, 2 (February 1997), 99–116.

[20] Welc, A., Hosking, A. L., and Jagannathan, S. Preemption-based avoidance of priority inversion for Java. In *Proceedings of the International Conference on Parallel Processing* (2004), IEEE Computer Society, pp. 529–538.

[21] Welc, A., Jagannathan, S., and Hosking, A. L. Transactional monitors for concurrent objects. In *Proceedings of the European Conference on Object-Oriented Programming* (2004), vol. 3086 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 519–542.