ABSTRACT
This paper proposes an efficient object-based implementation of non-blocking software transactions. We use ideas from software distributed shared memory to efficiently implement transactions with little overhead for non-transactional code. Rather than emulating a flat transactional memory, our scheme is object-based, which allows compiler optimizations to provide better performance for long-running transactions. We present empirical data on transaction properties to support the design. A model for the software transaction implementation is given in Promela, whose correctness has been mechanically verified using the SPIN model checker. The design presented cooperates well with an HTM providing support for small short transactions.

Categories and Subject Descriptors
C.0 [Computer Systems Organization]: General—Hardware/software interfaces; D.2.4 [Software Engineering]: Software/Program Verification—Model checking; D.3.2 [Programming Languages]: Language Classifications—Object-oriented languages, Java; D.3.3 [Programming Languages]: Language Constructs and Features—Concurrent programming structures; D.3.4 [Programming Languages]: Processors—Compilers; E.2 [Data Storage Representations]: Object representation

General Terms
Algorithms, Verification, Languages, Experimentation

Keywords
Object-based transactions, synchronization, Java, Promela, SPIN

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

1. INTRODUCTION
The transaction model is a natural means to express atomicity, fault-tolerance, exception-handling, and back-tracking. Although transactions can be implemented using mutual exclusion (locks), the algorithms presented utilize non-blocking synchronization to exploit optimistic concurrency among transactions. Non-blocking synchronization offers a number of advantages; among them fault-tolerance: processes that fail in critical regions cannot prevent other processes from making progress.

In this paper we discuss the benefits and disadvantages of an object-based transaction system. We present a mechanically-checked non-blocking object-based implementation, and experimental measurements of transactional code justifying its design. We provide one solution to the “large object” problem, and describe a hybrid hardware/software transaction system.

2. USING OBJECTS FOR PERFORMANCE
Recent trends in lightweight transaction system design have used a transactional memory abstraction. The performance improvements and other advantages demonstrated by Hardware Transactional Memory (HTMs) [19, 16, 13, 8, 3], using word- or cache-line–based abstractions to match the standard cache/memory system, have inspired similar Software Transactional Memory (STM) systems [25, 9]. We suggest that the exclusive use of flat memory abstractions in designing a modern lightweight transaction system is a mistake.

There is no guarantee that the transactional data in an application will be matched to word, cache line, or page boundaries, leading to false sharing. This false sharing may have semantic implications (may create new deadlocks) when transactions that ought to be able to commit in parallel are instead made exclusive because some piece of data is colocated. To some degree authors of parallel code are already aware of these issues: locks are placed, for example, to minimize cache ping-pong on SMP systems. But this is an unnecessary burden.

Moreover, the flat memory model complicates basic and important optimizations. An alternative, “clone and mu-
tate” or an object-based transaction system, performs transactional operations on copies of the objects involved which are then atomically substituted for the originals at commit time. Compile- or JIT-time analysis can track objects that have already been cloned, and henceforth operate on them without any synchronization overhead. The details can be subtle, but a compiler can emit transactional code which is, after initial overhead, equal in performance with unsynchronized/non-transactional code. In Section 3 we will show that common applications may contain large and long-running transactions for which this behavior is important.

A number of object-based transaction systems have been proposed, starting with Herlihy’s “small object protocol” [11]. Shavit and Touitou’s STM [25] required that that all input and output locations touched by a transaction be known in advance. Herlihy, Luchangco, Moir, and Scherer’s scheme [12] allows transactions to touch a dynamic set of memory locations; however the user still has to explicitly “open” every object touched before it can be used in a transaction. Further the cost of opening R objects for reading and W objects for writing is \(O(R(R+W))\); that is, potentially quadratic in the number of objects involved. The transaction system we will present in Section 5 does not require foreknowledge of the behavior of a transaction, and does not blow up on transactions touching a large number of objects.

Large objects and arrays typically present problems for object-based transaction systems. Herlihy presented a “large object protocol” [11] which required the programmer to manually break up large objects; we present our own solution to this problem in Section 5.3.

In the following section we will examine the properties of transactional code that motivate our design.

### 3. PROPERTIES OF TRANSACTIONS

One of the difficulties of evaluating transaction implementations is the lack of benchmarks. Although there is no body of code yet using atomic regions, there is a substantial body of code that uses Java (locking) synchronization. We have implemented a compiler that substitutes atomic blocks/methods for synchronized blocks/methods in order to evaluate the properties Java transactions are likely to have. Note that the semantics are not precisely compatible [5]: the existing Java memory model allows unsynchronized updates to shared fields to be observed within a synchronized block, while such updates will never be visible to an atomic block. The proposed revision of the Java memory model [21] narrows the semantic gap, however we do not treat volatile fields in this work. In addition, updates from

<table>
<thead>
<tr>
<th>Program</th>
<th>Memory Ops</th>
<th>Transactional Memory Ops</th>
<th>Biggest Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>201_compress</td>
<td>2,981,777,890</td>
<td>2,272</td>
<td>&lt;0.1%</td>
</tr>
<tr>
<td>202_jess</td>
<td>405,153,255</td>
<td>4,892,829</td>
<td>9.1%</td>
</tr>
<tr>
<td>205_raytrace</td>
<td>420,005,763</td>
<td>4,177</td>
<td>1.7%</td>
</tr>
<tr>
<td>209_db</td>
<td>848,082,597</td>
<td>45,222,742</td>
<td>23.0%</td>
</tr>
<tr>
<td>213.javac</td>
<td>472,416,129</td>
<td>668</td>
<td>99.9%</td>
</tr>
<tr>
<td>222_mpegaudio</td>
<td>2,620,816,169</td>
<td>2,991</td>
<td>&lt;0.1%</td>
</tr>
<tr>
<td>228_jack</td>
<td>187,029,744</td>
<td>12,017,041</td>
<td>34.2%</td>
</tr>
</tbody>
</table>

Figure 1: Transactification of SPECjvm98 benchmark suite: resulting transaction counts and sizes, compared to total number of memory operations (loads and stores). These are full input size runs.

### 3.5. Classification of SPECjvm98 Benchmarks into Quadrants Based on Transaction Properties.

The SPECjvm98 benchmark suite represents a variety of typical Java applications which use the capabilities of the Java standard library. Although the SPECjvm98 benchmarks are largely single-threaded, since they use thread-safe Java standard libraries they contain synchronized code which is transformed into transactions. Because in this evaluation we are looking at transaction properties only, the multithreaded 227_srtt benchmark is identical to its serialization, 205_raytrace. For consistency, we present only the latter.

Figure 1 (previously presented in [3]) shows the raw sizes and frequency of transactions in the transactified SPECjvm98 suite. Figure 2 proposes a taxonomy for Java applications with transactions, grouping the SPECjvm98 applications into quadrants based on the number and size of the transactions that they perform. Any scheme that allows the programmer free choice of de-
sired transaction and/or atomicity properties will inevitably result in some applications in each of these categories. Existing hardware transactional memory schemes only efficiently handle relatively short-lived and small transactions (Quad I or II), although for these they are very efficient. However, large and long-lived transactions can be readily seen in Figure 3, which plots the distribution of transaction sizes in SPECjvm98 on a semi-log scale. Object-based transaction systems pay copying overhead up front, making them costly for small transactions, but have low subsequent per-operation costs, which make them the best choice for long-lived transactions.

An ideal system would combine an HTM and an object-based transaction system to obtain the strengths of both; we will briefly describe such a system in Section 5.4.

4. DESIGNING EFFICIENT TRANSACTIONS

In this section we briefly describe some desired properties of our object-based software transaction system.

4.1 Field Flags

We would like non-transactional code to execute with minimal overhead, however, transactions should still appear atomic to non-transactional code. Our basic mechanism is loosely based on the distributed shared memory implementation of Scales and Gharaorchoo [24]. We will pick a special “flag” value, and “cross-out” locations currently involved in a transaction by overwriting them with the flag value. Reading or attempting to overwrite a flagged value will indicate to non-transactional code that exceptional processing is necessary; all other non-transactional operations proceed as usual.

This read barrier is the only overhead required for non-transactional code. As the test is simple and predictable, the runtime cost can be very low: the compare-and-branch can be scheduled in otherwise unused instruction slots on a multiple-issue processor [18].

Note that our technique explicitly allows safe access to fields involved in a transaction from non-transactional code.

4.2 Object expansion

We will need to add some additional information to each object to track transaction state. We measured the slowdown caused by various amounts of object “bloat” to determine reasonable bounds on the size of this extra information. Figure 4 presents these results for the SPECjvm98 applications; we determined that two words (eight bytes) of additional storage per object would not impact performance unreasonably. This amount of bloat causes a geometric mean of 2% (and no more than 10%) slowdown on these benchmarks, including costs caused by increased heap size and more-frequent garbage collection. This is consistent with the results reported by Bacon, Fink, and Grove, who found that reducing object header size from two words to one gave speedups ranging from -1.5% to 2.2% [4].

4.3 Reads vs. Writes

Figure 5 shows that transactional reads typically outnum-
ber transactional writes by 3 to 1; in some cases reads outnumber writes by over 100 to 1. The read/write ratios in transactions do not depart much from observed data-cache read/write ratios \([10, pp. 105, 379]\). It is worthwhile, therefore, to make reads more efficient than writes. In particular, since the flag-overwrite technique discussed in Section 4.1 requires us to allocate additional memory to store the “real” value of the field, we wish to avoid this process for transactional reads, reserving the extra allocation effort for transactional writes.

5. SOFTWARE TRANSACTION MECHANISM

We now present an algorithm that has these desired properties. Our algorithms will be completely non-blocking, which allows good scaling and proper fault-tolerant behavior: one faulty or slow processor cannot hold up the remaining good processors.

We will implement the synchronization required by our algorithm using Load-Linked/Store-Conditional instructions. We require a particular variant of these instructions that allows the location of the Load-Linked to be different from the target of the Store-Conditional: this variant is supported on the PowerPC processor family, although it has been deprecated in the newest chips. This disjoint location capability is essential to allow us to keep a finger on one location while modifying another: a poor man’s “Double Compare And Swap” instruction. Load-Linked/Store-Conditional also provides a convenient solution to the so-called “ABA” problem of Compare-And-Swap.\(^3\)

We will describe our algorithms in the Promela modeling language \([17]\), which we used to allow mechanical model checking of the race-safety and correctness of the design. Portions of the model have been abbreviated for this presentation; the full Promela model is available from a URL given at the end of the paper. Appendix A provides a brief primer on Promela syntax and semantics.

5.1 Object Structures

Figure 6 illustrates the basic data structures of our software transaction implementation. Objects are extended with two additional fields. The first field, versions, points to a singly-linked list of object versions. Each one contains field values corresponding to a committed, aborted, or in-progress transaction, identified by its owner field. There is a single unique transaction object for each transaction.

The other added field, readers, points to a singly-linked list of transactions that have read from this object. Committed and aborted transactions are pruned from this list. The readers field is used to ensure that a transaction does not operate with out-of-date values if the object is later written non-transactionally.

There is a special flag value, here denoted by FLAG. It should be an uncommon value, i.e., not a small positive or negative integer constant, nor zero. In our implementation, we have chosen the byte 0xCA to be our flag value, repeated as necessary\(^4\) to fill out the width of the appropriate type. The semantic value of an object field is the value in the

\(^3\)The copy-back procedure will notify the caller if this is a

\(^4\)Store-Conditional will fail if the target of the Load-Linked is written to, even if the value written is identical to value previously in the location.

original object structure, unless that value is FLAG, in which case the field’s value is the value of the field in the first committed transaction in the object’s version list. A “false flag” occurs when the application wishes to “really” store the value FLAG in a field; this is handled by creating a fully-committed version attached to the object and storing FLAG in that version as well as in the object field.

5.2 Operations

We support transactional read/write and non-transactional read/write as well as transaction begin, transaction abort, and transaction commit. Transaction begin simply involves the creation of a new transaction identifier object. Transaction commit and abort are simply compare-and-swap operations that atomically set the transaction object’s status field appropriately if and only if it was previously in the WAITING state. The simplicity of commit and abort are appealing: our algorithm requires no complicated processing, delay, roll-back or validate procedure to commit or abort a transaction.

We will present the other operations one-by-one.

5.2.1 Read

The ReadNT function does a non-transactional read of field \(f\) from object \(o\), putting the result in \(v\). In the common case, the only overhead is to check that the read value is not FLAG. However, if the value read is FLAG, the thread must copy back the field value from the most-recently committed transaction (aborting all other transactions) and try again. The copy-back procedure will notify the caller if this is a
The thread must call the copy-back procedure (as in readNT passing the constant kill
be aborted, not transactional readers. All possible races are
confined to the copy-back procedure.

```c
inline readNT(o, f, v) {
    do
        v = object[o].field[f];
        if
            (v!=FLAG) -> break /* done! */
        else
            fi;
        copyBackField(o, f, kill_writers, _st);
        if
            (_st==false_flag) ->
            v = FLAG;
            break
        else
            fi
    od
}
```

### 5.2.2 Write

The writeNT function does a non-transactional write of new value nval to field f of object o. For correctness, the thread needs to ensure that the reader list is empty before it does the write. We implement this with a Load-Linked/Store-Conditional pair (modelled slightly differently in Promela), ensuring that the write only takes effect as long as the reader list remains empty. If it is not empty, the thread must call the copy-back procedure (as in readNT), passing the constant kill_all to indicate that both transactional readers and writers should be aborted during the copy-back. The copy-back procedure leaves the reader list empty.

If the value to be written is actually the FLAG value, things get a little bit trickier. This case does not occur often, and so the simplest correct implementation is to treat this non-transactional write as a short transactional write, creating a new transaction for this one write, and attempting to commit it immediately after the write. This is slow, but adequate for this uncommon case.

```c
inline writeNT(o, f, nval) {
    if
        (nval != FLAG) ->
        do
            atomic {
                if /* this is a LL(readerlist)/SC(field) */
                    (object[o].readerlist == NIL) ->
                        object[o].fieldLock[f] = _thread_id;
                        object[o].field[f] = nval;
                        break /* success! */
                    else /* try, try, again */
                        moveVersion(_r, NIL);
                    fi
                /* unsuccessful SC */
            } /* use val from committed vers.*/
        copyBackField(o, f, kill_all, _st)
        od
        else -> /* create false flag */
            /* implement this as a short *transactional* write. */
            /* start a new transaction, write FLAG, commit the */
            /* transaction; repeat until successful. */
            /* Implementation elided. */
            ...
            fi;
    }
```

### Notes

5. Note that a standard CAS would not suffice, as the Load-Linked targets a different location than the Store-Conditional.

6. Note again that a CAS does not suffice.

### 5.2.3 Field Copy-Back

Figure 7 presents the field copy-back routine. The thread creates a pseudo-version owned by a pre-aborted transaction which serves as a reservation on the head of the version list. It then writes to the object field (with a Load-Linked/Store-Conditional pair) if and only if its pseudo-version remains at the head of the versions list. We want to avoid being interrupted while writing back some value A, and having a concurrent transaction finish writing back A and then commit and write back some other value B, before we resume and clobber B with our (interrupted) write-back of A. The use of the LL/SC on the head of the version list prevents this possible race.

### 5.2.4 Transactional Read

A transactional read is split into two parts. Before the read, the thread must ensure that its transaction is on the reader list for the object. This is straightforward to do in a non-blocking manner as long as all inserts are to the head of the list. The thread must also walk the versions list and abort any uncommitted transaction other than its own. With sufficiently precise pointer analysis these steps can be combined and hoisted so that they are done once before the first read from an object and not repeated. Imprecision in the analysis may cause re-examination of the reader list.

At read time, the thread initially reads from the original object. If the value read is not FLAG, the read value is used. Otherwise, the thread uses `findVersion` to look up the version object `ver` associated with its transaction (this will typically be at the head of the version list) and then reads the appropriate value from that version. Note that the initial read-and-check can be omitted if it knows that it has already written to this field inside this transaction, and that once `ver` has been looked up, it can be used in subsequent calls to `readT` (it is both an input and an output argument to the function). If there is not an uncommitted version for the object, `findVersion` returns the committed version in `r`.

```c
inline readT(tid, o, f, ver, result) {
    do
        /* we should always either be on the readerlist or */
        /* aborted here */
        result = object[o].field[f];
        if
            (result==FLAG) ->
                if
                    (ver!=NIL) ->
                        result = version[ver].field[f];
                        break /* done! */
                    else
decided
                        if
                            (ver==NIL) ->/*use val from committed vers.*/
                                assert (_r!=NIL);
                                result = version[_r].field[f];/*false flag*/
                                break /* done */
                            else /* try, try, again */
                                fi
                        else /* try */
                            fi
                    fi
                else
decided
                    fi
        od
    }
```
```c
inline copyBackField(o, f, mode, st) {
    _nonceV = NIL; _ver = NIL; _r = NIL; st = success;
    /* try to abort each version. When abort fails, we've got a */
    /* committed version. */
    do
    : _ver = object[o].version;
    if
    : (_ver == NIL) ->
        st = saw_race; break /* someone's done the copyback for us */
    : else
        fi;
    /* move owner to local var to avoid races (owner set to NIL behind */
    /* our back) */
    _tmp_tid = version[_ver].owner;
    tryToAbort(_tmp_tid);
    if
    : (_tmp_tid == NIL || transid[_tmp_tid].status == committed) ->
        break /* found a committed version */
    : else
        fi;
    /* link out an aborted version */
    assert(transid[_tmp_tid].status == aborted);
    CAS_Version(object[o].version, _ver, version[_ver].next, _);
    od;
    /* okay, link in our nonce. This will prevent others from doing the */
    /* copyback. */
    if
    : (st == success) ->
        assert(_ver != NIL);
        allocVersion(_retval, _nonceV, aborted_tid, _ver);
        CAS_Version(object[o].version, _ver, _nonceV, _cas_stat);
        if
        : (!_cas_stat) ->
            st = saw_race_cleanup /* need to clean up nonce */
        : else
            fi;
        else /* !success */
            fi;
    fi;
    /* check that no one's beaten us to the copy back */
    if
    : (st == success) ->
        if
        : (object[o].field[f] == FLAG) ->
            _val = version[_ver].field[f];
            if
            : (_val == FLAG) -> /* false flag... */
                st = saw_race_cleanup /* ...no copy back needed */
            : else /* not a false flag */
                d_step { /* LL/SC */
                if
                : (object[o].version == _nonceV) ->
                    object[o].fieldLock[f] = _thread_id;
                    object[o].field[f] = _val; /* back through race cleanup */
                : else /* no race. Must retry. */
                    st = saw_race_cleanup /* need to clean up nonce */
                fi;
            fi;
        : else /* !success */
            fi;
        else /* always kill readers, whether successful or not. This ensures that we */
        /* make progress if called from writeNT after a readNT sets readerList */
        /* non-null without changing FLAG to _val (see immediately above; st will */
        /* equal saw_race_cleanup in this scenario). */
        if
        : (mode == kill_all) ->
            do /* kill all readers (except ourself) */
            : (st == success) ->
                assert(_r != NIL);
                assert(_r != version[_r].owner); /* we never write to a committed versions. */
                assert(_r == token); /* make new version for this transaction. */
                assert(_ver != Nil); /* link us back in. */
                moveReaderList(_r, object[o].readerList);
                if
                : (_r == NIL) -> break /* no more killing needed. */
                : else
                    fi;
                else /* !success */
                    fi;
            /* try again */
            : else
                fi;
            /* try again from the top */
            moveVersion(_r, _ver, NIL);
            od;
        : else /* no more killing needed. */
            fi;
        /* done */
    fi;
    /* per-object, before write. */
    inline ensureWriter(tid, o, ver) {
        assert(tid != NIL);
        ver = NIL; _r = NIL; _rr = NIL;
        do
        : (ver != NIL) ->
            assert(_r != NIL); /
                assert(_r != version[_r].owner); /* make new version for this transaction. */
                assert(_r == token); /* link us back in. */
                moveReaderList(_r, object[o].readerList);
                if
                : (_r == NIL) -> break /* no more killing needed. */
                : else
                    fi;
                else /* !success */
                    fi;
            /* try again */
            : else
                fi;
        /* try again from the top */
        moveVersion(_r, ver, NIL);
        od;
    }*/
}*/
Figure 7: The field copy-back routine.

Figure 8: The per-object version-setup routine for transactional writes.
```cpp
/* per-field, before write. */
inline checkWriteField(o, f) {
  _r = NIL; _rr = NIL;
do:
    /* set write flag, if not already set */
    _val = object[o].field[f];
    if (_val==FLAG) ->
      break; /* done! */
    else
      fi;
    /* okay, need to set write flag. */
    moveVersion(_rr, object[o].version);
    moveVersion(_r, _rr);
    assert (_r!=NIL);
    do:
      /* this next check ensures that concurrent copythroughs don’t stomp * on each other’s versions, because the field will become FLAG * before any other version will be written. */
      if (_r==NIL) -> break /* done */
      else
        object[o].fieldLock[f] = _thread_id;
        if (_val==FLAG) ->
          version[_r].field[f] = _val;
          else -> break /* abort */
        fi;
        else -> break /* abort */
        fi;
    moveVersion(_r, version[_r].next) /* on to next */
  od;
  if (_r==NIL) ->
    /* field has been successfully copied to all versions */
    atomic {
      if (_val!=NIL) ->
        assert(object[o].field[f]==_val ||
          /* we can race with another copythrough and that’s okay; */
          /* the locking strategy above ensures that we’re all */
          /* writing the same values to all the versions and not */
          /* overwriting anything. */
          object[o].field[f]==FLAG);
        object[o].fieldLock[f] = _thread_id;
        object[o].field[f] = FLAG;
        break; /* success! done! */
      else
        fi
      fi
    }
    else
      fi
    /* retry */
  od;
  /* clean up */
  moveVersion(_r, NIL);
  moveVersion(_rr, NIL);
}
```

Figure 9: The per-field copy-through routine for transactional writes.

5.2.5 Transactional Write

Like the transactional read, transactional writes are split. The first preparatory step is done once for each object written by the transaction. The version list for each object is traversed, aborting other versions and locating or creating a version `ver` for the object corresponding to the current transaction. We retain `ver` for the actual write, later. The reader list must also be traversed, aborting all transactions on the list except the current transaction. This is shown in the `ensureWriter` routine in Figure 8.

The second preparatory step is done once for each field we intend to write. The thread must perform a copy-through on each field: copy the object’s (pre-write) field value into all the versions and then write `FLAG` to the object’s field. We use Load-Linked/Store-Conditional to update versions only if the object’s field has not already been set to `FLAG` behind our backs by a concurrent copy-through. The `checkWriteField` routine is shown in Figure 9.

Finally, for each transactional write to an object, we simply write to the identified version `ver` for that object.

```cpp
inline writeT(ver, f, nval) {
  /* easy enough: */
  version[ver].field[f] = nval;
}
```

Note that we have chosen not to allow concurrent writes by separate transactions to different fields in the same object.

5.3 Large objects

Our software transactions implementation clones objects on transactional writes, so that the previous state of the object can be restored if the transaction aborts. Figure 10 shows the object size distribution of transactional writes for SPECjvm98, and indicates that over 10% of writes may be to large objects. Obviously the copying cost would be prohibitive.

Our solution is to represent large objects as functional arrays. Functional arrays allow access to both committed and uncommitted versions of the (large) object without requiring multiple complete copies. O’Neill and Burton [23] give a fairly inclusive overview of functional array data structures; we will review very briefly here.

Functional arrays are persistent; that is, after an element is updated both the new and the old contents of the array...
are available for use. Since arrays are simply maps from integers (indexes) to values; any functional map datatype (for example, a functional balanced tree) can be used to implement functional arrays.

For concreteness, functional arrays have the following three operations defined:

- **FA-CREATE(n)**: Return an array of size n. The contents of the array are initialized to zero.
- **FA-UPDATE(A_j, i, v)**: Return an array A_j that is functionally identical to array A_j except that A_j(i) = v. Array A_j is not destroyed and can be accessed further.
- **FA-READ(A_j, i)**: Return A_j(i).

We are interested in lock-free implementations, thus we allow any of these operations to fail. Failed operations can be safely retried, as all operations are idempotent by definition.

For the moment, consider the following naïve implementation:

- **FA-CREATE(n)**: Return an ordinary imperative array of size n.
- **FA-UPDATE(A_j, i, v)**: Create a new imperative array A_j and atomically copy the contents of A_j to A_j'. Return A_j'.
- **FA-READ(A_j, i)**: Return A_j[i].

This implementation has O(1) read and O(n) update, so it matches the performance of imperative arrays only when n = O(1), where n is the size of the array (read, object). We will therefore call these small object functional arrays. We can view the version objects we have so far described as implementations of such an algorithm, where FA-UPDATE_ao takes an object and returns a version (slightly abusing the types). This is implemented within **ensureWriter**, Figure 8.

To handle large objects, we replace this naïve implementation in order to scale better with large n. We would like to prevent small changes to a large object from taking a large amount of time. We could consider using a functional array implementation based on functional balanced trees. However, this yields O(lg n) access or update, which is not as good as the O(1) element access and update characteristic of imperative arrays. Instead, we’ve chosen Tyng-Ruey Chuang’s version [6] of shallow binding, which gives O(1) cost for FA-UPDATE and for single-threaded FA-READ. Our use of functional arrays is single-threaded in the common case, when transactions do not abort. Chuang’s scheme uses randomized cuts to the version tree to limit the expected per-write overhead to O(n) in the worst case (that is, the expected cost C(l, m) of l updates and m reads obeys C(l, m) ≤ l + m + 2nl). So with little complexity we can get worst case expected performance for aborts equivalent to the naïve implementation, while obtaining “imperative-speed” common-case (commit) accesses.

Of course, Chuang’s published algorithm is not lock free. The crucial operation is a rotation of a difference node with the main body of the array. Elsewhere we present a lock-free version of Chuang’s algorithm [2]; for the purposes of the present paper we can trivially make Chuang’s published algorithm lock-free by implementing the rotation using some existing small-transaction HTM.

5.4 Hybrid STM/HTM

It is worth considering if a low-level HTM can yield benefits other than efficient implementations of large-object operations. In fact, Figure 11 presents research showing that we can combine the strengths of our object-based software transaction system with a fast bounded-size HTM. The numbers presented are cycle counts from a cycle accurate multiprocessor simulator (UVSIM [26]), extended to implement a HTM for [3]. More details are available in [20].

In the figure, combining the systems is done in the most simple-minded way: all transactions are begun in the hardware transactional memory, and after any abort the transaction is restarted in the object-based software system. The field flag mechanism described in Section 4.1 ensures that software transactions properly abort conflicting hardware transactions — when the software scribbles FLAG over the original field the hardware will detect the conflict. Hardware transactions must perform the ReadNT and WriteNT algorithms to ensure they interact properly with concurrent software transactions, although these checks can be done in software (they do not need to be part of the hardware HTM mechanism). In the figure, the read barriers were done in software, and caused a 2.2x slowdown for the (very small) hardware transactions. This is a pessimistic figure: no special effort was made to tune code or otherwise minimize slowdown, and the processor simulated had limited ability to exploit ILP (2 ALUs and 4-instruction issue width). Even so, the read barriers might be a worthwhile target for hardware support [7].

As a fortuitous synergy, hardware support for small transactions may also be used to implement the software transaction implementation’s Load-Linked/Store-Conditional sequences, which may not otherwise be available on a target processor.

6. VERIFICATION

The Promela model of this software transaction system was model-checked with SPIN version 4.1.0 and verified to operate correctly and without races in the scenarios described below. The verification was done on an SGI 64-processor MIPS machine with 16G of memory.
Sequences of transactional and non-transactional load and store operations were checked using two concurrent processes, and all possible interleavings were found to produce results consistent with the semantic atomicity of the transactions. Several test scripts were run against the model using separate processors of the verification machine (SPIN cannot otherwise exploit SMP). Some representative costs include:

- testing two concurrent writeT operations (including “false flag” conditions) against a single object required \(3.8 \times 10^8\) states and 170M memory;
- testing sequences of transactional and non-transactional reads and writes against two objects (checking that all views of the two objects were consistent) required \(4.6 \times 10^6\) states and 207M memory; and
- testing a pair of concurrent increments at values bracketing the FLAG value to 99.8% coverage of the state space required \(7.6 \times 10^7\) states and 4.3G memory. Simultaneously model-checking a range of values caused the state space explosion in this case.

SPIN’s unreachable code reporting was used to ensure that our test cases exercised every code path, although this doesn’t guarantee that every interesting interaction is checked.

In the process one bug in SPIN was discovered and a number of subtle race conditions in the model were discovered and corrected. These included a number of modelling artifacts: in particular, we were extremely aggressive about control the size of the state space, and this proved difficult to systematically model-checking a range of values caused the state space explosion in this case.

A race allowed conflicting readers to be created while a writer was inside ensureWriter creating a new version object.

- Allowing already-committed version objects to be mutated when writeT or writeNT was asked to store a “false flag” produced races between ensureWriter and copyBackField. The code that was expected to manage these races had unexpected corner cases.
- Using a bitmask to provide per-field granularity to the list of readers proved unmanageable as there were three-way races between the bitmask, the readers list, and the version tree.

In addition, the model-in-progress proved a valuable design tool, as portions of the algorithm could readily be mechanically checked to validate (or discredit) the designer’s reasoning about the concurrent system. Humans do not excel at exhaustive state space exploration.

\footnote{SPIN is not particularly suited to checking models with dynamic allocation and deallocation. In particular, the order of allocation artificially enlarges the state space. A great deal of effort was expended tweaking the model to approach a canonical allocation ordering. A better solution to this problem would allow larger model instances to be checked.}

\footnote{Breadth-first search of atomic regions was performed incorrectly in SPIN 4.0.7; this was fixed in SPIN 4.1.0.}

7. CONCLUSIONS

We have described an efficient object-based implementation of non-blocking software transactions, along with empirical data backing its design and extensions for large objects and HTM hybridization. The design has been model-checked with the SPIN model checker, and implemented in the FLEX Java compiler.

Spin model for Software Transactions

The complete SPIN 4.1.0 model for the FLEX software transaction system is available for download at http://flex-master.csail.mit.edu/Harpoon/swx.pml.

Acknowledgments

The work described in Section 5.4 was joint with Sean Lie, who provided Figure 11. Some additional details are in [20].

8. REFERENCES

APPENDIX

A. PROMELA PRIMER

A concise Promela reference is available at http://spinroot.com/spin/Man/Quick.html; we will here attempt to summarize just enough of the language to allow the model we’ve presented in this paper to be understood.

Promela syntax is C-like, with the same lexical and commenting conventions. Statements are separated by either a semi-colon, or, equivalently, an arrow. The arrow is typically used to separate a guard expression from the statements it is guarding.

The program counter moves past a statement only if the statement is enabled. Most statements, including assignments, are always enabled. A statement consisting only of an expression is enabled iff the expression is true (non-zero). Our model uses three basic Promela statements: selection, repetition, and atomic.

The selection statement,

if :: guard -> statements
:: statements
fi

selects one among its options and executes it. An option can be selected iff its first statement (the guard) is enabled. The special guard else is enabled iff all other guards are not.

The repetition statement,

do :: statements
:: statements
fi

is similar: one among its enabled statements is selected and executed, and then the process is repeated (with a different statement possibly being selected each time) until control is explicitly transferred out of the loop with a break or goto.

Finally, atomic { statements }

executes the given statements in one indivisible step. For the purposes of this model, a dstep block is functionally identical. Outside atomic or dstep blocks, Promela allows interleaving before and after every statement, but statements are indivisible.

Functions as specified in this model are similar to C macros: every parameter is potentially both an input and an output. Calls to functions with names starting with move are simple assignments; they’ve been turned into macros so that reference counting can be performed.