Everything in the universe relates to [transactions], one way or another, given enough ingenuity on the part of the interpreter.

Principia Discordia (amended)

Chapter 8

Related work

A number of researchers have been investigating transactional memory systems. This thesis is the first to present a hybrid hardware/software model-checked non-blocking object-oriented system which allows co-existence of non-transactional and transactional accesses to a dynamic set of object fields.

8.1 Non-blocking synchronization

Lamport presented the first alternative to synchronization via mutual exclusion in [48], for a limited situation involving a single writer and multiple readers. Lamport's technique relies on reading guard elements in an order opposite to that in which they are written, guaranteeing that a consistent data snapshot can be recognized. The writer always completes its part of the algorithm in a constant number of steps; readers are guaranteed to complete only in the absence of concurrent writes.

Herlihy formalized *wait-free* implementations of concurrent data objects in [37]. A wait-free implementation guarantees that any process can complete any operation in a finite number of steps, regardless of the activities of other processes. Lamport's algorithm is not wait-free because readers can be delayed indefinitely.

Massalin and Pu introduced the term *lock-free* to describe algorithms with weaker progress guarantees. A lock-free implementation guarantees only that *some* process will complete in a finite number of steps [52]. Unlike a wait-free implementation, lock-freedom allows starvation. Since other simple techniques can be layered to prevent starvation (for example, exponential backoff), simple lock-free implementations are usually seen as worthwhile practical alternatives to more complex wait-free implementations.

An even weaker criterion, *obstruction-freedom*, was introduced by Herlihy, Luchangco, and Moir in [39]. Obstruction-freedom only guarantees progress for threads executing in isolation; that is, although other threads may have partially completed operations, no other thread may take a step until the isolated thread completes. Obstruction-freedom not only allows starvation of a particular thread, it allows contention among threads to halt all progress in all threads indefinitely. External mechanisms are used to reduce contention (thus, achieve progress) including backoff, queueing, or timestamping.

We will use the term *non-blocking* to describe generally any synchronization mechanism which doesn't rely on mutual exclusion or locking, including wait-free, lock-free, and obstruction-free implementations. We will be concerned mainly with lock-free algorithms.[1]

8.2 Efficiency

Herlihy presented the first *universal* method for wait-free concurrent implementation of an arbitrary sequential object [37, 33]. This original method was based on a *fetch-and-cons* primitive, which atomically places an item

[1]Note that some authors use "non-blocking" and "lock-free" as synonyms, usually meaning what we here call *lock-free*. Others exchange our definitions for "lock-free" and "non-blocking", using lock-free as a generic term and non-blocking to describe a specific class of implementations. As there is variation in the field, we choose to use the parallel construction *wait-free*, *lock-free*, and *obstruction-free* for our three specific progress criteria, and the dissimilar *non-blocking* for the general class.

on the head of a list and returns the list of items following it; all concurrent primitives capable of solving the n-process consensus problem—*universal* primitives—were shown powerful enough to implement *fetch-and-cons*. In Herlihy's method, every sequential operation is translated into two steps. In the first, *fetch-and-cons* is used to place the name and arguments of the operation to be performed at the head of a list, returning the other operations on the list. Since the state of a deterministic object is completely determined by the history of operations performed on it, applying the operations returned in order from last to first is sufficient to locally reconstruct the object state prior to our operation. We then use the prior state to compute the result of our operation without requiring further synchronization with the other processes.

This first universal method was not very practical, a shortcoming which Herlihy soon addressed [34]. In addition, his revised universal method can be made lock-free, rather than wait-free, resulting in improved performance. In the lock-free version of this method, objects contain a shared variable holding a pointer to their current state. Processes begin by loading the current state pointer and then copying the referenced state to a local copy. The sequential operation is performed on the copy, and then if the object's shared state pointer is unchanged from its initial load it is atomically swung to point at the updated state.

Herlihy called this the "small object protocol" because the object copying overhead is prohibitive unless the object is small enough to be copied efficiently (in, say, $O(1)$ time). He also presented a "large object protocol" which requires the programmer to manually break the object into small blocks, after which the small object protocol can be employed. [This trouble with large objects is common to many non-blocking implementations; our solution is presented in chapter 5.]

Barnes provided the first universal non-blocking implementation method which avoids object copying [10]. He eliminates the need to store "old" object state in case of operation failure by having all threads cooperate to apply

operations. For example, if the first processor begins an operation and then halts, another processor will complete the operation of the first before applying its own. Barnes proposes to accomplish the cooperation by creating a parallel state machine for each operation, so that each thread can independently try to advance the machine from state to state and thus advance incomplete operations.[2] Although this avoids copying state, the lock-step cooperative process is extremely cumbersome and does not appear to have ever been implemented. Furthermore, it does not protect against errors in the implementation of the operations, which could cause *every* thread to fail in turn as one by one they attempt to execute a buggy operation.

Alemany and Felten [2] identified two factors hindering the performance of non-blocking algorithms to date: resources wasted by operations that fail, and the cost of data copying. Unfortunately, they proceeded to "solve" these problems by ignoring short delays and failures and using operating system support to handle delays caused by context switches, page faults, and I/O operations. This works in some situations, but obviously suffers from a bootstrapping problem as the means to implement an operating system.

Although lock-free implementations are usually assumed to be more efficient that wait-free implementations, LaMarca [47] showed experimental evidence that Herlihy's simple wait-free protocol scales very well on parallel machines. When more than about twenty threads are involved, the wait-free protocol becomes faster than Herlihy's lock-free small-object protocol, three OS-aided protocols of LaMarca and Alemany and Felten, and a *test-and-Compare&Swap* spin-lock.

[2] It is interesting to note that Barnes' cooperative method for non-blocking situation plays out in a real-time system very similarly to priority inheritance for locking synchronization.

8.3 Transactional Memory systems

Transactions are described in the database context by Gray [25], and [26] contains a thorough treatment of database issues. Hardware Transactional Memory (HTM) was first proposed by Knight [46], and Herlihy and Moss coined the term "transactional memory" and proposed HTM in the context of lock-free data structures [40, 36]. The BBN Pluribus [64, Ch. 23] provided transactions, with an architectural limit on the size of a transaction. Experience with Pluribus showed that the headaches of programming correctly with such limits can be at least as challenging as using locks. The ***Oklahoma Update*** is another variation on transactional operations with an architectural limit on the number of values in a transaction [65].

Transactional memory is sometimes described as an extension of Load-Linked/Store-Conditional [43] and other atomic instruction sequences. In fact, some CISC machines, such as the VAX, had complex atomic instructions such as enqueue and dequeue [19].

Of particular relevance are ***Speculative Lock Elision*** (SLE) [57] and ***Transactional Lock Removal*** (TLR) [58], which speculatively identify locks and use the cache to give the appearance of atomicity. SLE and TLR handle mutual exclusion through a standard programmer interface (locks), dynamically translating locks into transactional regions. My research thrust differs from theirs in that I hope to free programmers from the protocol complexities of locking, not just optimize existing practice. The quantitative results presented in this thesis confirm their finding that transactional hardware can be more efficient than locks.

Martinez and Torrellas proposed ***Speculative Synchronization*** [51], which allows some threads to execute atomic regions of code speculatively, using locks, while guaranteeing forward progress by maintaining a nonspeculative thread. These techniques gain many of the performance advantages of transactional memory, but they still require new code to obey a locking protocol to avoid deadlock.

The recent work on *Transactional memory Coherence and Consistency* (TCC) [29] is also relevant to our work. TCC uses a broadcast bus to implement the transaction protocols, performing all the writes of a particular transaction in one atomic bus operation. This strategy limits scalability, whereas both the UTM and LTM proposals in chapter 6 can employ scalable cache-consistency protocols to implement transactions. TCC affirms the conclusion we draw from our own Figure 3.3: most transactions are small, but some are very large. TCC supports large transactions by locking the broadcast bus and stalling all other processors when any processor buffer overflows, whereas UTM and LTM allow overlapped execution of multiple large transactions with local overflow buffers. TCC is similar to LTM in that transactions are bound to processor state and cannot extend across page faults, timer interrupts, or thread migrations.

Several software transaction systems have been proposed. Some constrain the programmer and make transactions difficult to use. All have relatively high overheads, which make transactions unattractive for uniprocessor and small SMP systems. [Once the number of processors is large enough, the increased parallelism which can be provided by optimistic transactions may cancel out the performance penalty of their use.]

The first proposal for software transactional memory was proposed by Shavit and Touitou [63]; their system requires that all input and output locations touched by a transaction be known in advance, which limits its application. It performs at least 10 fetches and 4 stores per location accessed (not counting the loads and stores directly required by the computation). The benchmarks presented were run on between 10 and 64 processors.

Rudys and Wallach [60] proposed a copying-based transaction system to allow rollback of hostile codelets. They show an order of magnitude slowdown for field and array accesses, and 6x to 23x slowdown on their benchmarks.

Herlihy, Luchango, Moss, and Scherer's scheme [35] 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. This implementation is based on object copying, and so has poor performance for large objects and arrays. Not including work necessary to copy objects involved in writes, they require $O(R(R+W))$ work to open R objects for reading and W objects for writing, which may be quadratic in the number of objects involved in the transaction. A list insertion benchmark which they present shows 9x slowdown over a locking scheme, although they beat the locking implementation when more than 5-10 processors are active. They present benchmark data with up to 576 threads on 72 processors.

Harris and Fraser built a software transaction system on a flat word-oriented transactional memory abstraction [30], roughly similar to simulating Herlihy's original hardware transactional memory proposal in software. This avoids problems with large objects. Performing m memory operations touching l distinct locations costs at least $m+l$ extra reads and $l+1$ CAS operations, in addition to the reads and writes required by the computation. They appear to execute about twice as slowly as a locking implementation on some microbenchmarks. They benchmark on a 4-processor as well as a 106-processor machine; their crossover point (at which the blocking overhead of locks matches the software transaction overhead) is around 4 processors. Note that Harris and Fraser do not address the problem of concurrent non-transactional operations on locations involved in a transaction. Java synchronization allows such concurrent operations, with semantics given by the Java memory model [50]. We support these operations safely using the mechanisms presented in chapter 3.

Programmers will be reluctant to use transactions to synchronize their code when it results in their code running more slowly on the uniprocessor and small-SMP systems which are most common today.

Herlihy and Moss [36] suggested that small transactions might be handled in cache with overflows handled by software. These software overflows must interact with the transactional hardware in the same way that the hardware interacts with itself, however. In chapter 6.4 we present just such

a system.

8.4 Language-level approaches to synchronization

Our work on integrating transactions into the Java programming language is related to prior work on integrating synchronization mechanisms for multiprogramming, and in particular, to prior work on synchronization in an object-oriented framework.

The Emerald system [12, 45] introduced *monitored objects* for synchronization. Emerald code to implement a simple directory object is shown in Figure 8.1. Each object is associated with Hoare-style monitor, which provides mutual exclusion and process signaling. Each Emerald object is divided into a monitored part and a non-monitored part. Variables declared in the monitored part are shared, and access to them from methods in the non-monitored part is prohibited—although non-monitored methods may call monitored methods to effect the access. Methods in the monitored part acquire the monitor lock associated with the receiver object before entry and release it on exit, providing for mutual exclusion and safe update of the shared variables. Monitored objects naturally integrate synchronization into the object model.

Unlike Emerald monitored objects, where methods can only acquire the monitor of their receiver and where restricted access to shared variables is enforced by the compiler, Java implements a loose variant where any monitor may be explicitly acquired and no shared variable protection exists. As a default, however, Java methods declared with the `synchronized` keyword behave like Emerald monitored methods, ensuring that the monitor lock of their receiver is held during execution.

Java's synchronization primitives arguably allow for more efficient concurrent code than Emerald's—for example, Java objects can use multiple locks to protect disjoint sets of fields, and coarse-grain locks can be used which protect multiple objects—but Java is also more prone to programmer

```
const myDirectory == object oneEntryDirectory
  export Store, Lookup
  monitor
    var name : String
    var AnObject : Any

    operation Store [ n : String, o : Any ]
      name ← n
      AnObject ← o
    end Store
    function Lookup [ n : String ] → [ o : Any ]
      if n = name
        then o ← AnObject
        else o ← nil
      end if
    end Lookup

    initially
      name ← nil
      AnObject ← nil
    end initially

  end monitor
end oneEntryDirectory
```

Figure 8.1: A directory object in Emerald, from [12], illustrating the use of monitor synchronization.

```
class Account {

  int balance = 0;

  atomic int deposit(int amt) {
    int t = this.balance;
    t = t + amt;
    this.balance = t;
    return t;
  }

  atomic int readBalance() {
    return this.balance;
  }

  atomic int withdraw(int amt) {
    int t = this.balance;
    t = t - amt;
    this.balance = t;
    return t;
  }

}
```

Figure 8.2: A simple bank account object, adapted from [20], illustrating the use of the `atomic` modifier.

error. However, even Emerald's restrictive monitored objects are not sufficient to prevent data races. As a simple example, imagine that an object provided two monitored methods `read` and `write` which accessed a shared variable. Non-monitored code can call `read`, increment the value returned, and then call `write`, creating a classic race condition scenario. The atomicity of the parts is not sufficient to guarantee atomicity of the whole [20].

This suggests that a better model for synchronization in object-oriented systems is *atomicity*. Figure 8.2 shows Java extended with an `atomic` keyword to implement an object representing a bank account. Rather than explicitly synchronizing on locks, I simply require that the methods marked `atomic` execute atomically with respect to other threads in the system; that is, that every execution of the program computes the same result as some execution where all atomic methods were run *in isolation* at a certain point in time, called the ***linearization point***, between their invocation and return. Note that atomic methods invoked directly or indirectly from an atomic method are subsumed by it: if the outermost method appears atomic, then by definition all inner method invocations will also appear atomic. Flanagan and Qadeer provide a more formal semantics in [20]. Atomic methods can be analyzed using sequential reasoning techniques, which significantly simplifies reasoning about program correctness.

Atomic methods can be implemented using locks. A simple if inefficient implementation would simply acquire a single global lock during the execution of every atomic method. Flanagan and Qadeer [20] present a more sophisticated technique which proves that a given implementation using standard Java monitors correctly guarantees method atomicity.

The transaction implementations presented in this thesis will use non-blocking synchronization to implement atomic methods.

"Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop."

Lewis Carroll, *Alice's Adventures in Wonderland*

Chapter 9

Conclusion

In this thesis I have shown that it is possible to implement an efficient strongly-atomic software transaction system, and that non-blocking transactions can be used in applications beyond synchronization, including fault tolerance and backtracking search. I have presented implementation details to address the practical problems of building such a system. I have argued the transactions should not be subject to limits on size or duration, and presented both software and hardware implementations free of such restrictions. Finally, the low overhead of my software system allows it to be profitably combined with a hardware transaction system; I have shown how this hybrid yields fast execution of short and small transactions while allowing fallback to software for large or complicated transactions.

misplaced in the flow

There is no escape: parallel systems are in our future. However, programming them does not have to be as fraught as it is presently. I believe that transactions provide a programmer-friendly model of concurrency which eliminates many potential pitfalls of our current locking-based methodologies.

In this thesis I have presented several designs for efficient transaction systems that will enable us to take advantage of the transactional programming model. The software-only system runs on current hardware, and LTM and UTM indicate possible directions for future hardware. However, there

are challenges and design decisions remaining: how should I/O be handled? What are the proper semantics for nested transactions? What loophole mechanisms are necessary to allow information about a transaction's progress to escape?

I believe hybrid systems are the best answer to these challenges: they combine the inherent speed of hardware systems with the flexibility of software, allowing novel solutions to be attempted without requiring that design decisions be cast in silicon. The flag-based software transaction system described in this thesis imposes very low overhead, allowing transactional programming to get off the ground without hardware support in the near term, while later supporting the development of new transactional models and methodologies as part of a hybrid system.

Designing correct transaction systems is not easy, however. In the appendix you will find a Promela model of my software transaction system. Automated verification was essential when designing and debugging the system, uncovering via exhaustive enumeration race conditions much too subtle for me to discover by other means. I believe any credible transaction system must be buttressed by formal verification.

Essentially, all models are
wrong, but some are useful.

George E. P. Box, *Empirical Model-Building and Response Surfaces*

Appendix A

Model-checking the software implementation

My work on both software and hardware transaction systems has reiterated the difficulty of creating correct implementations of concurrent and fault-tolerant constructs. Automatic model checking is a prerequisite to achieving confidence in the design and implementation. Versions of the software transaction system have been modeled in Promela using SPIN 4.1.0 and verified on an SGI 64-processor MIPS machine with 16G of main 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 3.8×10^6 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 4.6×10^6 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 7.6×10^7 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[1] and a number of subtle race conditions in the model were discovered and corrected. These included a number of modeling artifacts: in particular, we were extremely aggressive about reference-counting and deallocating objects in order to control the size of the state space, and this proved difficult to do correctly. We also discovered some subtle-but-legitimate race conditions in the transactions algorithm. For example:

- 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 dis-

[1] Breadth-first search of atomic regions was performed incorrectly in SPIN 4.0.7; this was fixed in SPIN 4.1.0.

credit) the designer's reasoning about the concurrent system. Humans do not excel at exhaustive state space exploration.

SPIN is not particularly suited to checking models with dynamic allocation and deallocation. In particular, it considers the location of objects part of the state space, and allocating object A before object B produces a different state than if object B were allocated first. This 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.

A.1 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 thesis 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
...
:: guard -> 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 `d_step` block is functionally identical. Outside `atomic` or `d_step` 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.

A.2 Spin model for software transaction system

The complete SPIN 4.1.0 model for the FLEX software transaction system is presented here. It may also be downloaded from http://flex-master.csail.mit.edu/Harpoon/swx.pml.

```
/*************************************************************************
 * Very detailed model of software transaction code.
 * Checking for safety and correctness properties.  Not too worried about
 * liveness at the moment.
```

```
 *
 * (C) 2006 C. Scott Ananian <cananian@alumni.princeton.edu>
 ************************************************************************/

/* CURRENT ISSUES:
 *  none known.
 */
/* MINOR ISSUES:
 * 1) use smaller values for FLAG and NIL to save state space?
 */

/* Should use Spin 4.1.0, for correct nested-atomic behavior. */

#define REFCOUNT

#define NUM_OBJS 2
#define NUM_VERSIONS 6 /* each obj: committed and waiting version, plus nonce
                        * plus addition nonce for NT copyback in test3 */
#define NUM_READERS 4 /* both 'read' trans reading both objs */
#define NUM_TRANS 5 /* two 'real' TIDs, plus 2 outstanding TIDs for
     * writeNT(FLAG) [test3], plus perma-aborted TID. */
#define NUM_FIELDS 2

#define NIL 255 /* special value to represent 'alloc impossible', etc. */
#define FLAG 202 /* special value to represent 'not here' */

typedef Object {
  byte version;
  byte readerList; /* we do LL and CAS operations on this field */
  pid fieldLock[NUM_FIELDS]; /* we do LL operations on fields */
  byte field[NUM_FIELDS];
};
typedef VersiOn { /* 'Version' misspelled because spin #define's it. */
  byte owner;
  byte next;
  byte field[NUM_FIELDS];
#ifdef REFCOUNT
  byte ref; /* reference count */
#endif /* REFCOUNT */
```

```
};
typedef ReaderList {
  byte transid;
  byte next;
#ifdef REFCOUNT
  byte ref; /* reference count */
#endif /* REFCOUNT */
};
mtype = { waiting, committed, aborted };
typedef TransID {
  mtype status;
#ifdef REFCOUNT
  byte ref; /* reference count */
#endif /* REFCOUNT */
};
Object object[NUM_OBJS];
VersiOn version[NUM_VERSIONS];
ReaderList readerlist[NUM_READERS];
TransID transid[NUM_TRANS];
byte aborted_tid; /* global variable; 'perma-aborted' */

/* --------------------------- alloc.pml ----------------------- */
mtype = { request, return };

inline manager(NUM_ITEMS, allocchan) {
  chan pool = [NUM_ITEMS] of { byte };
  chan client;
  byte nodenum;
  /* fill up the pool with node identifiers */
  d_step {
    i=0;
    do
    :: i<NUM_ITEMS -> pool!!i; i++
    :: else -> break
    od;
  }
end:
  do
  :: allocchan?request(client,_) ->
```

```
      if
      :: empty(pool) -> assert(0); client!NIL /* deny */
      :: nempty(pool) ->
pool?nodenum;
client!nodenum;
nodenum=0
      fi
  :: allocchan?return(client,nodenum) ->
      pool!!nodenum; /* sorted, to reduce state space */
      nodenum=0
  od
}

chan allocObjectChan = [0] of { mtype, chan, byte };
active proctype ObjectManager() {
  atomic {
    byte i;
    manager(NUM_OBJS, allocObjectChan)
  }
}
chan allocVersionChan = [0] of { mtype, chan, byte };
active proctype VersionManager() {
  atomic {
    byte i=0;
    d_step {
      do
      :: i<NUM_VERSIONS ->
 version[i].owner=NIL; version[i].next=NIL;
 version[i].field[0]=FLAG; version[i].field[1]=FLAG;
 assert(NUM_FIELDS==2);
 i++
      :: else -> break
      od;
    }
    manager(NUM_VERSIONS, allocVersionChan)
  }
}
chan allocReaderListChan = [0] of { mtype, chan, byte };
active proctype ReaderListManager() {
```

```
  atomic {
    byte i=0;
    d_step {
      do
      :: i<NUM_READERS ->
 readerlist[i].transid=NIL; readerlist[i].next=NIL;
 i++
      :: else -> break
      od;
    }
    manager(NUM_READERS, allocReaderListChan)
  }
}
chan allocTransIDChan = [0] of { mtype, chan, byte };
active proctype TransIDManager() {
  atomic {
    byte i=0;
    d_step {
      do
      :: i<NUM_TRANS -> transid[i].status=waiting; i++
      :: else -> break
      od;
    }
    manager(NUM_TRANS, allocTransIDChan)
  }
}

inline alloc(allocchan, retval, result) {
  result = NIL;
  do
  :: result != NIL -> break
  :: else -> allocchan!request(retval,0) ; retval ? result
  od;
  skip /* target of break. */
}
inline free(allocchan, retval, result) {
  allocchan!return(retval,result)
}
inline allocObject(retval, result) {
```

```
  atomic {
    alloc(allocObjectChan, retval, result);
    d_step {
      object[result].version = NIL;
      object[result].readerList = NIL;
      object[result].field[0] = 0;
      object[result].field[1] = 0;
      object[result].fieldLock[0] = _thread_id;
      object[result].fieldLock[1] = _thread_id;
      assert(NUM_FIELDS==2); /* else ought to initialize more fields */
    }
  }
}
inline allocTransID(retval, result) {
  atomic {
    alloc(allocTransIDChan, retval, result);
    d_step {
      transid[result].status = waiting;
#ifdef REFCOUNT
      transid[result].ref = 1;
#endif /* REFCOUNT */
    }
  }
}
inline moveTransID(dst, src) {
  atomic {
#ifdef REFCOUNT
    _free = NIL;
    if
    :: (src!=NIL) ->
       transid[src].ref++
    :: else
    fi;
    if
    :: (dst!=NIL) ->
       transid[dst].ref--;
       if
       :: (transid[dst].ref==0) -> _free=dst
       :: else
```

```
        fi
    :: else
    fi;
#endif /* REFCOUNT */
    dst = src;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeTransID(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif /* REFCOUNT */
  }
}
proctype freeTransID(byte result; chan retval) {
  chan _retval = [0] of { byte };
  atomic {
#ifdef REFCOUNT
    assert(transid[result].ref==0);
#endif /* REFCOUNT */
    transid[result].status = waiting;
    free(allocTransIDChan, _retval, result)
    retval!0; /* done */
  }
}
inline allocVersion(retval, result, a_transid, tail) {
  atomic {
    alloc(allocVersionChan, retval, result);
    d_step {
#ifdef REFCOUNT
      if
      :: (a_transid!=NIL) -> transid[a_transid].ref++;
      :: else
      fi;
      if
      :: (tail!=NIL) -> version[tail].ref++;
      :: else
      fi;
      version[result].ref = 1;
```

```
#endif /* REFCOUNT */
      version[result].owner = a_transid;
      version[result].next = tail;
      version[result].field[0] = FLAG;
      version[result].field[1] = FLAG;
      assert(NUM_FIELDS==2); /* else ought to initialize more fields */
    }
  }
}
inline moveVersion(dst, src) {
  atomic {
#ifdef REFCOUNT
    _free = NIL;
    if
    :: (src!=NIL) ->
       version[src].ref++
    :: else
    fi;
    if
    :: (dst!=NIL) ->
       version[dst].ref--;
       if
       :: (version[dst].ref==0) -> _free=dst
       :: else
       fi
    :: else
    fi;
#endif /* REFCOUNT */
    dst = src;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeVersion(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif /* REFCOUNT */
  }
}
proctype freeVersion(byte result; chan retval) {
```

```
  chan _retval = [0] of { byte };
  byte _free;
  atomic { /* zero out version structure */
#ifdef REFCOUNT
    assert(version[result].ref==0);
#endif /* REFCOUNT */
    moveTransID(version[result].owner, NIL);
    moveVersion(version[result].next, NIL);
    version[result].field[0] = FLAG;
    version[result].field[1] = FLAG;
    assert(NUM_FIELDS==2);
    free(allocVersionChan, _retval, result)
    retval!0; /* done */
  }
}

inline allocReaderList(retval, result, head, tail) {
  atomic {
    assert(head!=NIL);
    alloc(allocReaderListChan, retval, result);
    d_step {
#ifdef REFCOUNT
      readerlist[result].ref = 1;
      transid[head].ref++;
      if
      :: (tail!=NIL) -> readerlist[tail].ref++
      :: else
      fi;
#endif /* REFCOUNT */
      readerlist[result].transid = head;
      readerlist[result].next = tail;
    }
  }
}
inline moveReaderList(dst, src) {
  atomic {
#ifdef REFCOUNT
    _free = NIL;
    if
```

```
    :: (src!=NIL) ->
       readerlist[src].ref++
    :: else
    fi;
    if
    :: (dst!=NIL) ->
       readerlist[dst].ref--;
       if
       :: (readerlist[dst].ref==0) -> _free=dst
       :: else
       fi
    :: else
    fi;
#endif /* REFCOUNT */
    dst = src;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeReaderList(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif
  }
}
proctype freeReaderList(byte result; chan retval) {
  chan _retval = [0] of { byte };
  byte _free;
  atomic {
#ifdef REFCOUNT
    assert(readerlist[result].ref==0);
#endif /* REFCOUNT */
    moveTransID(readerlist[result].transid, NIL);
    moveReaderList(readerlist[result].next, NIL);
    free(allocReaderListChan, _retval, result)
    retval!0; /* done */
  }
}

/* ---------------------------- atomic.pml ----------------------- */
```

```
inline DCAS(loc1, oval1, nval1, loc2, oval2, nval2, st) {
  d_step {
    if
    :: (loc1==oval1) && (loc2==oval2) ->
       loc1=nval1;
       loc2=nval2;
       st=true
    :: else ->
       st=false
    fi
  }
}
inline CAS(loc1, oval1, nval1, st) {
  d_step {
    if
    :: (loc1==oval1) ->
       loc1=nval1;
       st=true
    :: else ->
       st=false
    fi
  }
}
inline CAS_Version(loc1, oval1, nval1, st) {
  atomic {
    _free = NIL;
    if
    :: (loc1==oval1) ->
#ifdef REFCOUNT
       if
       :: (nval1!=NIL) -> version[nval1].ref++;
       :: else
       fi;
       if
       :: (oval1!=NIL) -> version[oval1].ref--;
  if
  :: (version[oval1].ref==0) -> _free = oval1
  :: else
  fi
```

```
        :: else
        fi;
#endif /* REFCOUNT */
        loc1=nval1;
        st=true
    :: else ->
        st=false
    fi;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeVersion(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif /* REFCOUNT */
  }
}
inline CAS_Reader(loc1, oval1, nval1, st) {
  atomic {
    /* save oval1, as it could change as soon as we leave the d_step */
    _free = NIL;
    if
    :: (loc1==oval1) ->
#ifdef REFCOUNT
        if
        :: (nval1!=NIL) -> readerlist[nval1].ref++;
        :: else
        fi;
        if
        :: (oval1!=NIL) -> readerlist[oval1].ref--;
  if
  :: (readerlist[oval1].ref==0) -> _free = oval1
  :: else
  fi
        :: else
        fi;
#endif /* REFCOUNT */
        loc1=nval1;
        st=true
```

```
    :: else ->
       st=false
    fi;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeReaderList(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif /* REFCOUNT */
  }
}

/* ---------------- end atomic.pml ----------------- */

mtype = { kill_writers, kill_all };
mtype = { success, saw_race, saw_race_cleanup, false_flag };

inline tryToAbort(t) {
  assert(t!=NIL);
  CAS(transid[t].status, waiting, aborted, _);
  assert(transid[t].status==aborted || transid[t].status==committed)
}
inline tryToCommit(t) {
  assert(t!=NIL);
  CAS(transid[t].status, waiting, committed, _);
  assert(transid[t].status==aborted || transid[t].status==committed)
}
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
  :: moveVersion(_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=NIL;
      moveTransID(_tmp_tid, version[_ver].owner);
      if
      :: (_tmp_tid==NIL) ->
break; /* found a committed version */
      :: else
      fi;
      tryToAbort(_tmp_tid);
      if
      :: (transid[_tmp_tid].status==committed) ->
moveTransID(_tmp_tid, NIL);
moveTransID(version[_ver].owner, NIL); /* opportunistic free */
moveVersion(version[_ver].next, NIL); /* opportunistic free */
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, _);
      moveTransID(_tmp_tid, NIL);
  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
     :: else
     fi
  :: else
  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 = false_flag /* ...no copy back needed */
:: else -> /* not a false flag */
   d_step { /* could be DCAS */
     if
     :: (object[o].version == _nonceV) ->
object[o].fieldLock[f] = _thread_id;
object[o].field[f] = _val;
     :: else /* hmm, fail.  Must retry. */
st = saw_race_cleanup /* need to clean up nonce */
     fi
   }
fi
     :: else /* may arrive here because of readT, which doesn't set _val=FLAG*/
st = saw_race_cleanup /* need to clean up nonce */
     fi
  :: else /* !success */
  fi;

  /* 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 */
     :: moveReaderList(_r, object[o].readerList);
if
:: (_r==NIL) -> break
:: else
fi;
tryToAbort(readerlist[_r].transid);
/* link out this reader */
CAS_Reader(object[o].readerList, _r, readerlist[_r].next, _);
```

```
      od;
  :: else /* no more killing needed. */
  fi;

  /* finally, clean up our mess. */
  moveVersion(_ver, NIL);
  if
  :: (st == saw_race_cleanup || st == success || st == false_flag) ->
     CAS_Version(object[o].version, _nonceV, version[_nonceV].next, _);
     moveVersion(_nonceV, NIL);
     if
     :: (st==saw_race_cleanup) -> st=saw_race
     :: else
     fi
  :: else
  fi;
  /* done */
  assert(_nonceV==NIL);
}

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
}
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
  fi
}
/* unsuccessful SC */
copyBackField(o, f, kill_all, _st)
/* ignore return status */
      od
  :: else -> /* create false flag */
     /* implement this as a short *transactional* write.  this may be slow,
      * but it greatly reduces the race conditions we have to think about. */
     do
     :: allocTransID(_retval, _writeTID);
ensureWriter(_writeTID, o, _tmp_ver);
checkWriteField(o, f);
writeT(_tmp_ver, f, nval);
tryToCommit(_writeTID);
moveVersion(_tmp_ver, NIL);
if
:: (transid[_writeTID].status==committed) ->
   moveTransID(_writeTID, NIL);
   break /* success! */
:: else ->/* try again */
   moveTransID(_writeTID, NIL)
fi
      od
  fi;
}
inline readT(tid, o, f, ver, result) {
  do
  ::
     /* we should always either be on the readerlist or aborted here */
```

```
    atomic { /* complicated assertion; evaluate atomically */
      if
      :: (transid[tid].status == aborted) -> skip /* okay then */
      :: else ->
  assert (transid[tid].status == waiting);
  _r = object[o].readerList;
  do
  :: (_r==NIL || readerlist[_r].transid==tid) -> break
  :: else -> _r = readerlist[_r].next
  od;
  assert (_r!=NIL); /* we're on the list */
  _r = NIL /* back to normal */
       fi
    }
    /* okay, sanity checking done -- now let's get to work! */
    result = object[o].field[f];
    if
    :: (result==FLAG) ->
if
:: (ver!=NIL) ->
   result = version[ver].field[f];
   break /* done! */
:: else ->
   findVersion(tid, o, ver);
   if
   :: (ver==NIL) -> /* use value from committed version */
      assert (_r!=NIL);
      result = version[_r].field[f]; /* false flag? */
      moveVersion(_r, NIL);
      break /* done */
   :: else /* try, try, again */
   fi
fi
     :: else -> break /* done! */
     fi
  od
}
inline writeT(ver, f, nval) {
  /* easy enough: */
```

```
  version[ver].field[f] = nval;
}

/* make sure 'tid' is on reader list. */
inline ensureReaderList(tid, o) {
  /* add yourself to readers list. */
  _rr = NIL; _r = NIL;
  do
  :: moveReaderList(_rr, object[o].readerList); /* first_reader */
     moveReaderList(_r, _rr);
     do
     :: (_r==NIL) ->
break /* not on the list */
     :: (_r!=NIL && readerlist[_r].transid==tid) ->
break /* on the list */
     :: else ->
/* opportunistic free? */
if
:: (_r==_rr && transid[readerlist[_r].transid].status != waiting) ->
   CAS_Reader(object[o].readerList, _r, readerlist[_r].next, _)
   if
   :: (_cas_stat) -> moveReaderList(_rr, readerlist[_r].next)
   :: else
   fi
:: else
fi;
/* keep looking */
moveReaderList(_r, readerlist[_r].next)
     od;
     if
     :: (_r!=NIL) ->
break /* on the list; we're done! */
     :: else ->
/* try to put ourselves on the list. */
assert(tid!=NIL && _r==NIL);
allocReaderList(_retval, _r, tid, _rr);
CAS_Reader(object[o].readerList, _rr, _r, _cas_stat);
if
:: (_cas_stat) ->
```

```
   break /* we're on the list */
:: else
fi
/* failed to put ourselves on the list, retry. */
     fi
  od;
  moveReaderList(_rr, NIL);
  moveReaderList(_r, NIL);
  /* done. */
}

/* look for a version read/writable by 'tid'.  Returns:
 *   ver!=NIL -- ver is the 'waiting' version for 'tid'. _r == NIL.
 *   ver==NIL, _r != NIL -- _r is the first committed version in the chain.
 *   ver==NIL, _r == NIL -- there are no commited versions for this object
 *                          (i.e. object[o].version==NIL)
 */
inline findVersion(tid, o, ver) {
  assert(tid!=NIL);
  _r = NIL; ver = NIL; _tmp_tid=NIL;
  do
  :: moveVersion(_r, object[o].version);
     if
     :: (_r==NIL) -> break /* no versions. */
     :: else
     fi;
     moveTransID(_tmp_tid, version[_r].owner);/*use local copy to avoid races*/
     if
     :: (_tmp_tid==tid) ->
ver = _r; /* found a version: ourself! */
_r = NIL; /* transfer owner of the reference to ver, without ++/-- */
break
     :: (_tmp_tid==NIL) ->
/* perma-committed version.  Return in _r. */
moveVersion(version[_r].next, NIL); /* opportunistic free */
break
     :: else -> /* strange version.  try to kill it. */
/* ! could double-check that our own transid is not aborted here. */
tryToAbort(_tmp_tid);
```

```
if
:: (transid[_tmp_tid].status==committed) ->
   /* committed version.  Return this in _r. */
   moveTransID(version[_r].owner, NIL); /* opportunistic free */
   moveVersion(version[_r].next, NIL); /* opportunistic free */
   break /* no need to look further. */
:: else ->
   assert (transid[_tmp_tid].status==aborted);
   /* unlink this useless version */
   CAS_Version(object[o].version, _r, version[_r].next, _)
   /* repeat */
fi
     fi
  od;
  moveTransID(_tmp_tid, NIL); /* free tmp transid copy */
  assert (ver!=NIL -> _r == NIL : 1)
}

inline ensureReader(tid, o, ver) {
  assert(tid!=NIL);
  /* make sure we're on the readerlist */
  ensureReaderList(tid, o)
  /* now kill any transactions associated with uncommitted versions, unless
   * the transaction is ourself! */
  findVersion(tid, o, ver);
  /* don't care about which committed version to use, at the moment. */
  moveVersion(_r, NIL);
}

/* per-object, before write. */
/* returns NIL in ver to indicate suicide. */
inline ensureWriter(tid, o, ver) {
  assert(tid!=NIL);
  /* Same beginning as ensureReader */
  ver = NIL; _r = NIL; _rr = NIL;
  do
  :: assert (ver==NIL);
     findVersion(tid, o, ver);
     if
```

```
     :: (ver!=NIL) -> break /* found a writable version for us */
     :: (ver==NIL && _r==NIL) ->
/* create and link a fully-committed root version, then
 * use this as our base. */
allocVersion(_retval, _r, NIL, NIL);
CAS_Version(object[o].version, NIL, _r, _cas_stat)
     :: else ->
_cas_stat = true
     fi;
     if
     :: (_cas_stat) ->
/* so far, so good. */
assert (_r!=NIL);
assert (version[_r].owner==NIL ||
transid[version[_r].owner].status==committed);
/* okay, make new version for this transaction. */
assert (ver==NIL);
allocVersion(_retval, ver, tid, _r);
/* want copy of committed version _r.  Race here because _r can be
 * written to under peculiar circumstances, namely: _r has
 * non-flag value, non-flag value is copied back to parent,
 * flag_value is written to parent -- this forces flag_value to
 * be written to committed version. */
/* IF WRITES ARE ALLOWED TO COMMITTED VERSIONS, THERE IS A RACE HERE.
 * But our implementation of false_flag writes at the moment does
 * not permit *any* writes to committed versions. */
version[ver].field[0] = version[_r].field[0];
version[ver].field[1] = version[_r].field[1];
assert(NUM_FIELDS==2); /* else ought to initialize more fields */
CAS_Version(object[o].version, _r, ver, _cas_stat);
moveVersion(_r, NIL); /* free _r */
if
:: (_cas_stat) ->
   /* kill all readers (except ourself) */
   /* note that all changes have to be made from the front of the
    * list, so we unlink ourself and then re-add us. */
   do
   :: moveReaderList(_r, object[o].readerList);
      if
```

```
      :: (_r==NIL) -> break
      :: (_r!=NIL && readerlist[_r].transid!=tid)->
 tryToAbort(readerlist[_r].transid)
      :: else
      fi;
      /* link out this reader */
      CAS_Reader(object[o].readerList, _r, readerlist[_r].next, _)
   od;
   /* okay, all pre-existing readers dead & gone. */
   assert(_r==NIL);
   /* link us back in. */
   ensureReaderList(tid, o);
   break
:: else
fi;
/* try again */
     :: else
     fi;
     /* try again from the top */
     moveVersion(ver, NIL)
  od;
  /* done! */
  assert (_r==NIL);
}
/* per-field, before read. */
inline checkReadField(o, f) {
  /* do nothing: no per-field read stats are kept. */
  skip
}
/* 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
     :: (_r==NIL) -> break /* done */
     :: else ->
object[o].fieldLock[f] = _thread_id;
if
/* 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. */
:: (object[o].field[f]==_val) ->
   if
   :: (object[o].version==_rr) ->
      atomic {
if
:: (object[o].fieldLock[f]==_thread_id) ->
   version[_r].field[f] = _val;
:: else -> break /* abort */
fi
      }
   :: 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
  :: (object[o].version==_rr) ->
     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
}
      :: else
      fi
      /* retry */
  od;
  /* clean up */
  moveVersion(_r, NIL);
  moveVersion(_rr, NIL);
}
```

Bibliography

[1] San Jose, California, Oct. 5–9 2002. ACM Press. [2 citations on pages 188 and 189.]

[2] J. Alemany and E. W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *PODC '92*, pages 125–134, Vancouver, British Columbia, Canada, Aug. 1992. [1 citation on page 144.]

[3] C. S. Ananian. The FLEX compiler project. http://flex-compiler.csail.mit.edu/. [2 citations on pages 33 and 63.]

[4] C. S. Ananian. The static single information form. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, Sept. 1999. [1 citation on page 69.]

[5] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA-11*, pages 316–327, San Francisco, California, Feb. 2005. [6 citations on pages 22, 28, 38, 109, and 128.]

[6] A. W. Appel. Simple generational garbage collection and fast allocation. *Software–Practice and Experience*, 19(2):171–183, Feb. 1989. [1 citation on page 89.]

[7] A. W. Appel and G. J. Jacobson. The world's fastest Scrabble program.

Communications of the ACM, 31(5):572–578, May 1988. [1 citation on page 26.]

[8] H. G. Baker. Shallow binding makes functional arrays fast. *ACM SIGPLAN Notices*, 26(8):145–147, Aug. 1991. [1 citation on page 102.]

[9] H. G. Baker, Jr. Shallow binding in Lisp 1.5. *Communications of the ACM*, 21(7):565–569, July 1978. [1 citation on page 102.]

[10] G. Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 261–270. ACM Press, June 1993. [1 citation on page 143.]

[11] A. Bensoussan, C. Clingen, and R. Daley. The Multics virtual memory: Concepts and design. *CACM*, 15(5):308–318, May 1972. [1 citation on page 119.]

[12] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 78–86. ACM Press, Sept. 1986. [3 citations on pages 29, 148, and 149.]

[13] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, Madison, Wisconsin, June 2005. [2 citations on pages 17 and 133.]

[14] H. Boehm, A. Demers, and M. Weiser. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/, 1991. [1 citation on page 70.]

[15] T.-R. Chuang. A randomized implementation of multiple functional arrays. In *LFP*, pages 173–184, June 1994. [3 citations on pages 91, 101, and 102.]

[16] C. Click, G. Tene, and M. Wolf. The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual Execution Environments*, pages 46–56, Chicago, Illinois, June 2005. [1 citation on page 129.]

[17] S. Cohen. Multi-version structures in Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 265–274, Nov. 1984. [1 citation on page 100.]

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 2nd edition, 2001. [1 citation on page 27.]

[19] Digital Equipment Corporation. *VAX MACRO and Instruction Set Reference Manual*, Nov. 1996. [1 citation on page 145.]

[20] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI '98*, pages 338–349, Montreal, Quebec, Canada, June 1998. [5 citations on pages 29, 150, and 151.]

[21] Freescale Semiconductor, Chandler, Arizona. *MPC7450 RISC Microprocessor Family Reference Manual, Rev. 5*, Jan. 2005. MPC7450UM. [1 citation on page 42.]

[22] J. E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, 2nd edition, July 2002. [1 citation on page 26.]

[23] The GNU Classpath project. `http://classpath.org/`. [1 citation on page 63.]

[24] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Prentice Hall PTR, 3rd edition, June 2005. [1 citation on page 78.]

[25] J. Gray. The transaction concept: Virtues and limitations. In *VLDB*, pages 144–154, Sept. 1981. [1 citation on page 145.]

[26] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. [1 citation on page 145.]

[27] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *USENIX OSDI '96*, pages 123–136, Oct. 1996. [2 citations on pages 15 and 28.]

[28] D. Grunwald and S. Ghiasi. Microarchitectural denial of service: insuring [sic] microarchitectural fairness. In *MICRO-35*, pages 409–418, Istanbul, Turkey, Nov. 2002. [1 citation on page 132.]

[29] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA 31*, pages 102–113, München, Germany, June 2004. [1 citation on page 146.]

[30] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03*, pages 388–402, Anaheim, California, Oct. 2003. [5 citations on pages 17, 28, 39, 82, and 147.]

[31] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press. [1 citation on page 23.]

[32] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996. [4 citations on pages 41, 57, 60, and 61.]

[33] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, Jan. 1991. [1 citation on page 142.]

[34] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM TOPLAS*, 15(5):745–770, Nov. 1993. [2 citations on pages 95 and 143.]

[35] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03*, pages 92–101, Boston, Massachusetts, July 2003. [4 citations on pages 15, 16, and 146.]

[36] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA 20*, pages 289–300, San Diego, California, May 1993. [11 citations on pages 15, 16, 28, 36, 39, 107, 112, 117, 145, and 147.]

[37] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *PODC '88*, pages 276–290, Toronto, Ontario, Canada, Aug. 1988. [3 citations on pages 15, 141, and 142.]

[38] M. P. Herlihy. The transactional manifesto: Software engineering and non-blocking synchronization. Invited talk at PLDI, June 2005. http://research.ihost.com/pldi2005/manifesto.pldi.ppt. [2 citations on pages 21 and 23.]

[39] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, Providence, Rhode Island, May 2003. [2 citations on pages 15 and 142.]

[40] M. P. Herlihy and J. E. B. Moss. Transactional support for lock-free data structures. Technical Report 92/07, Digital Cambridge Research Lab, Dec. 1992. [2 citations on pages 120 and 145.]

[41] G. J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003. [1 citation on page 43.]

[42] IBM, Austin, Texas. *PowerPC Virtual Environment Architecture, Book II, Version 2.02*, Jan. 2005. [1 citation on page 42.]

[43] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, LLNL, Livermore, California, Nov. 1987. [1 citation on page 145.]

[44] M. Jones. What really happened on Mars? http://research.microsoft.com/~mbj/Mars_Pathfinder/, Dec. 1997. [1 citation on page 15.]

[45] E. Jul and B. Steensgaard. Implementation of distributed objects in Emerald. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 130–132, Palo Alto, CA, Oct. 1991. IEEE. [2 citations on pages 29 and 148.]

[46] T. Knight. An architecture for mostly functional languages. In *LFP*, pages 105–112. ACM Press, 1986. [5 citations on pages 15, 112, 117, 120, and 145.]

[47] A. LaMarca. A performance evaluation of lock-free synchronization protocols. In *PODC '94*, pages 130–140, Los Angeles, CA, Aug. 1994. [1 citation on page 144.]

[48] L. Lamport. Concurrent reading and writing. *CACM*, 20(11):806–811, Nov. 1977. [2 citations on pages 15 and 141.]

[49] R. B. Lee. Precision architecture. *IEEE Computer*, 22(1):78–91, Jan. 1989. [1 citation on page 119.]

[50] J. Manson and W. Pugh. Semantics of multithreaded Java. Technical Report UCMP-CS-4215, Department of Computer Science, University of Maryland, College Park, Jan. 2002. [2 citations on pages 34 and 147.]

[51] J. F. Martinez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X* [1], pages 18–29. [1 citation on page 145.]

[52] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, New York, NY 10027, June 1991. [4 citations on pages 15, 28, 137, and 142.]

[53] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed packet switching for local computer networks. *CACM*, 19(7):395–404, July 1976. [1 citation on page 117.]

[54] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96*, pages 267–276, Philadelphia, PA, May 1996. [1 citation on page 23.]

[55] C. Okasaki. Purely functional random-access lists. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 86–95. ACM Press, June 1995. [1 citation on page 100.]

[56] M. E. O'Neill and F. W. Burton. A new method for functional arrays. *J. Func. Prog.*, 7(5):487–514, Sept. 1997. [3 citations on pages 91, 100, and 102.]

[57] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO-34*, pages 294–305, Austin, Texas, Dec. 2001. [2 citations on pages 124 and 145.]

[58] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In Rajwar [1], pages 5–17. [2 citations on pages 15 and 145.]

[59] M. C. Rinard. Implicitly synchronized abstract data types: Data structures for modular parallel programming. *Journal of Programming Languages*, 6:1–35, 1998. [1 citation on page 22.]

[60] A. Rudys and D. S. Wallach. Transactional rollback for language-based systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN)*, pages 439–448. IEEE Computer Society, June 2002. [1 citation on page 146.]

[61] D. J. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In *SOSP '97*, pages 157–169, Oct. 1997. [1 citation on page 42.]

[62] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *CACM*, 10(8):501–506, Aug. 1967. [1 citation on page 24.]

[63] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95*, pages 204–213, Ottawa, Ontario, Canada, Aug. 1995. [3 citations on pages 15, 28, and 146.]

[64] D. Siewiorek, G. Bell, and A. Newell. *Computer Structures: Principles and Examples*. McGraw-Hill, 1982. [1 citation on page 145.]

[65] J. M. Stone, H. S. Stone, P. Heidelberg, and J. Turek. Multiple reservations and the oklahoma update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, Nov. 1993. [2 citations on pages 15 and 145.]

[66] I. Sun Microsystems. Java native interface. `http://java.sun.com/j2se/1.5.0/docs/guide/jni/`, 2004. [3 citations on pages 69, 71, and 72.]

[67] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, NJ, 1992. [1 citation on page 14.]

[68] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA '99*, pages 187–206, Denver, Nov. 1999. [1 citation on page 89.]

[69] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanović. Direct addressed caches for reduced power consumption. In *MICRO-34*, Austin, Texas, Dec. 2001. [2 citations on pages 43 and 77.]

[70] S. Zakhour, S. Hommel, J. Royal, I. Rabinovitch, T. Risser, and M. Hoeber. *The Java Tutorial: A Short Course on the Basics.* Prentice Hall PTR, 4th edition, Sept. 2006. [1 citation on page 81.]

[71] L. Zhang. UVSIM reference manual. Technical Report UUCS-03-011, University of Utah, Mar. 2003. [1 citation on page 120.]

[72] Y. Zibin, A. Potanin, S. Artzi, A. Kieżun, and M. D. Ernst. Object and reference immutability using Java generics. Technical Report MIT-CSAIL-TR-2007-018, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, March 16, 2007. [1 citation on page 89.]

[73] C. Zilles and D. H. Flint. Challenges to providing performance isolation in transactional memories. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, Madison, Wisconsin, June 2005. [2 citations on pages 131 and 132.]

Index