declarations for the components. The runtime system orders the lock acquisitions to prevent deadlock. This process suffices for conservative mutual exclusion, but it is limited in its ability to express alternative operation orders that preserve atomicity of the composed operation while overlapping its components.

Transactions do not suffer from the composability problem [31]. Because transactions only specify the atomicity properties, not the locks required, the programmer's job is eased and implementations are free to order operations in any way that preserves atomicity.

2.1.3 Create a thread-safe double-ended queue

Herlihy suggests creating a thread-safe double-ended queue using locks as "sadistic homework" for computer science students [38]. Although double-ended queues are a simple data structure, creating a scalable locking protocol is a non-trivial exercise: one wants dequeue and enqueue operations to complete concurrently when the ends of the queue are "far enough" apart, while safely handling the interference in the small-queue case. In fact, the solution to this assignment was a publishable result, as Michael and Scott demonstrated in 1996 [54].

The simple "one lock" solution to the double-ended queue problem, ruled out as unscalable in the locking case, is scalable and efficient for non-blocking transactions.

2.1.4 Handle asynchronous exceptions

Properly handling asynchronous events is difficult with locks, because it is impossible to safely go off to handle the event while holding an arbitrary set of locks—and it is impossible to safely drop the locks. The solution implemented in the Real-Time Specification for Java and similar systems is to generally forbid asynchronous events within locked regions, allowing the programmer to explicitly specify certain points within the region at which

```
int expr() {
   try {
     return sum();
   } else {
     return difference();
   }
}
int sum() {
   int a = number();
   eat(ADD);
   int b = number();
   return a+b;
}
int difference() {
   int a = number();
   eat(MINUS);
   int b = number();
   return a-b;
}
```

Figure 2.2: A simple backtracking recursive-decent parser.

succeed or not. This can be straight-forwardly implemented with a transaction around the traversal, always initially attempting the `try`. Exceptions or faults cause the transaction to abort; when it does so all the heap side-effects of the `try` block disappear.

2.2.2 Backtracking search

Introducing an explicit `fail` statement allows us to use the same `try/else` to facilitate backtracking search. Backtracking search is used to implement practical[1] regular expressions, parsers, logic programming languages, Scrabble-playing programs [7], and (in general) any problem with multiple solutions or multiple solution techniques.

As a simple example, let us consider a recursive-decent parser such as that shown in Figure 2.2. We don't know whether to apply the `sum()` or `difference()` production until after we've parsed some common left prefix. We can use backtracking to attempt one rule (`sum`) and `fail` out of it

[1] As opposed to the limited regular expressions demonstrated in theory classes which are always neatly compiled to deterministic finite automata [22].

inside the `eat()` method, in the process undoing any data structure updates performed on this path, and then attempt the other possible production.

2.2.3 Network flow

Let's now turn our attention now to parallel codes, the more conventional application of transaction systems. Consider a serial program for computing network flow (see, for example, [18, Chapter 26]). The inner loop of the code pushes flow across an edge by increasing the "excess flow" on one vertex and decreasing it by the same amount on another vertex. One might see the following Java code:

```
void pushFlow(Vertex v1, Vertex v2, double flow) {
  v1.excess += flow; /* Move excess flow from v1 */
  v2.excess -= flow; /* to v2.                   */
}
```

To parallelize this code, one must preclude multiple threads from modifying the excess flow on those two vertices at the same time. Locks provide one way to enforce this mutual exclusion:

```
void pushFlow(Vertex v1, Vertex v2, double f) {
  Object lock1, lock2;
  if (v1.id < v2.id) {       /* Avoid deadlock. */
    lock1 = v1; lock2 = v2;
  } else {
    lock1 = v2; lock2 = v1;
  }
  synchronized(lock1) {
    synchronized(lock2) {
      v1.excess += f; /* Move excess flow from v1 */
      v2.excess -= f; /* to v2.                   */
    } /* unlock lock2 */
  } /* unlock lock1 */
}
```

This code is surprisingly complicated and slow compared to the original. Space for each object's lock must be reserved. To avoid deadlock, the code must acquire the locks in a consistent linear order, resulting in an unpredictable branch in the code. In the code shown, I have required the programmer to insert an `id` field into each vertex object to maintain a

total ordering. The time required to acquire the locks may be an order of magnitude larger than the time to modify the excess flow. What's more, all of this overhead is rarely needed! For a graph with thousands or millions of vertices, the number of threads operating on the graph is likely to be less than a hundred. Consequently, the chances are quite small that two different threads actually conflict. Without the locks to implement mutual exclusion, however, the program would occasionally fail.

Software transactions (and some language support) allow the programmer to parallelize the original code using an `atomic` keyword to indicate that the code block should appear to execute atomically:

```
void pushFlow(Vertex v1, Vertex v2, double flow) {
  atomic { /* Transaction begin. */
    v1.excess += flow; /* Move excess flow from v1 */
    v2.excess -= flow; /* to v2.                   */
  } /* Transaction end. */
}
```

This `atomic` region can be implemented as a transaction, and with an appropriately non-blocking implementation, it will scale better and execute faster than the locking version [5, 30, 27, 52, 36, 63]. From the programmer's point of view, I have also eliminated the convoluted locking protocol which must be observed rigorously everywhere the related fields are accessed if deadlock and races are to be avoided.

Further, I can implement `atomic` using the `try/else` exception-handling mechanism I have already introduced:

```
for (int b=0; ; b++) {
  try {
    // atomic actions
  } else {
    backOff(b);
    continue;
  }
  break; // success!
}
```

I non-deterministically choose to execute the body of the `atomic` block if and only if it will be observed by all to execute atomically. The same lin-

guistic mechanism I introduced for fault tolerance and backtracking provides atomic regions for synchronization as well.

2.2.4 Bug fixing

The existing *monitor synchronization* methodology for Java, building on such features in progenitors such as Emerald [12, 45],[2] implicitly associates an lock with each object. Data races are prevented by requiring a thread to acquire an object's lock before touching the object's shared fields. However, the lack of races is not sufficient to prevent unanticipated parallel behavior.

Flanagan and Qadeer [20] demonstrated this insufficiency with an actual bug they discovered in the Sun JDK 1.4.2 Java standard libraries. The `java.lang.StringBuffer` class, which implements a mutable string abstraction, is implemented as follows:

```
public final class StringBuffer ... {
  private char value[];
  private int count;
  ...
  public synchronized
  StringBuffer append(StringBuffer sb) {
    ...
A:  int len = sb.length();
    int newcount = count + len;
    if (newcount > value.length)
      expandCapacity(newcount);
    // next statement may use stale len
B:  sb.getChars(0, len, value, count);
    count = newcount;
    return this;
  }
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }
}
```

The library documentation indicates that the methods of this class are meant to execute atomically, and the `synchronized` modifiers on the methods are meant to accomplish this.

[2] See chapter 8.4.

However, the `append()` method is *not* atomic. Another thread may change the length of the parameter `sb` (by adding or removing characters) between the call to `sb.length()` at label A and the call to `sb.getChars(...)` at label B. This non-atomicity may cause incorrect data to be appended to the target or a `StringIndexOutOfBoundException` to be thrown. Although the calls to `sb.length()` and `sb.getChars()` are individually atomic, they do not compose to form an atomic implementation of `append()`.

Note that replacing `synchronized` with `atomic` in this code gives us the semantics we desire: the atomicity of nested `atomic` blocks is guaranteed by the atomicity of the outermost block, ensuring that the entire operation appears atomic.

Both the network flow example and this `StringBuffer` example require synchronization of changes to more than one object. Monitor synchronization is not well-suited to this task. Atomic regions implemented with transactions can be used to simplify the locking discipline required to synchronize multi-object mutation and provide a more intuitive specification for the desired concurrent properties. Further, the `StringBuffer` example shows that simply replacing `synchronized` with `atomic` provides a alternative semantics that may in fact correct existing synchronization errors. For many Java programs, the semantics of `atomic` and `synchronized` are identical; see chapter 7.3.

2.3 Some things we still can't (easily) do

The transaction mechanism presented here is not a universal replacement for all synchronization. In particular, transactions cannot replace mutual exclusion required to serialize I/O, although the needed locks can certainly be built with transactions. The challenge of integrating I/O within a transactional context is discussed in chapter 7.4. However, large programs—the Linux kernel, for example—have been written such that locks are never held across context switches or I/O operations. Transactions provide a complete

solution for this limited synchronization.

Blocking producer-consumer queues, or other options that require a transaction to wait upon a condition variable, may introduce complications into a transaction system: transactions cannot immediately retry when they fail, but instead must wait for some condition to become true. Chapter 4.4.4 describes some solutions to this problem, ranging from naïve (keep retrying and aborting with exponential backoff until the condition is finally true) to clever; the clever solutions require additional transaction machinery.

Easy things should stay easy,
hard things should get easier,
and impossible things should get
hard.

Motto for Perl 6 development

Chapter 3

Designing a software transaction system

In this chapter I will detail the design of a high-performance software transaction system. I will first present our methodology for isolating likely transactions from our benchmarks. I will then describe the system goals, and use quantitative data from our benchmarks as I explain the design choices. Then I will formalize an implementation meeting those goals in the modeling language Promela. The correctness of this implementation can be model-checked using the SPIN tool; the details of this effort are in Appendix A.

After outlining the design in this chapter, chapter 4 will discuss a practical implementation.

3.1 Finding transactions

One of the difficulties of proposing a novel language feature is the lack of benchmarks for its evaluation. Although there is no existing body of code that uses transactions, there is a substantial body of code that uses Java (locking) synchronization. This thesis will utilize the FLEX compiler [3] to substitute `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: 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 a transaction expressed with an `atomic` block.[1] Despite the differences in semantics, the automatic substitution of `atomic` for `synchronized` does, in fact, preserve the correctness of the benchmarks I examine here.

The initial results of this chapter explore the implications of exposing the transaction mechanism to user-level code through a compiler. I compiled the SPECjvm98 benchmark suite with the FLEX Java compiler, modified to turn synchronized blocks and methods into transactions, in order to investigate the properties of the transactions in such "automatically converted" code. FLEX performed method cloning to distinguish methods called from within a transactions, and implemented nested locks as a single transaction around the outermost.[2] I instrumented this transformed program to produce a trace of memory references and transaction boundaries for analysis. I found both large transactions (involving millions of cache lines) and frequent transactions (up to 45 million of them).

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

Figure 3.1 shows the raw sizes and frequency of transactions in the transactified SPECjvm98 suite. Figure 3.2 proposes a taxonomy for Java applications with transactions, grouping the SPECjvm98 applications into quad-

[1] The proposed revision of the Java memory model [50] narrows the semantic gap, however I do not plan to treat `volatile` fields in this work. See chapter 7.3 for more details.

[2] See chapter 4.2 for more details on this transformation.

program	total memory ops	transactions	transactional memory ops	biggest transaction
201_compress	2,981,777,890	2,272	<0.1%	2,302
202_jess	405,153,255	4,892,829	9.1%	7,092
205_raytrace	420,005,763	4,177	1.7%	7,149,099
209_db	848,082,597	45,222,742	23.0%	498,349
213_javac	472,416,129	668	99.9%	118,041,685
222_mpegaudio	2,620,818,169	2,991	<0.1%	2,281
228_jack	187,029,744	12,017,041	34.2%	14,266

Figure 3.1: 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.

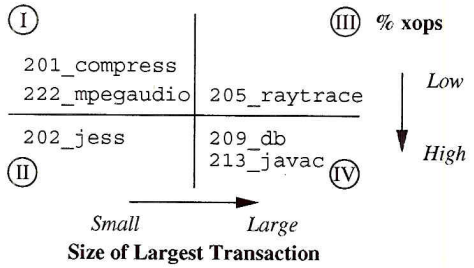

Figure 3.2: Classification of SPECjvm98 benchmarks into quadrants based on transaction properties.

rants based on the number and size of the transactions that they perform. Applications in Quads II and IV require an efficient transaction implementation, because they contain many transactional operations. Quads III and IV contain at least some very large transactions, which pose difficulties for currently-proposed hardware transactional memory schemes. We will now examine the benchmarks in each quadrant to determine why its program logic caused it to be classified in that quadrant.

Quad I applications perform few (up to about 2000) small transactions. These applications include 201_compress, an implementation of gzip compression, and 222_mpegaudio, an MP3 decoder. Both of these applications perform inherently serial tasks. They perform quite well with locks, and would likely execute with acceptable performance even with a naïve software implementation of transactions, as long as the impact on non-transactional operations was minimal.

Quad II applications perform a large number of small transactions. The expert system 202_jess falls in this category, as do small input sizes of 209_db, a database. These benchmarks perform at least an order of magnitude more transactions than Quad I applications, and all of the transactions are small enough to comfortably fit the known hardware transactional memory schemes [36, etc], if one were to be implemented.

Quad III includes 205_raytrace, a ray-tracing renderer. A small number of transactions are performed, but they may grow very large. Existing bounded hardware transactional schemes will not suffice. The large transactions may account for a large percentage of total memory operations, which may make software schemes impractical.

Finally, Quad IV applications such as 209_db and the 213_javac Java compiler application perform a large number of transactional memory operations with at least a few large transactions.

The 213_javac Java compiler application and the large input size of the 209_db benchmark illustrate that some programs contain *extremely* large transactions. When 213_javac is run on its full input set, it contains 4

very large transactions, each of which contains over 118 million transactional memory operations. Closer examination reveals that the method `Javac.compile()`, which implements the entire compilation process, is marked as synchronized: the programmer has explicitly requested that the entire compilation occur atomically.

The large transactions in Quad III and IV may be, as in this case, a result of overly-coarse–grained locking, but our goal is to relieve the programmer from the burden of specifying correct atomic regions of smaller granularity. Performance may benefit from narrowing the atomic regions, but execution with coarse regions should be possible and not prohibitively slow.

The 209_db benchmark suffers from a different problem: at one point the benchmark atomically scans an index vector and removes an element, creating a potentially large transaction if the index is large. The size of this index is correlated in these benchmarks with the input size, but it need not be: a large input could still result in a small index, and (to some degree) vice-versa.

A similar situation arises in the `java.lang.StringBuffer` code shown in chapter 2.2.4: a call to the synchronized `sb.getChars()` method means that the size of the transaction for this method will grow like the length of the parameter `sb`. In other words, the transaction can be made arbitrarily large by increasing the length of `sb`; or, equivalently, there is no bound on transaction size without a bound on the size of the string `sb`.

Any scheme that allows the programmer free reign over specifying desired 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. Object-based transaction systems can asymptotically approach that efficiency for very long-lived transactions; the existence of such is shown in Figure 3.3, which plots the distribution of transaction sizes in SPECjvm98 on a semi-log

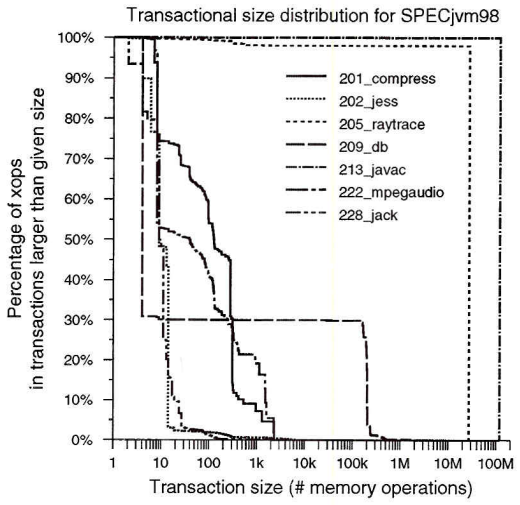

Figure 3.3: Distribution of transaction size in the SPECjvm98 benchmark suite. Note that the x-axis uses a logarithmic scale.

scale.

These initial results indicate that real applications can be transactified with modest effort, yielding significant gains in concurrency. In other work [5] we have shown that a factor of 4 increase in concurrency can be obtained by doing nothing more than converting locks to transactions. Since the transactified applications may contain large transactions, proposed hardware support for transactions is inadequate.

3.2 Designing efficient transactions

In this section I briefly describe some desired properties of the software transaction system of this thesis. Where possible I will justify these desiderata using quantitative data obtained from analyses of the SpecJVM98 benchmarks, which I implemented using the Flex Java compiler framework.

3.2.1 Weak vs. strong atomicity

As previously discussed in chapter 1.4, ***strongly atomic*** transaction systems protect transactions from interference from "non-transactional" code, while ***weakly atomic*** transaction systems do not afford this protection. Consider unsynchronized code directly altering the length field of `StringBuffer` (chapter 2.2.4). In a weakly atomic system this will cause the `atomic` `StringBuffer.append()` method to appear non-atomic. Current software transaction systems implement only weak atomicity, because of the assumed expense of implementing strong atomicity. The system we will design in this chapter will achieve strong atomicity without adding excessive overhead, so that correct operation is assured even despite concurrent non-transactional operations on locations involved in a transaction.

3.2.2 Object-oriented vs. flat TM

This transaction system, unlike most current proposals [30, 36] (including the hardware system presented in chapter 6), uses an "object-oriented" design. Much contemporary research is focused on implementing a flat (transactional) memory abstraction in software, primarily because this solves some of the issues I will present in chapter 5. However, the object-oriented approach offers several benefits:

Efficient execution of long-running transactions. As discussed briefly in Chapter 8.2 and 8.3, flat word-oriented transaction schemes require overhead proportional to the number of words read/written in the transaction, even if these locations had been accessed before inside the transaction. Object-oriented schemes impose a cost proportional to the number of *objects* touched by the transaction — but once the cost of "opening" those objects is paid, the transaction can continue to work indefinitely upon those objects without paying any further penalty. Object-oriented schemes are thus seen to be more efficient for *long-running transactions.*

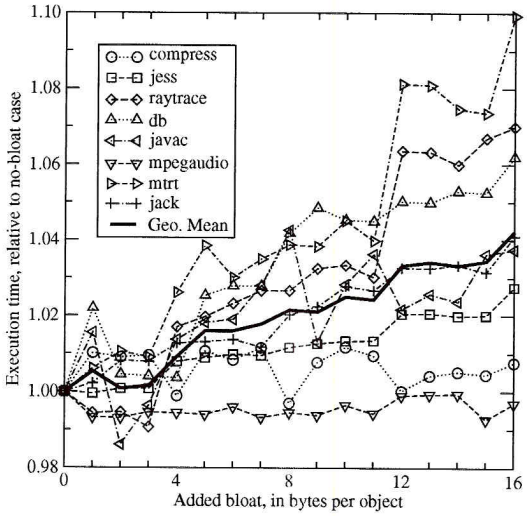

Figure 3.4: Application slowdown with increasing object bloat for the SPECjvm98 benchmark applications.

Preservation of optimization opportunities. Furthermore, transaction-local objects can be identified (statically or dynamically) and creation/updates to these objects can be done without any transaction tax at all. Word-oriented schemes discard the high-level information required to implement these optimizations.

I believe that the problems with previous object-oriented schemes can be solved while preserving the inherent benefits of an object-oriented approach, and the current thesis presents one such solution.

3.2.3 Tolerable limits for object expansion

I will need to add some additional information to each object to track transaction state. I measured the slowdown caused by various amounts of object "bloat" to determine reasonable bounds on the size of this extra information. Figure 3.4 presents these results for the SPECjvm98 applications; I determined that two words (eight bytes) of additional storage per object

program	transactional memory ops	transactional stores %
201_compress	50,029	26.2%
202_jess	36,701,037	0.6%
205_raytrace	7,294,648	23.2%
209_db	195,374,420	6.3%
213_javac	472,134,289	22.9%
222_mpegaudio	41,422	18.6%
228_jack	63,912,386	17.0%

Figure 3.5: Comparison of loads and stores inside transactions for the SPECjvm98 benchmark suite, full input runs.

would not impact performance unreasonably. This amount of bloat causes a geometric mean of 2% slowdown on these benchmarks.

3.2.4 Designing for reads vs. writes

I also measured the number and types of reads and writes for the SpecJVM98 benchmarks. Figure 3.5 shows that transactional reads typically outnumber transactional writes by at least 4 to 1; in some cases reads outnumber writes by over 100 to 1.[3] It is worthwhile, therefore, to make reads more efficient than writes. In particular, since the flag-overwrite technique discussed in chapter 3.2.5 requires us to allocate additional memory to store the "real" value of the field, I wish to avoid this process for transactional reads, reserving the extra allocation effort for transactional writes.

3.2.5 The big idea: waving FLAGs

I would like non-transactional code to execute with minimal overhead, but transactions should still appear atomic to non-transactional code. My basic

[3]The typical ratio roughly matches the 3:1 average observed in Hennessy and Patterson [32, pp. 105, 379].

mechanism is loosely based on the distributed shared memory implementation of Scales and Gharachorloo [61]. I 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.

Note that this technique explicitly allows safe access to fields involved in a transaction from non-transactional code, which is another design goal of the system. This eases "transactification" of legacy code (but see chapter 7.3!).

3.3 Specifying the basic mechanism

I now present an algorithm that has these desired properties. My 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.

I will implement the synchronization required by my algorithm using load-linked/store-conditional instructions. I 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 many chips in the PowerPC processor family, although it has been deprecated in version 2.02 of the PowerPC architecture standard.[4] This disjoint location

[4] Version 2.02 of the PowerPC architecture standard says, "If a reservation exists but the storage location specified by the `stwcx.` is not the same as the location specified by the *Load And Reserve* instruction that established the reservation... it is undefined whether [the operand is] stored into the word in storage addressed by [the specified effective address]" and states that the condition code indicating a successful store is also undefined in this circumstance [42, p 25]. The user manual for the MPC7447/7457 ("G4") PowerPC chips states, however, "The `stwcx.` instruction does not check the reservation for a matching address. The `stwcx.` instruction is only required to determine whether a reservation exists. The `stwcx.` instruction performs a store word operation only if the reservation exists," [21, Chapter 3.3.3.6] which is the behavior we require. I believe

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.

I will describe my algorithms in the Promela modeling language [41], which I 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 given in Appendix A, along with a brief primer on Promela syntax and semantics.

3.3.1 Object structures

Figure 3.6 illustrates the basic data structures of my 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 my implementation, I have chosen the byte `0xCA` to be our flag value, repeated as necessary to fill out the width of the appropriate type. The semantic value of an object field is the value in the original object

version 1.10 of the PowerPC Architecture specification required this behavior, although I have not been able to confirm this. The Cell architecture specification follows version 2.02 of the PowerPC specification, although it adds cache-line reservation operations that can also be used to implement our algorithm for reasonably-sized objects aligned within cache lines; see [69] for an implementation of Java I created that observes the appropriate alignments.

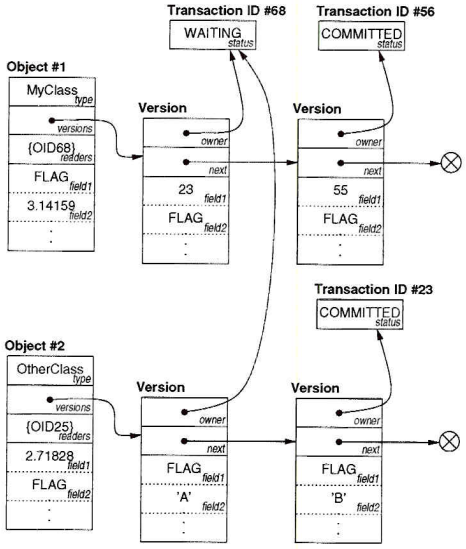

Figure 3.6: Implementing software transactions with version lists. A transaction object consists of a single field *status*, which indicates if it has COMMITTED, ABORTED, or is WAITING. Each object contains two extra fields: *readers*, a singly-linked list of transactions that have read this object; and *versions* a linked list of version objects. If an object field is FLAG, then the value for the field is obtained from the appropriate linked version object.

```
#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];
};
typedef ReaderList {
  byte transid;
  byte next;
};
mtype = { waiting, committed, aborted };
typedef TransID {
  mtype status;
};
```

Figure 3.7: Declaring objects with version lists in Promela. Note that we are using the `byte` datatype to encode pointers. The `fieldLock` field assists in the implementation of the load-linked/store-conditional pair of operations in Promela.

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.

Figure 3.7 declares these object structures in Promela.

3.3.2 Operations

I 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: my algorithm requires no complicated processing, delay, roll-back or validate procedure to commit or abort a transaction.

Note that we could support non-transactional read and write (that is, reads and writes that take place outside of any transaction) by creating a new very short transaction that encloses only the single read or write. Non-transactional accesses to objects can be very frequent, however, so I provide more efficient implementations with the same semantics.

I will present the operations one-by-one.

Non-transactional 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`, we copy back the field value from the most-recently committed transaction (aborting all other transactions) and try again. The copy-back procedure will notify us if this is a "false flag", in which case the value of this field really is `FLAG`. We pass the `kill_writers` constant to the copy-back procedure to indicate that only transactional writers need be aborted, not transactional readers. All possible races are confined to the copy-back procedure, which I will describe on page 48.

The top of Figure 3.8 specifies the non-transactional read operation in Promela.

Non-transactional write

The `writeNT` function does a non-transactional write of new value `nval` to field `f` of object `o`. For correctness, we need to ensure that the reader list is empty before we do the write. I implement this with a load-linked/store-conditional pair, which is modelled in Promela slightly differently, ensuring

```
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)
     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;
}
```

Figure 3.8: Promela specification of non-transactional read and write operations.

that our write only succeeds so long as the reader list remains empty.[5] If it is not empty, we 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.

The bottom of Figure 3.8 specifies the non-transactional write operation in Promela.

Field Copy-Back

Figure 3.9 presents the field copy-back routine. We create a new version owned by a pre-aborted transaction which serves as a reservation on the head of the version list. We then write to the object field with a load-linked/store-conditional pair if and only if our version is still at the head of the versions list.[6] This addresses the major race possible in this routine.

Transactional Read

A transactional read is split into two parts. Before the read, we must ensure that our transaction is on the reader list for the object. This is straightforward to do in a non-blocking manner as long as we always add ourselves to the head of the list. We must also walk the versions list, and abort any uncommitted transaction other than our own. These steps can be combined

[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.

```
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
     :: 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 { /* LL/SC */
             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) ->
     killAllReaders(o, _r); /* see Appendix for details */
  :: else /* no more killing needed. */
  fi;
  /* done */
}
```

Figure 3.9: The field copy-back routine.

and hoisted so that they are done once before the first read from an object and not repeated.

At read time, we initially read from the original object. If the value read is not FLAG, we use it. Otherwise, we look up the version object associated with our transaction (this will typically be at the head of the version list) and read the appropriate value from that version. Note that the initial read-and-check can be omitted if we know that we have already written to this field inside this transaction.

The top of Figure 3.10 specifies the transactional read operation in Promela.

Transactional Write

Again, writes are split in two. Once for each object we must traverse the version list, aborting other versions and locating or creating a version corresponding to our transaction. We must also traverse the reader list, aborting all transactions on the list except ourself. This is shown in the `ensureWriter` routine in Figure 3.11.

Once for each field we intend to write, we must perform a copy-through: copy the object's 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 another copy-through. The `checkWriteField` routine is shown in Figure 3.12.

Then, for each write, we simply write to the identified version, as shown at the bottom of Figure 3.10.

3.4 Performance limits for non-transactional code

The software transaction system outlined in this section depends on the insertion of simple read and write checks in non-transactional code. The performance of read and write barriers have been well-studied in the garbage

```
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 ->
           findVersion(tid, o, ver);
           if
           :: (ver==NIL) ->/*use val from committed vers.*/
              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;
}
```

Figure 3.10: Promela specification of transactional read and write operations.

```
/* per-object, before write. */
inline ensureWriter(tid, o, ver) {
  assert(tid!=NIL);
  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.  No race because
         * we never write to a 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);
}
```

Figure 3.11: The per-object version-setup routine for transactional writes.

```
/* 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);
}
```

Figure 3.12: The per-field copy-through routine for transactional writes.

collection community, but our checks are slightly different: in particular, they are checks on the *contents* of a memory cell, rather than on its address. This introduces a more direct dependency which may affect performance. Further, our write check involves a LL/SC instruction pair, which may behave differently from the standard loads and stores used in barriers.

In this section we use a simple counter microbenchmark to evaluate the "best worst case" non-transactional performance of our transaction system. It is an idealized "best case" in that we won't benchmark the effects of "false flags" or other forays into the transactional code path, nor will we account for double-word or sub-word writes, cache effects due to code duplication, or other details of a real implementation. These effects we will investigate with full benchmarks in the chapter 4.5. We will also feel free to optimize down to the assembly level to determine our fundamental performance limits. However, the tight read/write dependency of a counter increment makes it a "worst case" benchmark: in general modern compilers are very good at separating reads from writes in "real" code to mask load latency, but our particular microbenchmark can not be reasonably unrolled to accomplish this separation. Our conclusions about the fundamental costs of read and write costs should thus be tempered by the knowledge that some of these costs are in practice be masked by the same standard compiler techniques used to mitigate memory access latencies.

Figure 3.13 presents the basic structure of the counter microbenchmark. The `read()` and `write()` methods will have appropriate definitions inlined for each variant of the benchmark. Note that the `readerList` and `field` fields of the `struct oobj` are marked `volatile` to prevent the C compiler from optimizing away the accesses we are interested in benchmarking; without these declarations `gcc` 4.1.2 will optimize away the entire benchmark loop and replace it with a direct addition of `REPETITIONS`.

Figure 3.14 shows the baseline `read()` implementation, which inlines to a single `lwz` instruction on PowerPC, along with the C implementation of the read check necessary for non-transactional code under our transaction

```
typedef int32_t field_t;

struct oobj {
  struct version *version;
  struct readerList * volatile readerList;
  volatile field_t field[NUM_FIELDS];
};
void do_bench(struct oobj *obj) {
  int i;
  for (i=0; i<REPETITIONS; i++) {
    field_t v = read(obj, 0);
    v++;
    write(obj, 0, v);
  }
}
```

Figure 3.13: Counter microbenchmark to evaluate read- and write-check overhead for non-transactional code.

```
#if !defined(WITH_READ_CHECKS)

static inline field_t read(struct oobj *obj, int idx) {
  field_t val = obj->field[idx];
  return val;
}

#else

static inline field_t read(struct oobj *obj, int idx) {
  field_t val = obj->field[idx];
  if (unlikely(val==FLAG))
    return unusualRead(obj,idx);
  else return val;
}

#endif
```

Figure 3.14: C implementation of read checks for counter microbenchmark.

```
#if !defined(WITH_WRITE_CHECKS)

static inline void write(struct oobj *obj, int idx, field_t val) {
  obj->field[idx] = val;
}

#else

static inline void write(struct oobj *obj, int idx, field_t val) {
  if (unlikely(val==FLAG))
    unusualWrite(obj,idx,val); // never called
  else {
    do {
      if (unlikely(NULL != LL(&(obj->readerList)))) {
        unusualWrite(obj,idx,val); // never called
        break;
      }
    } while (unlikely(!SC(&(obj->field[idx]), val)));
  }
}

#endif
```

Figure 3.15: C implementation of write checks for counter microbenchmark.

system. We use the `unlikely()` macro, implemented using the `gcc` extension `__builtin_expect()`, to apply the appropriate static prediction bits indicating that `FLAG` is expected to be an unusual value. In our microbenchmark, this prediction will always correct. The C compiler must still save whatever registers are necessary to allow the call to `unusualRead()`, although this function is never actually called during the microbenchmark. The `unusualRead` function would be expected to perform the remainder of the `readNT` algorithm from Figure 3.8, including looking up and copying back the most recently-committed value for this field.

Figure 3.15 shows the equivalent implementation pairs for a non-transactional write. The basic implementation inlines to a single `stw` instruction. The write-check version at the bottom must perform three tests. First, it must ensure that the value to be written is not `FLAG`: if it is we must use the "false flag" mechanism to write it, modeled here by a call to `unusualWrite()`. This

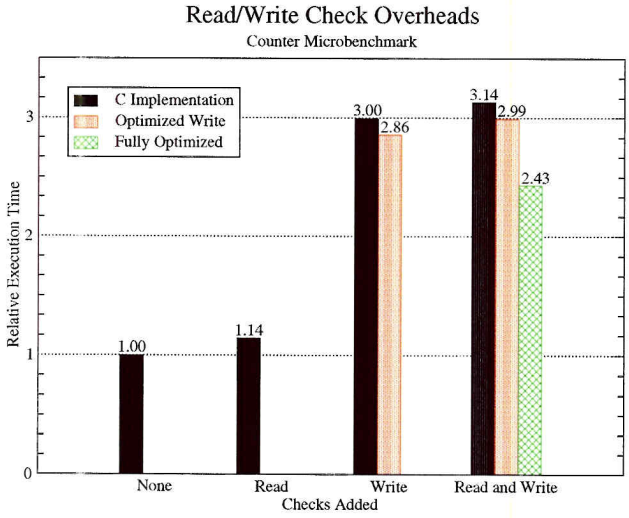

Figure 3.16: Time overhead of read checks, write checks, and both read and write checks for non-transactional code, in both a pure C implementation and optimized assembly.

check can be optimized away when writing a constant,[7] but the `volatile` declarations ensure that our benchmark will always perform the check. Second, we must perform a load-linked of the object's `readerList` to check that there are not any current transactional readers of this object. Last we perform a store-conditional of the value we wish to write, and check that it was successful. Unlike the other two tests, this check will fail a number of times[8] during the benchmark run due to context switches that may occur between the load-linked and the store-conditional instructions.

The performance of this benchmark is shown in Figure 3.16. Read checks, even in this worst case where the value read is demanded imme-

[7] Hennessy and Patterson indicate that 35% of integer instructions use an immediate operand; their methodology does not allow breaking out the percentage of stores specifically [32, p. 78].

[8] Experimentally, about 3,600 times in the 1 billion repetitions of the write during the counter microbenchmark.

diately, only add 14% overhead. Write checks are more costly, due to the load-linked and store-conditional pair; they add 200% overhead to the benchmark. Combining read and write checks yields the expected 214% overhead; our microbenchmark was carefully constructed to avoid the opportunities for instruction-level parallelism one might expect in real-world applications.

Examination of the assembly code generated for this simple benchmark seems to indicate substantial scope for improvement. The inline assembly mechanism of gcc/C provides no inherent support for instructions like the PowerPC's "store-conditional" (`stwcx.`) that leave their results in a condition code register. Figure 3.17 shows the assembly emitted for the write check version of the microbenchmark, with the cumbersome mechanism required to move the condition code to a register so that it can then be retested. We can easily hand-code a better write-check mechanism, shown in the right-hand column of the figure.[9] The optimized store-conditional test reduces write-check overhead to 186% and combined read- and write-check overhead to 199%, as shown in Figure 3.16.

Further performance improvement is possible by optimizing the benchmark with both read and write checks as a whole. Figure 3.18 shows an optimized assembly version of our `do_bench()` function. I've primarily rescheduled the code here to separate dependent instructions as much as possible, but I've also ensured repeatable instruction cache alignment,[10] replaced our canonical `FLAG` value with `0xFFFFCACA`, which can be represented by the PowerPC's 16-bit signed immediate instruction field (eliminating the need for a register), and combined the flag checks in the read and write

[9] This version has a stub where a function call to `unusualWrite` would usually go; the correct code here would branch to a small thunk that will perform the frame operations and register saves necessary to adhere to the C calling convention; gcc makes it difficult to ensure that this thunk is "near enough" for a direct branch (within a 24-bit signed displacement) without duplicating it needlessly.

[10] The MPC7447 (G4) PowerPC has a 32-byte cache line, although fetches occur in 16-byte (4 instruction) chunks.

```
do_bench:
  mflr 0
  stwu 1,-16(1)
  lis 5,0x3b9a
  li 8,0
  ori 5,5,51712
  addi 6,3,4
  addi 7,3,8
  stw 0,20(1)
  b .L4
.L22:
  mr 10,6
  mr 11,7
.L5:
  lwarx 0,0,10
  cmpwi 7,0,0
  bne- 7,.L19
  stwcx. 9,0,11
  li 0,0
  bne- 0f
  li 0,1
0:
  cmpwi 7,0,0
  beq- 7,.L5
  addi 8,8,1
  cmpw 7,8,5
  beq- 7,.L21
.L4:
  lwz 9,8(3)
  cmpwi 7,9,-13623
  addi 9,9,1
  bne+ 7,.L22
.L21:
  lwz 0,8(3)

  cmpw 7,0,8
  bne- 7,.L23
  lwz 0,20(1)
  addi 1,1,16
  mtlr 0
  blr
```

```
do_bench:
  mflr 0
  stwu 1,-16(1)
  addi 10,3,8
  addi 11,3,4
  stw 0,20(1)
  lis 0,0x3b9a
  ori 0,0,51712
  mtctr 0
  b .L4

.L5:
0:
  lwarx 0,0,11
  cmpwi 0,0
  beq+ 1f
  b . # stub for unusualWrite branch
1:stwcx. 9,0,10
  bne 0b

  bdz .L14

.L4:
  lwz 9,8(3)
  cmpwi 7,9,-13623
  addi 9,9,1
  bne+ 7,.L5
.L14:
  lwz 0,8(3)
  xoris 9,0,0xc465
  cmpwi 7,9,-13824
  bne 7,.L15
  lwz 0,20(1)
  addi 1,1,16
  mtlr 0
  blr
```

Figure 3.17: PowerPC assembly for counter microbenchmark with write checks. The right-hand column is generated from the C implementation; the left-hand column has an optimized version of the write check inlined into the code using the `asm` keyword. Italicized code is essentially unchanged; the optimized section is shown in boldface.

```
    # repetition count is in the count register to start
    b 0f
    .balign 32 # align to 32-byte cache-line boundary
    nop
    nop
    nop
    nop
    nop
    nop
0:  lwarx 5,0,0
    lwz 6, 0(9)
1:  ori 8, 6, 3
    addi 7, 6, 1
    cmpwi 1, 5,0
    cmpwi 2, 8, 0xFFFFCACB
    bne- 1, 0b # stub: unusualWrite
    beq- 2, 0b # stub: unusualRead or unusualWrite
    stwcx. 7,0,9
    lwarx 5,0,0
    lwz 6, 0(9)
    bne- 0, 1b
    bdnz 1b
```

Figure 3.18: Optimized PowerPC assembly for counter microbenchmark with both read and write checks.

routines.[11] The read-and-write check overhead is improved to 143% with these optimizations, due mostly to the irreducible dependency between the load-linked and store-conditional instructions.

Hennessy and Patterson's measurements indicate that load and store instructions comprise respectively 26% and 9% of dynamic instruction counts in SPECint92 on a RISC microarchitecture (DLX) [32, p. 105]. As a rough estimate, using dynamic instruction count as a proxy for instruction time, the 14% read overhead and 186% write overhead measured in this microbenchmark thus translate into 20% net overhead for non-transactional code.[12] If we conservatively assume that most of the improvement in Figure 3.18's optimized checks should be credited to reduced write-check over-

[11] Again, the branches to `unusualRead()` and `unusualWrite()` are difficult to express in this hybrid of C and assembly, but small stubs would be written to save registers and perform a branch with the appropriate calling conventions.

[12] $0.65 + (1.14 \times 0.26) + (2.86 \times 0.09) = 1.20$

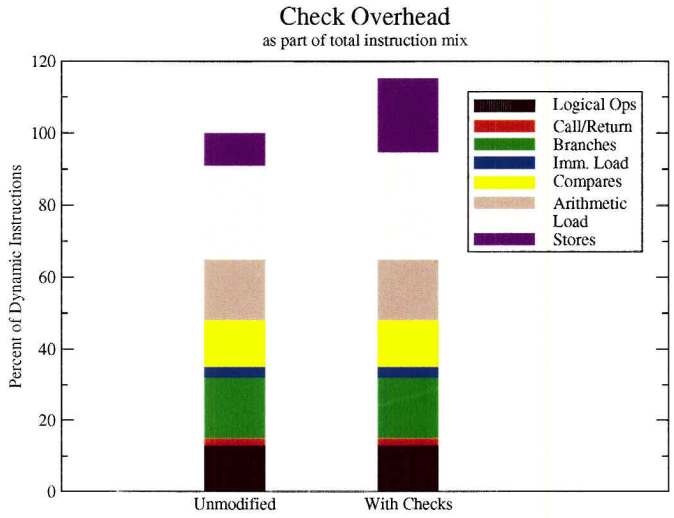

Figure 3.19: Check overhead as percentage of total dynamic instruction count. The bar on the left is the DLX instruction mix for SPECint92 from [32, p.105]. On the right is the same instruction mix after scaling the loads and stores according to Figure 3.16.

head, giving the same 14% read overhead but only 129% write overhead, the same metric gives only 15% net overhead, as shown in Figure 3.19. This should be considered a rough lower bound for the time overhead, possible with aggressive optimization and scheduling.[13]

Implementation

There are a number of details that are not yet covered by our model, for example: how are fields of different byte lengths handled? How does transactional Java code interact with the garbage collector, or with native code in the runtime system? The next chapter addresses the real-world implementation details of our software transaction system.

[13]Note that this calculation combines over- and underestimation, making this number only a rough bound. The use of dynamic instruction count, rather than dynamic instruction time, likely underestimates the contribution of loads and stores to the run time. However, scheduling, prefetching, and store buffers can pipeline the cost of the loads and stores, which might cause our overhead estimate to exceed the actual cost. Chapter 4.5 will present actual overhead measurements of real Java applications.