In theory, there is no difference
between theory and practice.

Jan L. A. van de Snepscheut

Chapter 4

Implementing efficient software transactions

In the previous chapter I described the design of an efficient software transaction system. In this chapter I will discuss the challenges involved in translating this design to a practical Java compiler, the implementation techniques I used, and the performance achieved. I will conclude by listing some limitations of the implementation, and describing how they may be overcome in a production-quality compiler.

4.1 The FLEX compiler infrastructure

In 1998 I began implementing the FLEX compiler infrastructure for Java [3], which I will use to evaluate the transaction system implementations in this thesis. FLEX has been used in over 20 published papers, on a wide range of topics. → cite them here

FLEX is a whole-program static compiler for Java, with a runtime system built around the GNU Classpath implementation of the Java standard libraries [23]. FLEX takes as input Java bytecode, generated by any Java 1.0-1.6 compiler from user code and the GNU Classpath 0.08 library implementation, and emits either assembly code for MIPS, Sparc, or StrongARM,

or else "portable assembly language" written in gcc 3.4's variant of C. The emitted code is compiled and linked against FLEX's runtime system and the GNU Classpath 0.08 native libraries to produce a stand-alone binary for the target system. The FLEX compiler and analysis code is written in Java, while the FLEX runtime system is written in C.

Using C as a target

The experiments in this thesis were conducted on either the x86 or PowerPC architectures, for which FLEX does not have a native assembly backend. The "Precise C" backend was thus used, so named because, aside from emitting C code, it's original purpose was to investigate the possibility of precise garbage collection whist emitting high-level code. To that end, the backend contains code to maintain a separate stack for values of non-primitive types, and to push and pop all live variables to this stack at gc points. Since FLEX's low-level IR may create derived pointers that point inside objects, for example during loop optimizations, the Precise C backend will also reconstruct the derived pointers after the gc point, in case the objects involved have moved.

Experiments showed that this mechanism had minimal impact on performance, because variables pushed onto the explicitly managed "object stack" were then dead across the call from the perspective of the underlying C compiler; in effect our explicit stack management replaced the implicit stack save/restore which the C compiler would otherwise have performed to maintain its calling convention.

Java exceptions presented another difficulty when implementing a C backend. The mechanisms used in our assembly backends (separate return addresses for "returning" an exception, either derived by rule from the normal return address or stored in a sorted table keyed by the normal return address) can not be implemented in C. FLEX supports two alternate translations: the first uses `setjmp` and `longjmp` to branch directly from the site where the exception is thrown to an appropriate handler, and the

second returns a pair value from every function call. The `setjmp` method only incurs cost when a new exception-handling region (`try/catch` block) is entered, or when an exception is thrown, but `setjmp` and `longjmp` are rather expensive. The pair-return method returns a C `struct` consisting of the "real" return value along with an exception value.[1] When the function returns, the caller must first test the exception value; if it is non-NULL, then an exception has been thrown and the caller must handle if (if it can) or re-throw it.

The mandatory test in the pair-return method adds overhead to every function call, but it is minimal. Experiments on x86 indicated that a pair-return call cost 2ms for both the "normal return" and "exception thrown" case, while the `setjmp` method cost only 1ms for the "normal return" but 73ms for "exception thrown". Thus `setjmp` is dramatically slower for applications that use exceptions, such as the SPECjvm98 benchmark `202_jess`.

The experiments in this thesis all use pair-return to implement exceptions, and conservative garbage collection, so the "Precise C" stack is not required.

4.2 Transforming synchronization

Aside from implementing the read and write mechanisms of the software transaction design presented in the previous chapter, a number of other transformations and analyses of our Java benchmarks must be performed: transactions must be synthesized from the benchmark's monitor synchronization, methods must be cloned according to the context (whether inside a transaction or not) of their call site, analyses must be performed in order to reduce redundant checks in the transformed code, and some minor desugaring is done to ease implementation.

```
synchronized {
    ...
    x = o.f;
    ...
    o.f = y;
    ...
    z = foo(a, b, c);
    ...
}
```

⇒

```
t = CommitRecord.newTransaction();
while(true) {
  try {
    ...
    v = ensureReader(o, t);
    x = readT(o, FIELD_F, &v, t);
    ...
    v = ensureWriter(o, t);
    checkWriteField(o, FIELD_F);
    writeT(o, FIELD_F, y, v); /* v.f = y */
    ...
    z = foo$$withtrans(t, a, b, c);
    ...
    t.commitTransaction();
    break;
  } catch (TransactionAbortException tex) {
    t = t.retryTransaction();
  }
}
```

Figure 4.1: Software transaction transformation. The code on the left is the original Java source; the transformed source is on the right.

4.2.1 Method transformation

We automatically create transactions in our benchmarks from Java synchronized blocks and methods. Figure 4.1 illustrates the transformation applied.

Entering a top-level synchronized block creates a new transaction object and starts a new exception handler context. Exiting the block causes a call to the transaction's commit method. If the commit, or any other operation inside the block, throws `TransactionAbortException` indicating that the transaction must be aborted, we recreate the transaction object and loop to retry the transaction's operations. The `retryTransaction` method performs backoff; it may also perform livelock detection or other transaction management functions.

Inside the transaction context, reads and writes must be transformed. Before the read, the `ensureReader` algorithm must be invoked (see chapter 3.3.2 and Appendix A). This need only be done once for each object read in this transaction; chapter 4.2.2 described how these checks are hoisted

[1] Methods declared as `void` return only the exception value, of course.

and combined to reduce their number. The ensureReader routine returns a pointer to a *version object* containing this transaction's field values for the object. The pointer may be NULL, however, if this transaction has not written the object. The actual read is done via the readT algorithm described in Figure 3.10 and chapter 3.3.2; it may update the cached version object for the given object (for example, if the object has been written within this transaction since the point at which ensureReader was invoked).[2]

Before each write, ensureWriter and checkWriteField must be executed. Like ensureReader, ensureWriter (Figure 3.11) need only be performed once for each object written in the transaction, and is hoisted and combined in the same way. The checkWriteField algorithm (Figure 3.12) need only be executed once for each object field written; if the same field in the same object is written multiple times, the subsequent checkWriteField invocations can be eliminated. The actual write is performed via writeT (Figure 3.10), which is a simple store to the version object.

Nested transactions are implemented via subsumption; that is, nested synchronized blocks inside a transaction context are ignored: the inner transaction is subsumed by the outermost.

Method invocations inside transaction context must be transformed, since read and write operations are implemented differently depending on whether or not the current execution is inside a transaction. We create a transactional clone from each method, named with a $$withtrans suffix, which is invoked when we are executing inside a transaction. We pass the current transaction as the first parameter of the cloned method.[3]

Non-transactional code inside a method must be transformed to use the readNT and writeNT mechanisms to perform object reads and writes

[2]The FLEX infrastructure cannot create pointers to temporaries, or alternately return multiple values from function calls, so the version object in my current implementation cannot be updated by readT. This impairs our ability to hoist and combine ensureReader and requires us to make redundant calls.

[3]Note that cloning must take virtual dispatch into account: cloning a method in a superclass must also create appropriate cloned subclass implementations.

(Figure 3.8); the implementation is similar to that shown in Figure 3.14 and Figure 3.15.

4.2.2 Analyses

Our performance will be improved if we can identify objects or fields that are guaranteed not to be concurrently mutated, and replace our checks and read/write protocols with direct accesses. We perform two analyses of this sort; chapter 4.6 contains additional analyses which we have not implemented.

Our first analysis classifies all fields in the program according to their use in transactional and non-transactional regions. This analysis is encapsulated in a FLEX class named `GlobalFieldOracle`. Fields that can never be written within a transaction do not need the special `readNT` mechanism from non-transactional code; their values can be loaded directly without testing for `FLAG`. The only way that the field can have the `FLAG` value, if it is not written within a transaction, is if it is a false flag, in which case its value really is `FLAG`. Similarly, fields that can never be read or written within a transaction do not need to use the `writeNT` mechanism; they can just store directly to the field.[4] Thus `GlobalFieldOracle` can make non-transactional code more efficient by eliminating the overhead of the software transaction system in some cases.

While `GlobalFieldOracle` targets non-transactional code, the `CheckOracle` analysis classes make transactional code more efficient by removing unneeded read and write checks. As mentioned above, the `ensureReader` and `ensureWriter` checks need only be invoked once on each unique object read/written by the transaction. Similarly, `checkWriteField` need only be performed once on each object field written by the transaction. The `CheckOracle` classes hoist each check to its immediate dominator iff this new location for the check is postdominated by the current location and

[4]Note that its not enough for the field not to be written; we must also protect against write-after-write conflicts.

the new location is dominated by the definition of the variable referenced in the check. This ensures that all paths to the original location of the check must pass through the new location, and that the object involved in the check is still defined appropriately at the new check location, since the analysis is performed on a single-assignment intermediate representation [4]. The check-hoisting process is repeated until no checks can be moved higher, then we eliminate any check that is dominated by an identical check.

4.2.3 Desugaring

We also desugar some Java idioms to ease implementation. Java specifies a `clone()` method of `java.lang.Object`, from which every object is derived. The top-level `clone()` method is somewhat magical, as it creates an exact copy of its subclass, including all its fields, which are otherwise hidden from methods of a superclass. The transactional translation of this method is also somewhat baroque: it would have to mark all fields of the original method as "read" by the transaction, and then construct a versioned object written by the transaction. As an end-run around this complexity, we desugar `clone()` into individual implementations in appropriate classes, each of which reads all fields of the class, and constructs a new object bypassing its constructors, and then individually sets all the fields to their new values. Arrays get a similar `clone()` implementation that reads and writes the elements of the array. These new implementations can then be transformed by the transaction pass like any other piece of code that reads and writes fields.

The FLEX infrastructure also contains a mechanism for fast initialization of arrays; this mechanism is desugared into individual array set operations so that it, too, can be uniformly transformed by the transaction pass.

4.3 Runtime system implementation

The FLEX runtime system was extended to support transactions. The runtime uses the standard Java Native Interface (JNI) [66] for native methods

(written in C) called from Java. Certain parts of the transaction system were written as native methods invoked via JNI, but others interact with the runtime system at a much lower level.

The runtime system is parameterized to allow a large number of memory allocation strategies and collectors, but for the experiments reported here I used the Boehm-Demers-Weiser conservative garbage collector [14].[5] Ideally the garbage collector would know enough about the transaction implementation to be able to automatically prune unneeded committed or aborted versions from the object's version list during collection. However, we relied on opportunistically nulling out tail pointers during version list traversal (for example, while looking for the last committed version), which may result in incomplete collections (and more memory usage than strictly necessary).

Each transaction had a `CommitRecord` object storing its state, whether `COMMITTED`, `ABORTED`, or still `WAITING`. Most `CommitRecord` methods were written in Java, including object creation, exponential backoff on retry, and throwing the appropriate `TransactionAbortException` on abort. Only the crucial `commit()` and `abort()` operations, which atomically set the transaction status iff it is still `WAITING`, were written in C, as JNI methods.

4.3.1 Implementing the JNI

No Java transaction implementation is complete without some mechanism for executing native methods, that is, C code. One cannot banish all native methods, since the Java standard library is built on them. If one cannot make native code observe the `readNT` and `writeNT` protocols, one must ensure that the native code never reads any object written in a transaction, or writes any object read or written in a transaction.[6] This is very inconvenient.

[5] The use of a conservative collector means that the mechanism described in chapter 4.1 to enable precise garbage collection was unneeded and thus disabled for these experiments.

[6] If one can implement `writeNT` but not `readNT`, one need only ensure that native code does not write fields that are also written inside a transaction, as discussed in chapter 4.2.2.

```
struct JNINativeInterface {
  ...
  /* Calling instance methods */
  jmethodID (*GetMethodID)
    (JNIEnv *env, jclass clazz, const char *name, const char *sig);
  jobject (*CallObjectMethod)
    (JNIEnv *env, jobject obj, jmethodID methodID, ...);
  ...
  jboolean (*CallBooleanMethod)
    (JNIEnv *env, jobject obj, jmethodID methodID, ...);
  ...
  jbyte (*CallByteMethod)
    (JNIEnv *env, jobject obj, jmethodID methodID, ...);
  ...
  /* Accessing fields of objects */
  jfieldID (*GetFieldID)
    (JNIEnv *env, jclass clazz, const char *name, const char *sig);
  jobject (*GetObjectField)
    (JNIEnv *env, jobject obj, jfieldID fieldID);
  jboolean (*GetBooleanField)
    (JNIEnv *env, jobject obj, jfieldID fieldID);
  ...
  void (*SetObjectField)
    (JNIEnv *env, jobject obj, jfieldID fieldID, jobject value);
  void (*SetBooleanField)
    (JNIEnv *env, jobject obj, jfieldID fieldID, jboolean value);
  ...
  /* Array Operations */
  jobject (*GetObjectArrayElement)
    (JNIEnv *env, jobjectArray array, jsize index);
  void (*SetObjectArrayElement)
    (JNIEnv *env, jobjectArray array, jsize index, jobject value);
  ...
  jboolean* (*GetBooleanArrayElements)
    (JNIEnv *env, jbooleanArray array, jboolean *isCopy);
  jbyte* (*GetByteArrayElements)
    (JNIEnv *env, jbyteArray array, jboolean *isCopy);
  ...
};
```

Figure 4.2: A portion of the Java Native Interface for interacting with the Java runtime from C native methods [66]. There are function variants for all of the basic Java types: boolean, byte, char, short, int, long, float, double, and Object. Note that the `jobject` and `j*Array` types are not direct references to the heap objects; they are opaque wrappers that preserve the garbage collector's invariants and protect the runtime system's private implementation.

Our solution ensures not only that (separately compiled) native code properly uses the `readNT` and `writeNT` protocols outside a transaction, but also that reads and writes inside a transaction are handled properly. This allows the use of "safe" native methods (those without external side effects) inside a transaction.

We are able to do this because FLEX uses the Java Native Inteface [66] to interact with native code. The JNI, a portion of which is shown in Figure 4.2, abstracts field accesses and Java method invocations from native code, so we can substitute implementations that behave appropriately whether in a transaction or not.

We distinguish between three classes of native methods. The first are native methods that are "safe" in a transactional context. That is, they have no external side effects (which would need to be undone if the transaction aborted), and will behave correctly in a transaction if all reads, writes, and method calls are simply replaced by the appropriate transactional protocol. The second are methods that have a different implementation in a transactional context. One example would be the native methods that implement file I/O; in a transactional context these should interface with a underlying transactional file system and arrange for the I/O operations to commit iff the current transaction commits. Another example is the `Object.wait()` method; the appropriate implementation is described in chapter 4.4.4. The last class of native methods are those that are inherently impossible in a transactional context, usually due to irreversible external I/O (chapter 7.4 discusses I/O interactions in more detail). A compiler configuration file identifies the safe native methods.

For "safe" native methods, the FLEX compiler creates a thunk that stores the transaction object in the JNI environment structure (`JNIEnv`) which is passed to every JNI method. When that native method invokes the field or array operations in the JNI (for example, `Get*Field`, `Set*Field`, `Get*ArrayElements`, etc.) the runtime checks the environment structure to determine whether to use the appropriate transactional or non-transactional

```
/* Framework for use of preprocessor specialization. */
/* (from readwrite.c) */

#define VALUETYPE jboolean
#define VALUENAME Boolean
#include "transact/readwrite-impl.c"
#define ARRAY
#include "transact/readwrite-impl.c"
#undef ARRAY
#undef VALUENAME
#undef VALUETYPE

/* ... etc ... */

#define VALUETYPE jdouble
#define VALUENAME Double
#include "transact/readwrite-impl.c"
#define ARRAY
#include "transact/readwrite-impl.c"
#undef ARRAY
#undef VALUENAME
#undef VALUETYPE

#define VALUETYPE struct oobj *
#define VALUENAME Object
#include "transact/readwrite-impl.c"
#define ARRAY
#include "transact/readwrite-impl.c"
#undef ARRAY
#undef VALUENAME
#undef VALUETYPE
```

Figure 4.3: Specializing transaction primitives by field size and object type: `readwrite.c`. This figure shows how `readwrite.c` specializes the code in `readwrite-impl.c` (Figure 4.4) through the use of multiple `#include` statements.

read or write protocol. When Java methods are invoked, via the `Call*Method` family of methods, in a transactional context the runtime looks up the `$$withtrans` suffixed version of the method (see chapter 4.2.1) and invokes it, passing the transaction object from the environment as the first parameter.

For unsafe native methods, FLEX generates a call to a `$$withtrans`-suffixed JNI method, passing the transaction as the first parameter as is done for pure Java calls inside a transaction context. The implementer is then responsible for correct transactional behavior.

```
/* Implementing generic functions. */
/* (from readwrite-impl.c) */
#include "transact/preproc.h" /* Defines 'T()' and 'TA'() macros. */

VALUETYPE TA(EXACT_readNT)(struct oobj *obj, unsigned offset);

VALUETYPE TA(EXACT_readNT)(struct oobj *obj, unsigned offset) {
  do {
    VALUETYPE f = *(VALUETYPE*)(FIELDBASE(obj) + offset);
    if (likely(f!=T(TRANS_FLAG))) return f;
    if (unlikely(SAW_FALSE_FLAG ==
                 TA(copyBackField)(obj, offset, KILL_WRITERS)))
      return T(TRANS_FLAG); // "false" transaction: field really is FLAG.
    // okay, we've done a copy-back now.  retry.
  } while(1);
}

/* ... undefine macros at end of readwrite-impl.c ... */
```

Figure 4.4: Specializing transaction primitives by field size and object type: `readwrite-impl.c`. In this snippet the `readNT` algorithm is implemented using macros (defined in Figure 4.5, `preproc.h`) to implement generic field types and naming conventions. Note the use of the macros `likely` and `unlikely`, which communicate static branch prediction information to the C compiler.

```
/* Preprocessor specialization macros */
/* (from preproc.h) */
#define x1(x,v) x2(x,v)
#define x2(x,v) x ## _ ## v

#if defined(ARRAY)
# define A(x) x1(x,Array)
# define OBJ_OR_ARRAY(x,y) (y)
#else
# define A(x) x
# define OBJ_OR_ARRAY(x,y) (x)
#endif

#define T(x) x1(x,VALUENAME)
#define TA(x) T(A(x))

#define FIELDBASE(obj) \
        OBJ_OR_ARRAY(obj->field_start,((struct aarray *)obj)->element_start)
```

Figure 4.5: Specializing transaction primitives by field size and object type: `preproc.h`. In this portion of the file we define the specialization macros used in Figure 4.4.

4.3.2 Preprocessor specialization

Part of the challenge in engineering a practical implementation is properly accounting for the wide range of data types in a real programming language. Java contains 8 primitive types in addition to types deriving from `java.lang.Object`, that are themselves divided into array and non-array types.[7]

Figures 4.3, 4.4, and 4.5 demonstrate how field size and object type specialization was implemented in the FLEX transaction runtime. The main header and source files did nothing but repeatedly `#include` an `-impl`-suffixed version of the file with `VALUETYPE`, `VALUENAME`, and `ARRAY` defined to range over the possible primitive and array types. This allows us to compactly name and define specialized functions, accounting for array/object (among other) differences. For example, the `FIELDBASE` macro (defined in Figure 4.4, used in Figure 4.3) allows us to treat object fields and array elements uniformly, even though the first array element starts at a different offset from the first field (since array data starts with an immutable length field).

Chapter 4.4.3 will discuss some of the other challenges involved in synthesizing atomic operations on subword and multi-word datatypes.

4.4 Limitations

The present implementation contains a number of non-fundamental limitations with well-understood solutions that, nonetheless, would add additional implementation complexity. These involve static fields, coarse granularity of LL/SC instructions, sub-word and multi-word fields, and condition variables (`Object.wait()`). In addition, there are limitations related to the

[7] Our flag value was carefully chosen so as not to be a NaN for either of the floating-point types, since comparisons against NaN can be surprising, and to consist of a repeated byte value so that a new object can be initialized with all fields `FLAG` without knowledge of the exact types and locations of each field.

```
class C {                          class C$$static {
  static int f;                      int f;
  ...                              }
  void foo() {                     class C {
    ... = C.f;          ⇒            final static C$$static $static = new C$$static();
    C.f = ...;                       ...
  }                                  void foo() {
}                                      ... = C.$static.f;
                                       C.$static.f = ...;
                                     }
                                   }
```

Figure 4.6: Static field transformation. Static fields of class `C` have been hoisted into a new class `C$$static`. Since the new field `$static` is write-once during initialization, it is safe to read/write directly, unlike the old field `f`.

manipulation of large objects (usually arrays); we will put off to chapter 5 a full discussion of the large object problem and its solutions.

4.4.1 Static fields

Static fields are not encapsulated in an object as other fields are; they are really a form of controlled-visibility global variable. The present implementation ignores static fields when performing the synchronization transformation; reads and writes to static fields are always direct. In theory this could cause races, but in our benchmarks most static fields are written only during initialization. An analysis that proved that writes only occurred during a single-threaded class initialization phase of execution could prove that these direct accesses are safe in most cases. Proper handling of static fields that do not meet this condition is straight-forward: new singleton classes would be created encapsulating these fields, and access would be via an extra indirection. Figure 4.6 illustrates this transformation.

4.4.2 Coarse-grained LL/SC

In our description of the software transaction algorithms we have so far neglected to state the size of the reservation created by the platform load-

linked instruction. The implicit assumption is that the reservation is roughly word-sized: our algorithm uses load-linked to watch the pointer-valued `readerList` (figures 3.8, 3.11) and `version` (figures 3.9, 3.11, 3.12) fields, and to perform the compare-and-swap operation on the transaction status field necessary for aborting and committing transactions. However, implementations of load-linked and store-conditional on most architectures usually place a reservation on a specific cache line, which spans many words. In the case of our MPC7447 (G4) PowerPC processor, a reservation (and cache line) is 32 bytes.

In our present implementation we have not taken any special action to account for this. In general, a larger-than-expected reservation may cause false conflicts between independent transactions, and this can lead to deadlock or livelock if contention is not managed properly. Our implementation uses randomized[8] exponential backoff, which ensures that every transaction eventually has an opportunity to execute in isolation, which will moot any false conflict.

Other mechanisms to resolve contention can also be used to address false conflicts. In an extreme case, a contention manager could invoke a copying collector to relocate objects involved in a false conflict, although this is likely merited only when the conflicts are extremely frequent or when long-lived transactions make the copying cost less than the cost of serializing the transactions involved.

Alternatively, false conflicts can be managed with careful allocation policy. The allocator in the present implementation ensures 8-byte alignment of objects. As we have done in prior work [69], the Java allocator can be modified to align objects to cache line (32-byte) boundaries, to ensure there are no false conflicts between objects. Some of the wasted space can be reclaimed by using a smaller alignment for objects known not to escape from their thread, or by packing multiple objects into a cache line iff they are

[8]The randomization is via the varying latency of the Unix `sleep` syscall; stronger randomization could be implemented if it mattered.

known never to participate in transactions that might conflict.

4.4.3 Handling subword and multi-word reads/writes

The granularity of the store-conditional instruction is more problematic. Although we have specified our algorithms assuming that a store-conditional of any field is possible, in practice store-conditional operations are only provided for a restricted set of data types: the PowerPC architecture defines only 32-bit and 64-bit stores, and 32-bit implementations (like our G4) don't implement the 64-bit variant. We need to carefully consider how to safely implement our algorithms with only a word-sized store-conditional.

First let us consider the case where the field size is larger than a word; for a Java implementation this corresponds to the `double` and `long` types. Within a transaction we can simply write two `int`s (for example) to implement the store of a `long`; the transactional mechanism already ensures that the multiple writes will appear atomic.

This leaves large writes outside of a transaction. One solution is to decompose these fields into multiple smaller fields as we did in the transactional case; we don't give up strong atomicity but we allow other readers (both transactional and non-transactional) to see the partial writes. The Java memory model actually permits this behavior (Java Language Specification chapter 17.7, "Non-atomic Treatment of double and long" [24]). The specification continues, "VM implementers are encouraged to avoid splitting their 64-bit values where possible." We can easily implement this behavior as an alternative: non-transactional writes of doubles and longs can be converted to small transactions encompassing nothing but the multiple word-size writes.

Now let us consider sub-word-sized writes. Chapter 17.6 of the Java Language Specification specifically says:

> One implementation consideration for Java virtual machines is that every field and array element is considered distinct; updates

> to one field or element must not interact with reads or updates of any other field or element. In particular, two threads that update adjacent elements of a byte array separately must not interfere or interact and do not need synchronization to ensure sequential consistency.
>
> Some processors do not provide the ability to write to a single byte. It would be illegal to implement byte array updates on such a processor by simply reading an entire word, updating the appropriate byte, and then writing the entire word back to memory. This problem is sometimes known as *word tearing*, and on processors that cannot easily update a single byte in isolation some other approach will be required.

Again, writes within a transaction are straight-forward: reading an entire word, updating it, and rewriting will appear atomic due to the transaction mechanism, and no observable word tearing will occur. A minor difficulty is the possibility of "false" conflicts between updates to adjacent fields of an object; these are handled as described in chapter 4.4.2.

The final case is then non-transactional writes to sub-word values, which occur frequently in real programs during string manipulations. We could address this by using a short transaction to do the read-modify-write of the word surrounding the sub-word field, but this is likely too expensive, considering the number of sub-word manipulations in most programs. A better algorithm to address this problem is presented in Figure 4.7; compare this to the earlier expression of the `writeNT` algorithm in Figure 3.15. The revised algorithm steals the lower two bits of the `readerList` pointer to serve as subword reservations, ensuring that racing writes to different subwords in the same word are prevented. The cost of this algorithm is mostly extra masking when the readerList is read, and an extra store (to set the `readerList` reservation) on every sub-word write.[9]

[9]It would be nice if we could avoid updating the `readerList` pointer on every write

```
void writeNT(struct oobj *obj, int byte_idx, small_field_t val) {
  if (unlikely(val==FLAG))
    unusualWrite(obj,byte_idx,val); // do a short transactional write
  else {
    do {
      int r = (int) LL(&(obj->readerList));
      if (unlikely((r & ~3) != 0)) {
        // readerList is not "NULL"; have to kill readers, etc.
        unusualWrite(obj,idx,val);
        return;
      } else if (r == (byte_idx & 3)) {
        // okay to write to this subword
        word_t w = obj->word[byte_idx>>2];
        if (SC(&(obj->word[byte_idx>>2]), combine(w, val, byte_idx)))
          return;
      } else
        // last write was to some other subword; switch to this one.
        SC(&(obj->readerList), byte_idx & 3);
    } while (1);
  }
}
```

Figure 4.7: Non-transactional write to a small (sub-word) field. Compare to Figure 3.15. The `byte_idx` value is the offset of the field within the objects, in bytes, and the `combine` function does the appropriate shifting and masking to combine the new sub-word value with the read value of the surrounding word.

Our present implementation does not specially handle sub-word-sized writes, resulting in programmer-visible word tearing.

4.4.4 Condition variables

A condition variable is a synchronization device that allows threads to suspend execution and relinquish the processors until some predicate on shared data is satisfied. When the predicate becomes true, a thread signals the condition (`Object.notify()` and `Object.notifyAll()` in Java); other threads can suspend themselves waiting for the signal (`Object.wait()`). In order to prevent races between notification and waiting, condition variables are

when using common access patterns, like stepping linearly through the array, but I don't see a way to do that.

```
public class Drop {
    //Message sent from producer to consumer.
    private String message;
    //True if consumer should wait for producer to send message, false
    //if producer should wait for consumer to retrieve message.
    private boolean empty = true;

    public synchronized String take() {
        //Wait until message is available.
        while (empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        //Toggle status.
        empty = true;
        //Notify producer that status has changed.
        notifyAll();
        return message;
    }

    public synchronized void put(String message) {
        //Wait until message has been retrieved.
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        //Toggle status.
        empty = false;
        //Store message.
        this.message = message;
        //Notify consumer that status has changed.
        notifyAll();
    }
}
```

Figure 4.8: Drop box example illustrating the use of condition variables in Java, from [70].

usually associated with a mutex (the object monitor, in Java) and `wait` and `notify` require the lock to be held when they are called. The `wait` operation atomically releases the lock and waits for the variable; when notification is received `wait` re-acquires the lock before resuming. An example of the use of these operations in Java is provided in Figure 4.8.

Implementing these semantics properly in a transactional context is challenging; a better solution to the fundamental problem is a mechanism like Harris and Fraser's ***guarded transactions*** [30]. However, a reasonable implementation is possible.

The JDK1.6 specification for the `Object.wait()` method says:

> A thread can also wake up without being notified, interrupted, or timing out, a so-called spurious wakeup. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied. In other words, waits should always occur in loops...

This simplifies our implementation of the notification methods, since we needn't wait until the transaction doing the notify is sure to commit; it is always safe to go ahead and do the notify immediately; if we end up getting aborted we'll just redo the notify later when we retry. So we acquire a special mutex (either per-object as with standard Java or global; the lock is not held long), do the notification, and release the mutex.

The `wait` method, however, is a commit point, since it releases and reacquires the monitor lock. Thus it splits the transaction containing it into two.[10] The fundamental mechanism is straight-forward. We first acquire our special mutex, then attempt to commit the transaction, and check the result. If we don't commit successfully, we unlock the mutex and throw the usual `TransactionAbortException`. If we have committed successfully, we

[10] Which arguably calls for a special type annotation on methods that may call `wait()`, so that users are not misled into thinking a block is atomic that may invoke something that calls `wait()`.

can safely wait (releasing our mutex atomically). When we're woken, we create a new transaction, release our mutex, and continue. The mutex prevents notification from occurring between the commit and the wait.[11]

Implementation in a practical system introduces some difficulties. As described in chapter 4.2, the transaction object is passed as the first argument when calling methods within a transaction context. Since `Object.wait()` commits the initial transaction and begins a new one, we must either factor out a continuation after the call to `Object.wait()` which can be invoked with the new transaction after the wait is complete, or else provide a means to update the active transaction pointer and the retry point. One way to update the active transaction pointer is to use indirection: instead of keeping a direct pointer to the transaction object, we can keep a pointer to a cell containing the pointer; the cell can then be updated to point to the new transaction by `Object.wait()`. Alternatively, an extra return value can be added to methods called within a transaction, so that method can return an updated transaction object as well as their usual return value. Either of these transformations can be applied selectively to only those methods that could possibly invoke, directly or indirectly, `Object.wait()`. The more difficult challenge is to update the location at which we'll resume for another attempt when the transaction is aborted; this retry point should be immediately following the `wait()`. The retry mechanism becomes much more complicated, and some means to reconstruct the appropriate call-stack is necessary.

Note that for code like the example given in Figure 4.8 the "correct" resumption behavior is actually identical to restarting the transaction from

[11] Note in the drop box example in Figure 4.8 that the standard monitor lock is present to prevent (for example) `put()` from setting `empty` to false and doing the `notifyAll()` *after* `take()` has seen that `empty` is true but *before* the `wait()` has occurred (hence the notification would be lost). With the transaction transformation, the lost notification is prevented because put's write to `empty` will abort the take transaction if it occurs before take is committed (causing the `wait()` to be retried), and the special mutex will prevent the notification from occurring between the commit and the `wait`.

the original start point (ie, the beginning of the `put()` or `take()` method), assuming that the `put()` or `take()` isn't in a subsumed nested transaction. It is probably worth identifying this situation through compile-time analysis, as it simplifies the resumption transformation considerably.

In our present implementation we implement correct `notify` semantics, but we do not allow `wait` to be a commit point. Instead we take the special mutex and sleep on the condition variable without attempting to commit the transaction, and resume executing in the same transaction when we awake. This is sufficient for the limited uses of `wait` in our chosen benchmarks.

4.5 Performance

In this section we will evaluate the performance of our implementation. All experiments were performed on a 1GHz MPC7447 (G4) PowerPC processor, with 512kB unified L2 cache and 512MB of main memory, running Ubuntu 6.10 on a Linux 2.6.17 kernel. The limitations noted in the previous section counsel that the numbers we obtain ought to be considered guidelines to potential performance, not fundamental limits. Previous work has adequately demonstrated the scalability benefits of non-blocking synchronization; I will concentrate in this thesis on single-threaded efficiency measures.

We will be examining the SPECjvm98 benchmark suite, a collection of practical Java programs including an expert system, a simple database, a Java compiler, an audio encoder, and a parser generator. To clarify our measurements, our baseline for all of our comparisons will be a version of the benchmarks compiled with the same method cloning and desugaring (chapter 4.2) that is done for the transaction transformation.

In chapter 3.4 we examined a microbenchmark to discover the lower limits on our non-transactional overhead. These are the fundamental costs of strong atomicity: the penalty we must pay even if we are not using transactions at all.[12] Figure 4.9 shows equivalent measurements on full Java

[12] Although, of course, in practice one would turn off the transaction transformation

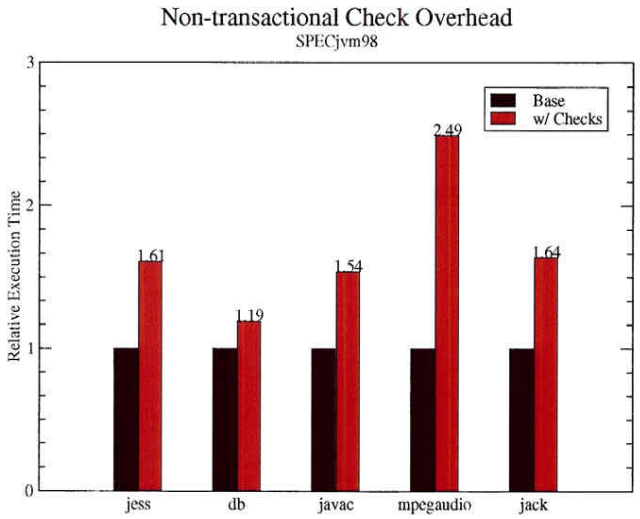


Figure 4.9: Non-transactional check overhead for SPECjvm98. Both benchmarks in each pair were compiled with the transformations in chapter 4.2 (method cloning, desugaring) and all synchronization was removed. The version "with checks" had all reads and writes transformed using the `readNT` and `writeNT` protocols, while the base version performed direct field access.

Benchmark	reads	false flags read	writes	false flags written
jess	25,583,878	0 (0%)	703,303	0 (0%)
db	11,078,177	0 (0%)	912,965	0 (0%)
javac	10,315	0 (0%)	2,312	0 (0%)
mpegaudio	239,928,276	2,056 (< 0.001%)	42,164,416	9,132 (.02%)
jack	9,882,846	0 (0%)	4,595,774	0 (0%)

Figure 4.10: Number of false flags read/written in SPECjvm98 benchmarks. To compile this table the applications were run with input size "10".

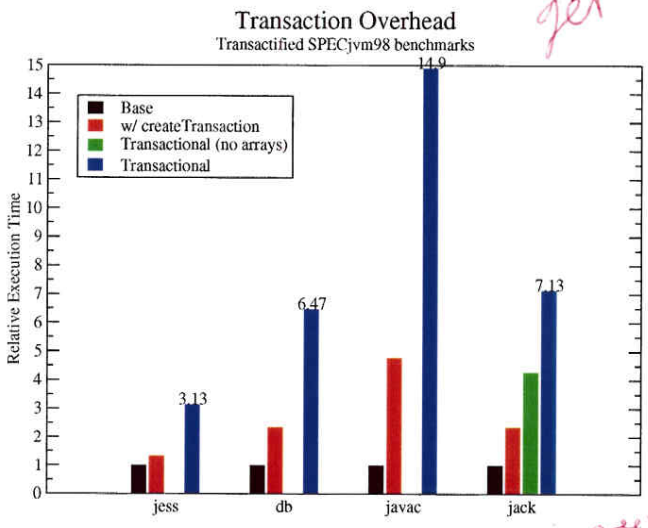


Figure 4.11: Transaction overhead for SPECjvm98. The leftmost bar is compiled with method cloning and desugaring, all transactions removed, and direct reads and writes. The next bar adds creation of commit records at the start of each transaction and the CAS commit operation at the end. The next bar uses the transactional and non-transactional read and write protocols for non-array objects. The final bar is fully transactional.

applications. We have removed all the synchronization in these benchmarks (safe because they are single-threaded) leaving only the costs of the `readNT` and `writeNT` protocols.

Our earlier microbenchmark hinted that we might expect non-transactional overheads as low as 15%, although 20% was the best we could obtain with a C implementation. In fact, we see a 19% overhead on the 209_db benchmark, although the other benchmarks show that an overhead of around 60% is more typical. Close inspection of the assembly code emitted by the C compiler for these benchmarks indicates that the compiler's limitations are the primary reason our performance falls short of mark set by the microbenchmark. As discussed in chapter 4.2, it is hard to obtain maximum performance with a C implementation (such as we are using here) due to expressive limits, and the emitted assembly indicates that the compiler is not optimally acting on the hints we can give it. In particular, we want as little extraneous code as possible on the fast path through the common case, but the compiler adds register spills and moves here that are necessary only in unusual cases. Further, because the fundamental load-linked and store-conditional instructions are inside inline assembly blocks and thus opaque to the compiler, many opportunities for code motion and scheduling are being missed.

The 222_mpegaudio benchmark shows another potential liability: it incurred a 149% overhead for the non-transactional code. The "false flag" counts shown in Figure 4.10 hint at the problem: this benchmark is the only one to encounter false flags. The hidden portion of the iceberg may partly consist of inefficiencies in our current implementation of byte-sized reads and writes, but since false flags cause creation of new version objects, most of the cost is probably a reflection of the "large object problem", which we will discuss further in a moment.

Figure 4.11 demonstrates the performance of transactional code, broken

completely if static or dynamic analysis indicated a program is single-threaded, for example if no Thread objects are every created.

down into the components of the final cost. We see that creation of transaction objects at the start of every atomic region is a considerable fraction of the total cost. If transactions are going to be frequent and small, this cost must be kept in line.

Transactions that mutate arrays also account for a large amount of the total cost. In the extreme case, the SPECjvm98 benchmark `201_compress` allocates a 1MB array and writes compressed data into it. The design of our transaction system necessitates an object copy for every unique object written to inside a transaction; if the objects are large (such as the arrays in `201_compress` and `222_mpegaudio`), then this cost becomes excessive. The next chapter will discuss this problem at length and propose a solution based on functional array data structures.

Although our transaction design ought to be able to perform direct read and writes on version objects in long-lived transactions, these performance numbers indicate that we are not achieving our goal.

4.6 Additional optimizations

In this section we will discuss some additional optimizations that could improve our performance on transactional code; they have not been implemented in the present compiler. The improvements include version passing, escape analysis, object immutability analysis, and fast allocation.

As described in chapter 4.2, the basic transaction transformation inserts a call to `ensureReader/ensureWriter` once per object prior to a read/write of a field in the object. These methods return a version object (the "current version") which is then passed to the `readT/writeT` methods. A straightforward improvement would avoid redundant calls to `ensureReader/ensureWriter` by passing the "current version" of the various parameters when a method is invoked. A strictness analysis should be performed, to avoid marking fields as read/written that are not guaranteed to be read/written by the transaction, and the "current version" added as

an extra argument for the strict parameters of the method. This allows the caller to combine redundant `ensureReader`/`ensureWriter` invocations for the argument.[13]

Escape analysis [68] is a standard technique to reduce unnecessary synchronization in Java programs. In a typical application, monitor locks on objects that do not escape their thread are eliminated; these lock eliminations can translate to transaction eliminations as well. Further, escape analysis can identify individual objects that do not escape their method, thread, or call context, and replace the `readT`/`writeT` or `readNT`/`writeNT` protocol with direct access.

Another orthogonal analysis can identify object immutability [72]. Immutable objects can be read directly and their field values can be safely cached across transaction boundaries. Further, the initialization of immutable objects typically can be done with direct writes as well.

Finally, a substantial portion of the cost of small transactions is spent allocating the version object, especially for the conservative collector used in the present implementation. Fast allocators could alleviate this bottleneck [6]. Alternatively, a free list of versions could be hung from objects involved in many transactions: as soon as a version commits, the previous version can be added to the free list for the use of the next transaction.

[13] One could also consider adding "current version" arguments even for non-strict parameters, with the caveat that the caller pass in NULL if it had not already read/written the non-strict parameter. However, all variables used in a method or the method's callees would then be potentially eligible for version passing; some more sophisticated heuristic would be needed to determine for which objects version passing is worthwhile.

Well, you go in and you ask for some toothpaste—the small size—and the man brings you the large size. You tell him you wanted the small size but he says the large size is the small size. I always thought the large size was the largest size, but he says that the family size, the economy size and the giant size are all larger than the large size—that the large size is the smallest size there is.

Charade (1963)

Chapter 5

Arrays and large objects

The software transaction system implemented in the previous chapter clones objects on transactional writes, so that the previous state of the object can be restored if the transaction aborts. Figure 5.1 shows the object size distribution of transactional writes for SPECjvm98, and indicates that over 10% of writes may be to large objects. As we've seen in chapter 4.5, the copying cost can become excessive.

The solution I will propose will represent objects as *functional arrays*. O'Neill and Burton [56] give a fairly inclusive overview of such algorithms; I've chosen Tyng-Ruey Chuang's version [15] of *shallow binding*, which uses randomized cuts to the version tree to limit the cost of a read to $O(n)$ in the worst case. Single-threaded accesses to the array are $O(1)$. Our use of functional arrays is single-threaded in the common case, when transactions do not abort. Chuang's scheme is attractive because it limits the worst-case cost of an abort, with very little added complexity.

In this chapter I will recast the transaction system design of chapter 3 as a "small object protocol," then show how to extend it to a "large object protocol," in the process addressing the large object performance problems. The large object protocol will use a lock-free variant of Chuang's algorithm, which I will present in chapter 5.4.

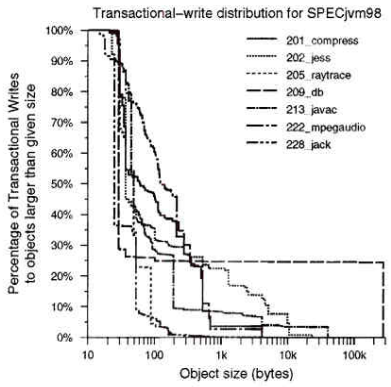


Figure 5.1: Proportion of transactional writes to objects equal to or smaller than a given size.

5.1 Basic operations on functional arrays

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

However, the distinguishing characteristic of an imperative array is its time complexity: $O(1)$ access or update of any element. Implementing functional arrays with a functional balanced tree yields $O(\lg n)$ worst-case access or update.[1]

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

- FA-CREATE(n): Return an array A of size n. The contents of the array are initialized to zero.

[1] I will return to a discussion of operational complexity in chapter 5.4.

- FA-UPDATE(A, i, v): Return an array A' that is functionally identical to array A except that FA-READ$(A', \mathrm{i}) = v$. Array A is not destroyed and can be accessed further.

- FA-READ(A, i): Return $A(\mathrm{i})$ (that is, the value of the ith element of array A).

We allow any of these operations to *fail*. Failed operations can be safely retried, as all operations are idempotent by definition.

For the moment, consider the following naïve implementation:

- FA-CREATE(n): Return an ordinary imperative array of size n.

- FA-UPDATE(A, i, v): Create a new imperative array A' and copy the contents of A to A'. Set $A'[\mathrm{i}] = v$. Return A'.

- FA-READ(A, i): Return $A[\mathrm{i}]$.

This implementation has $O(1)$ read and $O(\mathrm{n})$ update, so it matches the performance of imperative arrays only when $\mathrm{n} = O(1)$. I will therefore call these *small object functional arrays*. Operations in this implementation never fail. Every operation is non-blocking and no synchronization is necessary, since the imperative arrays are never mutated after they are created. In chapter 5.4 we will review better implementations of functional arrays, and present our own lock-free variant.

5.2 A single-object protocol

Given a non-blocking implementation of functional arrays, we can construct a transaction implementation for single objects. In this implementation, fields of at most one object may be referenced during the execution of the transaction.

I will consider the following two operations on objects:

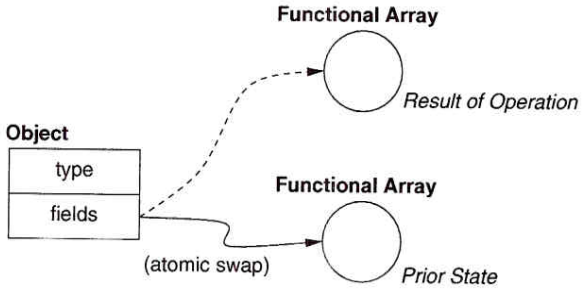


Figure 5.2: Implementing non-blocking single-object concurrent operations with functional arrays.

- READ(o, f): Read field f of o. We will assume that there is a constant mapping function that given a field name returns an integer index. We will write the result of mapping f as `f.index`. For simplicity, and without loss of generality, we will assume all fields are of equal size.
- WRITE(o, f, v): Write value v to field f of o.

All other operations on Java objects, such as method dispatch and type interrogation, can be performed using the immutable `type` field in the object. Because the `type` field is never changed after object creation, non-blocking implementations of operations on the `type` field are trivial.

As Figure 5.2 shows, our single-object transaction implementation represents objects as a pair, combining `type` and a reference to a functional array. When not inside a transaction, object reads and writes are implemented using the corresponding functional array operation, with the array reference in the object being updated appropriately:

- READ(o, f): Return FA-READ(`o.fields`, `f.index`).
- WRITE(o, f, v): Replace `o.fields` with the result of FA-UPDATE(`o.fields`, `f.index`, v).

The interesting cases are reads and writes inside a transaction. At entry to our transaction that will access (only) object o, we store `o.fields` in a

local variable u. We create another local variable u' which we initialize to u. Then our read and write operations are implemented as:

- READT(o, f): Return FA-READ$(u', f.\texttt{index})$.
- WRITET(o, f, v): Update variable u' to the result of FA-UPDATE$(u', f.\texttt{index}, v)$.

At the end of the transaction, we use Compare-And-Swap to atomically set $o.\texttt{fields}$ to u' iff it contained u. If the CAS fails, we the transaction is aborted (we simply discard u') and we retry.

With our naïve "small object" functional arrays, this implementation is exactly the "small object protocol" of Herlihy [34]. Herlihy's protocol is rightly criticized for an excessive amount of copying. I will address this with a better implementation of functional arrays in chapter 5.4. However, the restriction that only one object may be referenced within a transaction is overly limiting. I will first fix this problem.

5.3 Extension to multiple objects

I extend the implementation to allow the fields of any number of objects to be accessed during the transaction. Figure 5.3 shows our new object representation. Compare this to Figure 3.6; we've successfully recast our earlier transaction system design now in terms of operations on an array datatype. Objects consist of two slots, and the first represents the immutable `type`, as before. The second field, `versions`, points to a linked list of `Version` structures. The `Version` structures contain a pointer `fields` to a functional array, and a pointer `owner` to an *transaction identifier*. The transaction identifier contains a single field, `status`, which can be set to one of three values: *COMMITTED*, *IN-PROGRESS*, or *ABORTED*. When the transaction identifier is created, the status field is initialized to *IN-PROGRESS*, and it will be updated exactly once thereafter, to either *COMMITTED* or

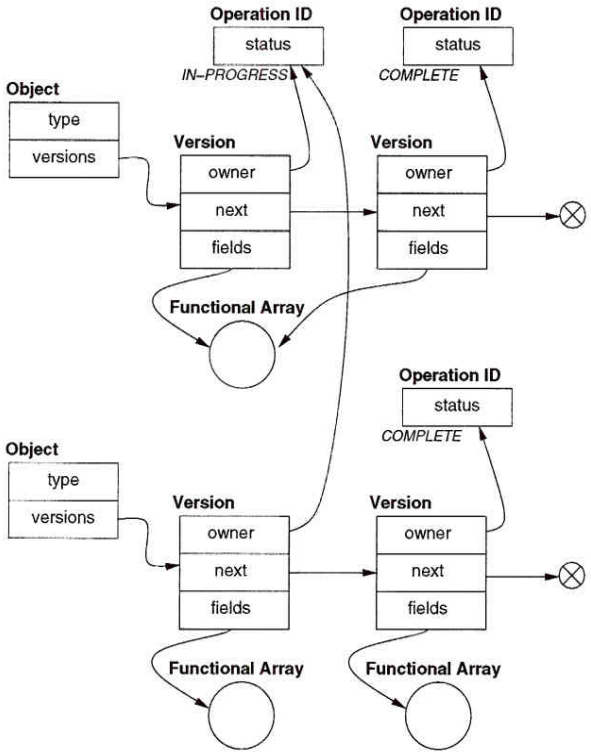


Figure 5.3: Data structures to support non-blocking multi-object concurrent operations. Objects point to a linked list of versions, which reference transaction identifiers. Versions created within the same execution of a transaction share the same transaction identifier. Version structure also contain pointers to functional arrays, which record the values for the fields of the object. If no modifications have been made to the object, multiple versions in the list may share the same functional array. (Compare this model of a transaction system to our concrete design in Figure 3.6.)

```
READ(o, f):
begin
retry:
  u ← o.versions
  u' ← u.next
  s ← u.owner.status
  if (s = DISCARDED)                          [Delete DISCARDED?]
    CAS(u, u', &(o.versions))
    goto retry
  else if (s = COMPLETE)
    a ← u.fields                                [u is COMPLETE]
    u.next ← null                               [Trim version list]
  else
    a ← u'.fields                               [u' is COMPLETE]
  return FA-READ(a, f.index)                    [Do the read]
end

READT(o, f):
begin
  u ← o.versions
  if (oid = u.owner)                          [My OID should be first]
    return FA-READ(u.fields, f.index)           [Do the read]
  else                                          [Make me first!]
    u' ← u.next
    s ← u.owner.status
    if (s = DISCARDED)                        [Delete DISCARDED?]
      CAS(u, u', &(o.versions))
    else if (oid.status = DISCARDED)            [Am I alive?]
      fail
    else if (s = IN-PROGRESS)                 [Abort IN-PROGRESS?]
      CAS(s, DISCARDED, &(u.owner.status))
    else                                        [Link new version in:]
      u.next ← null                             [Trim version list]
      u' ← new Version(oid, u, null)            [Create new version]
      if (CAS(u, u', &(o.versions)) ≠ FAIL)
        u'.fields ← u.fields                    [Copy old fields]
    goto retry
end
```

Figure 5.4: READ and READT implementations for the multi-object protocol.

```
WRITE(o, f, v):
begin
retry:
  u ← o.versions
  u′ ← u.next
  s ← u.owner.status
  if (s = DISCARDED)                          [Delete DISCARDED?]
    CAS(u, u′, &(o.versions))
  else if (s = IN-PROGRESS)                   [Abort IN-PROGRESS?]
    CAS(s, DISCARDED, &(u.owner.status))
  else                                        [u is COMPLETE]
    u.next ← null                             [Trim version list]
    a ← u.fields
    a′ ← FA-UPDATE(a, f.index, v)
    if (CAS(a, a′, &(u.fields)) ≠ FAIL)       [Do the write]
      return                                  [Success!]
  goto retry
end

WRITET(o, f, v):
begin
  u ← o.versions
  if (oid = u.owner)                          [My OID should be first]
    u.fields ← FA-UPDATE(u.fields, f.index, v)[Do write]
  else                                        [Make me first!]
    u′ ← u.next
    s ← u.owner.status
    if (s = DISCARDED)                        [Delete DISCARDED?]
      CAS(u, u′, &(o.versions))
    else if (oid.status = DISCARDED)          [Am I alive?]
      fail
    else if (s = IN-PROGRESS)                 [Abort IN-PROGRESS?]
      CAS(s, DISCARDED, &(u.owner.status))
    else                                      [Link new version in:]
      u.next ← null                           [Trim version list]
      u′ ← new Version(oid, u, null)          [Create new version]
      if (CAS(u, u′, &(o.versions)) ≠ FAIL)
        u′.fields ← u.fields                  [Copy old fields]
    goto retry
end
```

Figure 5.5: WRITE and WRITET implementations for the multi-object protocol.

ABORTED. A *COMMITTED* transaction identifier never later becomes *IN-PROGRESS* or *ABORTED*, and a *ABORTED* transaction identifier never becomes *COMMITTED* or *IN-PROGRESS*.

We create an transaction identifier when we begin or restart a transaction and place it in a local variable *tid*. At the end of the transaction, we use CAS to set *tid*.`status` to *COMMITTED* iff it was *IN-PROGRESS*. If the CAS is successful, the transaction has also executed successfully; otherwise *tid*.`status` = *ABORTED* (which indicates that our transaction has been aborted) and we must back off and retry. All `Version` structures created while in the transaction will reference *tid* in their `owner` field.

Semantically, the current field values for the object will be given by the first version in the versions list whose transaction identifier is *COMMITTED*. This allows us to link *IN-PROGRESS* versions in at the head of multiple objects' versions lists and atomically change the values of all these objects by setting the one common transaction identifier to *COMMITTED*. We only allow one *IN-PROGRESS* version on the versions list, and it must be at the head, so before we can link a new version at the head we must ensure that every other version on the list is *ABORTED* or *COMMITTED*.

Since we will never look past the first *COMMITTED* version in the versions list, we can free all versions past that point. In our presentation of the algorithm, we do this by explicitly setting the `next` field of every *COMMITTED* version we see to **null**; this allows the versions past that point to be garbage collected. An optimization would be to have the garbage collector do the list trimming for us when it does a collection.

We don't want to inadvertently chase the null `next` pointer of a *COMMITTED* version, so we always load the `next` field of a version *before* we load `owner.status`. Since the writes occur in the reverse order (*COMMITTED* to `owner.status`, then **null** to `next`) we have ensured that our `next` pointer is valid whenever the status is not *COMMITTED*.

We begin an atomic method with TRANSSTART and attempt to complete an atomic method with TRANSEND. They are defined as follows:

- TRANSSTART: create a new transaction identifier, with its status initialized to *IN-PROGRESS*. Assign it to the thread-local variable *tid*.
- TRANSEND: If

  $$\text{CAS}(\textit{IN-PROGRESS}, \textit{COMMITTED}, \&(\textit{tid}.\texttt{status}))$$

  is successful, the transaction as a whole has completed successfully, and can be linearized at the location of the CAS. Otherwise, the transaction has been aborted. Back off and retry from TRANSSTART.

Pseudo-code describing READ, WRITE, READT, and WRITET is presented in Figures 5.4 and 5.5. In the absence of contention, all operations take constant time plus an invocation of FA-READ or FA-UPDATE.

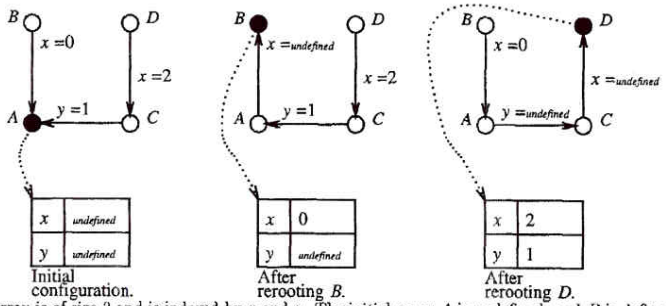
5.4 Lock-free functional arrays

In this section I will present a lock-free implementation of functional arrays with $O(1)$ performance in the absence of contention. The crucial operation is a rotation of a *difference node* with the main body of the array. Using this implementation of functional arrays in the multi-object transaction protocol of the previous chapter will complete our re-implementation of non-blocking transactions, solving the large object problem.

Let's begin by reviewing the well-known functional array implementations. As mentioned previously, O'Neill and Burton [56] give an inclusive overview. Functional array implementations fall generally into one of three categories: *tree-based*, *fat-elements*, or *shallow-binding*.

Tree-based implementations typically have a logarithmic term in their complexity. The simplest is the persistent binary tree with $O(\ln n)$ look-up time; Chris Okasaki [55] has implemented a purely-functional random-access list with $O(\ln i)$ expected lookup time, where i is the index of the desired element.

Fat-elements implementations have per-element data structures indexed by a master array. Cohen [17] hangs a list of versions from each element in



NOTE. The array is of size 2 and is indexed by x and y. The initial array A is undefined, and B is defined as an update to A at index x by value 0. Similarly for C and D. The dark node is the root node which has the cache. White nodes are differential nodes which must first be rerooted before be read. Note that only the root node has the cache.

Figure 5.6: Shallow binding scheme for functional arrays, from [15, Figure 1].

the master array. O'Neill and Burton [56], in a more sophisticated technique, hang a splay tree off each element and achieve $O(1)$ operations for single-threaded use, $O(1)$ amortized cost when accesses to the array are "uniform", and $O(\ln n)$ amortized worst case time.

Shallow binding was introduced by Baker [9] as a method to achieve fast variable lookup in Lisp environments. Baker clarified the relationship to functional arrays in [8]. Shallow binding is also called *version tree arrays*, *trailer arrays*, or *reversible differential lists*. A typical drawback of shallow binding is that reads may take $O(u)$ worst-case time, where u is the number of updates made to the array. Tyng-Ruey Chuang [15] uses randomized cuts to the version tree to limit the cost of a read to $O(n)$ in the worst case. Single-threaded accesses are $O(1)$.

Our use of functional arrays is single-threaded in the common case, when transactions do not abort. Chuang's scheme is attractive because it limits the worst-case cost of an abort, with very little added complexity. In this section I will present a lock-free version of Chuang's randomized algorithm.

In shallow binding, only one version of the functional array (the *root*) keeps its contents in an imperative array (the *cache*). Each of the other versions is represented as a path of *differential nodes*, where each node describes the differences between the current array and the previous array. The difference is represented as a pair $\langle index, value \rangle$, representing the new value to be stored at the specified index. All paths lead to the root. An update to the functional array is simply implemented by adding a differential node pointing to the array it is updating.

The key to constant-time access for single-threaded use is provided by the read operation. A read to the root simply reads the appropriate value from the cache. However, a read to a differential node triggers a series of rotations that swap the direction of differential nodes and result in the current array acquiring the cache and becoming the new root. This sequence of rotations is called *re-rooting*, and is illustrated in Figure 5.6. Each rotation exchanges the root nodes for a differential node pointing to it, after which

the differential node becomes the new root and the root becomes a differential node pointing to the new root. The cost of a read is proportional to its re-rooting length, but after the first read accesses to the same version are $O(1)$ until the array is re-rooted again.

Shallow binding performs badly if read operations ping-pong between two widely separated versions of the array, as we will continually re-root the array from one version to the other. Chuang's contribution is to provide for *cuts* to the chain of differential nodes: once in a while we clone the cache and create a new root instead of performing a rotation. This operation takes $O(n)$ time, so we amortize it over n operations by randomly choosing to perform a cut with probability $1/n$.

Figure 5.7 shows the data structures used for the functional array implementation, and the series of atomic steps used to implement a rotation. The `Array` class represents a functional array; it consists of a `size` for the array and a pointer to a `Node`. There are two types of nodes: a `CacheNode` stores a value for every index in the array, and a `DiffNode` stores a single change to an array. `Array` objects that point to `CacheNodes` are roots.

In step 1 of the figure, we have a root array A and an array B whose differential node d_B points to A. The functional arrays A and B differ in one element: element x of A is z, while element x of B is y. We are about to rotate B to give it the cache, while linking a differential node to A.

Step 2 shows our first atomic action. We have created a new `DiffNode` d_A and a new `Array` C and linked them between A and its cache. The `DiffNode` d_A contains the value for element x contained in the cache, z, so there is no change in the value of A.

We continue swinging pointers until step 5, when can finally set the element x in the cache to y. We perform this operation with a DCAS operation that checks that C.`node` is still pointing to the cache as we expect. Note that a concurrent rotation would swing C.`node` in its step 1. In general, therefore, the location pointing to the cache serves as a reservation on the cache.

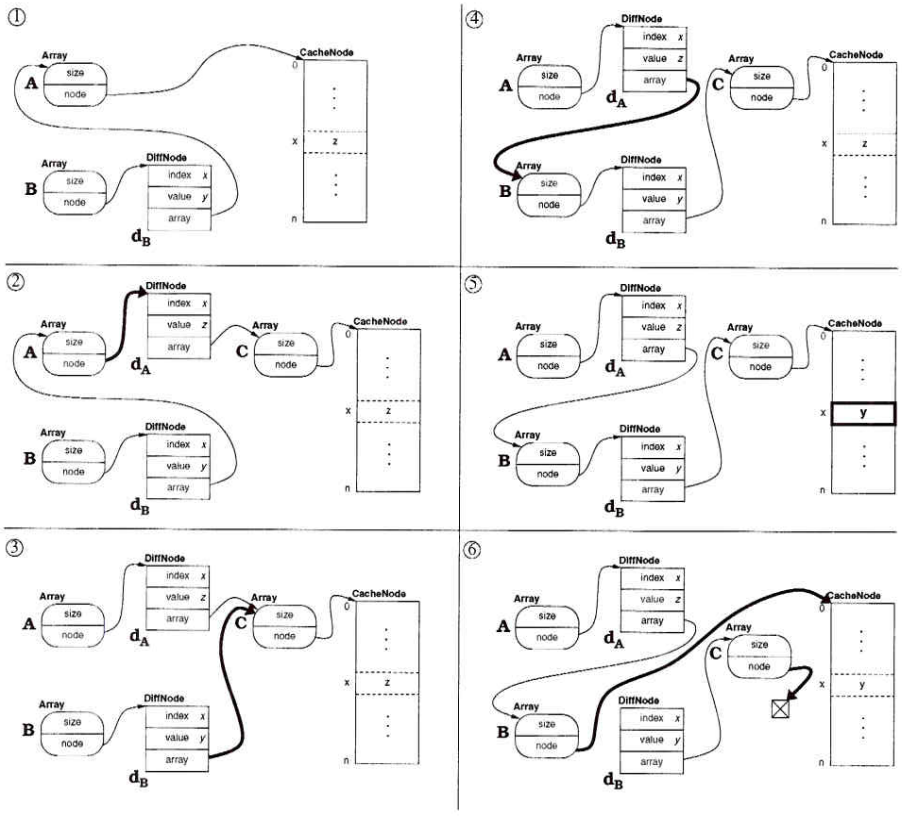


Figure 5.7: Atomic steps in FA-ROTATE(B). Time proceeds top-to-bottom on the left hand side, and then top-to-bottom on the right. Array A is a root node, and FA-READ(A, x) = z. Array B has the almost the same contents as A, but FA-READ(B, x) = y.

```
FA-UPDATE(A, i, v):
begin
  d ← new DiffNode(i, v, A)
  A′ ← new Array(A.size, d)
  return A′
end

FA-READ(A, i):
begin
retry:
  d_C ← A.node
  if d_C is a cache, then
    v ← A.node[i]
    if (A.node ≠ d_C)[consistency check]
      goto retry
    return v
  else
    FA-ROTATE(A)
    goto retry
end
```

Figure 5.8: Implementation of lock-free functional array using shallow binding and randomized cuts (part 1).

```
FA-ROTATE(B):
begin
retry:
  d_B ← B.node    [step (1): assign names as per Figure 5.7.]
  A ← d_B.array
  x ← d_B.index
  y ← d_B.value
  z ← FA-READ(A, x)                    [rotates A as side effect]

  d_C ← A.node
  if d_C is not a cache, then
    goto retry

  if (0 = (random mod A.size))                          [random cut]
    d'_C ← copy of d_C
    d'_C[x] ← y
    s ← DCAS(d_C, d_C, &(A.node), d_B, d'_C, &(B.node))
    if (s ≠ SUCCESS) goto retry
    else return

  C ← new Array(A.size, d_C)
  d_A ← new DiffNode(x, z, C)

  s ← CAS(d_C, d_A, &(A.node))                          [step (2)]
  if (s ≠ SUCCESS) goto retry

  s ← CAS(A, C, &(d_B.array))                           [step (3)]
  if (s ≠ SUCCESS) goto retry

  s ← CAS(C, B, &(d_A.array))                           [step (4)]
  if (s ≠ SUCCESS) goto retry

  s ← DCAS(z, y, &(d_C[x]), d_C, d_C, &(C.node))        [step (5)]
  if (s ≠ SUCCESS) goto retry

  s ← DCAS(d_B, d_C, &(B.node), d_C, nil, &(C.node))    [step (6)]
  if (s ≠ SUCCESS) goto retry
end
```

Figure 5.9: Implementation of lock-free functional array using shallow binding and randomized cuts (part 2).

Thus in step 6 we need to again use DCAS to simultaneously swing C.node away from the cache as we swing B.node to point to the cache.

Figures 5.8 and 5.9 present pseudocode for FA-ROTATE, FA-READ, and FA-UPDATE. Note that FA-READ also uses the cache pointer as a reservation, double-checking the cache pointer after it finishes its read to ensure that the cache hasn't been stolen from it.

Let us now consider cuts, where FA-READ clones the cache instead of performing a rotation. Cuts also check the cache pointer to protect against concurrent rotations. But what if the cut occurs while a rotation is mutating the cache in step 5? In this case the only array adjacent to the root is B, so the cut must be occurring during an invocation of FA-ROTATE(B). But then the differential node d_B will be applied after the cache is copied, which will safely overwrite the mutation we were concerned about.

Note that with hardware support for small transactions [36] we could cheaply perform the entire rotation atomically, instead of using this six-step approach.

So??
What is the point??
Does this help our
hurt performance??