Software Atomic Transactions in FLEX

C. Scott Ananian

November 28, 2001
Revision: 1.23

1 Data Structures

Transaction support uses the same "inflated object" extension mechanism which is used for thread synchronization, clustered heaps, JNI data, and finalization for certain garbage collectors. The basic object data structure, as shown in figure 1, is unchanged, although fields containing the specified `FLAG_VALUE` have different semantics. The flag value is used to indicate that the field value is "not here"; that is, the code must consult the transaction information to find the field's current value. This happens very rarely even when no transaction is associated with the object; Shasta [1] has shown that the overhead entailed by such "false" transactions is expected to be extremely low.

The "inflated object" data structure, shown in figure 2, has an "object versions" linked list associated with it. Each entry conceptually stores information about a different, possibly-uncommitted, version of the object. One of these versions is guaranteed to be "most recent committed", and it is guaranteed that the "most recent committed" version will be the first committed version in the list.

The commit record structure is shown in figure 4. The pointer to the commit record also serves as a transaction identifier. Transactions can be nested, therefore every commit record may contain a pointer to a "parent" transaction which it is dependent upon. The commit record also contains a three-value "state" field which indicates whether the transaction it represents has been committed or aborted yet.

The "object version" structure, shown in figure 3, is similar to the object structure shown in figure 1 so that field pointers can be converted easily. Each `vinfo` structure identifies an object version which is associated with a particular transaction (via `transid`) which has mutated the object. Readers are associated with each mutated version using the `readers` list, which is inlined to avoid an extra memory dereference in the hot path through `ReadTrans` (figure 7).

```
/* the oobj structure tells you what's inside the object layout. */
struct oobj {
  struct claz *claz;
  /* if low bit is one, then this is a fair-dinkum hashcode. else, it's a
   * pointer to a struct inflated_oobj. this pointer needs to be freed
   * when the object is garbage collected, which is done w/ a final-
izer. */
  union { ptroff_t hashcode; struct inflated_oobj *inflated; } hashunion;
#ifdef WITH_TRANSACTIONS
#define FLAG_VALUE 0xCACACACA
  /* consult transaction to determine value of any field with FLAG_VALUE */
#endif
  /* fields below this point */
  char field_start[0];
};
```

Figure 1: The object structure in the FLEX runtime.

Subtransactions cause a possible race condition when traversing the versions list. The parent of each subtransaction will have a separate "pre-subtransaction" version entry in order to allow retrying an aborting subtransaction. A non-transactional reader may examine the subtransaction and find that it is still waiting and move on to the next item in the versions list, which will be the subtransaction's parent. But if both the subtransaction and then its parent are then committed before the non-transactional reader proceeds, it may erroneously fetch the "pre-subtransaction" version of the now-committed parent instead of the correct version associated with the newly-committed subtransaction.

To prevent this race, there are two "next" fields for the versions list. The `anext` field is followed if the version's transaction has been aborted. This allows correct fail back to the parent transaction's version in this case. However, if the version's transaction is in the `WAITING` state (review figure 4), we skip examining its parent transaction(s), avoiding the possible race, by following the `wnext` field.

These data structures allow a single-writer multiple-readers transaction consistency scheme to be implemented. To avoid polling where possible, our transactions prefer committing suicide to aborting another transaction when consistency only allows a single outstanding transaction. At the end of every versions list we guarantee there is at least one committed transaction. Other ordering is also preserved in the list: for every `vinfo` structure v, v`->transid` is a subtransaction of v`->anext->transid` and *not* a subtransaction of v`->wnext->transid`. Every reader is a subtransaction of every writer; formally, for every reader r in v`->readers`, r`->transid` is a subtransaction of v`->transid`. Finally, to enforce consistency, there are never any non-aborted readers of v's parent transaction

```
/* the inflated_oobj structure has various bits of information that we
 * want to associate with *some* (not all) objects. */
struct inflated_oobj {
  ptroff_t hashcode; /* the real hashcode, since we've booted it */
  void *jni_data; /* information associated with this object by the JNI */
  void (*jni_cleanup_func)(void *jni_data);
  int use_count; /* determines when we can deflate the object */
  /* TRANSACTION SUPPORT */
#if WITH_TRANSACTIONS
  struct vinfo *first_version; /* linked list of object versions */
#endif
  /* locking information */
#if WITH_HEAVY_THREADS || WITH_PTH_THREADS
  pthread_t tid; /* can be zero, if no one has this lock */
  jint nesting_depth; /* recursive lock nesting depth */
  pthread_mutex_t mutex; /* simple (not recursive) lock */
  pthread_cond_t  cond; /* condition variable */
  pthread_rwlock_t jni_data_lock; /*read/write lock for jni_data field, above*/
#endif
#ifdef WITH_CLUSTERED_HEAPS
  struct clustered_heap * heap;
  void (*heap_release)(struct clustered_heap *);
#endif
#ifdef BDW_CONSERVATIVE_GC
  /* for cleanup via finalization */
  GC_finalization_proc old_finalizer;
  GC_PTR old_client_data;
#endif
};
```

Figure 2: The "inflated object" structure in the FLEX runtime. Fields specific to transaction support are in boldface.

```
#if WITH_TRANSACTIONS
/* A simple linked list of transaction identifiers. */
struct tlist {
  struct commitrec *transid; /* transaction id */
  struct tlist *next; /* next version. */
};

/* The vinfo structure provides values for a given version of the ob-
ject. */
struct vinfo {
  struct commitrec *transid; /* transaction id */
  struct tlist readers; /* list of readers.  first node is inlined. */
  struct vinfo *anext; /* next version to look at if tran-
sid is aborted. */
  struct vinfo *wnext; /* next version to look at if transid is wait-
ing. */
  /* fields below this point */
  char field_start[0];
};
#endif
```

Figure 3: The “version” structure in the FLEX runtime.

```
#if WITH_TRANSACTIONS
/* The commit record for a (possibly nested) transaction */
struct commitrec {
  /* The transaction that this depends upon, if any. */
  struct commitrec *parent;
  /* The 'state' should be initialized to W and write-once to C or A. */
  enum { WAITING=0, COMMITTED, ABORTED } state;
};
#endif
```

Figure 4: Commit record structure.

```
Read(struct oobj *o, int n) =
 1: x ← o->field[n]
 2: if x = FLAG_VALUE then
 3:    v ← o->hashunion.inflated->first_version
 4:    u ← 0 /* count uncommitted transactions */
 5:    loop
 6:       Assert: v ≠ null
 7:       case StateP''(v) matching
 8:          COMMITTED ⇒
 9:             Assert: ∀n, o->field[n] = FLAG_VALUE ⇒ v->field[n] is valid
10:             break loop
11:          ABORTED ⇒
12:             v ← v->anext
13:          WAITING ⇒
14:             u ← u + 1
15:             v ← v->wnext
16:       end case
17:    end loop
18:    x ← v->field[n]
19:    if u = 0 ∧ x ≠ FLAG_VALUE then
20:       /* XXX: race here */
21:       o->field[n] ← x /* store must be atomic */
22:       dec_use(o) /* will deflate when all fields are copied back */
23: return x
```

Figure 5: Algorithm for non-transactional read of a field.

which are subtransactions of v; that is, for all readers r in v`->anextreaders`, v`->transid` is a subtransaction of r`->transid` (and not the other way around), unless r`->transid` has been aborted.

2 Algorithms

Blah blah blah

References

[1] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceed-*

```
Write(struct oobj *o, int n, <ty> x) =
 1: if o->field[n] = FLAG_VALUE then
 2:    if PossiblyDownGrade(o, n) then /* try to avoid aborting anything */
 3:       goto line 1
 4:    v ← o->hashunion.inflated->first_version
 5:    while AbortTransaction(v->transid) ≠ COMMITTED do
 6:       /* prune list for the sake of writes w/o intervening reads */
 7:       /* XXX: potential race here — v could be already removed */
 8:       v ← v->anext
 9:       o->hashunion.inflated->first_version ← v
10:    /* v is a committed transaction */
11:    v->anext ← v->wnext ← null /* atomic stores */
12:    /* abort all readers of this committed transaction */
13:    r ← &(v->readers)
14:    while r ≠ null do
15:       AbortTransaction(r->transid)
16:       r ← r->next
17: else if x = FLAG_VALUE then
18:    /* XXX: define CreateNewVersion (watch those races) */
19:    v ← CreateNewVersion(o, null, o)
20: else
21:    v ← o
22: v->field[n] ← x
```

Figure 6: Algorithm for non-transactional write of a field.

ReadTrans(`struct commitrec *`c, `struct oobj *`o, `int` n) =

1: `InflateObject`(o)
2: $v \leftarrow o$`->hashunion.inflated->first_version`
3: **if** $v =$ `null` **then**
4: $v \leftarrow$ CreateNewVersion(o, `null`, o)
5: $r \leftarrow$ `&`(v`->readers`)
6: **repeat**
7: **if** r`->transid` $= c$ **then**
8: **goto** line 27
9: $r \leftarrow r$`->next`
10: **until** $r =$ `null`
11: **case** StateP″(v) **matching**
12: `ABORTED` ⇒
13: $v \leftarrow v$`->anext`
14: **goto** line 5
15: `COMMITTED` ⇒
16: **goto** line 26
17: `WAITING` ⇒
18: /* *abort unless c is a non-aborted subtransaction of v*`->transid` */
19: **if not** IsNAST(c, v) **then**
20: **if** PossiblyDownGrade(o, n) **then**
21: **goto** line 5 /* *last chance for life* */
22: **else**
23: commit suicide /* *can't guarantee consistency* */
24: **goto** line 26
25: **end case**
26: add c to head of readers list, atomically.
27: /* *v has correct version* */
28: **if** o`->field`$[n] \neq$ `FLAG_VALUE` **then**
29: /* *XXX: race here, most likely* */
30: copy current value somewhere (how far over?)
31: `inc_use`(o)
32: o`->field`$[n] \leftarrow$ `FLAG_VALUE`
33: /* *XXX: even if this transaction is aborted now, we still register a dependency on this field.* */
34: **return** v`->field`$[n]$

Figure 7: Algorithm for transactional read of a field.

WriteTrans(`struct commitrec *`c, `struct oobj *`o, `int` n, `<ty>` x) =

```
 1: InflateObject(o)
 2: v ← o->hashunion.inflated->first_version
 3: if v = null then
 4:    v ← CreateNewVersion(o, c, o)
 5: else if v->transid ≠ c then
 6:    case StateP″(v) matching
 7:       ABORTED ⇒
 8:          v ← v->anext
 9:          goto line 5
10:       COMMITTED ⇒
11:          goto line 21
12:       WAITING ⇒
13:          /* abort unless c is a non-aborted subtransaction of v->transid */
14:          if not IsNAST(c, v) then
15:             if PossiblyDownGrade(o, n) then
16:                goto line 5 /* last chance for life */
17:             else
18:                commit suicide /* can't guarantee consistency */
19:          goto line 21
20:    end case
21:    kill some of the readers
22:    v ← CreateNewVersion(o, c, v)
23: /* v has correct version */
24: if o->field[n] ≠ FLAG_VALUE then
25:    watch for races here
26:    copy current value somewhere (how far over?)
27:    inc_use(o)
28:    o->field[n] ← FLAG_VALUE
29: v->field[n] ← x
```

Figure 8: Algorithm for transactional write of a field.

StateP(`struct commitrec *`c) =

```
1: loop
2:    if c = null then
3:       return COMMITTED
4:    s ← c->state /* cache this read to prevent races */
5:    if s ≠ COMMITTED then
6:       return s
7:    c ← c->parent
8: end loop
```

Figure 9: Simple algorithm for determining transaction status.

StateP′(`struct commitrec **`c_p) =

```
1: c ← *c_p
2: s ← COMMITTED
3: l ← false
4: while c ≠ null do
5:    s ← c->state /* atomic */
6:    if s ≠ COMMITTED then
7:       break
8:    c ← c->parent
9:    l ← true
10: if l then /* l indicates a change has been made */
11:    *c_p ← c /* atomic */
12: return ⟨s, l⟩
```

Figure 10: Improved algorithm for determining transaction status — does pruning.

```
StateP''(struct vinfo *v) =
 1: case StateP'(&(v->transid)) matching
 2:   ⟨ABORTED, l⟩ ⇒
 3:     /* XXX: potential race here — v could be already removed */
 4:     /* the following store ought to be atomic */
 5:     o->hashunion.inflated->first_version ← v->anext
 6:     return ABORTED
 7:   ⟨COMMITTED, l⟩ ⇒
 8:     /* these stores should be atomic (but no one will read them) */
 9:     /* this might be better done by the gc */
10:     v->anext ← v->wnext ← null
11:     return COMMITTED
12:   ⟨WAITING, l⟩ ⇒
13:     if l then /* transid was pruned; snip out parents */
14:       while v->transid = v->anext->transid do
15:         v->anext ← v->anext->anext /* XXX: race here. */
16:     return WAITING
17: end case
```

Figure 11: Even better pruning algorithm for determining transaction status.

```
AbortTransaction(struct commitrec *c) =
1: if c = null then
2:   return COMMITTED
3: loop
4:   s ← c->state
5:   if s ≠ WAITING then
6:     return s
7:   compare_and_swap(&c->state, s, ABORTED)
8: end loop
```

Figure 12: Algorithm for (possibly) aborting a transaction. Change the constant `ABORTED` in line 7 to `COMMITTED` to obtain the `CommitTransaction` algorithm.

```
IsNAST(struct commitrec *c_s, struct vinfo *v) =
 1: c_p ← v->transid
 2: if c_s ≠ c_p then
 3:    if c_p ≠ null ∧ c_p->state = COMMITTED then
 4:       repeat
 5:          c_p ← c_p->parent
 6:       until c_p = null ∨ c_p->state ≠ COMMITTED
 7:       v->transid ← c_p /* atomic */
 8:       check to see whether successor needs to be pruned.
 9:    if c_p = null then /* committed all the way down */
10:       v->anext ← v->wnext ← null /* prune */
11:    else if c_p->state = ABORTED then
12:       return false /* c_s either not subtrans, or is aborted. */
13:    else
14:       /* the possible WAITING-becomes-COMMITTED race */
15:       /* between l.3/l.6 and l.11 is not harmful. */
16:       while c_s ≠ c_p do
17:          if c_s->state = ABORTED then
18:             return false /* aborted */
19:          c_s ← c_s->parent
20:          if c_s = null then
21:             return false /* not a subtransaction */
22: return (StateP(c_s) ≠ ABORTED)
```

Figure 13: Algorithm for determining whether c_s is a non-aborted subtransaction of v->transid

ings of the seventh international conference on architectural support for programming languages and operating systems, pages 174–185, Cambridge, Oct. 1996.