Software Transactional Memory & Automatic Mutual Exclusion

Martin Steffen

Oslo
10. Feb. 2009

Abstract

This is the handout version talk about software transactional memory and automatic mutual exclusion. The wisdom is taken from [Abadi et al., 2008] and [Jagannathan et al., 2005], for the most part.

1 Introduction

Motivation

- concurrency ⇒ concurrency control
- nowaday's languages: lock-based (good ol' mutex)
- disadvantages:
  - low-level of abstraction
  - difficult to reason about
  - "conservative" protection ⇒ performance penalty / deadlocks
  - pessimistic approach to concurrency control
- here: "optimistic" approach
  - reduce crit-secs, more concurrency ⇒ non-blocking

Transactions

- coming from the data-base community
- control abstraction
- important correctness/failure properties: ACID transaction semantics = "illusion" of mutex
  1. atomicity
  2. isolation
  3. consistency
  4. durability

2 Transactional Java

TFJ

- taken from [Jagannathan et al., 2005]
- extending Featherweight Java with transactions
  - state
  - multi-threading (of course)
  - transactions
- featuring: nested and multi-threaded transactions
- operational semantics, 2 concretizations
  - versioning
  - 2-phase locking
- correctness proof: serializability

Why are transactions more high-level?

Listing 1: TFJ example

```
class Transactor {
 u: Updater;
 r: Runner;
 init (r: Runner , u: Updater ) { this.u := u;
                                  this.r := r;
                                  this }

 run () {
   onacid
     this.u.update();              // write
     this.r.run();                 // spawn intervening activity
     thus.u.n.val;                 // read
   commit
 }
}
```

Syntax

P	$::=$	$0 \mid P \parallel P \mid t\langle e \rangle$	process
L	$::=$	$class C\{\vec{f}; \vec{M}\}$	class definition
M	$::=$	$m(\vec{x})\{e\}$	method
e	$::=$	$x \mid e.f \mid e, m(\vec{e}) \mid e.f := e$	expression
	$\mid$	$new C \mid spawn\ e \mid onacid \mid commit \mid null$	
v	$::=$	$r \mid v, f \mid v.m(\vec{v}) \mid b.f := v$	values/basic expressions

- basically 2 additions:

  - onacid : start a transaction
  - commit : end a transaction

Semantics

- given operationally (SOS, as usual . . .)
  - labelled transition system
  - evaluation-contexts
- 2 "stages":
  1. first "general" semantics
  2. afterwards: 2 concretizations
- 2-level semantics
  1. local = per thread
  2. global = many threads

2.1 Operational semantics without transactions

Underlying semantics: no transactions

- for illustration here, only
- no separation in local $\leftrightarrow$ global steps
- no transaction handling (but concurrency)
- heap-manipulations (read, write, extend) left "unspecified"
- configuration (local/global): $\Gamma \vdash e$

Operational semantics: no transactions

$$\frac{read(r,\Gamma) = C(\vec{u}) \qquad fields(C) = \vec{f}}{\Gamma \vdash r.f_i \xrightarrow{rd\ r} \Gamma \vdash u_i}\ \text{R-FIELD}$$

$$\frac{read(r,\Gamma) = C(\vec{r}) \qquad write(r \mapsto C(\vec{r}) \downarrow_i^{r'}, \Gamma) = \Gamma'}{\Gamma \vdash r.f_i := r' \xrightarrow{wr\ r r'} \Gamma' \vdash r'}\ \text{R-ASSIGN}$$

$$\frac{read(r,\Gamma) = C(\vec{r}) \qquad mbody(m,C) = (\vec{x}, e)}{\Gamma \vdash r.m(\vec{r}) \xrightarrow{rd\ r} \Gamma \vdash e[\vec{r}/\vec{x}][r/this]}\ \text{R-INVOKE}$$

$$\frac{r\ fresh \qquad extend(r \mapsto C(\vec{\mathsf{null}}), \Gamma) = \Gamma'}{\Gamma \vdash \mathsf{new}\, C() \xrightarrow{xt\ r} \Gamma' \vdash r}\ \text{R-NEW}$$

$$\frac{\begin{array}{c} P = P'' \parallel n\langle E[\mathsf{spawn}\, e]\rangle \qquad P' = P'' n\langle E[\mathsf{null}]\rangle \parallel n'\langle e'\rangle \\ n'\ fresh \qquad spawn(n, \mathcal{E}', \Gamma) = \Gamma' \end{array}}{\Gamma \vdash P \xrightarrow{sp\ n'} \Gamma' \vdash P'}\ \text{R-SPAWN}$$

2.2 Transactional semantics

Introducing transactions

- as said: syntax: onacid + commit
- steps: split into 2 levels
  1. local : per thread
  2. global : "inter"-thread
- more complicated " memory model "
  - each thread has a local copy
  - how that exactly works ⇒ depending on the kind of transaction implementation (see later)
- general idea: optimistic approach
  - each thread works on its local copy (no locks, no regard of others)
  - local copy ⇒ isolation
  - when committing : check for conflicts ⇒
    - no: ⇒ make the effect visible
    - yes: ⇒ abort

Transactions and threads

- both are dynamic
  - thread creation by spawn
  - transaction "creation" by onacid
- transaction structure: nested [1]
  - a transaction can contain inner transactions
  - child transactions must commit before outer transaction
  - child transaction
    - commits ⇒ effects become visible to outer transaction
    - aborts ⇒ outer transaction does not abort
- relationship:
  - each thread inside an enclosing transaction[2]
  - "multi" threads in one transaction

Local steps

- steps concerning one thread
- basic "single-threaded", "non-transactional" steps
- local state/configuration:
  - "simple" expression e + local environment $\mathcal{E}$ [3]

$$\mathcal{E} \vdash e$$

- $\mathcal{E}$:
  - per transaction (labelled with l): local (partial) "state" = assoc of references to values
  - manipulated by read/write/extend
  - details determine the transactional model
  - Note: read-access may change $\mathcal{E}$

[1] Thread structure: flat. One could make a hierarchical "father-child" structure, but it's irrelevant here.
[2] or toplevel
[3] The paper itself is undecided whether to call it transaction environment or a sequence of transaction environments.

Local steps: rules

$$\frac{read(r,\mathcal{E}) = \mathcal{E}', C(\vec{u}) \qquad fields(C) = \vec{f}}{\mathcal{E} \vdash r.f_i \xrightarrow{rd\ r} \mathcal{E}' \vdash u_i}\ \text{R-FIELD}$$

$$\frac{read(r,\mathcal{E}) = \mathcal{E}', C(\vec{r}) \qquad write(r \mapsto C(\vec{r}) \downarrow_i^{r'}, \mathcal{E}') = \mathcal{E}''}{\mathcal{E} \vdash r.f_i := r' \xrightarrow{wr\ r r'} \mathcal{E}'' \vdash r'}\ \text{R-ASSIGN}$$

$$\frac{read(r,\mathcal{E}) = \mathcal{E}', C(\vec{r}) \qquad mbody(m, C) = (\vec{x}, e)}{\mathcal{E} \vdash r.m(\vec{r}) \xrightarrow{rd\ r} \mathcal{E}' \vdash e[\vec{r}/\vec{x}][r/this]}\ \text{R-INVOKE}$$

$$\frac{r\ fresh \qquad extend(r \mapsto C(\vec{\mathsf{null}}), \mathcal{E}) = \mathcal{E}'}{\mathcal{E} \vdash \mathsf{new}\, C() \xrightarrow{xt\ r} \mathcal{E}' \vdash r}\ \text{R-NEW}$$

Global steps

- behavior of multiple interacting threads

$$n_1\langle e_1\rangle \parallel \ldots \parallel n_k\langle e_k\rangle = P$$

- global state/configuration

$$\Gamma \vdash P$$

  = program P + global environment Γ = local environment per thread:

$$n_1{:}\mathcal{E}_1, \ldots n_k{:}\mathcal{E}_k \vdash n_1\langle e_1\rangle \ldots n_k\langle e_k\rangle$$

- transitions

$$\Gamma \vdash P \overset{\alpha}{\Longrightarrow}_n \Gamma' \vdash P'$$

Global steps: rules (1)

$$\frac{P = P'' \parallel n\langle e\rangle \qquad \mathcal{E} \vdash e \xrightarrow{\alpha} \mathcal{E}' \vdash e' \qquad P' = P'' \parallel n\langle e'\rangle \qquad reflect(n, \mathcal{E}', \Gamma) = \Gamma'}{\Gamma \vdash P \overset{\alpha}{\Longrightarrow}_n \Gamma' \vdash P'}\ \text{G-PLAIN}$$

$$\frac{P = P'' \parallel n\langle E[\mathsf{spawn}\ e]\rangle \qquad P' = P'' \parallel n\langle E[\mathsf{null}]\rangle \parallel n'\langle e'\rangle \qquad n'\ fresh \qquad spawn(n, \mathcal{E}', \Gamma) = \Gamma'}{\Gamma \vdash P \overset{sp\ n'}{\Longrightarrow}_n \Gamma' \vdash P'}\ \text{G-SPAWN}$$

$$\frac{P = P' \parallel n\langle r\rangle \qquad \Gamma = n{:}\mathcal{E}, \Gamma'}{\Gamma \vdash P \overset{ki}{\Longrightarrow}_n \Gamma' \vdash P''}\ \text{G-THKILL}$$

Global steps: transaction handling

- start a transaction:
  - basically straightforward
  - create a new transaction label
- finish a transaction (commit)
  - "publish" the result
  - slightly more complex, because of *multi-threaded* transactions

  ⇒ join all threads that are about to commit the transaction in question
  - transaction in question: the "innermost" meant by the commit-action

Global steps: transaction rules (2)

$$\frac{P = P'' \parallel n\langle E[\mathsf{onacid}]\rangle \qquad P' = P'' \parallel n\langle E[\mathsf{null}]\rangle \qquad l \; \mathit{fresh} \qquad \mathit{start}(l, n, \Gamma) = \Gamma'}{\Gamma \vdash P \overset{ac}{\Longrightarrow}_n \Gamma' \vdash P'} \text{ G-TRANS}$$

$$\frac{\begin{array}{c} P = P'' \parallel n\langle E[\vec{\mathsf{commit}}]\rangle \qquad P' = P'' \parallel n\langle E[\vec{\mathsf{null}}]\rangle \\ \Gamma = \Gamma'', n{:}\mathcal{E} \qquad \mathcal{E} = \mathcal{E}', l{:}\varrho \qquad \mathit{intranse}(l, \Gamma) = \vec{n} = n_1 \ldots n_k \\ \mathit{commit}(\vec{n}, \vec{\mathcal{E}}, \Gamma) = \Gamma' \quad n_1{:}\mathcal{E}_1, n_2{:}\mathcal{E}_2, \ldots n_k{:}\mathcal{E}_k \in \Gamma \quad \vec{\mathcal{E}} = \mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_k \end{array}}{\Gamma \vdash P \overset{co}{\Longrightarrow}_n \Gamma' \vdash P'} \text{ G-COMM}$$

2.3 Versioning semantics

Versioning semantics

- so far: the core has been left abstract
- one concretization of the general semantics
- concretization of the memory manipulations
- local environment $\mathcal{E}$

  $l_1{:}\varrho_1, \ldots l_k{:}\varrho_k$
- l: transaction label
- ϱ:
  - log (of that transaction/of the given thread)
  - (part of the) dynamic context of the transaction l
- $\mathcal{E}$ is ordered ,
  - current enclosing one: on the right
  - reflects the nesting of transactions

Environment manipulations (local)

remember the local steps, for one thread $\mathcal{E} \vdash r \twoheadrightarrow \mathcal{E}' \vdash r'$

read: given a reference r, find the assoc. value

- look-up the value for r, not necessary in the innermost (= rightmost) transaction
- log the found value for the innermost transaction, i.e., copy/record it into that transactions log

write: similarly, the old value is logged locally, too

extend: similarly, no old value is logged (fresh reference)

Environment manipulation (local)

$$\frac{\mathcal{E} = \mathcal{E}', l{:}\varrho \qquad \mathit{findlast}(r, \mathcal{E}) = C(\vec{r}) \qquad \mathcal{E}'' = \mathcal{E}', l{:}(\varrho, r \mapsto C(\vec{r}))}{\mathit{read}(r, \mathcal{E}) = \mathcal{E}'', C(\vec{r})}\ \text{E-READ}$$

$$\frac{\mathcal{E} = \mathcal{E}', l{:}\varrho \quad \mathit{findlast}(r, \mathcal{E}) = D(\vec{r}') \quad \mathcal{E}'' = \mathcal{E}', l{:}(\varrho, r \mapsto D(\vec{r}'), r \mapsto C(\vec{r}))}{\mathit{write}(r \mapsto C(\vec{r}), \mathcal{E}) = \mathcal{E}''}\ \text{E-WRITE}$$

$$\frac{\mathcal{E} = \mathcal{E}', l{:}\varrho \qquad \mathcal{E}'' = \mathcal{E}', l{:}(\varrho, r \mapsto C(\vec{r}))}{\mathit{extend}(r \mapsto C(\vec{r}), \mathcal{E}) = \mathcal{E}''}\ \text{E-EXTEND}$$

$$\frac{\Gamma = n{:}\mathcal{E}, \Gamma' \qquad \Gamma'' = n'{:}\mathcal{E}', \Gamma}{\mathit{spawn}(n, n', \Gamma) = \Gamma''}\ \text{E-SPAWN}$$

Environment manipulation: for transactions

- 2 operations: *start* and *commit*

  start:
  - easy ("optimistic")
  - create a new label for the transaction
  - start with an empty log for the new transaction

  commit:
  - more tricky.
  - propagate ("reflect") bindings from the transaction to the parent
  - commit only, if no conflict is detected
  - conflict: values used (r/w) in l must coincide with values as in parent transaction

Environment manipulation: transactions

$$\frac{\Gamma = n{:}\mathcal{E}, \Gamma' \qquad \Gamma'' = (l{:}(\mathcal{E}, l{:}\langle\rangle)), \Gamma}{start(l, n, \Gamma) = \Gamma''}\ \text{E-START}$$

$$\frac{}{commit(\langle\rangle, \langle\rangle, \Gamma) = \Gamma}\ \text{E-COMMIT}_1$$

$$\frac{\begin{array}{c}\mathcal{E} = \mathcal{E}', l{:}\varrho \qquad readset(\varrho, \langle\rangle) = \varrho' \qquad writeset(\varrho, \langle\rangle) = \varrho'' \\ check(\varrho', \mathcal{E}') \qquad \mathcal{E}' = \mathcal{E}'', l'{:}\varrho''' \qquad reflect(n, (\mathcal{E}'', l'{:}\varrho''', \varrho''), \Gamma) = \Gamma' \\ commit(\vec{n}, \vec{\mathcal{E}}, \Gamma') = \Gamma''\end{array}}{commit(n\ \vec{n}, \mathcal{E}\ \vec{\mathcal{E}}, \Gamma) = \Gamma''}\ \text{E-COMMIT}_2$$

Checking an environment

Modsets

Modsets

$$\frac{}{readset(\langle\rangle, \_) = \langle\rangle}$$

$$\frac{\varrho = u \mapsto C(\vec{u}) \qquad u \notin \vec{r} \qquad readset(\varrho'', \vec{r}u) = \varrho'}{readset(\varrho, \vec{r}) = u \mapsto C(\vec{u}), \varrho'}$$

$$\frac{\varrho = u \mapsto C(\vec{u}), \varrho'' \qquad u \in \vec{r} \qquad readset(\varrho'', \vec{r}) = \varrho'}{readset(\varrho, \vec{r}) = \varrho'}$$

$$\frac{}{writeset(\langle\rangle, \_) = \langle\rangle}$$

$$\frac{\varrho?r \mapsto C(\vec{r}), \varrho'' \qquad writeset(\varrho'', \varrho') = \varrho''' \qquad r \mapsto C(\vec{r}) \neq first(r, \varrho')}{writeset(\varrho, \varrho') = u \mapsto D(\vec{u}), \varrho'''}$$

2.4 Two-phase locking

Two-phase locking

- different instantiation of the general semantics, slight alteration
- based on locks
- pessimistic
- two phases:

1. first get hold of all the locks needed for a transaction
2. then release them again

- strict: all acquiring is done before all releasing.

Two-phase locking transactional semantics

- "slight" alteration of the previous one
- transaction & locks
  - objects have locks for protection
  - locks are held by transactions[4].
  - enter a transaction: all locks held by transaction or prefix
  - creating an object.
- to support locking
  - unique transaction label l_L +
  - lock environment ϱ_L.
- ϱ stores lock ownership (per reference): which transactions hold the lock = sequence to reflected nesting
- given $l_1, l_2, \ldots, l_k$
- change of lock-ownership:
  - acquire by grabbing
  - commit by child, and propagate the lock upwards

Environment manipulation with locks (local)

$$\frac{\mathcal{E} = \mathcal{E}', l{:}\varrho \qquad \mathit{findlast}(r, \mathcal{E}) = C(\vec{r}) \qquad \mathcal{E}'' = \mathcal{E}', l{:}(\varrho, r \mapsto C(\vec{r})) \qquad \mathit{checklock}(r, \mathcal{E}) = \top}{\mathit{read}(r, \mathcal{E}) = \mathcal{E}'', C(\vec{r})} \text{ E-READ}$$

$$\frac{\mathit{findlast}(r, \mathcal{E}) = D(\vec{r}') \qquad \mathcal{E}' = \mathit{acquirelock}(r, E) \qquad \mathcal{E}' = \mathcal{E}'', l{:}\varrho \qquad \mathcal{E}''' = \mathcal{E}'', l{:}(\varrho, r \mapsto D(\vec{r}'), r \mapsto C(\vec{r}))}{\mathit{write}(r \mapsto C(\vec{r}), \mathcal{E}) = \mathcal{E}'''} \text{ E-WRITE}$$

$$\frac{\mathit{acquirelock}(r, E) = \mathcal{E}', l{:}\varrho \qquad \mathcal{E}'' = \mathcal{E}', l{:}(\varrho, r \mapsto C(\vec{r}))}{\mathit{extend}(r \mapsto C(\vec{r}), \mathcal{E}) = \mathcal{E}''} \text{ E-EXTEND}$$

---

[4] Note the difference to multi-threaded Java

Environment manipulation: transactions

$$\frac{\Gamma = n{:}\mathcal{E}, \Gamma' \qquad \Gamma'' = (l{:}(\mathcal{E}, l{:}\langle\rangle)), \Gamma}{start(l, n, \Gamma) = \Gamma''}\ \text{E-START}$$

$$\frac{}{commit(\langle\rangle, \langle\rangle, \Gamma) = \Gamma}\ \text{E-COMMIT}_1$$

$$\frac{\begin{array}{c}\mathcal{E} = l_L{:}\varrho_L, \mathcal{E}' \qquad \varrho'_L = release(l(\mathcal{E}), \varrho_L) \qquad \mathcal{E}'' = l_L{:}\varrho'_L, \mathcal{E}' \\ reflect(n, (\mathcal{E}'', l'{:}\varrho''', \varrho''), \Gamma) = \Gamma' \qquad commit(\vec{n}, \vec{\mathcal{E}}, \Gamma') = \Gamma''\end{array}}{commit(n\ \vec{n}, \mathcal{E}\ \vec{\mathcal{E}}, \Gamma) = \Gamma''}\ \text{E-COMMIT}_2$$

Further development in the paper

- After the formalization: prove some "soundness results"
  - ultimately: "ACID", serialization
  - techniques: "permutation lemmas"

3 Conclusion

Further reading

- wait-free data structures
- old, related theoretical results: [Lipton, 1975]: theory of left/right movers
- [Herlihy and Wing, 1990]: linearizability for concurrent objects
- futures [Welc et al., 2005]
- transactions for Java [Garthwaite and Nettles, 1996]
- software transactional memory [Shavit and Toitu, 1995]
- automatic mutual exclusion [Abadi et al., 2008] and originally [Isard and Birell, 2007]
- and another POPL'08 paper?

References

References

[Abadi et al., 2008] Abadi, M., Birell, A., Harris, T., and Isard, M. (2008). Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of POPL '08*. ACM.

[Garthwaite and Nettles, 1996] Garthwaite, A. and Nettles, S. (1996). Transactions for Java. In Aktinson, M. P. and Jordan, M. J., editors, *Proceedings of the First International Workshop on Persistence and Java. Sun Microsystems Laboratoris Technical Report 96-58*, pages 6–14.

[Herlihy and Wing, 1990] Herlihy, M. and Wing, J. (1990). Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492.

[Isard and Birell, 2007] Isard, M. and Birell, A. (2007). Automatic mutual exclusion. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*.

[Jagannathan et al., 2005] Jagannathan, S., Vitek, J., Welc, A., and Hosking, A. (2005). A transactional object calculus. *Science of Computer Programming*, 57(2):164–186.

[Lipton, 1975] Lipton, R. J. (1975). Reduction: A new method of proving properties of system processes. In *Second Annual Symposium on Principles of Programming Languages (POPL) (Palo Alto, CA)*, pages 78–86. ACM.

[Shavit and Toitu, 1995] Shavit, N. and Toitu, D. (1995). Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 204–213.

[Welc et al., 2005] Welc, A., Jagannathan, S., and Hosking, A. (2005). Safe futures in Java. In *Twentieth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '05*, pages 439 – 453. ACM. In *SIGPLAN Notices*.