Alternative Concurrency Models

CS 450 : Operating Systems
Michael Lee <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

"The free lunch is over. We have grown used to the idea that our programs will go faster when we buy a next-generation processor, but that time has passed. While that next-generation chip will have more CPUs, each individual CPU will be no faster than the previous year's model. If we want our programs to run faster, we must learn to write parallel programs."

- Simon Peyton Jones, *Beautiful Concurrency*

Position Sep 2012	Position Sep 2011	Delta in Position	Programming Language	Ratings Sep 2012	Delta Sep 2011	Status
1	2	↑	C	19.295%	+1.29%	A
2	1	↓	Java	16.267%	-2.49%	A
3	6	↑↑↑	Objective-C	9.770%	+3.61%	A
4	3	↓	C++	9.147%	+0.30%	A
5	4	↓	C#	6.596%	-0.22%	A
6	5	↓	PHP	5.614%	-0.98%	A
7	7	=	(Visual) Basic	5.528%	+1.11%	A
8	8	=	Python	3.861%	-0.14%	A
9	9	=	Perl	2.267%	-0.20%	A
10	11	↑	Ruby	1.724%	+0.29%	A
11	10	↓	JavaScript	1.328%	-0.14%	A
12	12	=	Delphi/Object Pascal	0.993%	-0.32%	A
13	14	↑	Lisp	0.969%	-0.07%	A
14	15	↑	Transact-SQL	0.875%	+0.02%	A
15	39	↑↑↑↑↑↑↑↑↑↑	Visual Basic .NET	0.840%	+0.53%	A
16	16	=	Pascal	0.830%	-0.02%	A
17	13	↓↓↓↓	Lua	0.723%	-0.43%	A-
18	18	=	Ada	0.700%	+0.02%	A--
19	17	↓↓	PL/SQL	0.604%	-0.12%	B
20	22	↑↑	MATLAB	0.563%	+0.02%	B

http://www.tiobe.com

TIOBE language popularity chart

most popular paradigms are

imperative and **object-oriented**

imperative: a program consists of a sequence of *statements* that read and alter process state

e.g.,
```
for (i=0; i<N; i++) {
        sum += arr[i];
    }
```

early on, *procedural* languages helped us modularize imperative programs by separating logic into different procedures

… not quite good enough.

Bad programmers can too easily write "spaghetti code" (e.g., with globs & gotos)

OOP: bundle data and methods that act on them into *objects*; goal is *encapsulation*

e.g.,
```
acc1 = BankAccount(balance=1000.0)
acc2 = BankAccount(balance=0.0)
acc2.deposit(500.0)
acc1.transfer_to(acc2, 250.0)
print(acc1.balance(), acc2.balance())
```

In most OO languages, objects are *mutable*; i.e., objects may consist of many pieces of *shareable*, *changeable state*

(aka "big mutable balls")

Most common concurrency model:

- explicitly created & managed *threads*
- *shared*, *freely mutable* state (memory)
- *lock*-based synchronization (e.g., semaphores, mutexes)

"Mutual-exclusion locks are one of the most widely used and fundamental abstractions for synchronization … Unfortunately, without specialist programming care, these benefits rarely hold for systems containing more than a handful of locks:

- *For correctness, programmers must ensure that threads hold the necessary locks to avoid conflicting operations being executed concurrently...*
- *For liveness, programmers must be careful to avoid introducing deadlock and, consequently, they may cause software to hold locks for longer than would otherwise be necessary ...*
- *For high performance, programmers must balance the granularity at which locking operates against the time that the application will spend acquiring and releasing locks."*

- Keir Fraser, *Concurrent Programming Without Locks*

i.e., implementing correct concurrent behavior via locks is difficult!

but correctness can be verified via testing, right?

"… one of the fundamental problems with testing … [is that] testing for one set of inputs *tells you* nothing at all *about the behaviour with a different* set of inputs*. In fact the problem caused by state is typically worse — particularly when testing large chunks of a system — simply because even though the number of possible* inputs *may be very large, the number of possible* states *the system can be in is often even larger."*

"One of the issues (that affects both testing and reasoning) is the exponential rate at which the number of possible states grows — for every single bit *of state that we add we* double *the total number of possible states."*

- Ben Moseley and Peter Marks, *Out of the Tar Pit*

"Concurrency also affects testing … Running a test in the presence of concurrency with a known initial state and set of inputs tells you nothing at all *about what will happen the next time you run that very same test with the very same inputs and the very same starting state. . . and things can't really get any worse than that."*

- Ben Moseley and Peter Marks, *Out of the Tar Pit*

Another issue: *composability*

I.e., after building and testing a software module, can we easily combine it with other (tested) modules to build a system?

"... consider a hash table with thread-safe insert and delete operations. Now suppose that we want to delete one item A *from table* t1*, and insert it into table* t2*; but the intermediate state (in which neither table contains the item) must not be visible to other threads. Unless the implementor of the hash table anticipates this need, there is simply no way to satisfy this requirement… In short, operations that are individually correct (insert, delete) cannot be* ***composed*** *into larger correct operations."*

- Tim Harris et al, *Composable Memory Transactions*

lack of composability is a big problem!

- code modules can not make use of each other without additional reasoning/testing

"Civilization advances by extending the number of important operations which we can perform without thinking."

- Alfred North Whitehouse

the root problem is *shared, freely mutable state*

requires the use of *synchronization*

leading to unnecessary, or *accidental*, complexity in the implementation

"Anyone who has ever telephoned a support desk for a software system and been told to "try it again", or "reload the document", or "restart the program", or "reboot your computer" or "re-install the program" or even "re-install the operating system and then the program" has direct experience of the problems that state causes for writing reliable, understandable software."

- Ben Moseley and Peter Marks, *Out of the Tar Pit*

"Complexity is the *root cause of the vast majority of problems with software today. Unreliability, late delivery, lack of security — often even poor performance in large-scale systems can all be seen as deriving ultimately from unmanageable complexity. The primary status of complexity as* the *major cause of these other problems comes simply from the fact that being able to* understand *a system is a prerequisite for avoiding all of them, and of course it is this which complexity destroys."*

- Ben Moseley and Peter Marks, *Out of the Tar Pit*

goal: avoid accidental complexity

— don't make concurrent programming harder than it needs to be!

Alternative concurrency models:

1. Actor model
2. Software Transactional Memory

1. Actor model

- no shared state, ever
- actors are *concurrent* & *independent*
- actors interact through *message passing*

e.g., Erlang

- created at Ericsson for telecom apps
- designed for concurrent, distributed, real-time systems
- "99.9999999 percent reliability (9 nines, or 31 ms. downtime a year!)"

- functional core
- messages are *asynchronous*
- creating actors (aka processes) is *cheap* (scales to millions of processes)
- essential architecture: client/server

```
% basic pattern matching; note vars in uppercase
factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).

% if expression: one guard must evaluate to true
max(A,B) -> if A < B -> B;
              true  -> A
            end.

% atoms (lowercase) and fixed-arity tuples
area({circle, R})       -> 3.1415 * R * R;
area({rectangle, L, W}) -> L * W;
area({square, L})       -> area({rectangle, L, L});
area(_)                 -> unknown.
```

```
> basics:factorial(10).
3628800
> basics:max(5, 10).
10
> basics:area({rectangle, 5, 10}).
50
> basics:area({triangle, 4, 5, 6}).
unknown
```

Creating processes:

```
Pid = spawn(Fun)
```

Sending messages (asynchronous):

```
Pid ! Message
```

Receiving messages (synchronous):

```
receive Pattern1 -> Exp1; ... end
```

Receiving with timeout:

```
receive ... after Millis -> Exp end
```

Server template:

```
loop() ->
    receive
        terminate -> done;
        Message   -> process(Message),
                     loop()
    end.
```

Server with "state":

```
loop(State) ->
    receive
        terminate -> done;
        Message   -> loop(process(Message, State))
    end.
```

```
loop() ->
    receive
        {From, Msg} ->
            io:format("Got ~s~n", [Msg]),
            From ! lists:reverse(Msg),
            loop();
        terminate   ->
            io:format("Stopping~n")
    end.

start() ->
    Pid = spawn(fun loop/0),
    Pid ! {self(), "hello!"},
    receive
        Msg -> io:format("Got ~s~n", [Msg])
    end,
    Pid ! terminate.
```

```
> basics.start().
Got hello!
Got !olleh
Stopping
```

```
consumer() ->
    receive
        % consumer blocks for message
        terminate -> done;
        Val -> io:format("C: got ~w~n", [Val]),
               consumer()
    end.

producer(Val, Consumer) ->
    if Val =:= ?MAX_VAL ->
            Consumer ! terminate;
       true ->
            % producer sends value to consumer
            Consumer ! Val,
            % then works on producing next value
            producer(Val + 1, Consumer)
    end.

start() ->
    C = spawn(fun consumer/0),
    % producer needs consumer pid & start value
    spawn(fun() -> producer(0, C) end).
```

- processes are automatically backed by "mailboxes" — by default unbounded
  - to simulate bounded buffer, must use messages to convey state & synch

```
producer(Val, Consumer, Ahead) ->
    if Val =:= ?MAX_VAL ->
            Consumer ! terminate;
       Ahead =:= ?MAX_AHEAD ->
            % throttle producer --- force to wait for ack
            receive
                ack -> producer(Val, Consumer, Ahead - 1)
            end;
       true ->
            Consumer ! {self(), Val}, % send my pid to consumer
            receive
                % try to get ack (immediate timeout)
                ack -> producer(Val + 1, Consumer, Ahead)
            after
                0 -> producer(Val + 1, Consumer, Ahead + 1)
            end
    end.

consumer() ->
    receive
        terminate -> done;
        {Producer, Val} -> io:format("C: got ~w~n", [Val]),
                           Producer ! ack,
                           consumer()
    end.
```

```
producer(Val, Consumer, Ahead) ->
    if Val =:= ?MAX_VAL ->
            Consumer ! terminate;
       Ahead =:= ?MAX_AHEAD ->
            % throttle producer --- force to wait for ack
            receive
                ack -> producer(Val, Consumer, Ahead - 1)
            end;
       true ->
            Consumer ! {self(), Val}, % send my pid to consumer
            receive
                % try to get ack (immediate timeout)
                ack -> producer(Val + 1, Consumer, Ahead)
            after
                0 -> producer(Val + 1, Consumer, Ahead + 1)
            end
    end.
```

subtle issue: once producer hits cap, will never drop below `?MAX_AHEAD-1`

```
producer(Val, Consumer, Ahead) ->
    if Val =:= ?MAX_VAL ->
            Consumer ! terminate;
       Ahead =:= ?MAX_AHEAD ->
            % throttle producer --- force to wait for ack
            receive
                ack -> producer(Val, Consumer, Ahead - 1)
            end;
       true ->
            receive
                % process all outstanding acks in mailbox
                ack -> producer(Val, Consumer, Ahead - 1)
            after
                0 -> Consumer ! {self(), Val},
                     producer(Val + 1, Consumer, Ahead + 1)
            end
    end.
```

takeaway: Erlang doesn't magically take care of synchronization issues!

General issues:

- Messages may be big — but no other way of sharing data!
- Process synchronization is still hard!
  - Typically use well known protocols

But we've eliminated shared state issues!

Huge boon to reasoning, composability, and robustness

- actors are independent — if one is unresponsive, can route around it

Also, makes deploying on distributed hardware transparent

Projects in Erlang:

- Facebook Chat
- RabbitMQ messaging framework
- lots of telephony and real-time (e.g., routing, VOIP) services

2. Software Transactional Memory (STM)

- supports shared memory
- but *all changes* are vetted by runtime

STM guarantees ACID properties:

- **A**tomicity
- **C**onsistency
- **I**solation

Atomicity:

- all requested changes take place (commit), or none at all (rollback)

Consistency:

- updates always leave data in a *valid state*
- i.e., allow validation hooks

Isolation:

- no transaction sees intermediate effects of other transactions

e.g., Clojure

- "invented" by Rich Hickey
- a (mostly functional) Lisp dialect
- primarily targets JVM

synchronization is *built into* the platform

based on a re-examination of *state vs. identity*

Tenet: most languages (notably, OOPLs) simplify but complicate *identity*

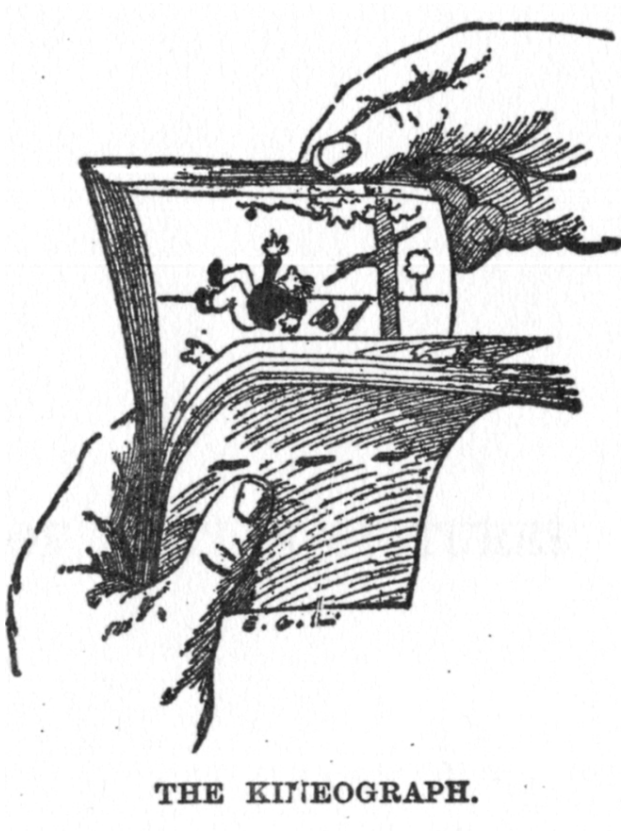
- *identity* is disconnected from *state*
- an object's state (attributes) can change, but it's still considered *the same object*
  - e.g., pointer based equality

Ramifications:

- threads can concurrently change the state of the same object
- objects that are entirely identical (state-wise) are considered different
  - requires comparators, .equals, etc.

Alternate view: objects perpetually advance through separate, *instantaneous states*

- we conveniently use names (i.e., *references*) to refer to the *most recent* state

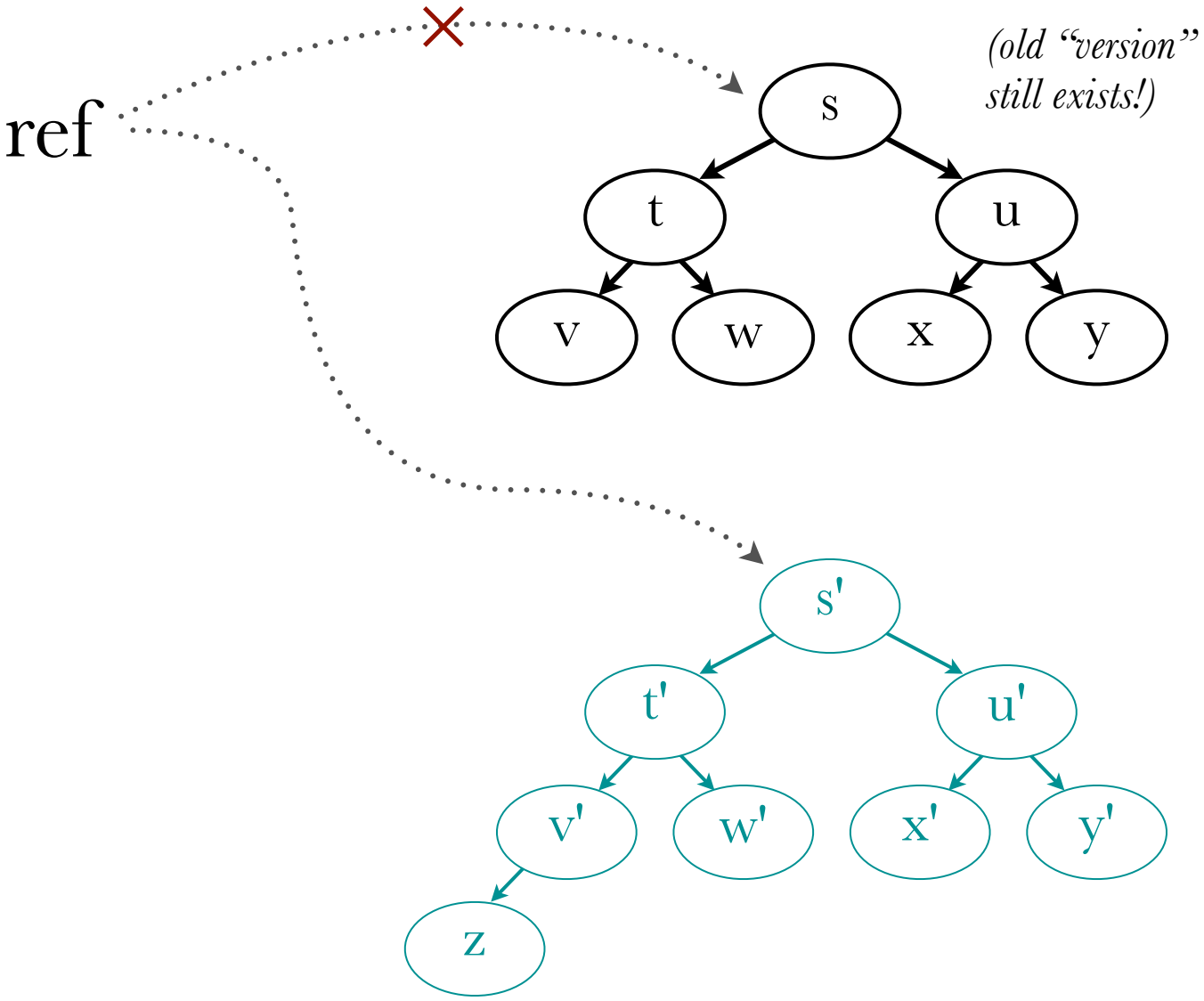


In Clojure, all values (state) are *immutable*

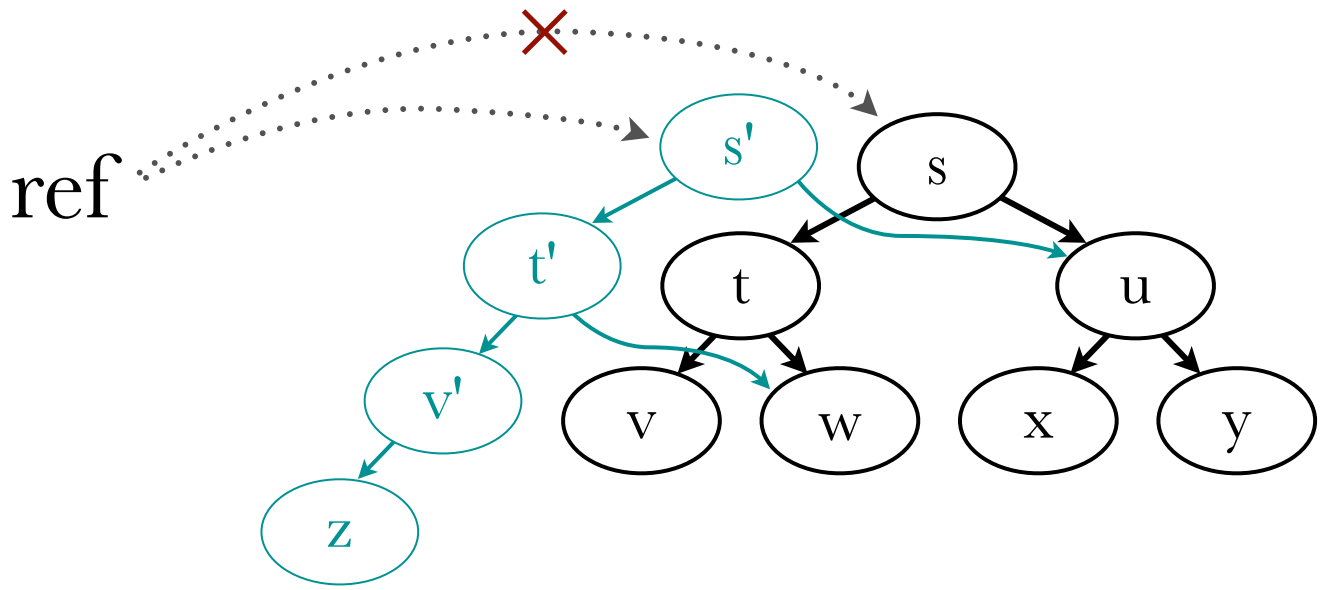
… but we can point a given *reference* at different states

to "update" a data structure:

1. access current value via reference
2. use it to create a new value
3. modify reference to refer to new value


ref
(old "version" still exists!)
s
t
u
v
w
x
y
s'
t'
u'
v'
w'
x'
y'
z

problem: very inefficient for large data structures



- in practice, share structure with old version

"persistent" data structures

- allow for structural sharing
  - ok because they are *immutable*
- allow *multiple versions* of a given data structure to be kept around

Multiversion Concurrency Control (**MVCC**)

- track versions of all data in history
- support "point-in-time" view of all data

Value vs. Reference dichotomy is crucial

- *immutable values* let us use data without concern that it will change under us
- *references* let us easily coordinate "changes"

important: how can we alter references?

- if arbitrarily, still have synch issue
- Clojure has multiple types of references, with different "change" semantics

Clojure reference types:

- vars
- atoms
- refs
- agents

vars are classic "variables"

- bound to *root values*, shared by all threads
- bad style to change at runtime
  - i.e., treat bound values as constants

```
;;; vars

(def x 10)
(inc x) ; => 11
x ; => 10 (unchanged)

(def acc {:name "checking" :balance 1000})

(defstruct account :name :balance)
(def acc2 (struct account "savings" 2000))

(= acc2 {:name "savings" :balance 2000}) ; => true

(def acc3 (assoc acc2 :name "business"))

acc3 ; => {:name "business" :balance 2000}
acc2 ; => {:name "savings" :balance 2000} (unchanged)
```

atoms support *isolated, atomic* updates

- provide with a function to compute new value from old value
  - atom is updated in mutex

```
;;; atoms

(def count (atom 0))

(deref count) ; => 0

@count ; => 0 (‘@’ is shortcut for deref)

(swap! count inc)

@count ; => 1

(reset! count 0)

@count ; => 0
```

`swap` runs function on atom's *current value*

- if another thread changes the atom before I write my update, retry!

refs support *coordinated* updates

- updates can only take place in *transactions*
  - within a transaction, we automatically get atomicity/isolation

```
;;; refs

(def a (ref 10))
(def b (ref 20))

(defn swap [ref1 ref2]
  (dosync ; start transaction
   (let [val1 @ref1
         val2 @ref2]
     (ref-set ref1 val2)
     (ref-set ref2 val1))))

(swap a b) ; @a = 20, @b = 10

(dosync (alter a inc)) ; @a = 21
```

Demo

important: transaction is run *optimistically*

- refs are all updated at single *commit-point*
- if another transaction changes *any of the refs* I'm altering before I commit, rerun!
  - alternative: `commute`

agents support *asynchronous* updates

- update functions run in separate thread
- at most one action being run at any time
  - i.e., updates (aka *actions*) are queued

```
;;; agents

(def bond (agent 007))

@bond ; => 7

(send bond inc)

;; a short while later...
@bond ; => 8
```

Demo

Benefits of STM:

- automatic support for mutex/isolation
- optimistic transactions maximize concurrency
- framework helps guarantee freedom from race conditions!

Clojure-specific benefits:

- modifications to refs *must happen* within transactions (i.e., not advisory)
- persistent data structures allow for "snapshot" MVCC (vs. logging in other implementations)

Drawbacks:

- transaction restarts = overhead
  - performance is not transparent
    - compared to locking?
- MVCC = overhead (need a lot of GC)
- snapshot isolation → *write skew*

Write skew scenario:

- Accounts X, Y with total min balance M
  - Thread A debits X, checks $X + Y \leq M$
  - Thread B debits Y, checks $X + Y \leq M$
- Only conflicting *writes* require rollback!
  - A may read old version of Y (and B of X)

Clojure "fix" for write-skew:

- can pretend to update reference: `(ref-set ref @ref)`
- or use `(ensure ref)` — requires rollback if ref has been changed at commit point

§Summary

Chips aren't getting any faster, but we are getting more processing cores

— we need a scalable way of writing concurrent programs

Mutable, shared memory and locks are hard to reason about and add *unnecessary complexity* to programs (especially in concurrent settings)

Two alternative models:

- Actor model: no shared state, ever; communicate via messages
- STM: transactional support for state transactions

Implementations:

- Actor model: Erlang, Scala, node.js
- STM: Clojure, Haskell, Python, etc.

No silver bullet!

- Actor model: still need to worry about synchronization, deadlock still possible
- STM: performance under scrutiny

What would *you* use?

Bibliography:

- Harris, T., Marlow, S., & Peyton-Jones, S. *Composable memory transactions*. In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '05).

- Peyton-Jones, S. *Beautiful concurrency*. Beautiful Code. 2007.

- Moseley, B & Marks, P. *Out of the tar pit*. 2006.

- Fraser, K., & Harris, T. *Concurrent programming without locks*. ACM Transactions on Computer Systems, 25(2), 5. 2007.

- Hansen, P. *Java's insecure parallelism*. SIGPLAN notices. 1999.

- Hickey, R. *Are we there yet?* JVM Languages Summit presentation. 2009.