Software Transactions:
A Programming-Languages Perspective

Dan Grossman
University of Washington

29 March 2007

Atomic

An *easier-to-use* and *harder-to-implement* primitive

```
void deposit(int x){
synchronized(this){
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

```
void deposit(int x){
atomic {
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

lock acquire/release

(behave as if)
no interleaved computation;
no unfair starvation

Viewpoints

Software transactions good for:

- Software engineering (avoid races & deadlocks)
- Performance (optimistic "no conflict" without locks)

Research should be guiding:

- New hardware with transactional support
- Inevitable software support
  - Legacy/transition
  - Semantic mismatch between a PL and an ISA
  - May be fast enough
- Prediction: hardware for the common/simple case

PL Perspective

Key complement to the focus on “transaction engines” and low-level optimizations

Language design:

interaction with rest of the language

– Not just I/O and exceptions (not this talk)

Language implementation:

interaction with the compiler and today’s hardware

– Plus new needs for high-level optimizations

Not today

"Across the lake" my students are busy with a variety of ongoing projects related to PL/TM

- Formal semantics
- Parallelism within transactions
- Interaction with first-class continuations
- "Transactional events" in the presence of mutation
- …

Happy to return in a year and tell you more; today focus on more mature results/questions

Today

Issues in language design and semantics

1. Transactions for software evolution
2. Transactions for strong isolation [PLDI07]*
3. The need for a memory model [MSPC06a]**

Software-implementation techniques

1. On one core [ICFP05]
2. Without changing the virtual machine [MSPC06b]
3. Static optimizations for strong isolation [PLDI07]*

* Joint work with Intel PSL

** Joint work with Manson and Pugh

Code evolution

Having chosen "self-locking" today, hard to add a correct transfer method tomorrow

```
void deposit(…)  { synchronized(this) { … }}
void withdraw(…) { synchronized(this) { … }}
int  balance(…)  { synchronized(this) { … }}
void transfer(Acct from, int amt) {
  synchronized(this) {
   //race
   if(from.balance()>=amt && amt < maxXfer) {
     from.withdraw(amt);
     this.deposit(amt);
   }
  }
}
```

Code evolution

Having chosen “self-locking” today, hard to add a correct transfer method tomorrow

```
void deposit(…)  { synchronized(this) { … }}
void withdraw(…) { synchronized(this) { … }}
int  balance(…)  { synchronized(this) { … }}
void transfer(Acct from, int amt) {
  synchronized(this) {
  synchronized(from) { //deadlock (still)
   if(from.balance()>=amt && amt < maxXfer) {
     from.withdraw(amt);
     this.deposit(amt);
   }
  }}
}
```

Code evolution

Having chosen "self-locking" today, hard to add a correct transfer method tomorrow

```
void deposit(…)  { atomic { … }}
void withdraw(…) { atomic { … }}
int  balance(…)  { atomic { … }}

void transfer(Acct from, int amt) {

   //race
   if(from.balance()>=amt && amt < maxXfer) {
      from.withdraw(amt);
      this.deposit(amt);
   }

}
```

Code evolution

Having chosen “self-locking” today, hard to add a correct transfer method tomorrow

```
void deposit(…)  { atomic { … }}
void withdraw(…) { atomic { … }}
int  balance(…)  { atomic { … }}
void transfer(Acct from, int amt) {
  atomic {
    //correct
    if(from.balance()>=amt && amt < maxXfer){
      from.withdraw(amt);
      this.deposit(amt);
    }
  }
}
```

Lesson

Locks do not compose; transactions do

Today

Issues in language design and semantics

1. Transactions for software evolution
2. Transactions for strong isolation [PLDI07]*
3. The need for a memory model [MSPC06a]**

Software-implementation techniques

1. On one core [ICFP05]
2. Without changing the virtual machine [MSPC06b]
3. Static optimizations for strong isolation [PLDI07]*

* Joint work with Intel PSL

** Joint work with Manson and Pugh

"Weak" atomicity

Widespread misconception:

"Weak" atomicity violates the "all-at-once" property of transactions only when the corresponding lock code has a data race

(May still be a bad thing, but smart people disagree.)

```
initially y==0
```

```
atomic {
  y = 1;
  x = 3;
  y = x;
}
```

```
x = 2;
print(y); //1? 2? 85?
```

Segregation

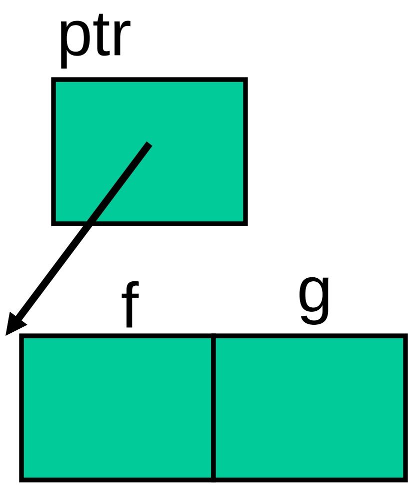
Segregation is not necessary in lock-based code

– Even under relaxed memory models

```
initially ptr.f == ptr.g
```

```
sync(lk) {
  r = ptr;
  ptr = new C();
}
assert(r.f==r.g);
```

```
sync(lk) {
  ++ptr.f;
  ++ptr.g;
}
```



(Example adapted from [Rajwar/Larus] and [Hudson et al])

It’s worse

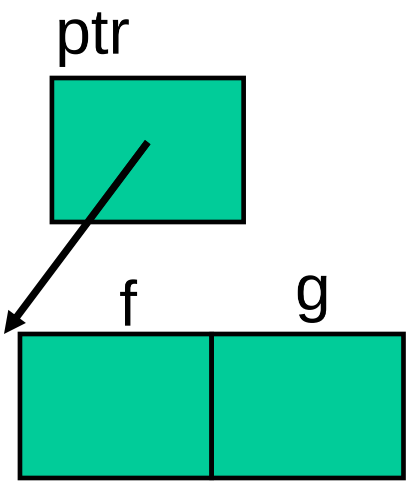
But every published weak-atomicity system allows the assertion to fail!

- Eager- or lazy-update

```
initially ptr.f == ptr.g
```

```
atomic {
  r = ptr;
  ptr = new C();
}
assert(r.f==r.g);
```

```
atomic {
  ++ptr.f;
  ++ptr.g;
}
```



(Example adapted from [Rajwar/Larus] and [Hudson et al])

“Weak” atomicity redux

“Weak” really means nontransactional code bypasses the transaction mechanism…

Weak STMs violate isolation on example:

- Eager-updates (one update visible before abort)
- Lazy-updates (one update visible after commit)

Imposes correctness burdens on programmers that locks do not

More examples (see paper for more)

With eager-update, speculative dirty read:

```
initially x==0 and y==0
```

```
atomic {
  if(y==0)
    x=1;
  /* abort */
}
assert(x==1);
```

```
if(x==1)
  y=1;
```

More examples (see paper for more)

With weak-update, can miss an initialization
(e.g., a **`readonly`** field)

```
initially x==null
```

```
atomic {
  t = new C();
  t.f = 42;
  x=t;
}
```

```
if(x!=null)
  assert(x.f==42);
```

Lesson

"Weak" is worse than most think; it can require segregation where locks do not

Corollary: "Strong" has easier semantics

- especially for a safe language

Today

Issues in language design and semantics

1. Transactions for software evolution
2. Transactions for strong isolation [PLDI07]*
3. The need for a memory model [MSPC06a]**

Software-implementation techniques

1. On one core [ICFP05]
2. Without changing the virtual machine [MSPC06b]
3. Static optimizations for strong isolation [PLDI07]*

* Joint work with Intel PSL

** Joint work with Manson and Pugh

Relaxed memory models

Modern languages don't provide sequential consistency

1. Lack of hardware support
2. Prevents otherwise sensible & ubiquitous compiler transformations (e.g., copy propagation)

So safe languages need two complicated definitions

1. What is "properly synchronized"?
2. What can compiler and hardware do with "bad code"?

(Unsafe languages need (1))

A flavor of simplistic ideas and the consequences…

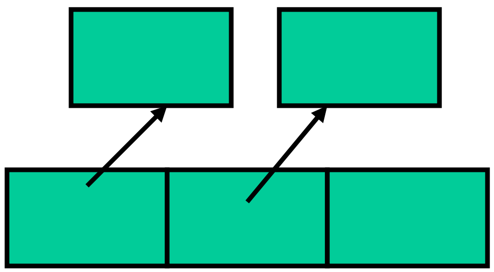
Simplistic ideas

"Properly synchronized" ➔ All thread-shared mutable memory accessed in transactions

Consequence: *Data-handoff* code deemed "bad"

```
//Producer
tmp1=new C();
tmp1.x=42;
atomic {
 q.put(tmp1);
}
```

```
//Consumer
atomic {
 tmp2=q.get();
}
tmp2.x++;
```



Ordering

Can get “strange results” for bad code

– Need rules for what is “good code”

```
initially x==0 and y==0
```

```
x = 1;

y = 1;
```

```
r = y;

s = x;
assert(s>=r);//invalid
```

Ordering

Can get "strange results" for bad code

– Need rules for what is "good code"

```
initially x==0 and y==0
```

```
x = 1;
sync(lk){}
y = 1;
```

```
r = y;
sync(lk){} //same lock
s = x;
assert(s>=r);//valid
```

Ordering

Can get "strange results" for bad code

– Need rules for what is "good code"

```
initially x==0 and y==0
```

```
x = 1;
atomic{}
y = 1;
```

```
r = y;
atomic{}
s = x;
assert(s>=r);//???
```

If this is good code, existing STMs are wrong

Ordering

Can get “strange results” for bad code

– Need rules for what is “good code”

```
initially x==0 and y==0
```

```
x = 1;
atomic{z=1;}
y = 1;
```

```
r = y;
atomic{tmp=0*z;}
s = x;
assert(s>=r);//???
```

“Conflicting memory” a slippery ill-defined slope

Lesson

It is not clear when transactions are ordered, but languages need memory models

Corollary: This could/should delay adoption of transactions in well-specified languages

Shameless provocation:

What is the C# memory model?

Today

Issues in language design and semantics

1. Transactions for software evolution
2. Transactions for strong isolation [PLDI07]*
3. The need for a memory model [MSPC06a]**

Software-implementation techniques

1. On one core [ICFP05]
2. Without changing the virtual machine [MSPC06b]
3. Static optimizations for strong isolation [PLDI07]*

* Joint work with Intel PSL

** Joint work with Manson and Pugh

Interleaved execution

The “uniprocessor (and then some)” assumption:

Threads communicating via shared memory don't execute in “true parallel”

Important special case:

- Uniprocessors still exist
- Many language implementations assume it (e.g., OCaml, DrScheme)
- Multicore may assign one core to an application

Implementing atomic

Key pieces:

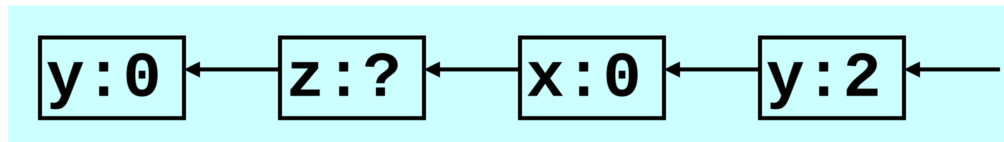
- Execution of an atomic block logs writes
- If scheduler pre-empts a thread in atomic, rollback the thread
- Duplicate code so non-atomic code is not slowed by logging
- Smooth interaction with GC

Logging example

```
int x=0, y=0;
void f() {
  int z = y+1;
  x = z;
}
void g() {
  y = x+1;
}
void h() {
  atomic {
    y = 2;
    f();
    g();
  }
}
```

Executing atomic block:

- build LIFO log of old values:



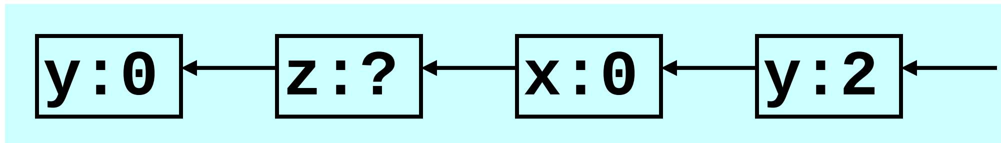
Rollback on pre-emption:

- Pop log, doing assignments
- Set program counter and stack to beginning of atomic

On exit from atomic:

- drop log

Logging efficiency



Keep the log small:

- Don't log reads (key uniprocessor advantage)
- Need not log memory allocated after atomic entered
  - Particularly *initialization writes*
- Need not log an address more than once
  - To keep logging fast, switch from array to hashtable after "many" (50) log entries

Duplicating code

```
int x=0, y=0;
void f() {
  int z = y+1;
  x = z;
}
void g() {
  y = x+1;
}
void h() {
  atomic {
    y = 2;
    f();
    g();
  }
}
```

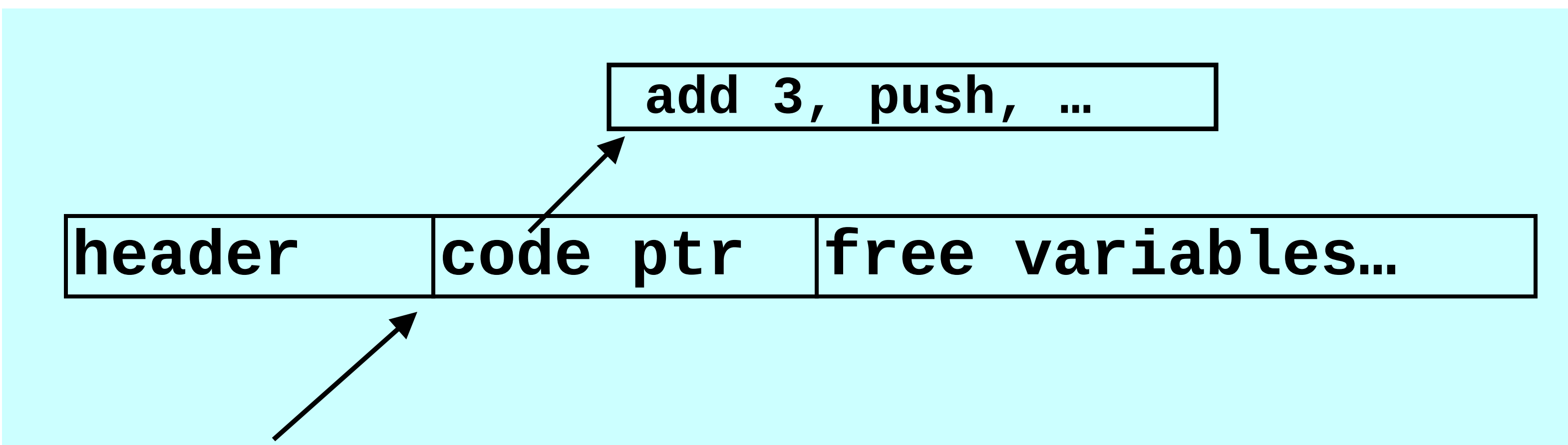
Duplicate code so callees know to log or not:

- For each function **`f`**, compile **`f_atomic`** and **`f_normal`**
- Atomic blocks and atomic functions call atomic functions
- Function pointers compile to pair of code pointers

Representing closures/objects

Representation of function-pointers/closures/objects an interesting (and pervasive) design decision

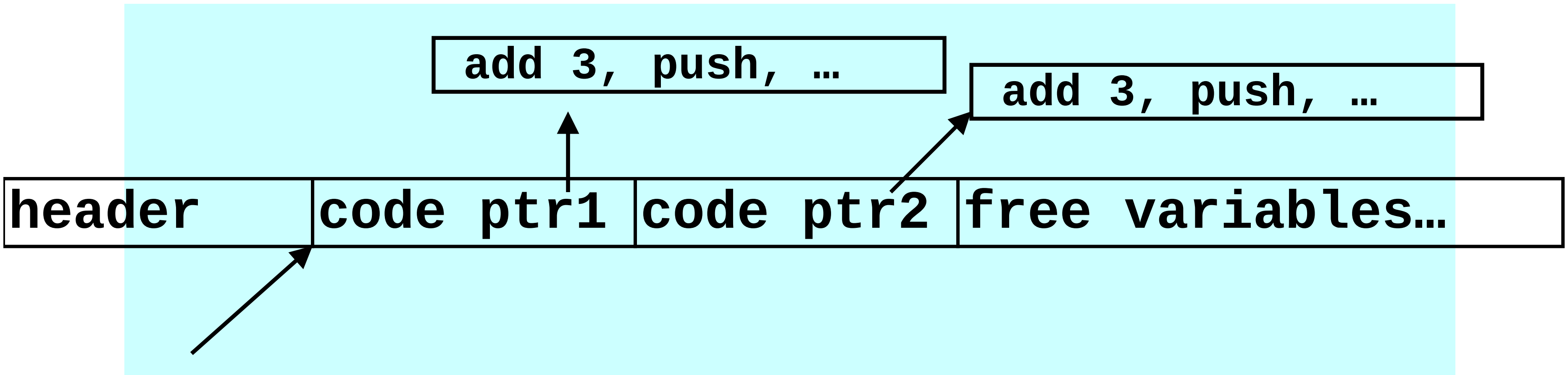
OCaml:



Representing closures/objects

Representation of function-pointers/closures/objects an interesting (and pervasive) design decision

One approach: bigger closures

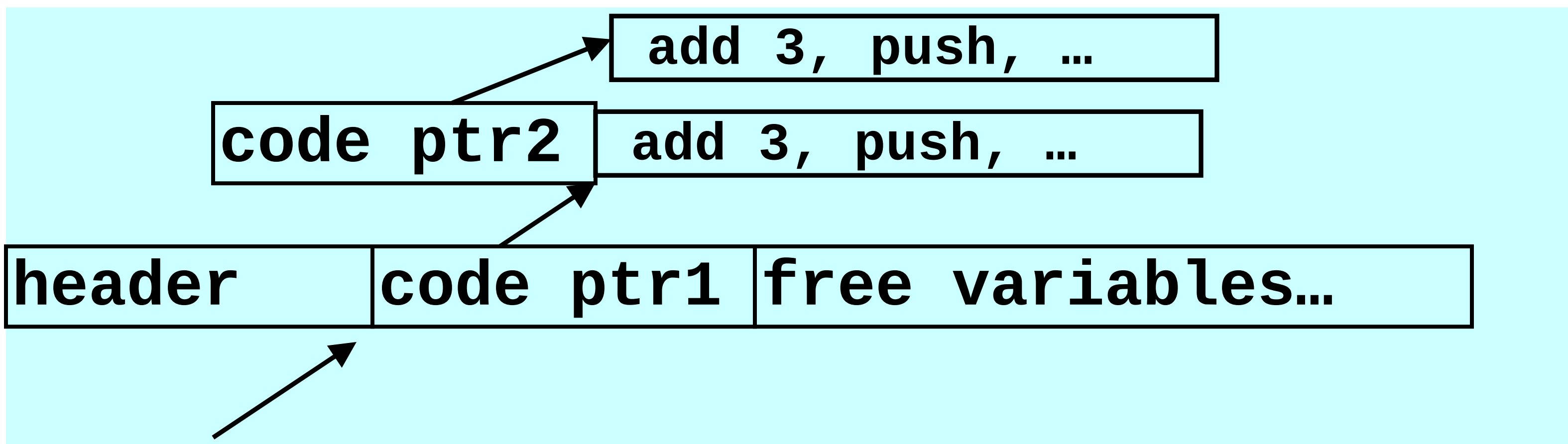


Note: atomic is first-class, so it is just one of these too!

Representing closures/objects

Representation of function-pointers/closures/objects an interesting (and pervasive) design decision

Alternate approach: slower calls in **`atomic`**



Note: Same overhead as OO dynamic dispatch

Interaction with GC

What if GC occurs mid-transaction?

- The log is a root (in case of rollback)
- Moving objects is fine
  - Rollback produces *equivalent* state
  - Naïve hardware solutions may log/rollback GC!

What about rolling back the allocator?

- Don’t bother: after rollback, objects allocated in transaction are unreachable
  - Naïve hardware solutions may log/rollback initialization writes!

Evaluation

Strong atomicity for Caml at little cost

- Already assumes a uniprocessor
- See the paper for “in the noise” performance

- Mutable data overhead

	not in atomic	in atomic
read	none	none
write	none	log (2 more writes)

- Rare rollback

Lesson

Implementing (strong) atomicity in software for a uniprocessor is so efficient it deserves special-casing

Note: Don’t run other multicore services on a uni either

Today

Issues in language design and semantics

1. Transactions for software evolution
2. Transactions for strong isolation [PLDI07]*
3. The need for a memory model [MSPC06a]**

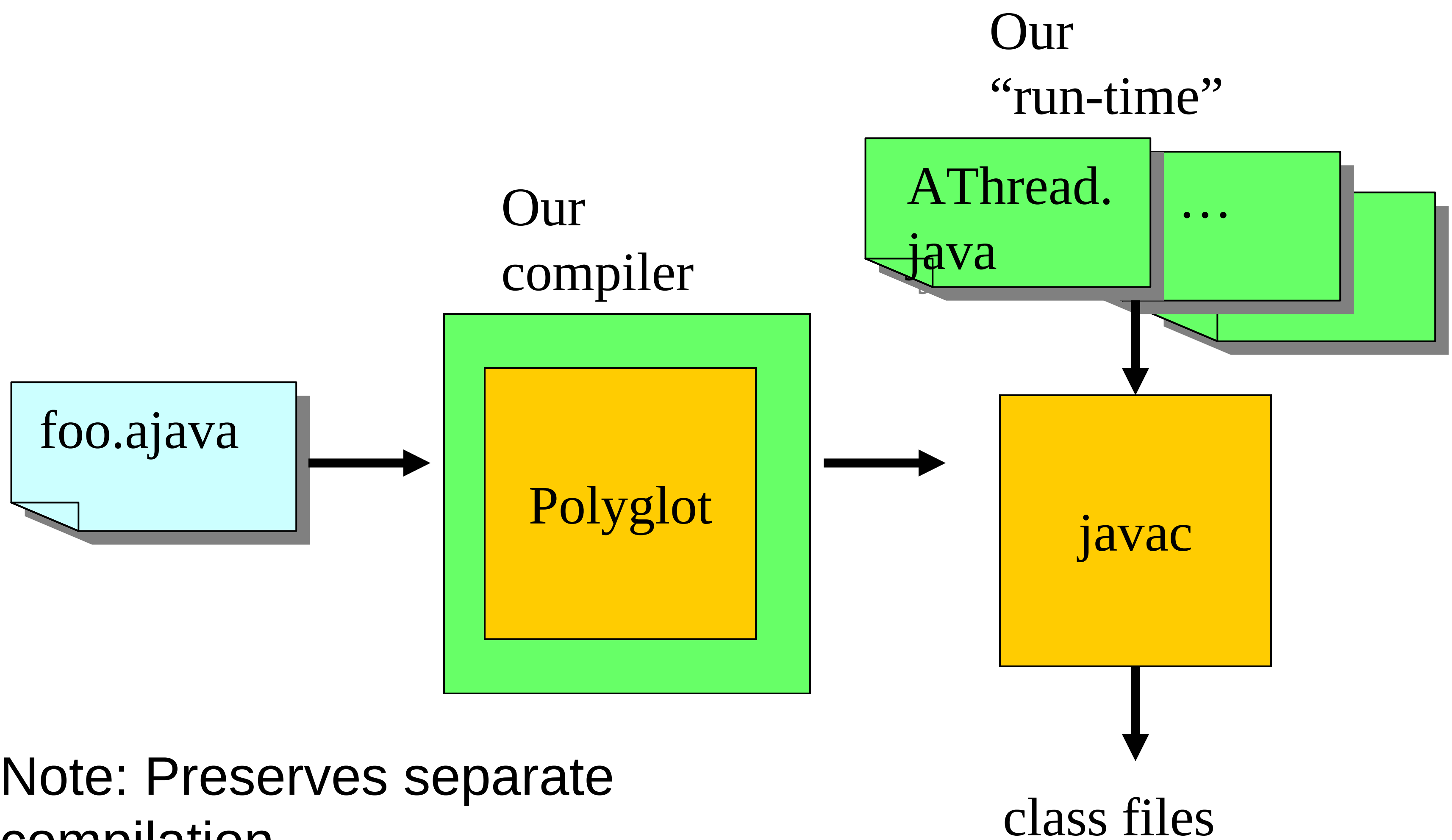
Software-implementation techniques

1. On one core [ICFP05]
2. Without changing the virtual machine [MSPC06b]
3. Static optimizations for strong isolation [PLDI07]*

* Joint work with Intel PSL

** Joint work with Manson and Pugh

System Architecture



Note: Preserves separate compilation

Key pieces

- A field read/write first *acquires ownership* of object
  - In transaction, a write also *logs the old value*
  - No synchronization if already own object
- *Polling* for releasing ownership
  - Transactions rollback before releasing
- Read/write barriers via method calls

  (JIT can inline them later)
- Some Java cleverness for efficient logging
- Lots of details for other Java features

Acquiring ownership

All objects have an **owner** field

```
class AObject extends Object {
  Thread owner; //who owns the object
  void acq(){   //owner=caller (blocking)
   if(owner==currentThread())
      return;
   … // complicated slow-path
  }
}
```

- Synchronization only when contention
- With "owner=currentThread()" in constructor, thread-local objects *never* incur synchronization

Releasing ownership

- Must “periodically” check “to release” set
  - If in transaction, first rollback
    - Retry later (backoff to avoid livelock)
  - Set owners to **`null`**
- Source-level “periodically”
  - Insert call to **`check()`** on loops and non-leaf calls
  - Trade-off synchronization and responsiveness:

```
int count = 1000; //thread-local
void check(){
 if(--count >= 0) return;
 count=1000; really_check();
}
```

But what about…?

Modern, safe languages are big…

See paper & tech. report for:

constructors, primitive types, static fields,

class initializers, arrays, native calls,

exceptions, condition variables, library classes,

…

Lesson

Transactions for high-level programming languages do not need low-level implementations

But good performance does tend to need parallel readers, which is future work for this system. ☹

Today

Issues in language design and semantics

1. Transactions for software evolution
2. Transactions for strong isolation [PLDI07]*
3. The need for a memory model [MSPC06a]**

Software-implementation techniques

1. On one core [ICFP05]
2. Without changing the virtual machine [MSPC06b]
3. Static optimizations for strong isolation [PLDI07]*

* Joint work with Intel PSL

** Joint work with Manson and Pugh

Strong performance problem

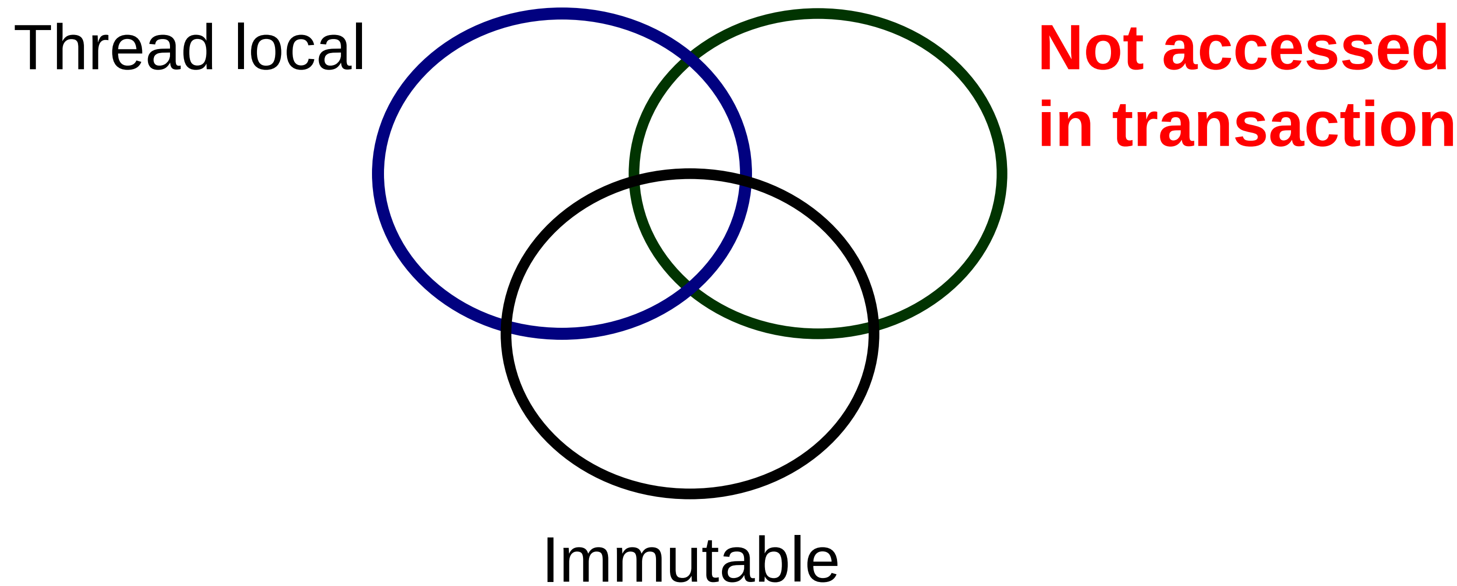
Recall uniprocessor overhead:

	not in atomic	in atomic
read	none	none
write	none	some

With parallelism:

	not in atomic	in atomic
read	none iff weak	some
write	none iff weak	some

Optimizing away barriers



New: static analysis for not-accessed-in-transaction …

Not-accessed-in-transaction

Revisit overhead of not-in-atomic for strong atomicity, given information about how data is used in atomic

	not in atomic			in atomic
	no atomic access	no atomic write	atomic write	
read	none	none	some	some
write	none	some	some	some

Yet another client of pointer-analysis

Analysis details

- Whole-program, context-insensitive, flow-insensitive
  - Scalable, but needs whole program
- Can be done before method duplication
  - Keep lazy code generation without losing precision
- Given pointer information, just two more passes
  1. How is an “abstract object” accessed transactionally?
  2. What “abstract objects” might a non-transactional access use?

Static counts

Not the point, but good evidence

- Usually better than thread-local analysis

App	Access	Total	Barrier removed by NAIT or TL	NAIT only	TL only
SpecJVM98	Read	12671	12671	8796	0
	Write	9885	9885	7961	0
Tsp	Read	106	93	89	0
	Write	36	17	16	0
JBB	Read	804	798	364	24
	Write	621	575	131	344

Experimental Setup

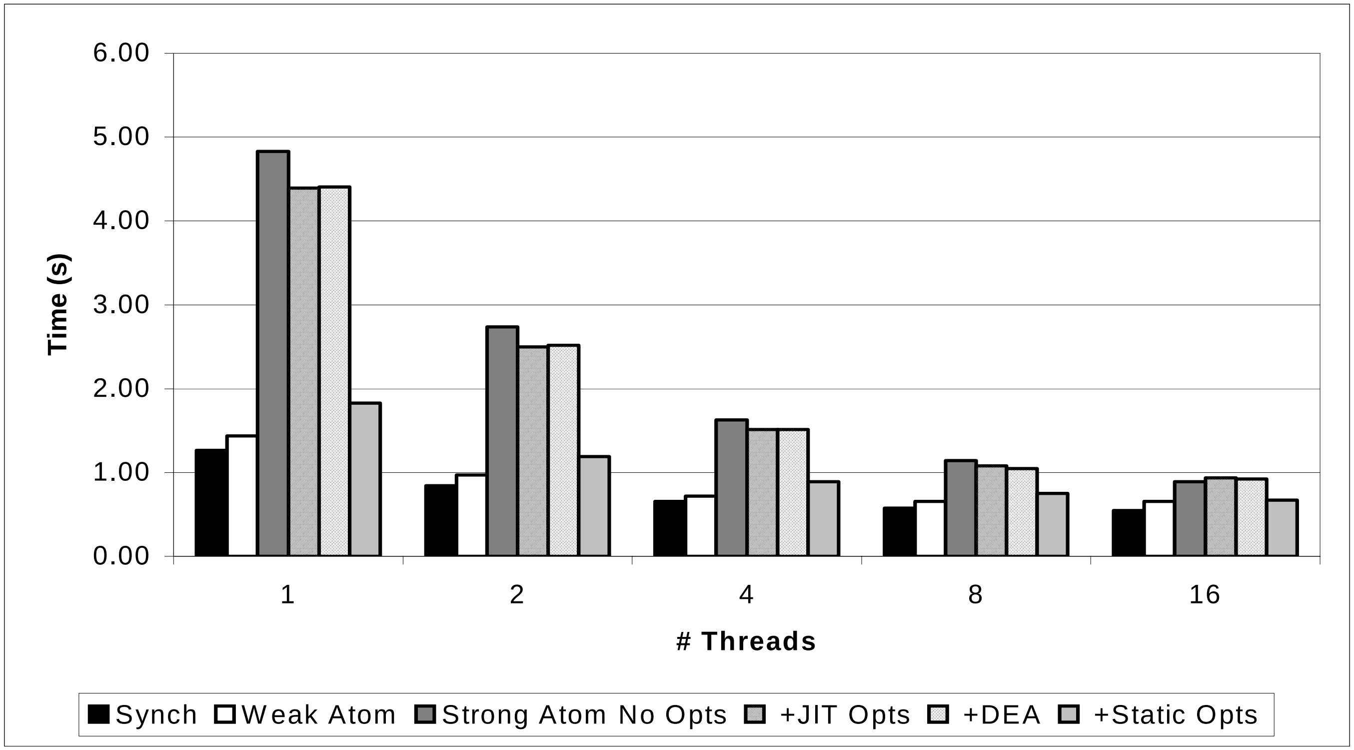
UW: static analysis using whole-program pointer analysis

- Scalable (context- and flow-insensitive) using Paddle/Soot

Intel PSL: high-performance strong STM via compiler and run-time

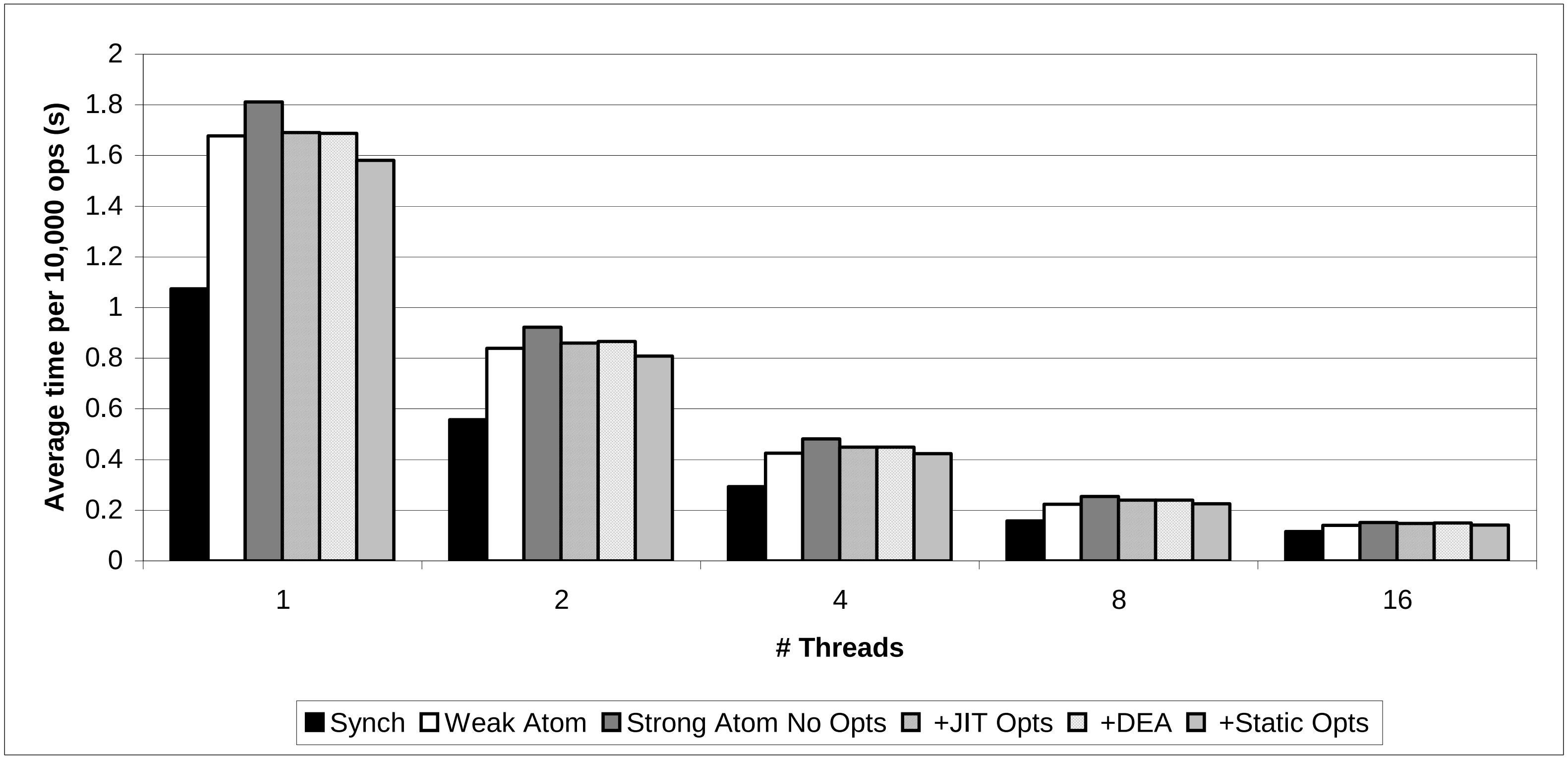
- StarJIT
  - IR and optimizations for transactions and isolation barriers
  - Inlined isolation barriers
- ORP
  - Transactional method cloning
  - Run-time optimizations for strong isolation
- McRT
  - Run-time for weak and strong STM

Benchmarks



Tsp

Benchmarks



JBB

Lesson

The cost of strong isolation is in nontransactional barriers and compiler optimizations help a lot

Credit

Uniprocessor: Michael Ringenburg

Source-to-source: Benjamin Hindman

Barrier-removal: Steven Balensiefer, Kate Moore

Memory-model issues: Jeremy Manson, Bill Pugh

High-performance strong STM: Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha

wasp.cs.washington.edu

Lessons

1. Locks do not compose; transactions do
2. "Weak" is worse than most think and sometimes worse than locks
3. It is unclear when transactions should be ordered, but languages need memory models

---

4. Implementing atomicity in software for a uniprocessor is so efficient it deserves special-casing
5. Transactions for high-level programming languages do not need low-level implementations
6. The cost of strong isolation is in nontransactional barriers and compiler optimizations help a lot