Open Nesting in Software Transactional Memories

DaCapo

Open Nesting in Software Transactional Memories

Tony Hosking

PURDUE UNIVERSITY

DaCapo

Open Nesting in Software Transactional Memories

Tony Hosking

Eliot Moss

Open Nesting in Software Transactional Memories

Yang Ni   Vijay Menon   Ali-Reza Adl-Tabatabai
Tony Hosking   Rick Hudson   Eliot Moss
Bratin Saha   Tatiana Shpeisman

PURDUE
UNIVERSITY

DaCapo

Abstract v. physical serializability

```
Set s;
```

Txn T_1:

```
atomic {
  s.add(x);
  s.add(y);
}
```

Txn T_2:

```
atomic {
  s.add(z);
}
```

- Linked list set implementation adds element to end of list
- Physical schedule results in (x,y,z) or (z,x,y)
- (x,z,y) violates atomicity of T_1
- However, (x,z,y) is consistent with "abstract" serializabilty: both transactions insert their elements; the result, in terms of s.contains, is the same
- Permitting the "abstractly" serializable result can increase concurrency

PURDUE UNIVERSITY

DaCapo

Abstract serializability

- Still need:
  - conflict detection
  - conflict resolution
- Set insertions must themselves happen atomically: (x,z) is not serializable either way
- Abstract conflicts must still be prevented

PURDUE UNIVERSITY

DaCapo

Abstract v. physical serializability

```
Set s;
```

Txn T_1:

```
atomic {
  s.add(x);
  if (!s.contains(z))
    s.add(y);
}
```

Txn T_2:

```
atomic {
  s.add(w);
  if (!s.contains(y))
    s.add(z);
}
```

- Abstract serializability allows (unordered) {w,x,y} or {w,x,z}
- If T_1 sees z not in s, t2 must not insert z without serializing with T_1
- T_2 must either wait for T_1 to complete or one of them must abort

PURDUE UNIVERSITY

DaCapo

Open nesting

- Supports abstract serializability
- Set operations implemented as openatomic actions that execute at a lower level of abstraction
- Individual set operations physically serializable, but once committed release memory resources to enhance concurrency
- Conflict detection via *abstract* locks:
  - T_1 must use abstract lock to assert absence of z
  - T_2 will try to obtain lock to insert z
- If both get past "if" at least one must abort

PURDUE UNIVERSITY

DaCapo

Abort

- To abort an open nested action programmers provide a *compensation* action
- In the example, T_1 must run an action that removes `x`

Atomic actions: syntax

(cf. Harris & Fraser)

`atomic` in place of `synchronized`

- succeeds: effects committed
- aborts: effects undone, action retried
- fails: effects undone, action not retried (for exceptions)
  - (others vary in semantics of exceptions)

PURDUE UNIVERSITY

DaCapo

Open atomic

Open atomicity as a property of *classes*:

- Can have `openatomic` methods; all public methods are implicitly so
- `openatomic` is inherited
- No public fields (else no access control)
- All fields implicitly `openatomic`, accessible only in `openatomic` methods

PURDUE UNIVERSITY

DaCapo

Open atomic methods

Signified by one of following handler blocks:

- `onabort`: undo global effects of committed child when parent fails
- `oncommit`: cleanup action for committed child when parent succeeds
- `onvalidation`: high-level checking for committed child when parent commits
- `ontopcommit`: finalisation action for all committed descendents of a committing top-level action

PURDUE UNIVERSITY

DaCapo

Handler syntax

MethodBody:

Block OpenAtomicHandlers$_{opt}$

OpenAtomicHandler:

onabort *Block*

oncommit *Block*

onvalidation *Block*

ontopcommit *Block*

PURDUE UNIVERSITY

DaCapo

Open atomic methods

May also have a `locks` clause:

- abstract locks that open atomic method must acquire before it can return

MethodDeclaration:

MethodHeader LocksClause$_{opt}$ *MethodBody*

PURDUE UNIVERSITY

DaCapo

Example

```
import java.util.*;
import java.openatomic.SXModes.*;
public openatomic class OpenMap implements Map {
  private final Map map;
  private final Object lock = new Object();
  public OpenMap(Map map) { this.map = map; }
  public Object get(Object key)
    locks(key:S) { return map.get(key); }
  public Object put(Object key, Object value)
    locks(key:X, lock:IX) { return map.put(key, value); }
    onabort {
      if (@result == null) map.remove(key);
      else map.put(key, @result);
    }
  public Object remove (Object key)
    locks(key:X, lock:IX) { return map.remove(key); }
    onabort { if (@result != null) map.put(key, @result); }
  public int size() locks(lock:S) { return map.size(); }
}
```

PURDUE UNIVERSITY

DaCapo

Log semantics: closed nesting

- New transaction starts with a clean log
- Log reads/writes + old values for writes
- Closed nested commit appends log to parent
- Top-level commit discards log
- Abort processes log in reverse, restores writes

PURDUE UNIVERSITY

DaCapo

Log semantics: conflicts

- Neither action an ancestor of the other and both access a location in conflicting modes (i.e., at least one writes)
- Conflicting actions must not both commit and must never write same locations

PURDUE UNIVERSITY

DaCapo

Log semantics: open nesting

- Add handlers and abstract locks to logs
- Nested commit:
  - invokes handlers (from children) in its log, in order, as open nested actions: `onvalidation` then `oncommit`
  - `ontopcommit` handlers appended to parent log (invoked on top-level commit)
  - appends own handlers/locks to parent log

PURDUE UNIVERSITY

DaCapo

Log semantics: nested abort

- Process log entries in reverse order:
  - invoke `onabort` handlers (from children)
  - undo writes
    - writes to same location may be undone more than once to preserve view for interleaved handlers

PURDUE UNIVERSITY

DaCapo

Log semantics: abstract locks

Lock consists of:

- *context* object (or class) whose method was run as openatomic
- *Locked* object mentioned in lock expression
- Lock *mode*

PURDUE UNIVERSITY

DaCapo

Log semantics: lock conflict

- Same context (==)
- Same locked object (`.equals`)
- Conflicting modes, as defined by lock class

Action T_a acquiring lock conflicts with T_h if T_h is not an ancestor of T_a and lock by T_a conflicts with some lock in T_h's log

PURDUE UNIVERSITY

DaCapo

Caveat Emptor

- Programmer errors bite!
  - programmer must specify conflicts and compensations
  - open nested aborts can deadlock
- Most problems arise because of mistakes in abstraction layering: accessing objects at different abstraction levels
- Avoiding mistakes can be achieved by observing guidelines that we hope to formalize (see paper)

PURDUE UNIVERSITY

DaCapo

Experiments

- Wrap `HashMap` and `TreeMap` using `synchronized`, `atomic`, `openatomic`
- 16 "transactions", each performing 4K puts:
  - partitioned across available threads
- All keys are different (so no logical conflicts)
- 16-way (4x4) Xeon
- "Transaction":

```
atomic {
  for (int i = 0; i < 4096; i++)
    map.put(key[i], value[i]);
}
```

PURDUE UNIVERSITY

DaCapo

Closed nested

- map is an `AtomicMap`:

```
import java.util.*;
public class AtomicMap implements Map {
    private final Map map;
    public AtomicMap(Map map) { this.map = map; }
    public Object put(Object key, Object value) {
        atomic { return map.put(key,value); }
    }
    ...
}
```

PURDUE UNIVERSITY

DaCapo

Open nested

- `map` is an `OpenMap`, defined earlier

Synchronized

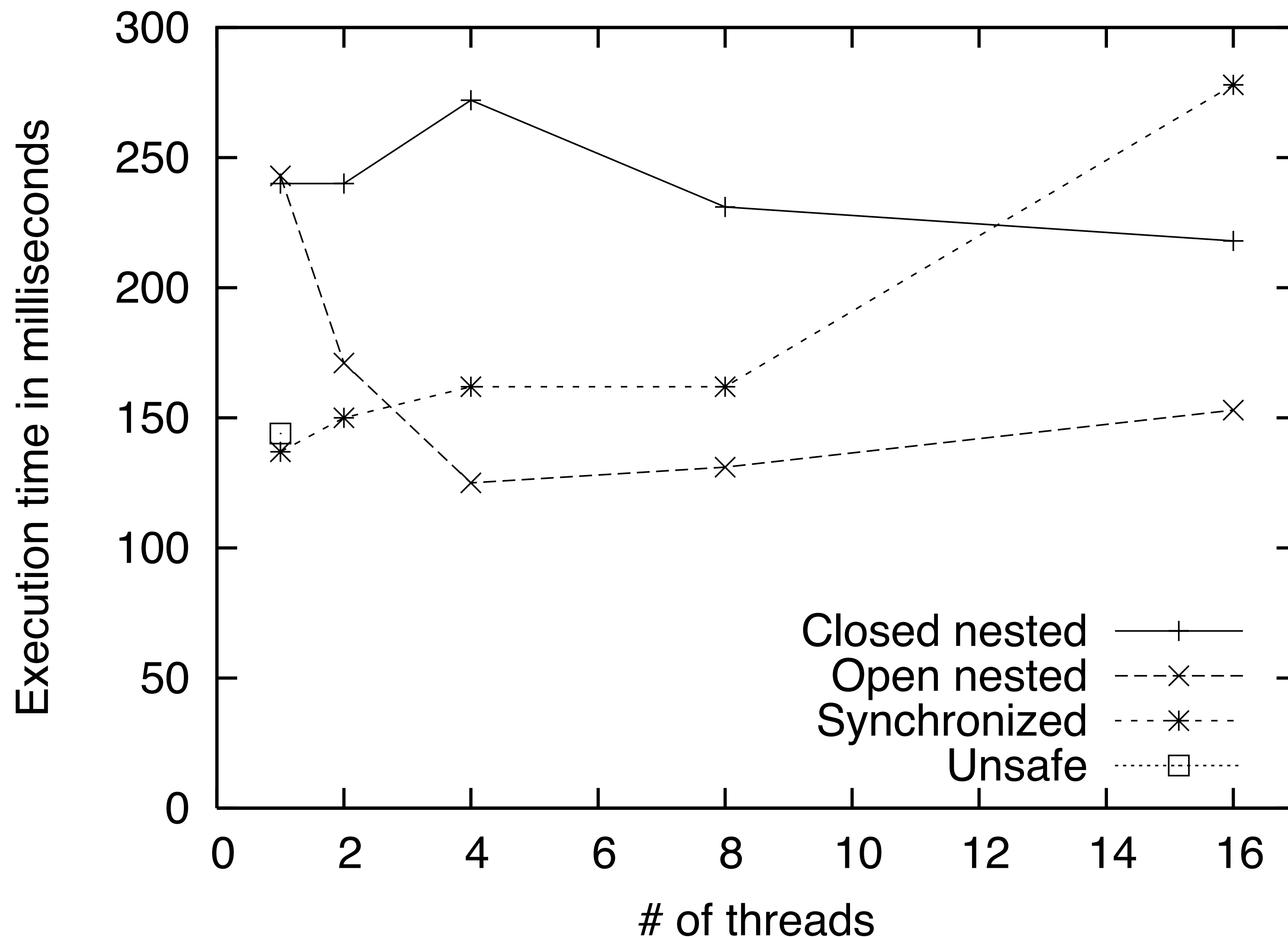
- `map` is unmodified `TreeMap/HashMap`
- `synchronized(map)` instead of `atomic`:

```
synchronized (map) {
                for (int i = 0; i < 4096; i++)
                  map.put(key[i], value[i]);
              }
```

PURDUE UNIVERSITY

DaCapo

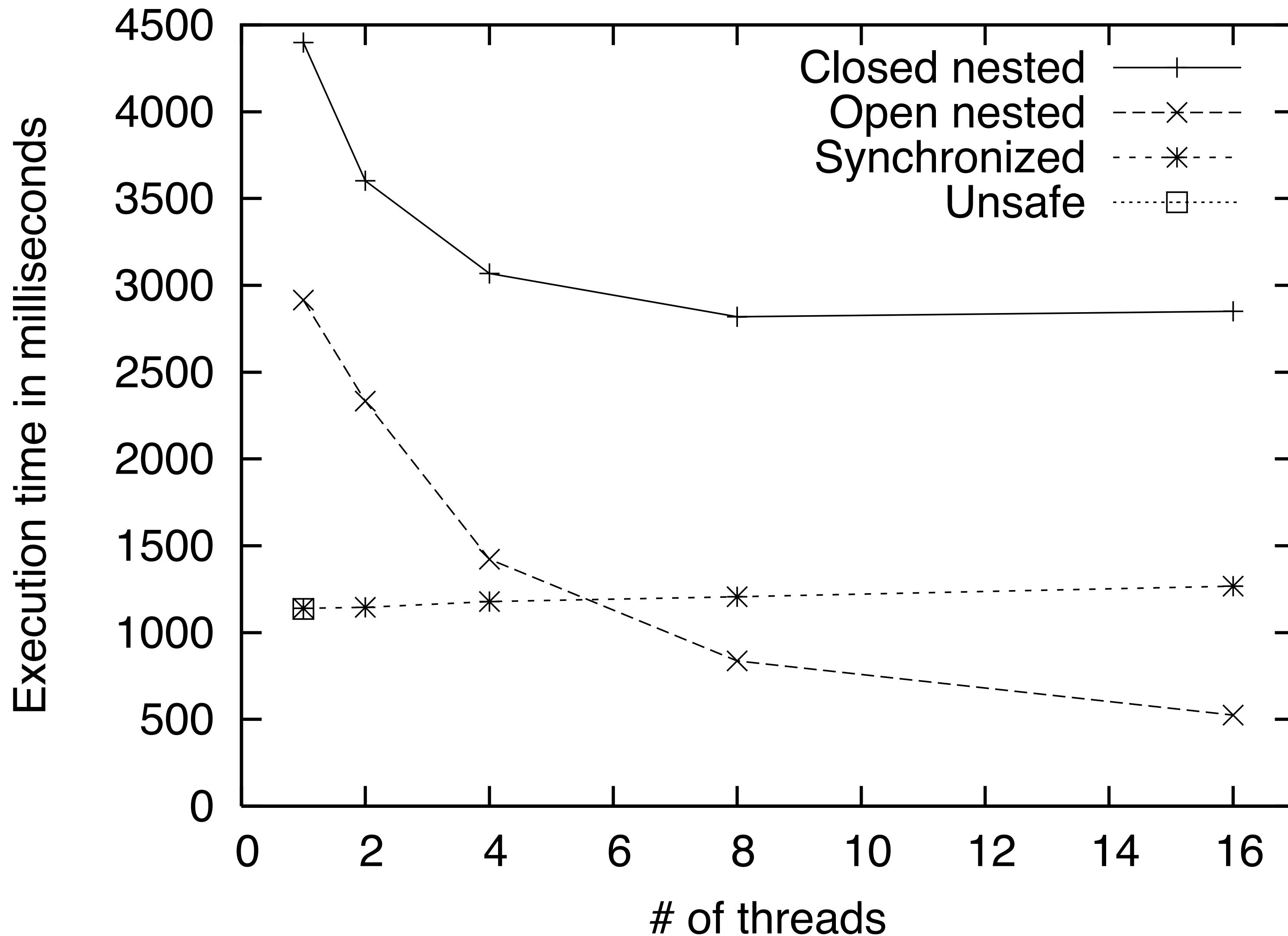
64K puts on a TreeMap



PURDUE
UNIVERSITY

DaCapo

64K puts on a HashMap

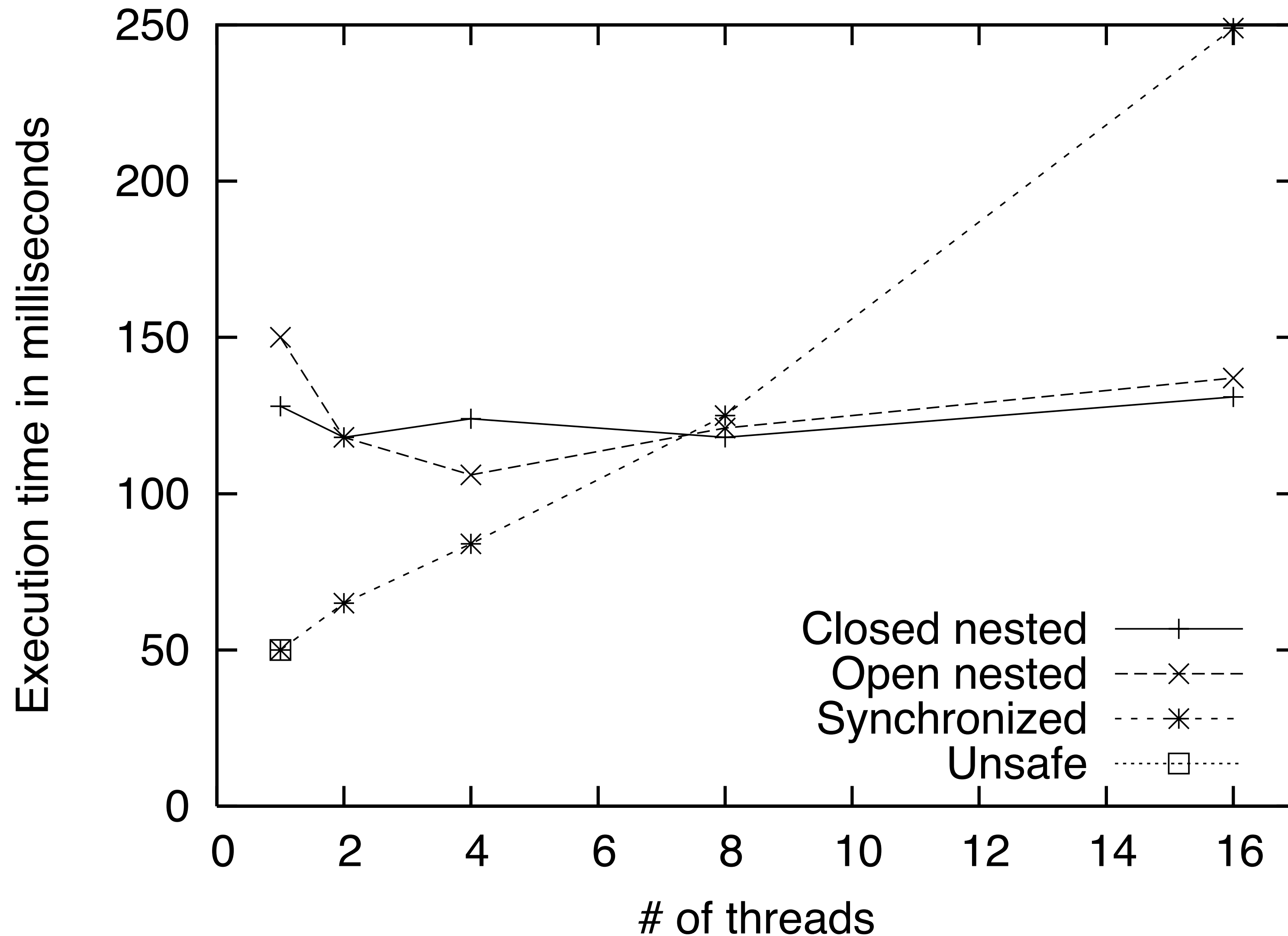


128 buckets

PURDUE UNIVERSITY

DaCapo

64K puts on a HashMap



64K buckets

PURDUE UNIVERSITY

DaCapo