CONTENTION MANAGEMENT IN DYNAMIC SOFTWARE-TRANSACTIONAL MEMORY

Dotan Nahum, STM091

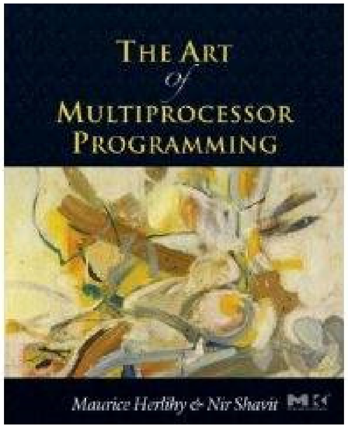
Overview

- Part I: DSTM review
  - Obstruction Freedom
  - Attributes

Part II: Contention Management
  - The Problem
  - DSTM Contention Management Mechanics
  - Contention Management Policies
  - Results, Conclusions, and Future
  - Extras

Dynamic STM (DSTM)

herlihy@cs.brown.edu



- the first published paper to describe a dynamic STM (DSTM) system that did not require specifying transaction's memory usage in advance.
- Used as API libraries
- Based on **obstruction freedom** as nonblocking progress condition
- Introduced an explicit **contention manager**

Obstruction Freedom

- Obstruction-freedom demands that a single thread completes its operations in bounded steps
- No concurrent assistance (locks)
- Ethernet CCMA/CD?
- Live locking is solved by the **contention manager**

Summary of DSTM Attributes

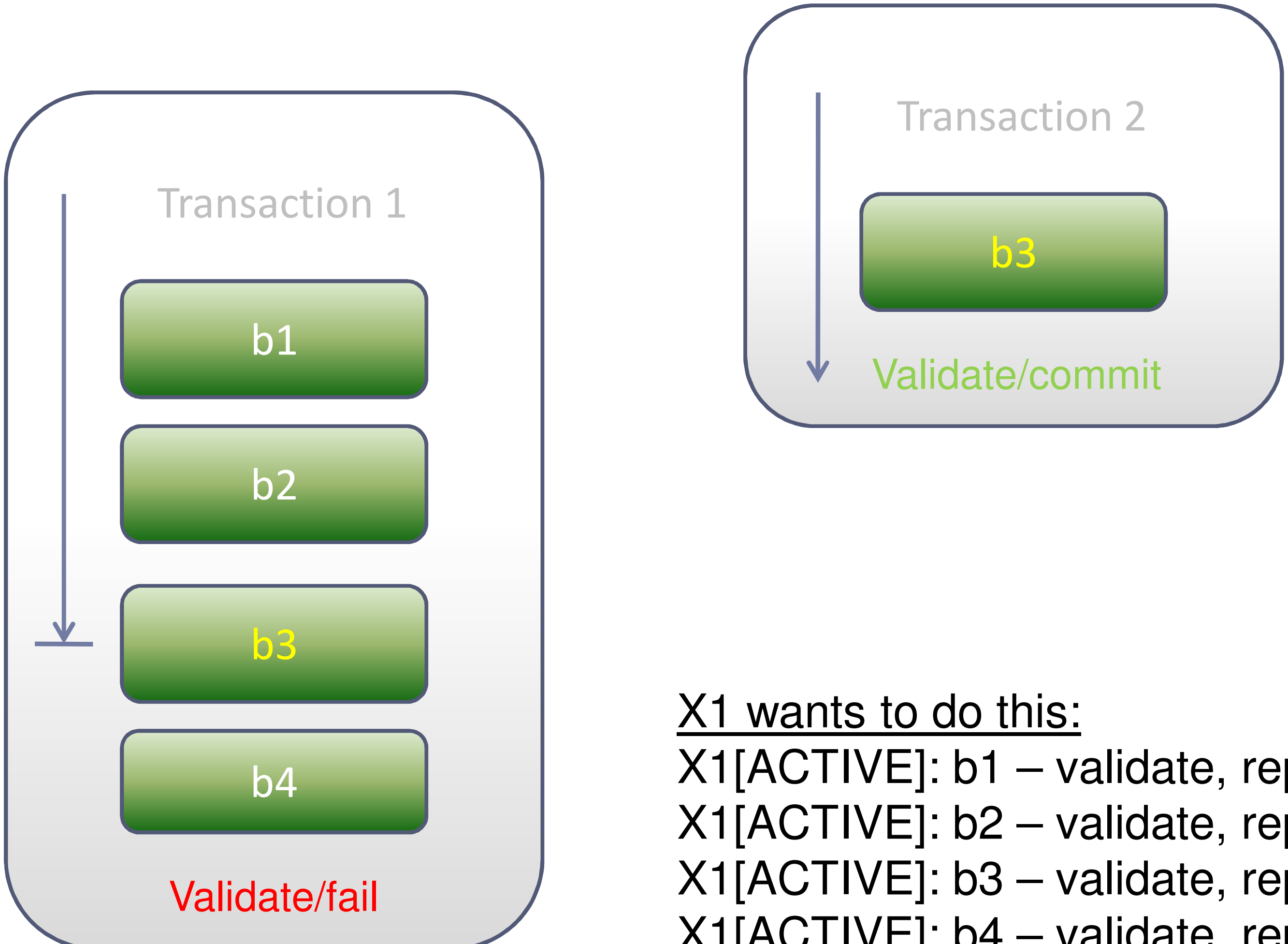
DSTM Attributes	
Transaction Granularity	Object
Concurrency Control	Optimistic
Synchronization	Obstruction Free
Conflict Detection	Early
Conflict Resolution	Contention Manager

Contention Management

Contention Management

- Running a transaction and an enemy transaction comes along wanting our block of memory. who wins?
- With obstruction freedom in mind, DSTM enables to divide the problem into 2 orthogonal concerns: progress and corectness
- Read – in order to conserve obs.freedom and progress, we must have contention management.

Running a transaction



<u>X1 wants to do this:</u>
X1[ACTIVE]: b1 – validate, replace
X1[ACTIVE]: b2 – validate, replace
X1[ACTIVE]: b3 – validate, replace
X1[ACTIVE]: b4 – validate, replace
X1[COMMITED]: validate, commit

Option 1

- Never abort enemy transaction
  - Priority inversion = deadlock
  - Starvation = enemy will always win, we always lose
  - Performance loss = move to enemy transaction = lose working set = page faults.

Option 2

- Always abort enemy transaction
  - Starvation = Starving the enemy
  - Livelock = both repeatedly restart, encounter each other, repeatedly restart…
- We need **policies**

The contention manager

- A set of policies for managing contention between transactions
- A policy must be deadlock free
- Manager must ensure progress
- Out of band: manager must not know about transaction inner-workings
- A good contention policy is between #1 and #2.

Impl. Considerations: “Visibility” of block reads(1)

- Invisible reads
  - Create a private copy of Locator, attach it to own transaction
  - Upon validation compare own and current, fail if mismatch
  - Invisible: Contention manager has no way to know who reads and cant expect potential conflicts

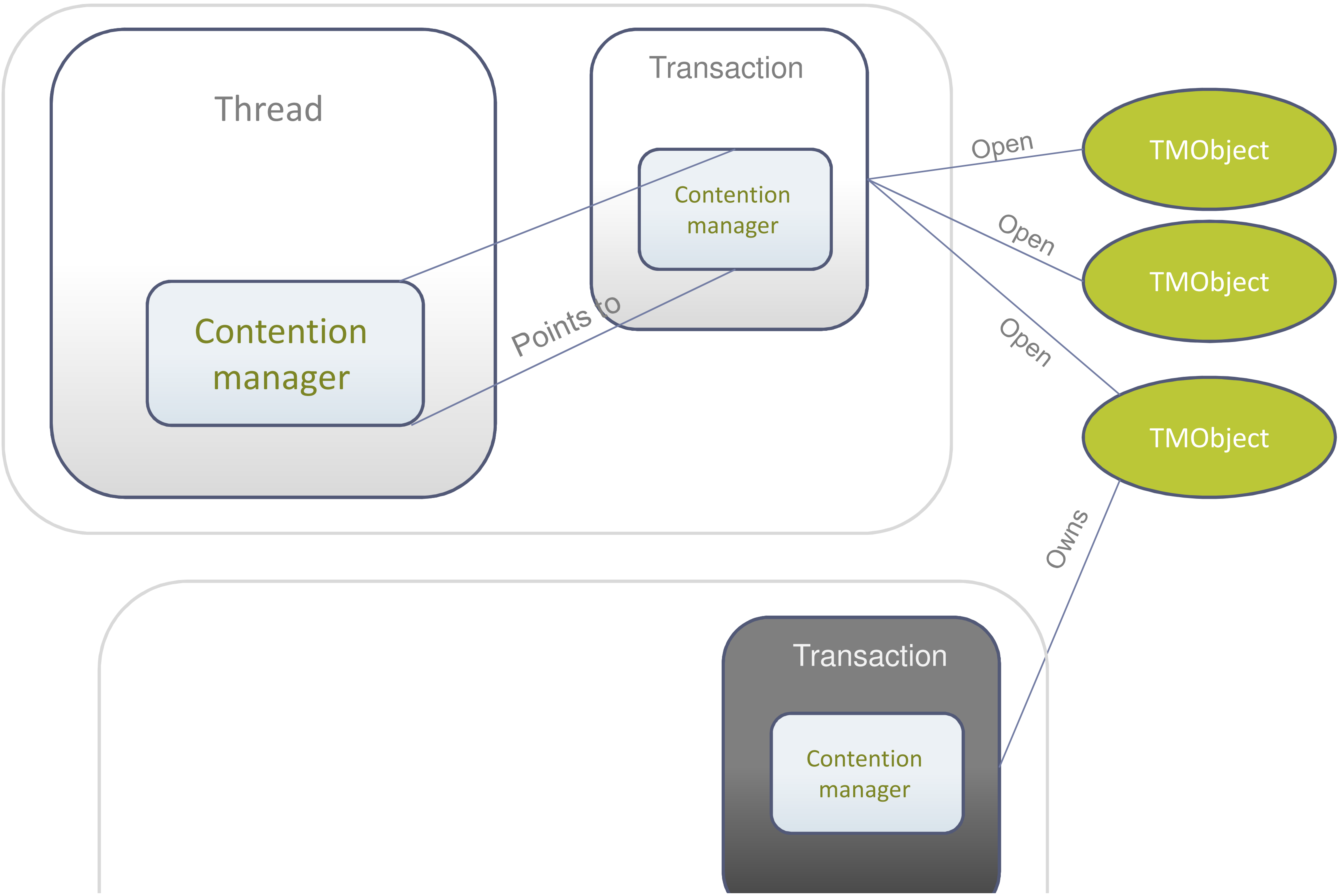
Impl. Considerations: “Visibility” of block reads(2)

- Visible reads
- Each block gets a linked list of readers
  - Added complexity:
  - Adding readers now has additional overhead
  - A new writer must traverse the list and (potentially) abort readers
- But we become more intelligible

Impl. Considerations: Mutual Abortion

- Problem: if a thread decides to abort an enemy, it does so without checking if it is itself already aborted.
- Fix: to minimize risk, a thread checks its own transaction status word just before aborting an enemy.

Contention Manager Mechanics



Contention Manager Interface

- **Notifications** announce events in a transaction:
  - begin
  - Commit-fail/commit-success
  - Self-abort
  - Block: begin-open/success-open/fail-open/change read access
  - Helps in making more informed decisions in the future
- **Requests** decide upon a conflict:
  - Abort enemy?

```java
public interface ContentionManager {
  /**
   * Either give the writer a chance to finish it, abort it, or both.
   * @param me Calling transaction.
   * @param other Transaction that's in my way.
   */
  void resolveConflict(Transaction me, Transaction other);

  /**
   * Either give the writer a chance to finish it, abort it, or both.
   * @param me Calling transaction.
   * @param other set of transactions in my way
   */
  void resolveConflict(Transaction me, Collection<Transaction> other);

  /**
   * Assign a priority to caller. Not all managers assign meaningful priorities.
   * @return Priority of conflicting transaction.
   */
  long getPriority();

  /**
   * Change this manager's priority.
   * @param value new priority value
   */
  void setPriority(long value);

  /**
   * Notify manager that object was opened.
   */
  void openSucceeded();

  /**
   * Notify manager that transaction committed.
   */
  void committed();

};
```

Contention Manager In Action(1)

Newer versions of DSTM will **emit the TMObjects** in real time, giving a proxy that will intercept open read/write. SXM uses .Net IL Emitting, DSTM2 uses BCEL.

OpenWrite(), resolve each of the readers

```
switch (me.State)
{
    case XStates.ABORTED:
        throw new AbortedException();
    case XStates.COMMITTED:
        return oldVersion;
    case XStates.ACTIVE:
        // check for read conflicts
        bool readConflict = false;
        foreach (XState reader in oldLocator.readers)
        {
            if (me.Conflict(reader))
            {
                manager.ResolveConflict(me, reader);
                readConflict = true;
            }
            if (readConflict)
            {
                goto retry;
            }
        }
```

SXM 1.1 (C#)

Locator.writePath(), used as part of getting a new locator

```
public void writePath(Transaction me, ContentionManager manager, Locator newLocator) {
  retry:
  while (true) {
    Copyable version = getVersion(me, manager);
    newLocator.oldVersion = version;
    newLocator.newVersion.copyFrom(version);
    for (Transaction t : readers) {
      if (t.isActive() && t != me) {
        manager.resolveConflict(me, t);
        continue retry;
      }
    }
    if (me.isAborted()) {
      throw new AbortedException();
    }
    return;
  }
}
```

DSTM2(Java)

Contention Manager In Action(2)

```java
public Copyable getVersion(Transaction me, ContentionManager manager) {
  while (true) {
    if (me != null && me.getStatus() == Status.ABORTED) {
      throw new AbortedException();
    }
    switch (writer.getStatus()) {
      case ACTIVE:
        if (manager == null) {
          throw new PanicException("Transactional/Non-Tranactional race");
        }
        manager.resolveConflict(me, writer);
        continue;
      case COMMITTED:
        return newVersion;
      case ABORTED:
        return oldVersion;
      default:
        throw new PanicException("Unexpected transaction state: " + writer.getStatus());
    }
  }
```

Contention Management Policies

1. Aggressive
2. Polite
3. Randomized
4. Karma
5. Eruption
6. KillBlocked
7. Kindergarten
8. Timestamp
9. QueueOnBlock

Characterizing a Policy

- Transaction history
  - # of open blocks
  - Time of start
- What do we do when a thread is dead
- Cyclic blocking
- Starvation
- Performance (working set/page faults)

1. Aggressive

- Ignore everything,
- Always abort enemy transactions
- Highly prone to **livelock**

```
public class AggressiveManager extends BaseManager {
  public AggressiveManager() {}
  public void resolveConflict(Transaction me, Transaction other) {
      other.abort();
  }
  ...
```

2. Polite

- Uses exponential backoff
- Spin period of time proportional to 2^n [ns]
  - n = number of retries on access to the block
  - 8 retries are maximum, after that abort enemy
- In other words, let enemy(ies) go, but there is a limit (n) to how nice we can be.
- Prone to **performance loss** based on preemption = page faults.

2. Polite (cont'd)

```java
static final int MIN_LOG_BACKOFF = 4;
static final int MAX_LOG_BACKOFF = 26;
static final int MAX_RETRIES = 22;
static final int BACKOFF_DIVISOR = 1000000;
Random random;
int currentAttempt = 0;

@Override
public void resolveConflict(Transaction me, Transaction other) {
  if (!other.isActive() || me.isAborted() || other == me) {
    return;
  }
  if (Thread.stop) {
    throw new GracefulException();
  }
  int logBackoff = currentAttempt - 2 + MIN_LOG_BACKOFF;
  if (logBackoff > MAX_LOG_BACKOFF) {
    logBackoff = MAX_LOG_BACKOFF;
  }
  int sleep = random.nextInt(1 << logBackoff);
  super.sleep(sleep);
  currentAttempt++;
  if (currentAttempt == MAX_RETRIES) {
    other.abort();
    currentAttempt = 0;
  }
}
```

3. Randomized

- Ignore all notifications
- Upon contention – flip a coin
- Coin bias tunable.

4. Karma

- Amount of work a transaction has performed is measured in **number of open blocks**.
- Tracks # of open blocks of a transaction which will be its priority:
  - On **commit**: reset priority (0)
  - On **open block**: priority +1 (good karma)
- On conflict compare priorities, abort enemy if lower.
- Karma effect: aborted enemy gets to keep his points. In the next life it will have a better chance.
- Note: each thread gains at least 1 point with every unsuccessful attempt. Short transactions can compete with longer ones that way.

4. Karma (cont'd)

```
public class KarmaManager extends BaseManager {
  static final int SLEEP_PERIOD = 100;
  @Override
  public void resolveConflict(Transaction me, Transaction other) {
    ContentionManager otherManager = other.getContentionManager();
    for (int attempts = 0; ; attempts++) {
        if(other.isActive()) {
                      long delta = otherManager.getPriority() - priority;
                      if (attempts > delta) {
                        other.abort();
                      }
        } else break;
    }
  }

  /**
   * Reset priority only on commit. On abort, restart with previous priority.
   * "Cosmic debt"?. More like cosmic credit.
   **/
  @Override
  public void committed() {
    setPriority(0);
  }

  @Override
  public void openSucceeded() {
    setPriority(getPriority() + 1);
  }
}
```

5. Eruption

- Increase pressure on transaction that a transaction is "waiting" on:
  - If a transaction is blocked on another transaction, it will give its priority points to it.
  - Eventually it will have enough priority points to win all other future conflicts and finish as soon as possible
  - When it finishes our guy will now have a clear road
- Popularity principle: if a transaction is taking a popular resources, everyone will "encourage" it to complete by handing in their priority points.

5. Eruption (cont’d)

```
@Override
public void resolveConflict(Transaction me, Transaction other) {
  ContentionManager otherManager = other.getContentionManager();
  for (int attempts = 0; ; attempts++) {
          if(other.isActive()) {
            long otherPriority = otherManager.getPriority();
            long delta = otherPriority - priority;
            if (attempts > delta) {
              other.abort();
              return;
            }
            // Unsafe increment, but too expensive to synchronize.
            if (priority > transferred) {
              otherManager.setPriority(otherPriority + priority - transf
erred);
              transferred = priority;
            }
          } else break;
  }
}

@Override
public void openSucceeded() {
  priority++;
  transferred = 0;
}
```

6. KillBlocked

- Mark transaction as blocked on first openblock-fail event (waiting)
- Abort enemy when:
  - Enemy is also blocked, or
  - Max. wait time expired – to break cyclic blocking

7. Kindergarten

- Each manager stores a list of enemies that it gave up its turn for.
- Upon conflict if the enemy is not in the list, add myself to its list and abort self – it will be his turn.
- If the enemy is in the list back off for a max. number of times
  - Once that number of times passed, abort the enemy.
  - Note: In the code we back off for 0 number of times.

7. Kindergarten (cont'd)

```java
public class KindergartenManager extends BaseManager {
  static final int SLEEP_PERIOD = 1000; // was 100
  static final int MAX_RETRIES = 100; // was 10
  TreeSet<KindergartenManager> otherChildren;
  /** Creates new Kindergarten manager */
  public KindergartenManager() {
    super();
    otherChildren = new TreeSet<KindergartenManager>();
    otherChildren.add(this);
  }

  @Override
  public void resolveConflict(Transaction me, Transaction other) {
    try {
        KindergartenManager otherManager =
            (KindergartenManager) other.getContentionManager();
        // first, check sharing records.
        if (otherChildren.contains(otherManager)) {
          otherChildren.remove(otherManager);
          other.abort();           // My turn! My turn!
          return;
        }
        else {    otherChildren.add(otherManager);   }
        me.abort(); // give up
        return;
    } catch (ClassCastException e) {
      other.abort(); // Oh, other not a Kindergartener. Kill it.
      return;
      }
  }}
```

8. Timestamp

- Record current time at transaction-start
- On conflict if our transaction is earlier, abort enemy
- Otherwise, we start a series of timed waits.
  - After half of the max. number of waits we mark the enemy 'defunct'.
  - Enemy has a chance to reset its own defunct flag after each transaction related action.
  - If the max. number of waits have been reached and the defunct flag have never been reset, enemy is aborted.
- Similar algorithms have been used with databases.

9. QueueOnBlock

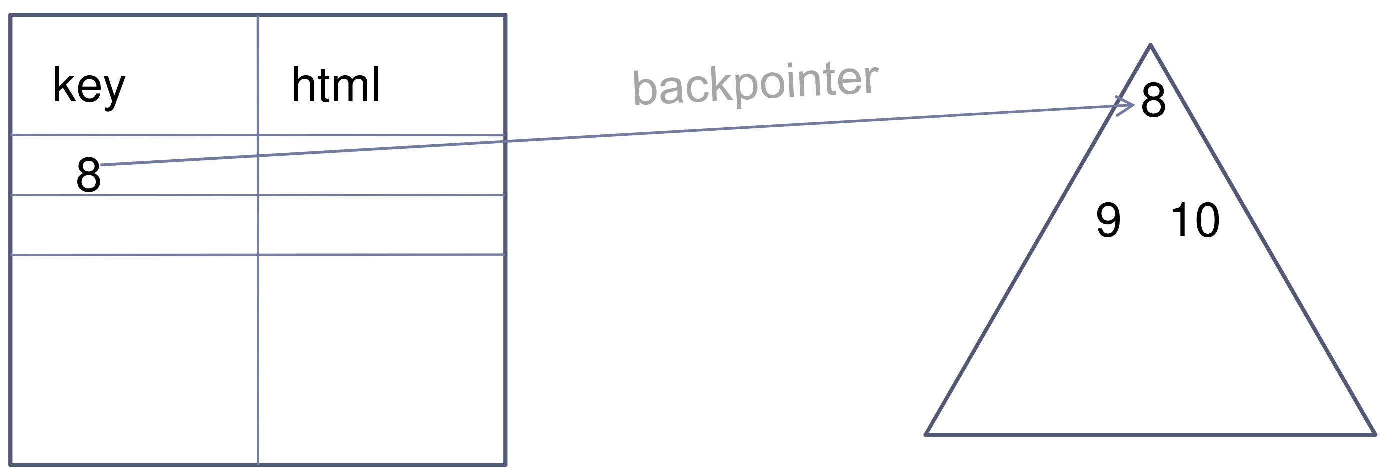
- Upon contention, our manager will register itself inside a list in the enemy transaction
- It will spin on a ‘finished’ flag which is set in the enemy transaction upon completion.
- If it waits too long will abort enemy.
- Not all of those wait on the same block. Those who do, will compete for it. Loser will enter the winner’s wait list.

Benchmarks(1)

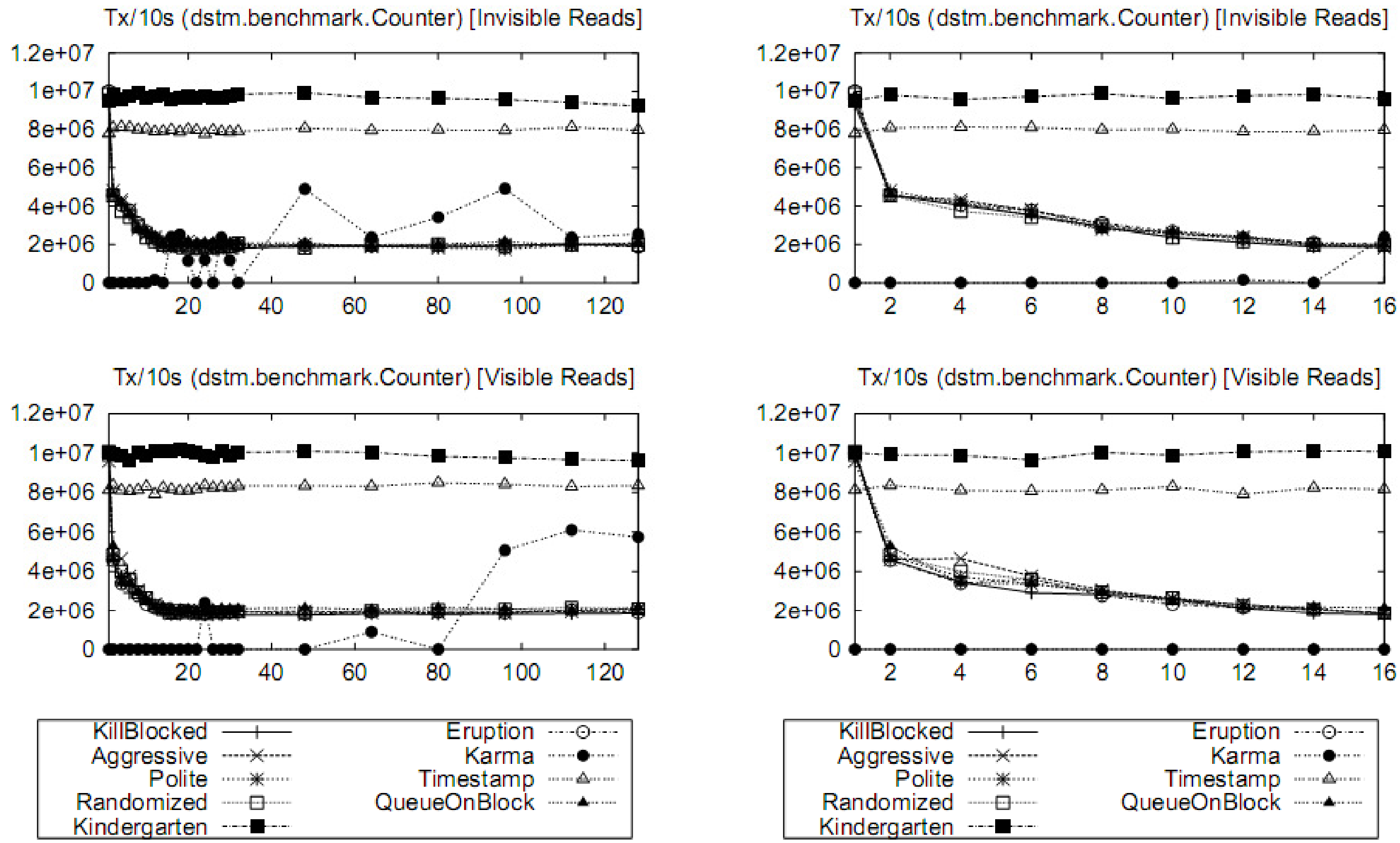
- Counter – shared counter, incremented via transactions
- Set for integers “IntSet”, implemented as:
  - Sorted linked list, each block opens for write
  - Sorted linked list, blocks open transiently, released as transaction approaches insertion/deletion. “Release”
  - Red-black tree, blocks open for read, upgraded to write when needed. “RBTree”
  - Random insert or delete integers in 0..255
  - Throughput = #completeTransactions /timeperiod

Benchmarks(1)

- LFUCache – least frequently used cache implementation for an HTTP proxy:
  - Lookup table, key is Integer stored as TMObject, value will be an HTML page.
  - Priority heap 255 entries for the LFU part, lower freq. near the root. Each entry is <key, freq.>

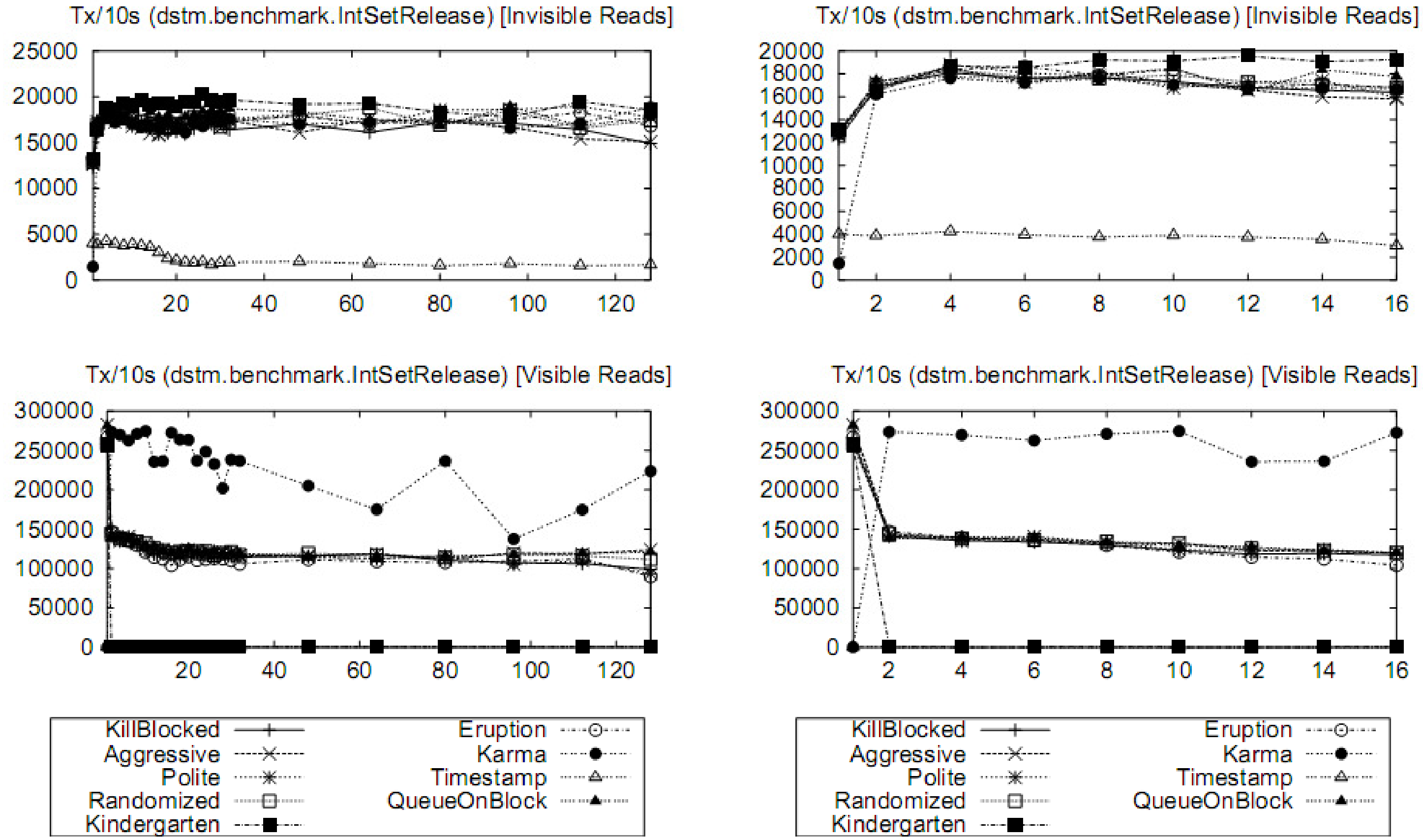


Counter



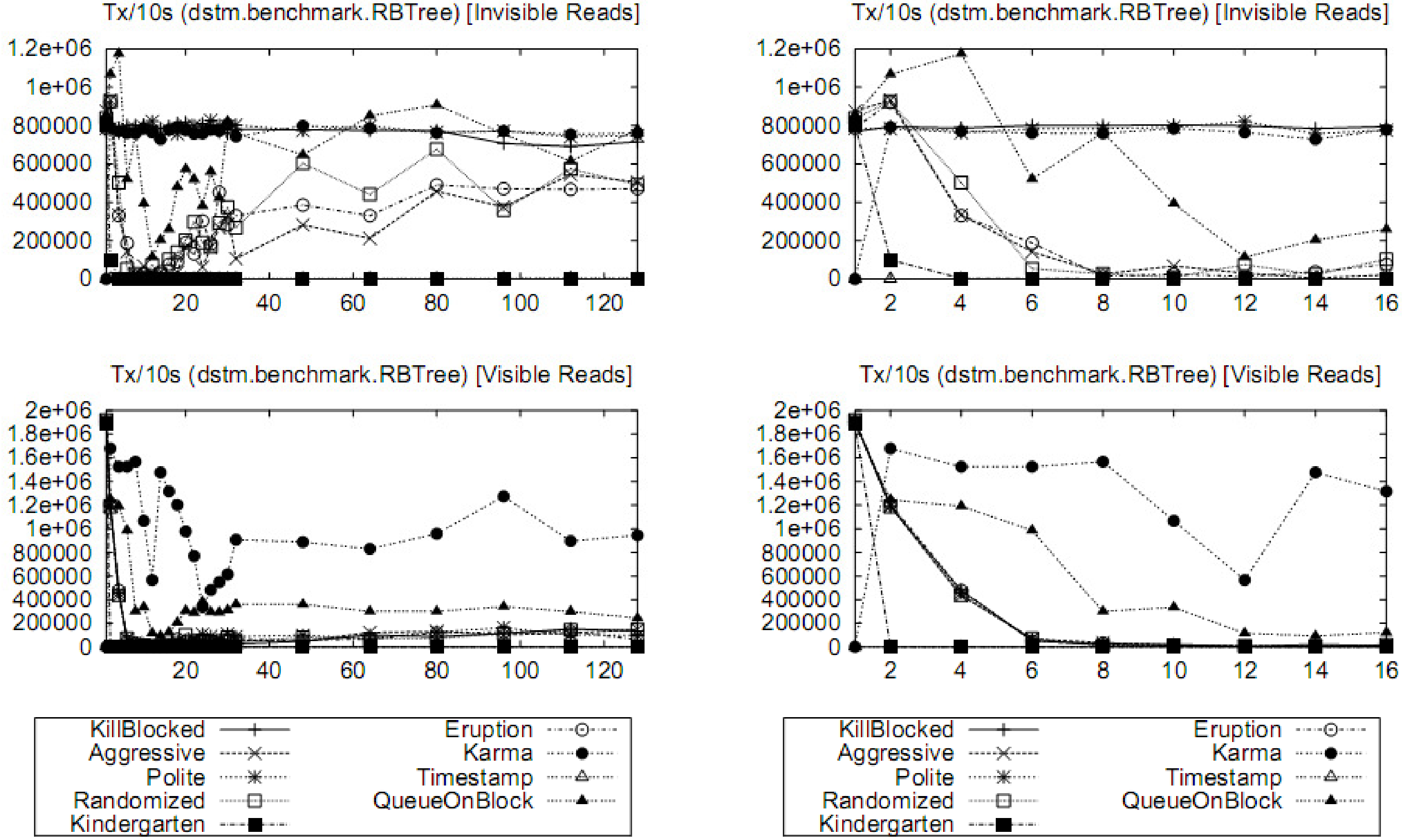
- Every transaction will collide with the other
- Kindergarten performs best:
    - There is a delay between two modes: aborting the enemy or aborting self.
      since the transactions for counter are short, some can squeeze into that delay and complete, without us being able to abort those.
- Datetime also perform better because arranging transaction by date will virtually serialize them.
- No difference between visible/invisible reads because this is not read bound

intSet-release



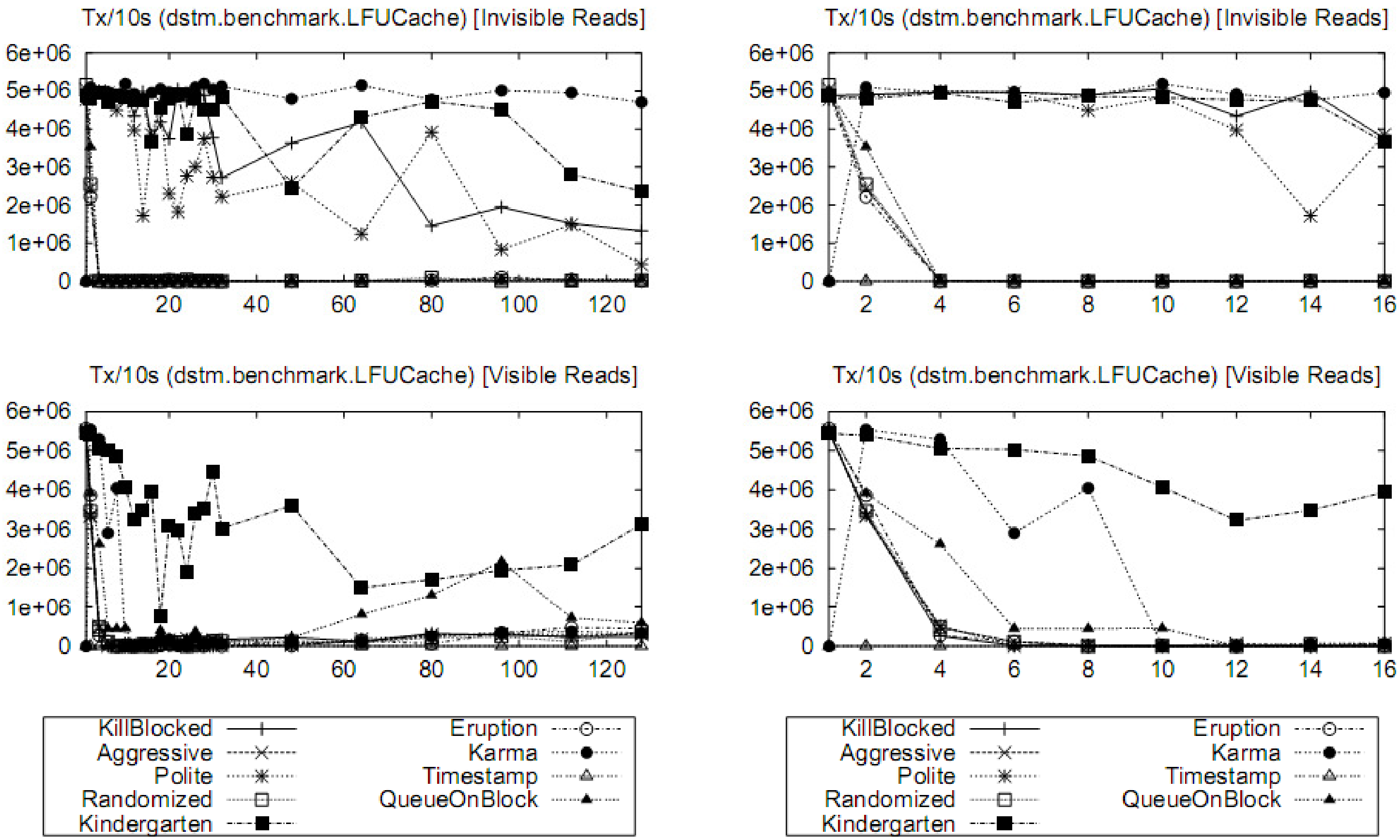
- Invisible reads: Timestamp perform badly, others comparable.
- Visible reads: Karma is a top performer.
- Kindergarten is bad, virually livelocks
- Visible reads are good here: they stall writers and let the readers finish their work since most reads on the list are **temporary** and short anyway.
- Nonrelease version mirrors the Counter benchmark.

RBTree



- •Karma outperforms all
- •Most managers improve as multiprogramming degree rises
- •For invisible reads Karma is joined with Aggressive and Polite as top performers
- •Big problem with visible vs invisible reads: by the nature of walking a redblack tree:
    - •reader work from the root towards insertion point
    - •Writer is working its way up from insertion point to root in order to restore the red black properties.
    - •Make reads visible and we have a mess when writer/reader meet and up towards every level in the tree.

LFUCache



- Managers do worse with visible reads
- Top performer is Karma for invisible reads, Kindergarten for visible reads.

Conclusions

- Performance should be considered according to many factors:
  - Visible/invisible reads
  - Type of problem
  - Implementation of solution (write or readbound, both?)
  - Type of manager
- This research only starts to examine what seems to be a very wide area of experimentation.

Snooping into the future (spoiler)

(Drawing from Adv. Contention Management article)

- Timestamp have been improved to include a periodic update that each xaction will maintain that indicates it is still alive
- Polite and Karma being top performers are merged into Polka. Uses Karma with Polite's exponential backoffs.
- We have a conclusion:
  - Visible reads good for read bound transactions
  - Invisible reads good for high contention in write bound transactions

Thank You
Questions?

STM Extras

- Google lecture about STM: http://www.youtube.com/watch?v=FHUFHCPh8Ms
- Garbage collection vs STM http://tm-101.blogspot.com/2008/03/garbage-collection-vs-transactional.html
- William s. Scherer http://www.cs.rice.edu/~wns1/
- Simon Peyton Jones http://research.microsoft.com/~simonpj/papers/stm/
- SXM 1.1 http://research.microsoft.com/research/downloads/Details/FBE1CF9A-C6AC-4BBB-B5E9-D1FDA49ECAD9/Details.aspx
- STM Articles http://www.cs.wisc.edu/trans-memory/
- Herlihy http://www.cs.brown.edu/~mph/
- Microsoft STM Teamhttp://blogs.msdn.com/stmteam/default.aspx
- STM Article http://channel9.msdn.com/shows/Going+Deep/Programming-in-the-Age-of-Concurrency-Software-Transactional-Memory/
- Improvements on DSTM http://tm-101.blogspot.com/2008/03/reading-5-improvements-on-dstm-and-wstm.html