Software Transactional Memory

Курс «Базы данных»

Цесько Вадим Александрович
http://incubos.org
@incubos

Computer Science Center

9 декабря 2013 г.

Содержание

Мотивация

Задача

- Множество изменяемых объектов в памяти
- Атомарность наборов операций с объектами

Решение

Software Transactional Memory[a]:

- Память как транзакционное хранилище
- Универсальная альтернатива ручным блокировкам
- ACI

[a] http://en.wikipedia.org/wiki/Software_transactional_memory

Идея

- (Очень) оптимистичные транзакции
- Если прочитанные значения не менялись, то `commit`
- В противном случае — `retry`
- Возможен `abort` в любой момент
- Как следствие – max concurrency

Ручные блокировки

- Нужно думать о перекрывающихся операциях
- Нужно «держать в голове» весь код
- Deadlocks, livelocks, progress, etc.
- Очень трудно воспроизвести и отладить
- Priority inversion[1]

[1] http://en.wikipedia.org/wiki/Priority_inversion

Подход STM

- Simple
- Maintainable
- **Composable**
- Не нужно думать о deadlocks и livelocks
- Priority inversion — `abort` низкоприоритетной транзакции
- Но (почти) недопустимы side effects (в т. ч. IO)

Реализации

- Множество реализаций для всех языков
- Встроено в Clojure
- **ScalaSTM**[2]

[2] http://nbronson.github.io/scala-stm/

Scala STM Expert Group

- Akka
- Stanford
- Tel-Aviv University
- EPFL
- Cisco
- etc.

С высоты птичьего полёта

- Живёт между `atomic`-блоком и Heap
- Перехватывает чтения и записи
- Чтения и записи из разных потоков перемешались — `rollback` записей и `retry`
- Иначе — `commit`
- Видны только закоммиченые изменения
- Принимаем во внимание только `Ref`ы
- Реализация в виде **библиотеки**
- `TSet` и `TMap`

Достоинства

- **Say what you mean**
  - (nested) `atomic`
- **Readers scale**
  - CPU cache friendly
- **Exceptions automatically trigger cleanup**
  - By default
- **Waiting for complex conditions is easy**
  - `retry`, chaining
- **Simple**
  - Just a library

Недостатки

- **Two extra characters per read or write**
  - `x: Ref`
  - Read: `x()`
  - Write: `x() = y`
- **Single-thread overheads**
  - Но полезен `rollback` при исключениях
- **Rollback doesn't mix well with I/O**
  - Но есть хуки

Basic

```scala
import scala.concurrent.stm._

val x = Ref(0) // allocate a Ref[Int]
val y = Ref.make[String]() // type-specific default
val z = x.single // Ref.View[Int]

atomic { implicit txn =>
  val i = x() // read
  y() = "x was " + i // write
  val eq = atomic { implicit txn => // nested atomic
    // both Ref and Ref.View can be used inside atomic
    x() == z()
  }
  assert(eq)
  y.set(y.get + ", long-form access")
}
```

Advanced

```
1  // only Ref.View can be used outside atomic
2  println("y was '" + y.single() + "'")
3  println("z was " + z())
4
5  atomic { implicit txn =>
6    y() = y() + ", first alternative"
7    if (x getWith { _ > 0 }) // read via a function
8      retry // try alternatives or block
9  } orAtomic { implicit txn =>
10   y() = y() + ", second alternative"
11 }
12
13 val prev = z.swap(10) // atomic swap
14 val success = z.compareAndSet(10, 11) // atomic compare-and-set
15 z.transform { _ max 20 } // atomic transformation
16 val pre = y.single.getAndTransform { _.toUpperCase }
17 val post = y.single.transformAndGet { _.filterNot { _ == ' ' } }
```

Use Ref for shared variables

```
1  import scala.concurrent.stm._
2
3  class ConcurrentIntList {
4    private class Node(
5        val elem: Int,
6        prev0: Node,
7        next0: Node) {
8      val isHeader = prev0 == null
9      val prev = Ref(if (isHeader) this else prev0)
10     val next = Ref(if (isHeader) this else next0)
11   }
12
13   private val header = new Node(-1, null, null)
```

Wrap your code in atomic

```
1 def addLast(elem: Int) {
2   atomic { implicit txn =>
3     val p = header.prev()
4     val newNode = new Node(elem, p, header)
5     p.next() = newNode
6     header.prev() = newNode
7   }
8 }
```

Compose atomic operations

```
1 def addLast(e1: Int, e2: Int, elems: Int*) {
2   atomic { implicit txn =>
3     addLast(e1)
4     addLast(e2)
5     elems foreach { addLast(_) }
6   }
7 }
```

Optimize single-operation transactions

`Ref.View`:

- Получение через `Ref.single`
- Можно использовать вне `atomic`
- Поддерживает транзакции из одной операции
- `swap`, `compareAndSet`, `transform`, etc.

Пример:

```
1 /*
2   def isEmpty = atomic { implicit t =>
3     header.next() == header
4   }
5 */
6
7 def isEmpty = header.next.single() == header
```

Wait for conditions to change

```
1  def removeFirst(): Int =
2    atomic { implicit txn =>
3      val n = header.next()
4      if (n == header)
5        retry
6      val nn = n.next()
7      header.next() = nn
8      nn.prev() = header
9      n.elem
10   }
```

Wait for multiple events

```
1 def maybeRemoveFirst(): Option[Int] = {
2   atomic { implicit txn =>
3     Some(removeFirst())
4   } orAtomic { implicit txn =>
5     None
6   }
7 }
```

Composition: select

```
1  object ConcurrentIntList {
2    def select(stacks: ConcurrentIntList*):
3      (ConcurrentIntList, Int) =
4      atomic { implicit txn =>
5        for (s <- stacks) {
6          s.maybeRemoveFirst() match {
7            case Some(e) => return (s -> e)
8            case None => _
9          }
10       }
11       retry
12     }
```

Be careful about rollback (1)

```
1  def badToString: String = {
2    val buf = new StringBuilder("ConcurrentIntList(")
3    atomic { implicit txn =>
4      var n = header.next()
5      while (n != header) {
6        buf ++= n.elem.toString
7        n = n.next()
8        if (n != header) buf ++= ","
9      }
10   }
11   buf ++= ")" toString
12 }
```

Be careful about rollback (2)

```
1  override def toString: String = {
2    atomic { implicit txn =>
3      val buf = new StringBuilder("ConcurrentIntList(")
4      var n = header.next()
5      while (n != header) {
6        buf ++= n.elem.toString
7        n = n.next()
8        if (n != header) buf ++= ","
9      }
10     buf ++= ")" toString
11   }
12 }
```

Цель

- In-memory DB
- User-defined indices

API

```
scala> case class User(id: Int, name: String, likes: Set[String])
scala> val m = new IndexedMap[Int, User]
scala> m.put(10, User(10, "alice", Set("scala", "climbing")))
res0: Option[User] = None
scala> val byName = m.addIndex { (id,u) => Some(u.name) }
byName: (String) => Map[Int,User] = <function1>
scala> val byLike = m.addIndex { (id,u) => u.likes }
byLike: (String) => Map[Int,User] = <function1>
scala> m.put(11, User(11, "bob", Set("scala", "skiing")))
res1: Option[User] = None
scala> byName("alice")
res2: Map[Int,User] = Map((10,User(10,alice,Set(scala,
    climbing))))
scala> byLike("scala").values map { _.name }
res3: Iterable[String] = List(alice, bob)
```

A high-level sketch

```
1  import scala.concurrent.stm._
2  class IndexedMap[A, B] {
3    private val contents = TMap.empty[A, B]
4    // TODO def addIndex(view: ?): ?
5    def get(key: A): Option[B] = contents.single.get(key)
6    def put(key: A, value: B): Option[B] =
7      atomic { implicit txn =>
8        val prev = contents.put(key, value)
9        // TODO: update indices
10       prev
11     }
12   def remove(key: A): Option[B] =
13     atomic { implicit txn =>
14       val prev = contents.remove(key)
15       // TODO: update indices
16       prev
17     }
18 }
```

Types for the view function and index

Помедитируем:

```
1 def addIndex(view: ((A, B) => Iterable[C])):
2   (C => Map[A, B]) = ...
```

Tracking and updating indices (1)

```
1  private class Index[C](view: (A, B) => Iterable[C])
2    extends (C => Map[A, B]) {
3    def += (kv: (A, B)) // TODO
4    def -= (kv: (A, B)) // TODO
5  }
6
7  private val indices = Ref(List.empty[Index[_]])
8
9  def addIndex[C](view: (A, B) => Iterable[C]):
10   (C => Map[A, B]) =
11   atomic { implicit txn =>
12     val index = new Index(view)
13     indices() = index :: indices()
14     contents foreach { index += _ }
15     index
16   }
```

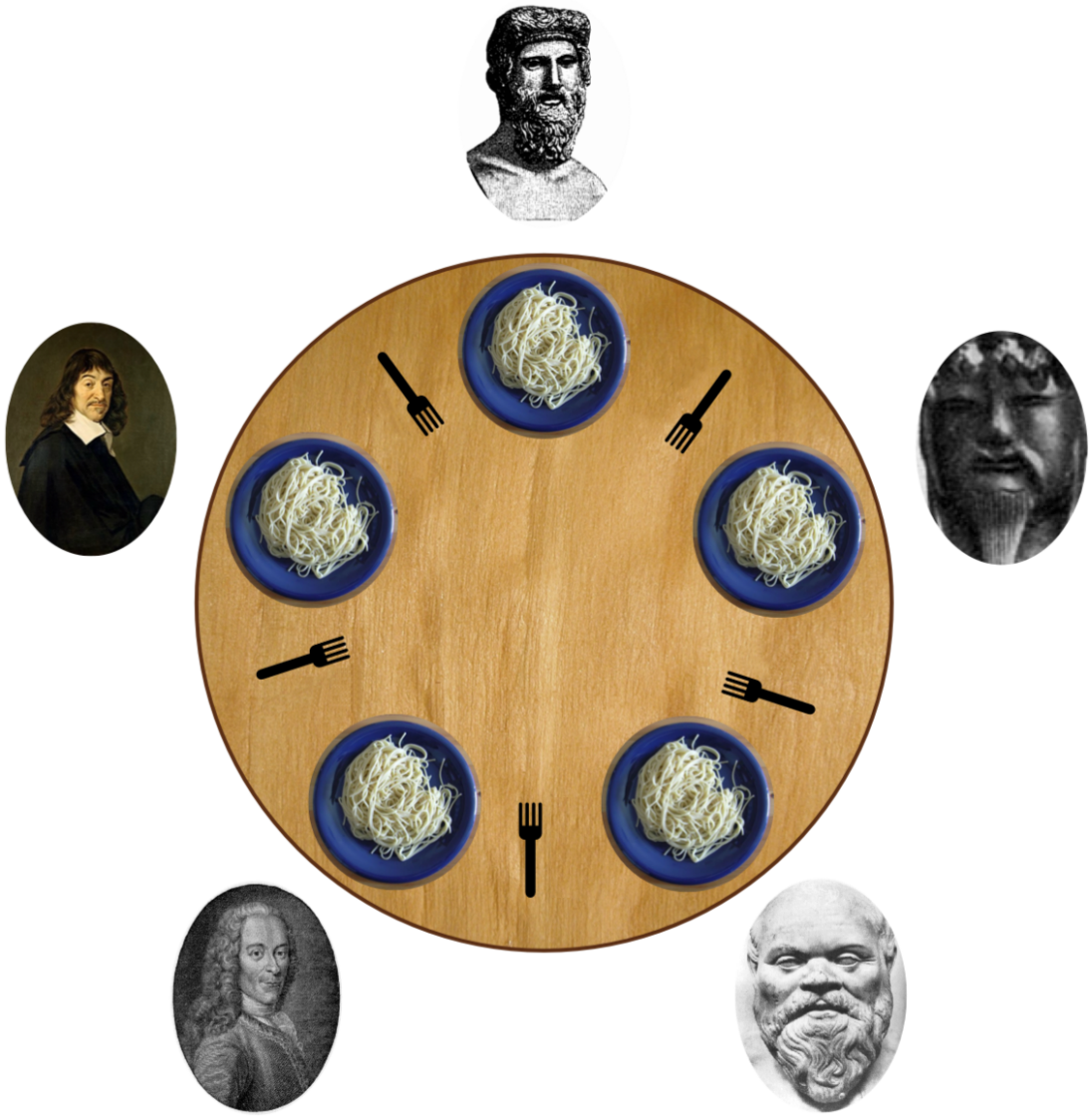
Tracking and updating indices (2)

```
1  def put(key: A, value: B): Option[B] =
2    atomic { implicit txn =>
3      val prev = contents.put(key, value)
4      for (p <- prev; i <- indices()) i -= (key -> p)
5      for (i <- indices()) i += (key -> value)
6      prev
7    }
8
9  def remove(key: A): Option[B] =
10   atomic { implicit txn =>
11     val prev = contents.remove(key)
12     for (p <- prev; i <- indices()) i -= (key -> p)
13     prev
14   }
```

Index internals

```
1  private class Index[C](view: (A, B) => Iterable[C])
2    extends (C => Map[A, B]) {
3    val mapping = TMap.empty[C, Map[A, B]]
4    def apply(derived: C) =
5      mapping.single.getOrElse(derived, Map.empty[A, B])
6    def += (kv: (A, B))(implicit txn: InTxn) {
7      for (c <- view(kv._1, kv._2))
8        mapping(c) = apply(c) + kv
9    }
10   def -= (kv: (A, B))(implicit txn: InTxn) {
11     for (c <- view(kv._1, kv._2)) {
12       val after = mapping(c) - kv._1
13       if (after.isEmpty)
14         mapping -= c
15       else
16         mapping(c) = after
17     }
18   }
```

Проблема



STM solution

```
1  class Fork { val inUse = Ref(false) }
2
3  def meal(left: Fork, right: Fork) {
4    // thinking
5    atomic { implicit txn =>
6      if (left.inUse() || right.inUse())
7        retry // forks are not both ready, wait
8      left.inUse() = true
9      right.inUse() = true
10   }
11   // eating
12   atomic { implicit txn =>
13     left.inUse() = false
14     right.inUse() = false
15   }
16 }
```

Waiting

`retry` в `atomic` ≈ `wait()` в `synchronized`, но `retry`:

- Безопаснее
  - STM определяет модификации `Ref`, ведущие к пробуждению (вместо `notifyAll`)
  - Невозможны «потерянные» пробуждения
- Эффективнее
  - Нет «лишних» пробуждений
  - Можно ожидать на любых условиях, а не только на предопределённых

Search with backtracking (1)

Optimistic concurrency control as a search with backtracking:

```
1  val (x, y) = (Ref(10), Ref(0))
2
3  def sum = atomic { implicit txn =>
4    val a = x()
5    val b = y()
6    a + b
7  }
8
9  def transfer(n: Int) {
10   atomic { implicit txn =>
11     x -= n
12     y += n
13   }
14 }
```

Search with backtracking (2)

```
1  // sum
2  atomic
3  |  begin txn attempt
4  |  |  read x -> 10
5  |  |     :
6  |  |
7  |  |     :
8  |  |
9  |  |  read y -> x read is invalid
10 |  roll back
11 |  begin txn attempt
12 |  |  read x -> 8
13 |  |  read y -> 2
14 |  commit
15 +-> 10
```

```
// transfer(2)
atomic
|  begin txn attempt
|  |  read x -> 10
|  |  write x <- 8
|  |  read y -> 0
|  |  write y <- 2
|  commit
+-> ()
```

Retry. Семантика

- Вызов `retry` — сигнал о dead end, даже если все чтения и записи консистентны
- STM откатится и попробует снова
- Если некоторые из прочитанных `Refов` изменились, то `atomic` блок может пойти по другому пути и избежать `retry`

Условие ожидания `retry`

Неявно задано потоком управления в `atomic`-блоке

Retry. Эффективность

Примеры:

- `if (x() <= 10) retry`
- `if (x() == 0 && y() == 0 && z() == 0) retry`

Реализация:

- STM отслеживает, к каким `Refam` обращались
- Под капотом — блокирующиеся конструкции
- `retry: Nothing`

Alternatives

`orAtomic/atomic.oneOf`:

- Несколько путей поиска
- Первый заканчивается `retry` — откат и идём по второму и т. д.
- Учитывается только явный `retry` (а не неконсистентные чтения)

Alternatives. Пример

```
1  val msg = atomic { implicit txn =>
2    if (x() == 0)
3      retry
4    x -= 1
5    "took one from x"
6  } orAtomic { implicit txn =>
7    if (y() == 0)
8      retry
9    y -= 1
10   "took one from y"
11 } orAtomic { implicit txn =>
12   if (z() == 0)
13     retry
14   z -= 1
15   "took one from z"
16 }
```

Timeouts

Зачем timeout при `retry`:

- Error logging/handling
- No work — waiting thread shutdown
- Timeouts in spec of higher-level interface

Способы ограничения `retry`:

- Модифицированный `TxnExecutor` (`InterruptedException`)
- `retryFor()`

Timeouts. TxnExecutor

```
1  atomic.withRetryTimeout(1000) { implicit txn =>
2    // any retries in this atomic block will wait for at most
         1000 milliseconds
3  }
4
5  val myAtomic = atomic.withRetryTimeout(1, TimeUnit.SECONDS)
6  myAtomic { implicit txn =>
7    // this atomic block has a timeout of 1 seconds
8  }
9  myAtomic { ... }
10
11 TxnExecutor.transformDefault( _.withRetryTimeout(1000) )
12 atomic { implicit txn =>
13   // all atomic blocks now default to a 1 second timeout
14 }
```

Timeouts. retryFor(timeout)

```
1 val instance = atomic { implicit txn =>
2   if (!pool.hasAvailable) {
3     retryFor(100)
4     pool.grow()
5   }
6   pool.take()
7 }
```

Сигнатура

```
1 retryFor(...): Unit
```

```
1 retry: Nothing
```

Timeouts. Views

Можно блокироваться на `Ref.View`:

- `Ref.View.await(pred): Unit`
- `Ref.View.tryAwait(duration)(pred): Boolean`

Maps + Sets

- Консистентные итераторы у `TMap.View` и `TSet.View`
- Быстрые слепки за $O(1)$

Consistent iteration

- `TMap.View` extends `mutable.MapLike`
- `get()/put()` outside `atomic` should be atomic
- `TMap.View.iterator/TSet.View.iterator` для атомарного слепка

Inconsistent iteration

```
1  val m = TMap("one" -> 1).single
2
3  (new Thread { override def run {
4    atomic { implicit txn =>
5      m -= "one"
6      m += ("ONE" -> 1)
7    }
8  } }).start
9
10 for ((k, v) <- m; if v == 1) println(k)
```

Manual snapshots

- `<TMap|TSet>[.View].snapshot()` возвращает `immutable.Map/immutable.Set`
- `<TMap|TSet>.clone()`

How does it work?

- Mutable hash tries from `Refs` with generation numbers that control copy-on-write
- N. G. Bronson, J. Casper, H. Chafi and K. Olukotun. *A Practical Concurrent Binary Search Tree*. 2010.
- N. G. Bronson, J. Casper, H. Chafi and K. Olukotun. *Transactional Predication: High-Performance Concurrent Sets and Maps for STM*. In PODC'10: Proceedings of the 29th Annual ACM Conference on Principles of Distributed Computing, 2010.
- N. G. Bronson. *Composable Operations on High-Performance Concurrent Collections*. Ph.D. Dissertation, Stanford University, 2011.

Exceptions

- `Exception` $\Rightarrow$ `rollback` + rethrow
- `scala.util.control.ControlThrowable` $\Rightarrow$ `commit` + rethrow

Exceptions and nesting

```
1  val last = Ref("none")
2  atomic { implicit txn =>
3    last() = "outer"
4    try {
5      atomic { implicit txn =>
6        last() = "inner"
7        throw new RuntimeException
8      }
9    } catch {
10     case _: RuntimeException =>
11   }
12 }
```

Benchmarking

TL;DR

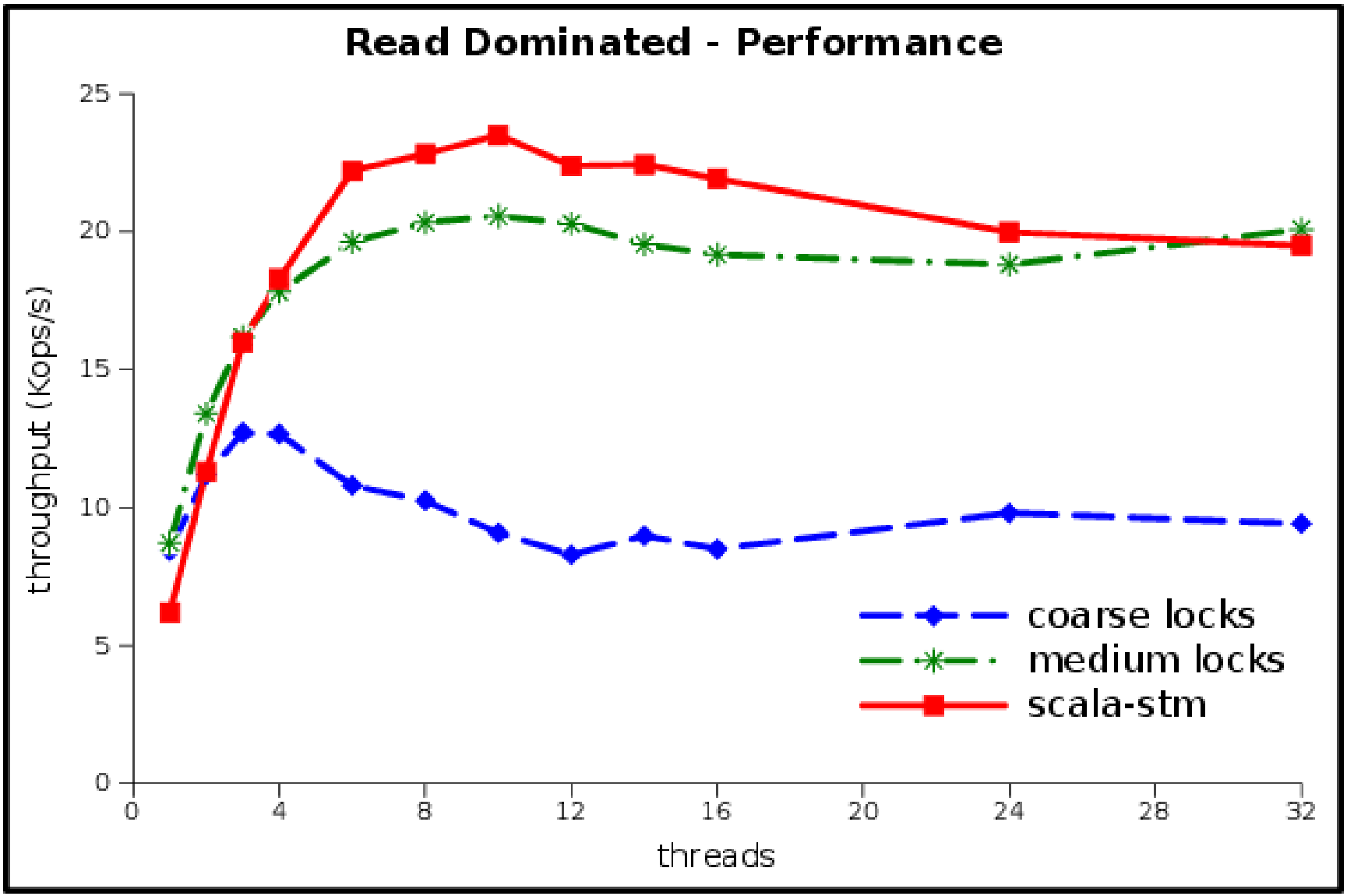
- Немного медленнее в однопоточном режиме по сравнению с хитрыми блокировками
- Лучше масштабируется по ядрам/нитям

STMBench7

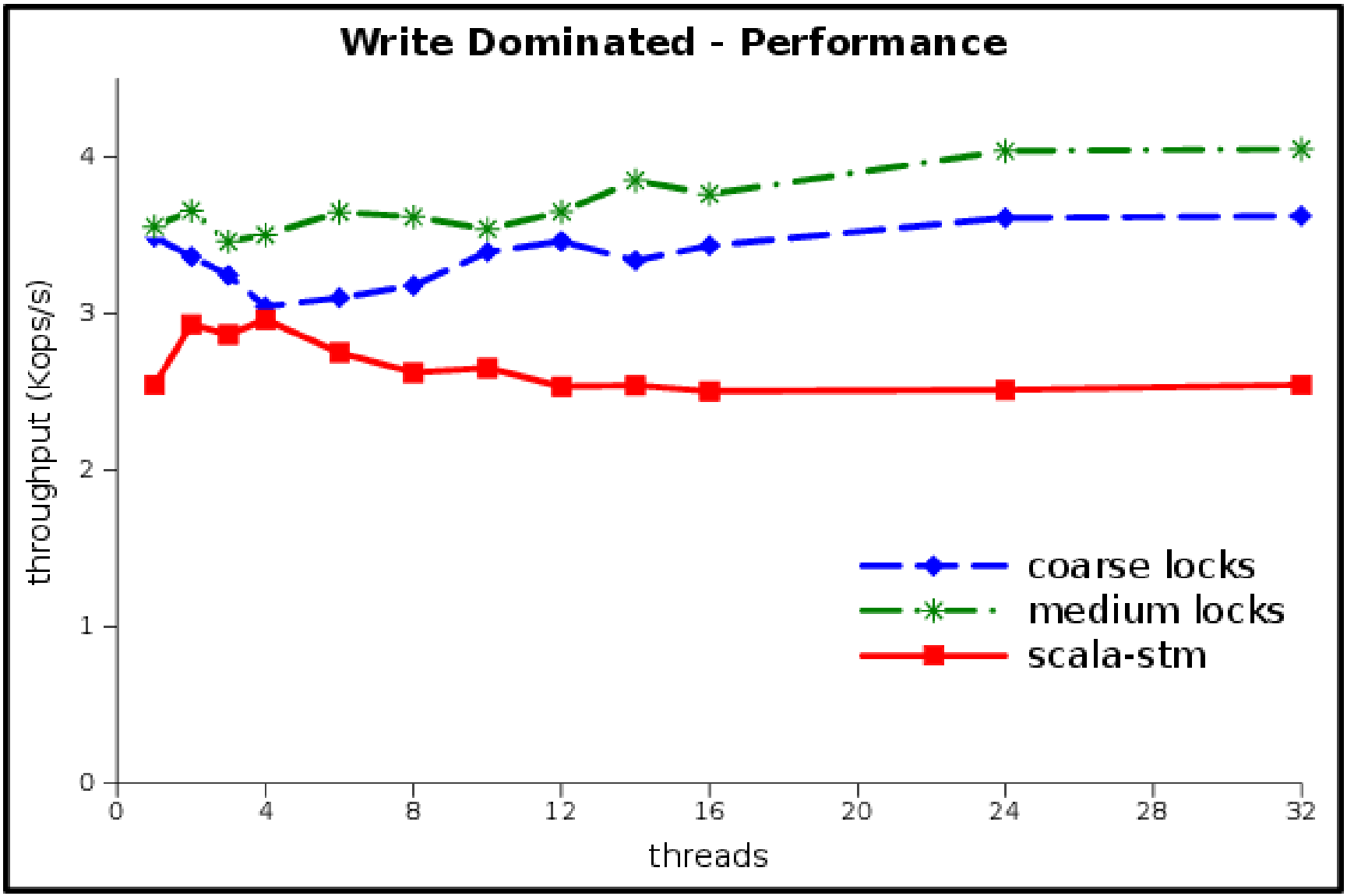
- R. Guerraoui, M. Kapalka and J. Vitek. *STMBench7: A Benchmark for Software Transactional Memory*. 2007.
- A. Dragojevic, R. Guerraoui and M. Kapalka. *Dividing Transactional Memories by Zero*. 2008.
- Comparison to coarse- and medium-grained locking
- See the details[4]

[4] `http://nbronson.github.io/scala-stm/benchmark.html`

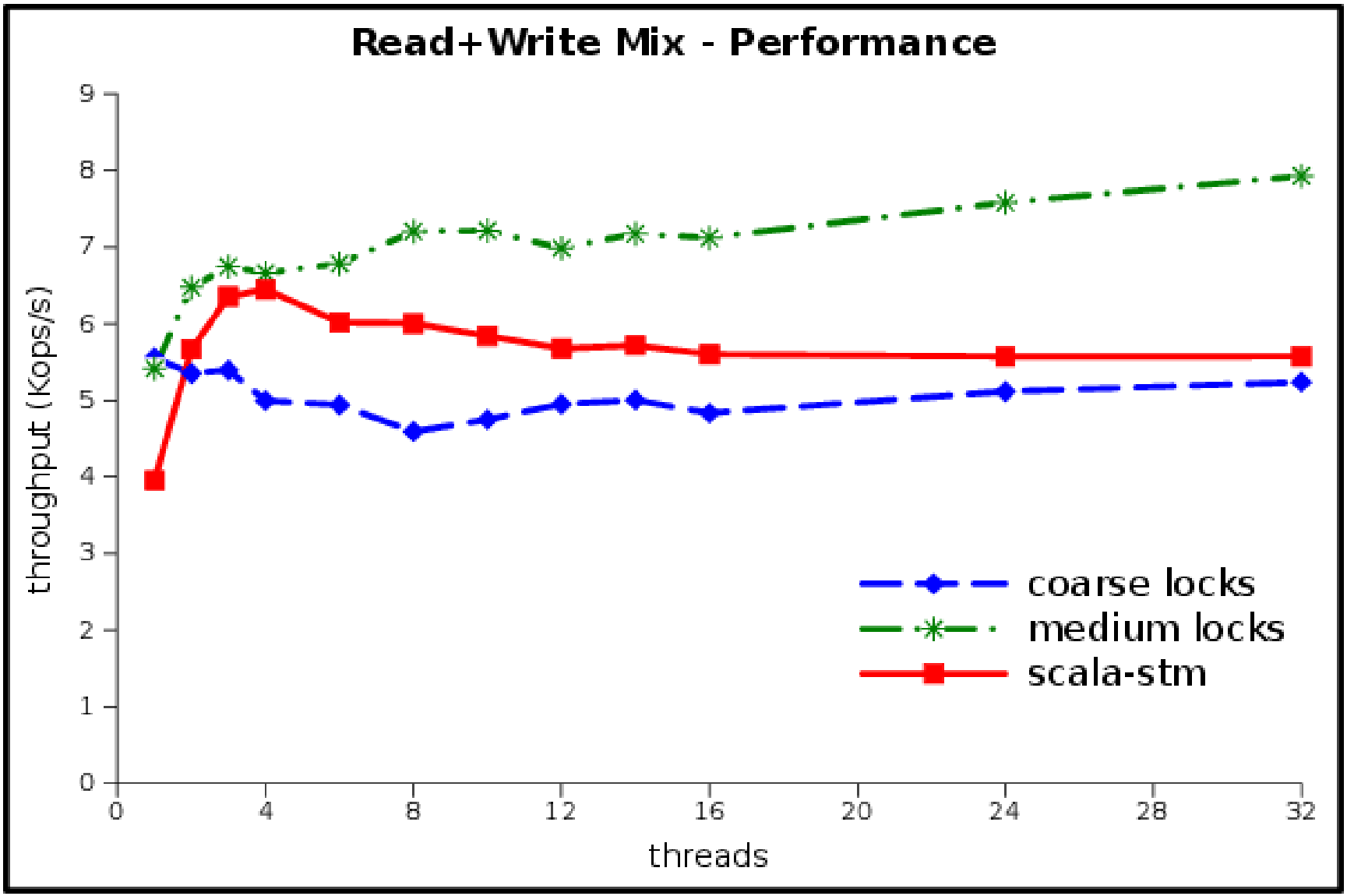
Read Dominated



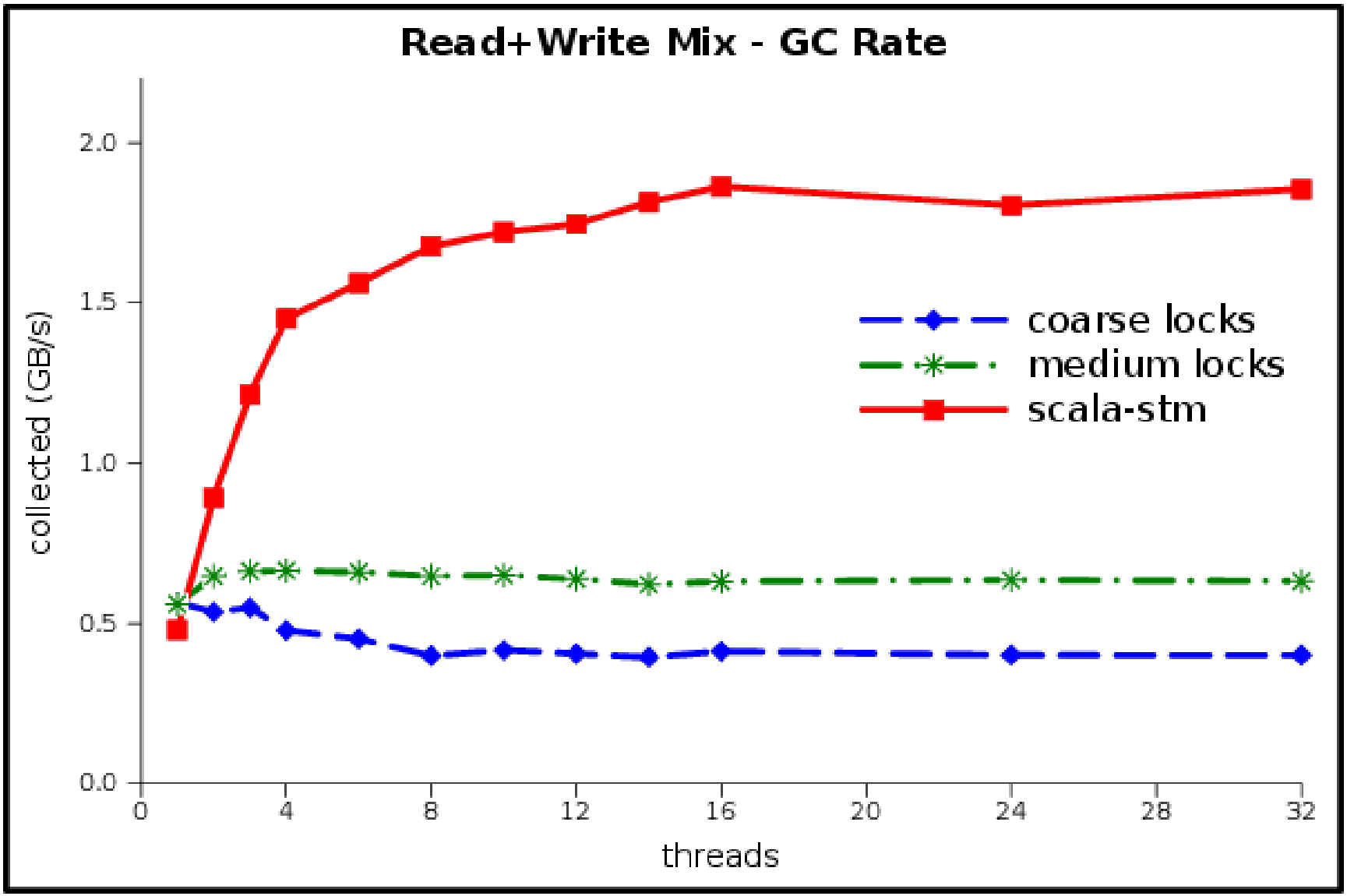
Write Dominated



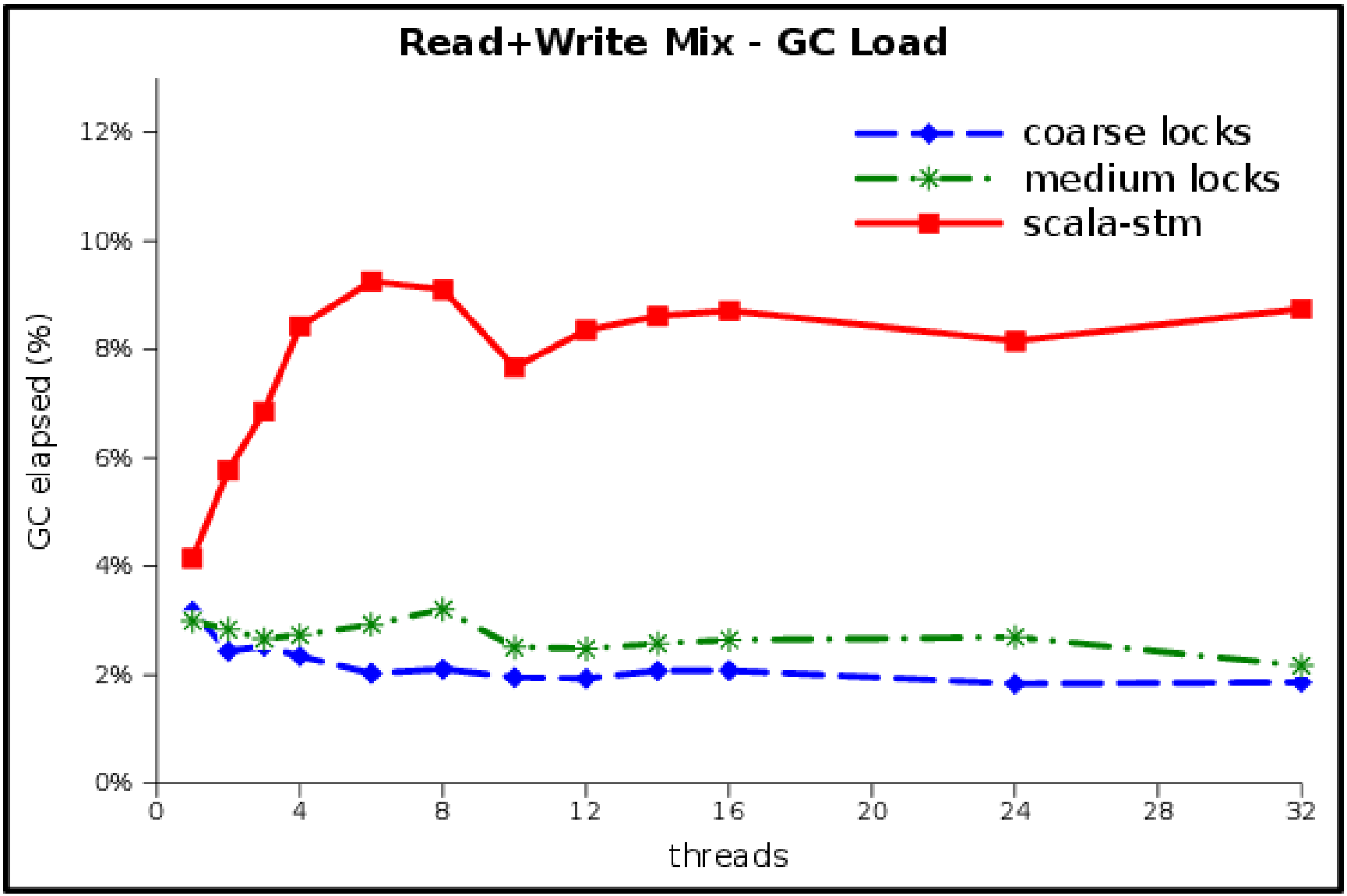
Read + Write Mix



Read + Write Mix GC Rate



Read + Write Mix GC Load



Bibliography

- F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. *Dynamic Optimization for Efficient Strong Atomicity*. 2008.
- N. G. Bronson, C. Kozyrakis and K. Olukotun. *Feedback-Directed Barrier Optimization in a Strongly Isolated STM*. 2009
- V. Menon, S. Balensieger, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha and A. Welc. *Practical Weak-Atomicity Semantics for Java STM*. 2008.
- K. F. Moore and D. Grossman. *High-Level Small-Step Operational Semantics for transactions*. 2008.

Bibliography continued

- T. Harris, S. Marlow, S. Peyton-Jones and M. Herlihy. *Composable Memory Transactions*. 2005.
- R. Hickey. *The Clojure Programming Language*. 2008
- D. Dice, O. Shalev and N. Shavit. *Transactional Locking II*. 2006.
- T. Riegel, P. Felber and C. Fetzer. *A Lazy Snapshot Algorithm with Eager Validation*. 2006.
- R. Guerraoui and M. Kapalka. *On the Correctness of Transactional Memory*. 2008.
- A. Dragojevic, R. Guerraoui and M. Kapalka. *Stretching Transactional Memory*. 2009.

Куда двигаться дальше

- Herb Sutter. *The Free Lunch Is Over*. 2009.
- Jonas Bonér. State: You're Doing It Wrong — Alternative Concurrency Paradigms For The JVM[5]. JavaOne 2009.
- ScalaDays 2013. Concurrency — The good, the bad, the ugly[6]
- Chris Okasaki. *Purely Functional Data Structures*. 1999
- JCIP 2nd edition + JMM

---

[5] http://www.slideshare.net/jboner/state-youre-doing-it-wrong-javaone-2009

[6] http://www.parleys.com/play/51c0bc58e4b0ed877035680a/

Получили зачёт

- Ершов В.
- Суворов Е.
- Фёдоров К.

Могут получить зачёт

- Бакрадзе Л.
- Грязнов С.
- Егоров Д.
- Королев Д.
- Хомутов В.
- Шашкова Е.

Last Feature Requests

- **2013-11-18** FR8: Redis Hashes[7]
- **2013-11-25** FR9: Compression
- **2013-12-02** FR10: Multithreaded Node + Atomic Batches
- **2013-12-09** FR11: Server Side Filtering and Processing

[7] `http://redis.io/commands#hash`

Вопросы?

- http://incubos.org/contacts/
- Общие вопросы — в Twitter: @incubos
- Вопросы по лекциям — в комментариях: http://incubos.org/blog/
- Частные вопросы — в почту vadim.tsesko@gmail.com