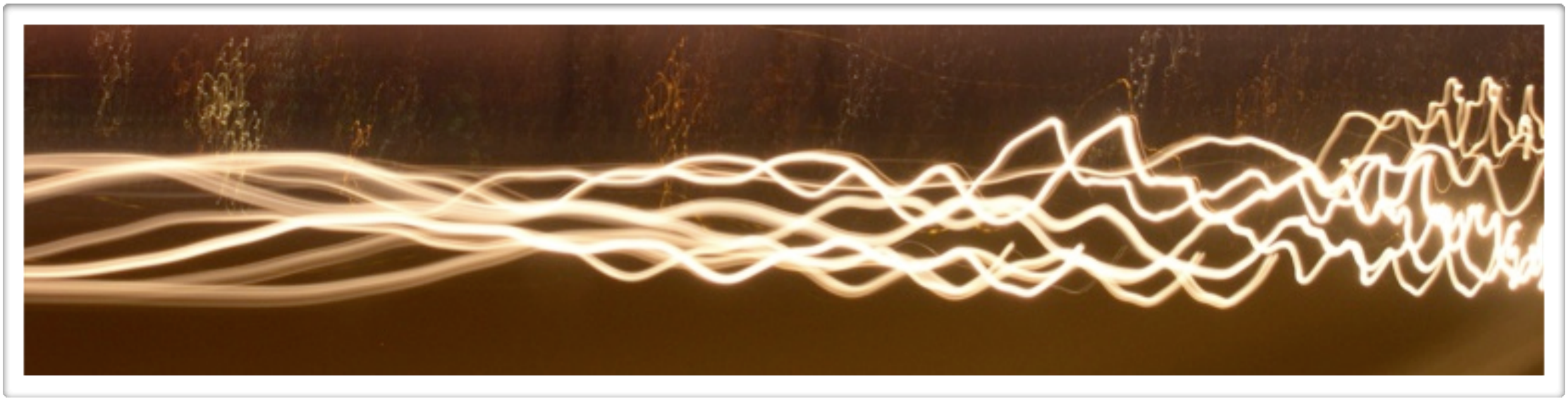


JVSTM
and its applications

João Cachopo @ Software Engineering Group

INSTITUTO SUPERIOR TÉCNICO

inesc id lisboa 10 anos

JAVA VERSIONED SOFTWARE TRANSACTIONAL MEMORY

First developed in 2005

First developed in 2005

(web page not update since 2006)

Outline

- Original design of the JVSTM
- Recent changes to the JVSTM
- What we have been doing with it

First **multi-version** STM

Designed for **very large** transactions
and **high read/write ratio**

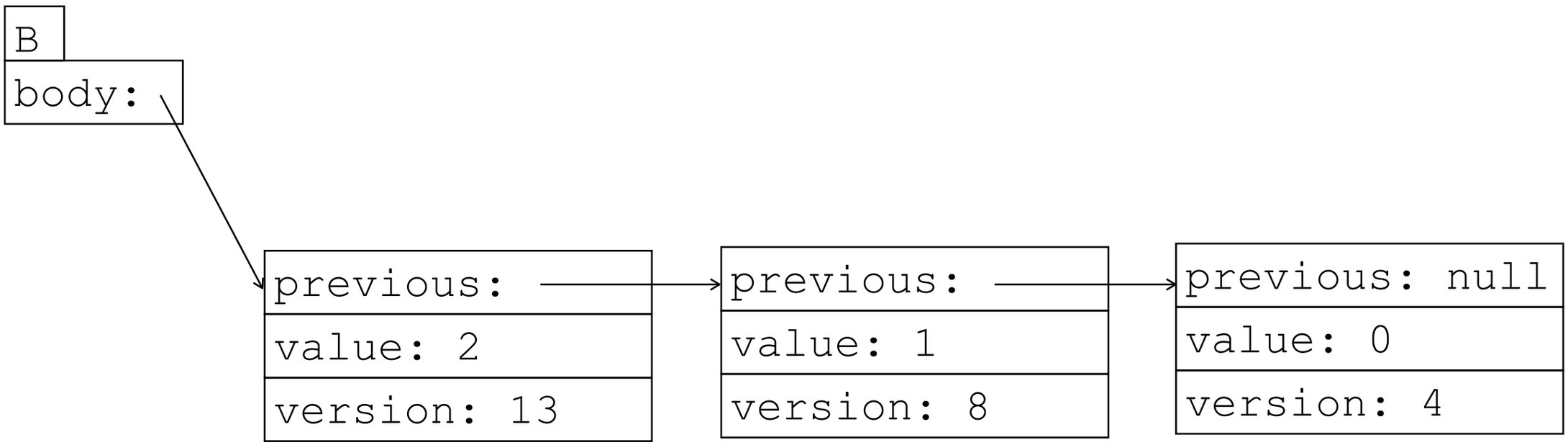
The goal:

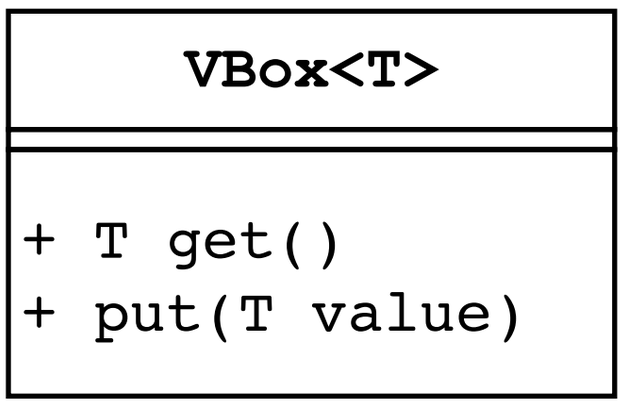
ease of programming

(without incurring into too much overhead)

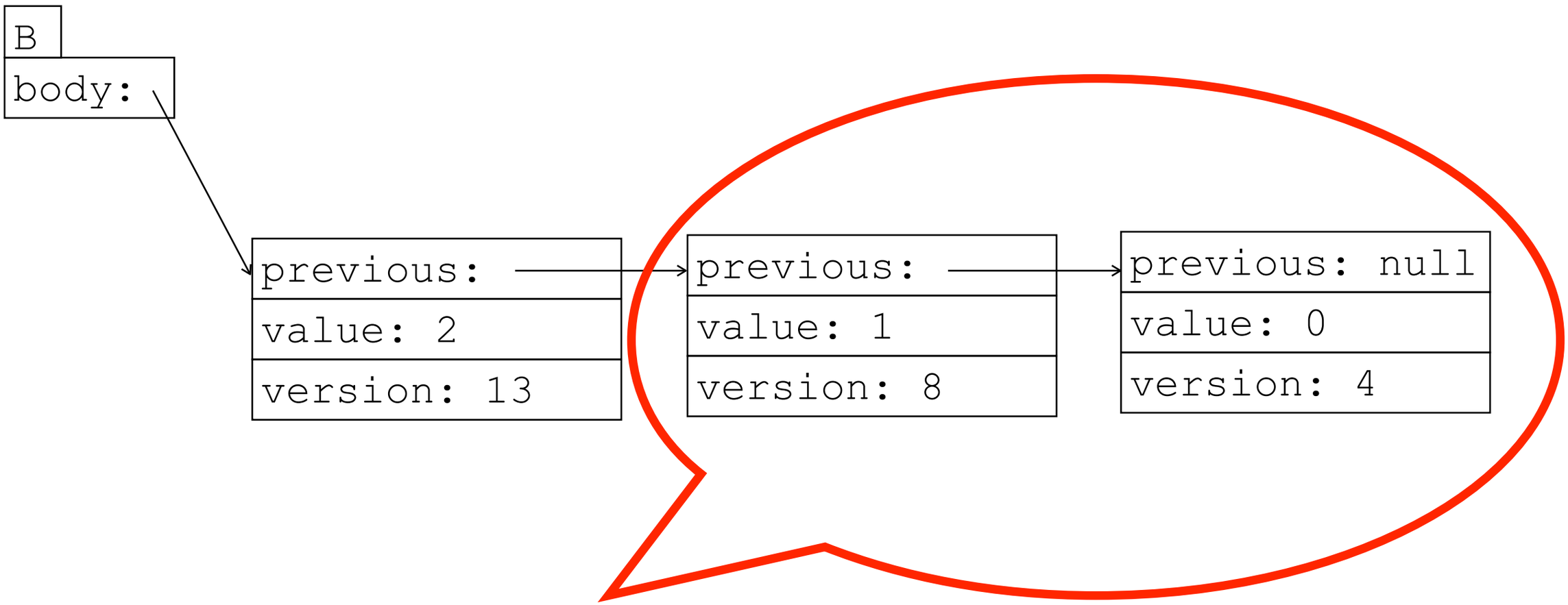
It is **not** a transparent STM

Versioned Boxes



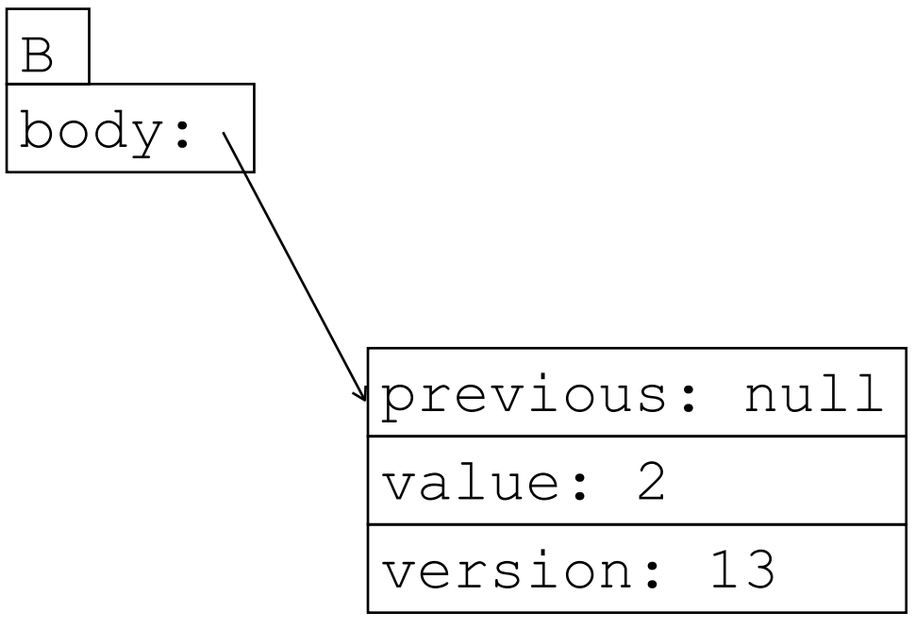


Versioned Boxes

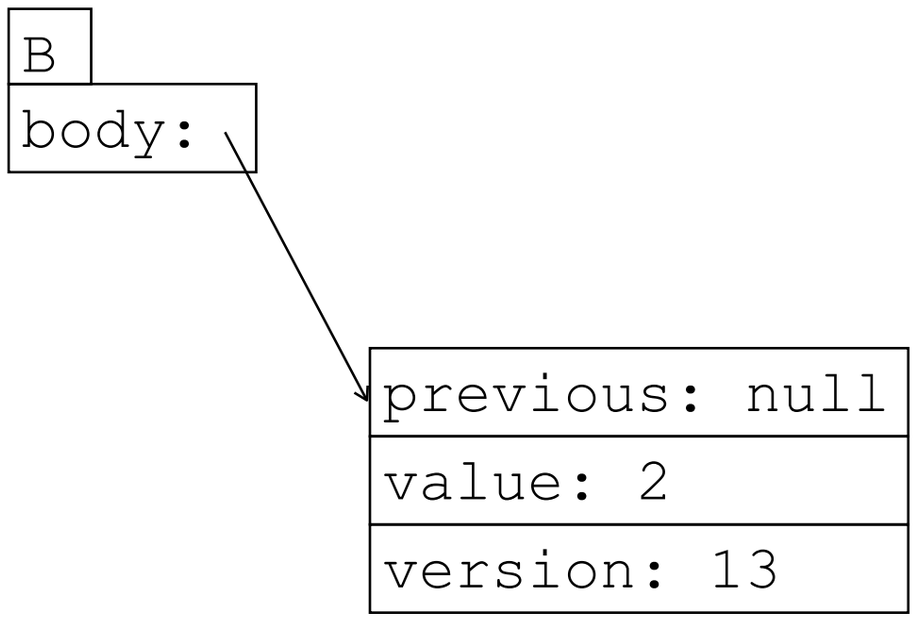


GCed when
no longer needed

Versioned Boxes



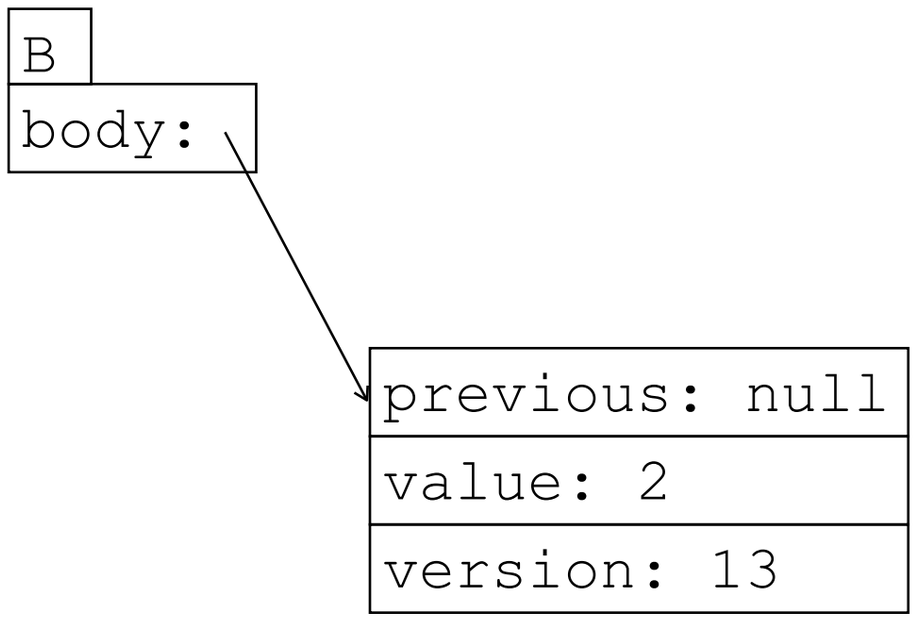
Versioned Boxes



Reading a box:

- follow body
- compare version
- return value

Versioned Boxes



Reading a box:

- follow body
- compare version
- return value

- wait-free
- no sync actions
- no allocation

No inconsistent reads

(JVSTM ensures **opacity**)

Read-only txs

- Begin Tx (lock-free)
- Read boxes (wait-free)
- Commit / abort Tx (lock-free)

Read-only txs

- Begin Tx (lock-free)
- Read boxes (wait-free)
- Commit / abort Tx (lock-free)

Entire TX lock-free

Read-only txs

- Begin Tx (lock-free)
- Read boxes (wait-free)
- Commit / abort Tx (lock-free)

Because of GC

Entire TX lock-free

Read-only transactions **never** conflict

(MV-permissiveness)

Read-write txs

- Begin Tx (lock-free)
- Read / write boxes (wait-free)
- Abort Tx (lock-free)
- Commit Tx (**global** lock)

Lock-based commit

```
commit() {
  GLOBAL_LOCK.lock();
  try {

  } finally {
    GLOBAL_LOCK.unlock();
  }
}
```

Lock-based commit

```
commit() {
  GLOBAL_LOCK.lock();
  try {
    if (validate()) {

    }
  } finally {
    GLOBAL_LOCK.unlock();
  }
}
```

Lock-based commit

```
commit() {
  GLOBAL_LOCK.lock();
  try {
    if (validate()) {
      int newTxNumber = globalCounter + 1;
      writeBack(newTxNumber);
      globalCounter = newTxNumber;
    }
  } finally {
    GLOBAL_LOCK.unlock();
  }
}
```

Very simple design...

Works **really** well **in practice**

Lock-based commit

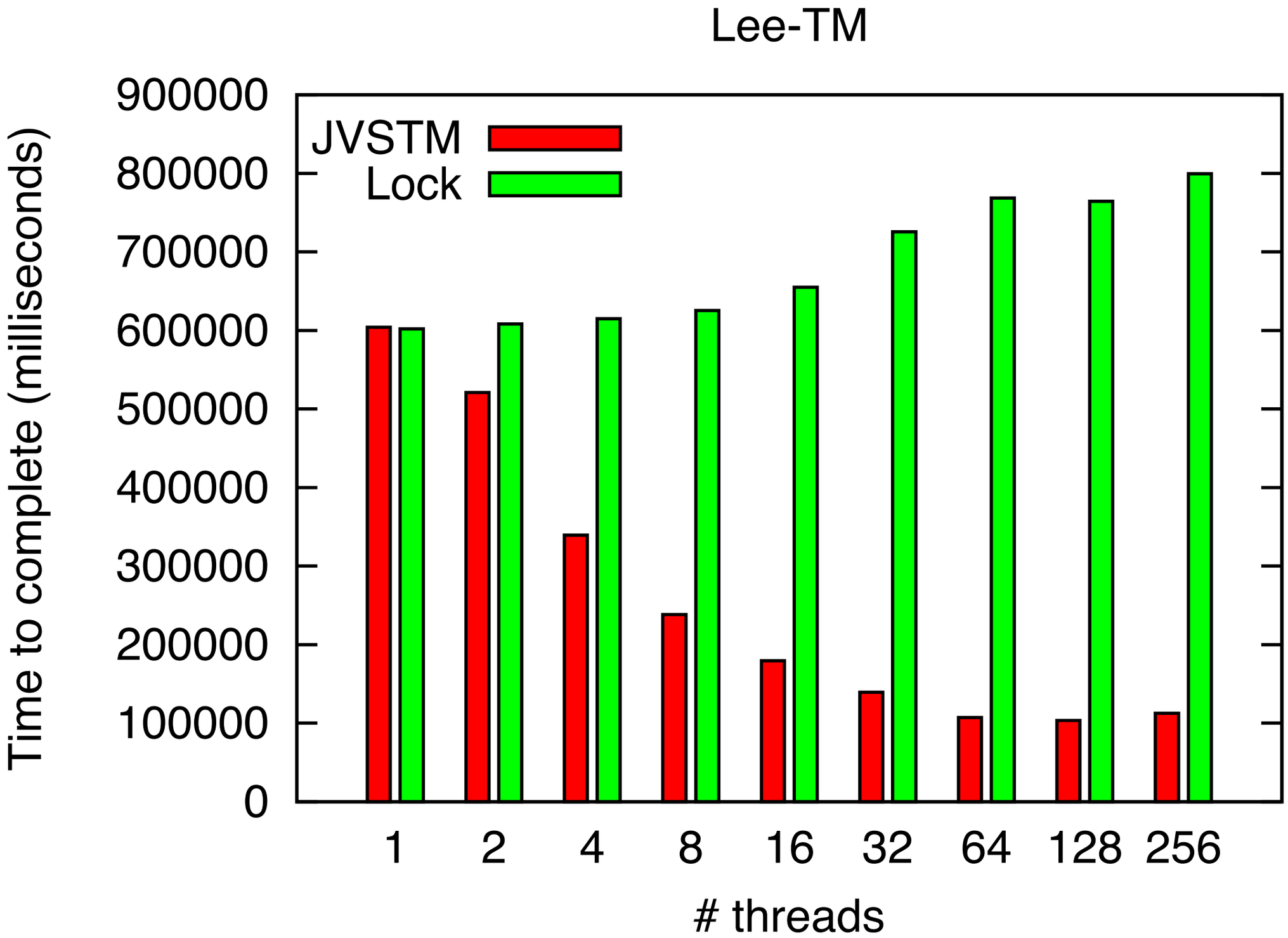
```
commit() {
  GLOBAL_LOCK.lock();
  try {
```

If uncontended, this is quite fast!

```
  } finally {
    GLOBAL_LOCK.unlock();
  }
}
```

Good when **few** read-write txs

or txs **long enough** so that commits do not overlap often



(machine with 196 physical cores)

Lock-based commit

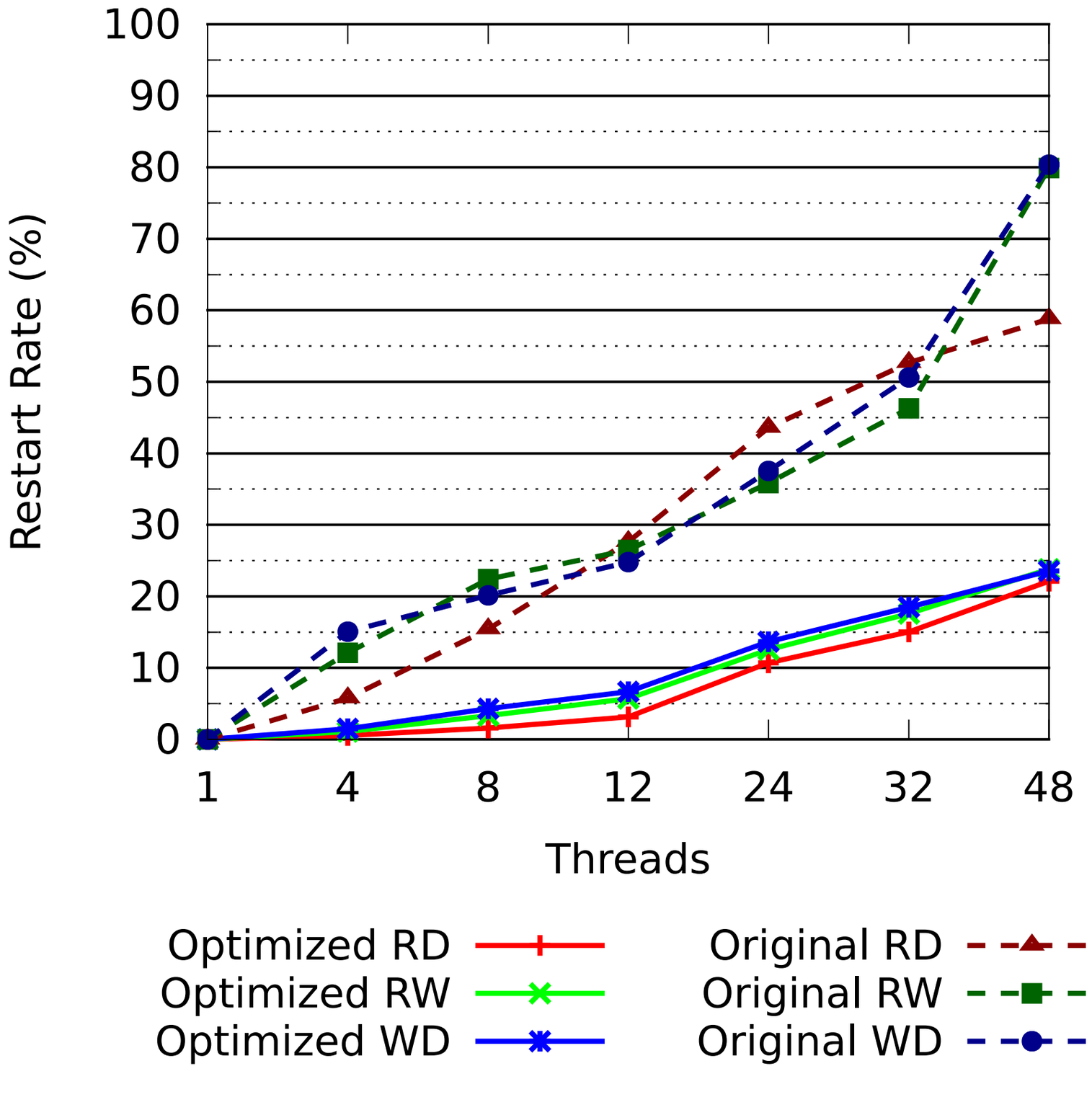
```
commit() {
  GLOBAL_LOCK.lock();
  try {
    if (validate()) {
```

We may do other things here...

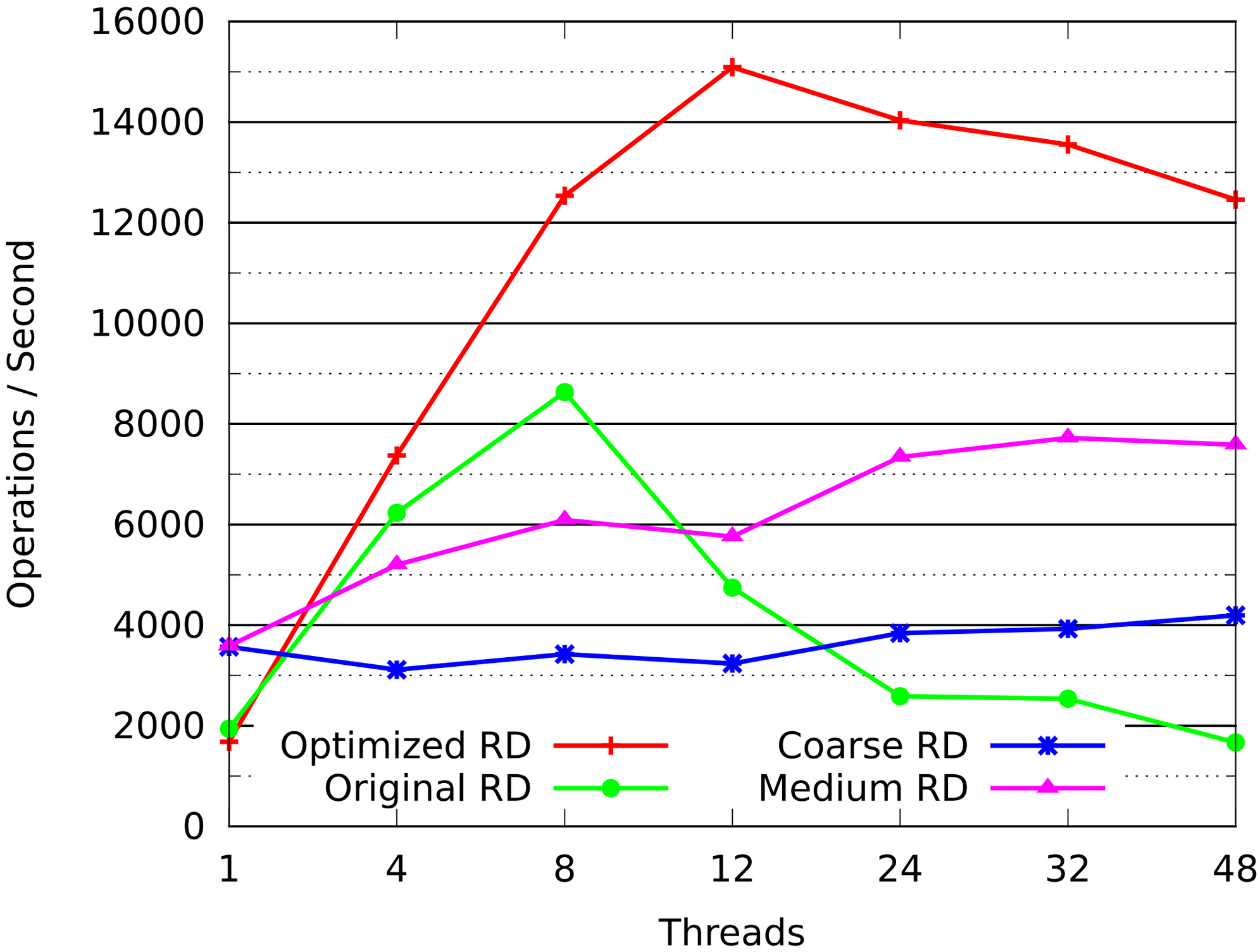
```
    }
  } finally {
    GLOBAL_LOCK.unlock();
  }
}
```

Delay computations until commit time, if they cause many conflicts

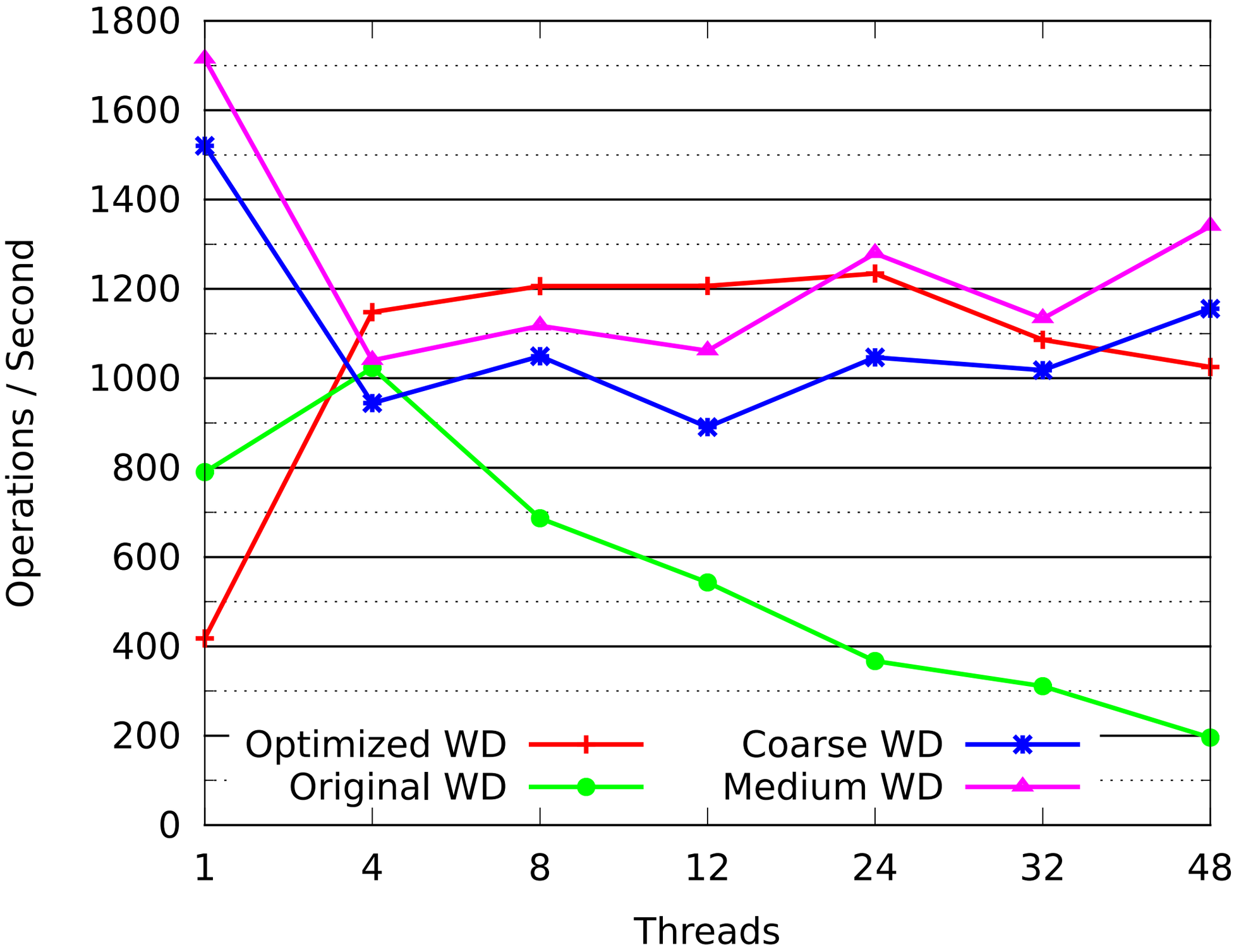
STMBench7


Restart Rate (%)
100
90
80
70
60
50
40
30
20
10
0
1
4
8
12
24
32
48
Threads
Optimized RD
Optimized RW
Optimized WD
Original RD
Original RW
Original WD

STMBench7


16000
14000
12000
10000
8000
6000
4000
2000
0
Operations / Second
1
4
8
12
24
32
48
Threads
Optimized RD
Original RD
Coarse RD
Medium RD

STMBench7


1800
1600
1400
1200
1000
800
600
400
200
0
Operations / Second
1
4
8
12
24
32
48
Threads
Optimized WD
Original WD
Coarse WD
Medium WD

What about **small** transactions?

Read-write txs

- Begin Tx (lock-free)
- Read / write boxes (wait-free)
- Abort Tx (lock-free)
- Commit Tx (**global** lock)

Read-write txs

- Begin Tx (lock-free)
- Read / write boxes (wait-free)
- Abort Tx (lock-free)
- Commit Tx (**global** lock)

Can we make it better?

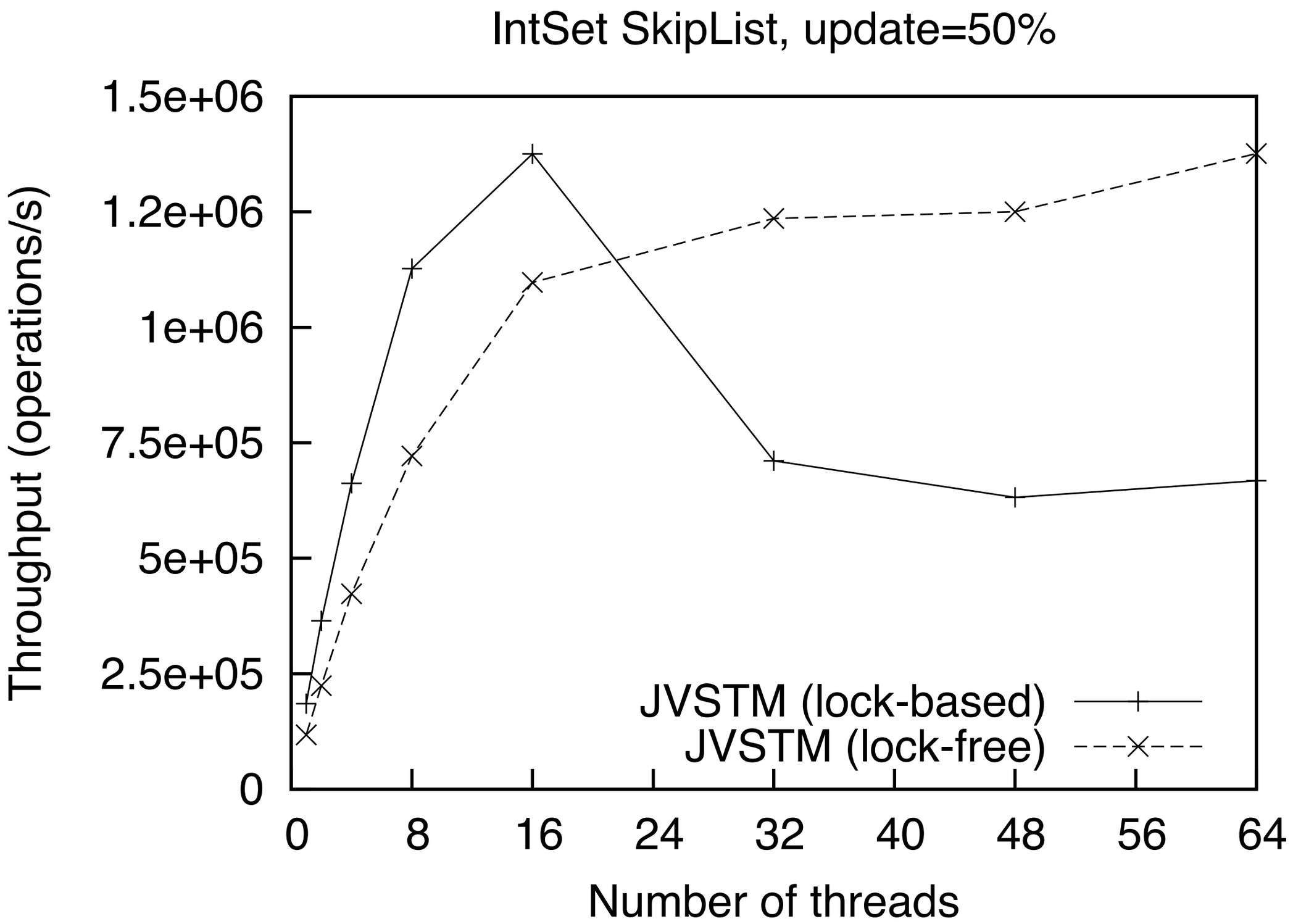
Read-write txs

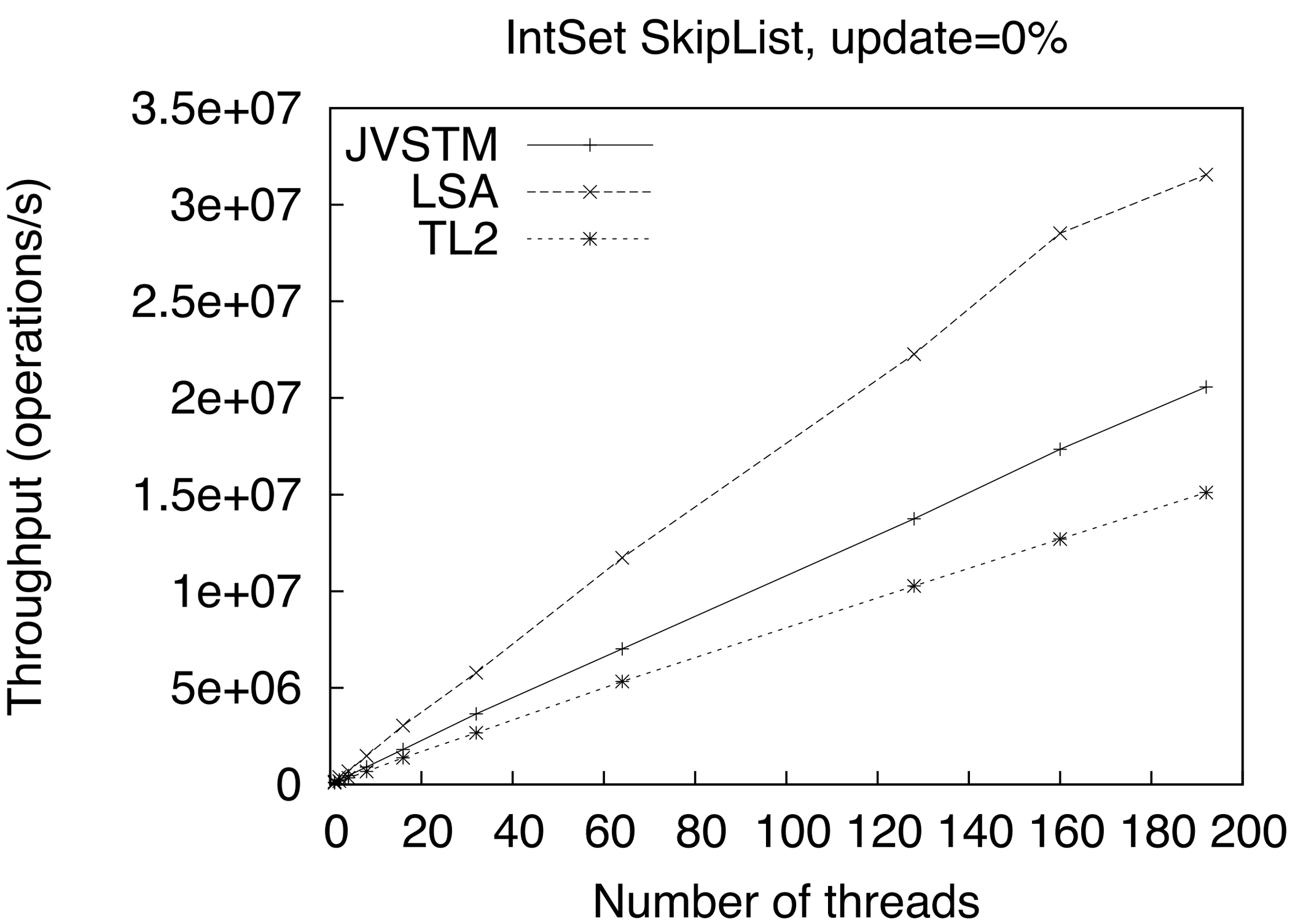
- Begin Tx (lock-free)
- Read / write boxes (wait-free)
- Abort Tx (**wait-free**)
- Commit Tx (**lock-free**)

JVSTM completely **lock-free**

Lock-free isn't that hard...

Efficient **and** lock-free is!


IntSet SkipList, update=50%
Throughput (operations/s)
1.5e+06
1.2e+06
1e+06
7.5e+05
5e+05
2.5e+05
0
0
8
16
24
32
40
48
56
64
Number of threads
JVSTM (lock-based)
JVSTM (lock-free)


IntSet SkipList, update=0%
JVSTM
LSA
TL2
Throughput (operations/s)
3.5e+07
3e+07
2.5e+07
2e+07
1.5e+07
1e+07
5e+06
0
0
20
40
60
80
100
120
140
160
180
200
Number of threads

JVSTM still not optimized for **tiny** transactions

We're working on it

- Eliminate memory allocation
- Stealth transactions
- Make beginTx wait-free
- Optimize fast path of read/write
- Versioned arrays
- ...

JVSTM applications?

More recently

- **STM-based** Thread Level Speculation
- Dynamic Software Updates
- Automatic memoization of functions with side-effects

This led to

- Transactional
  - collections (queues, maps, sets, etc)
  - I/O streams
- Inevitable transactions
- Exploring failure atomicity
- ...

But it all started with the

FénixEDU system

(> 1 million LOCs)

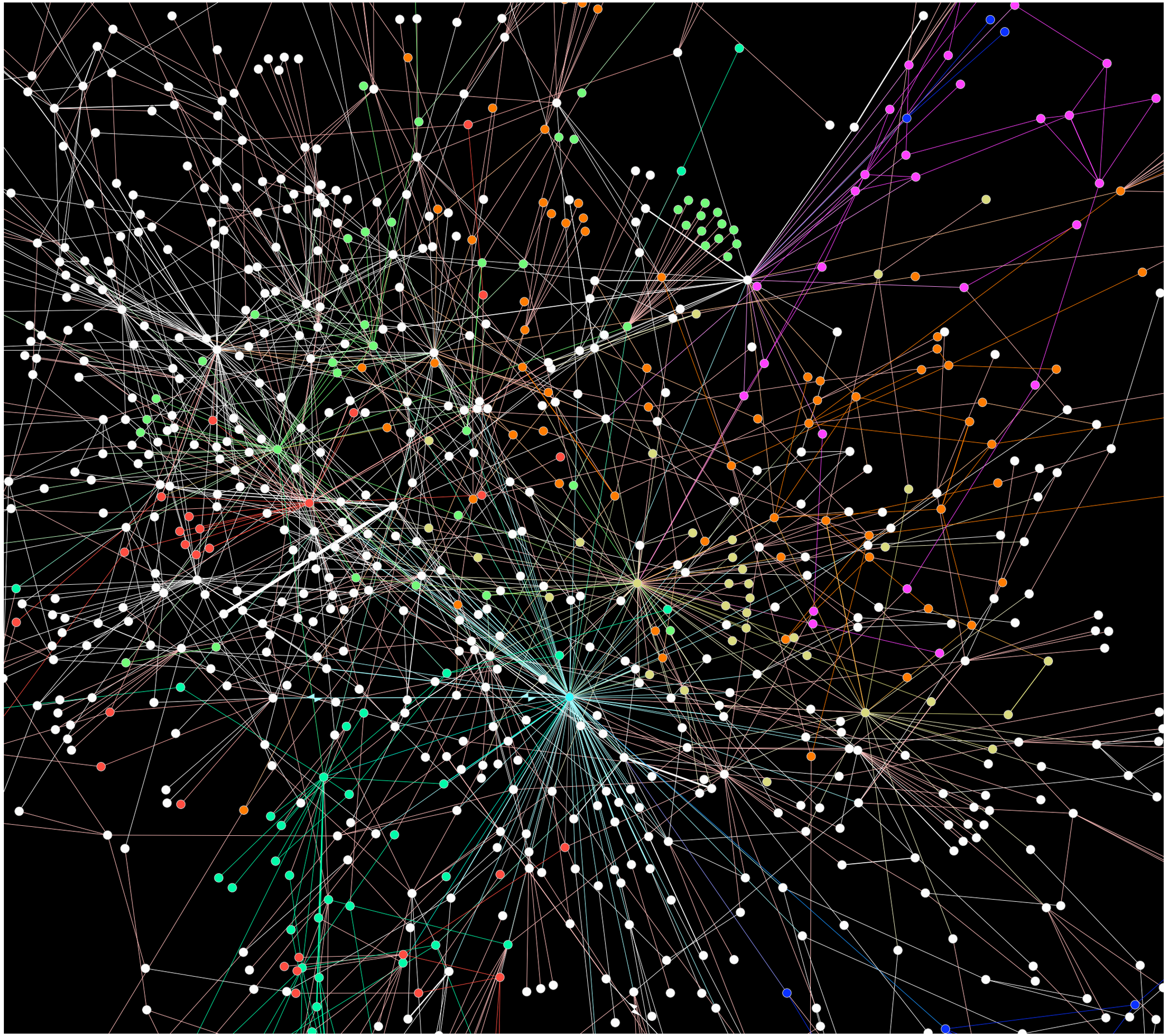
First **real-world application** of STMs

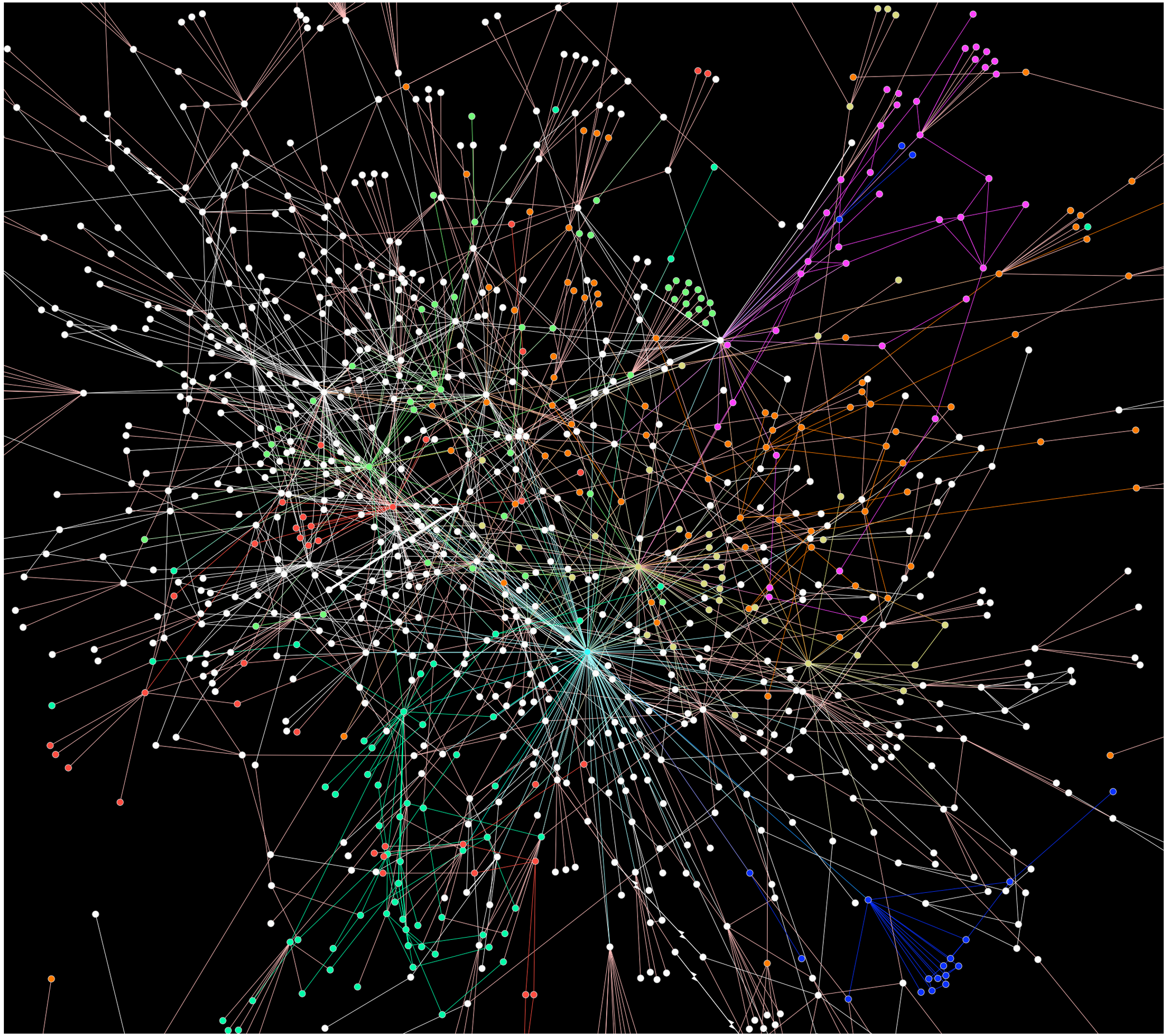
The Fénix Framework

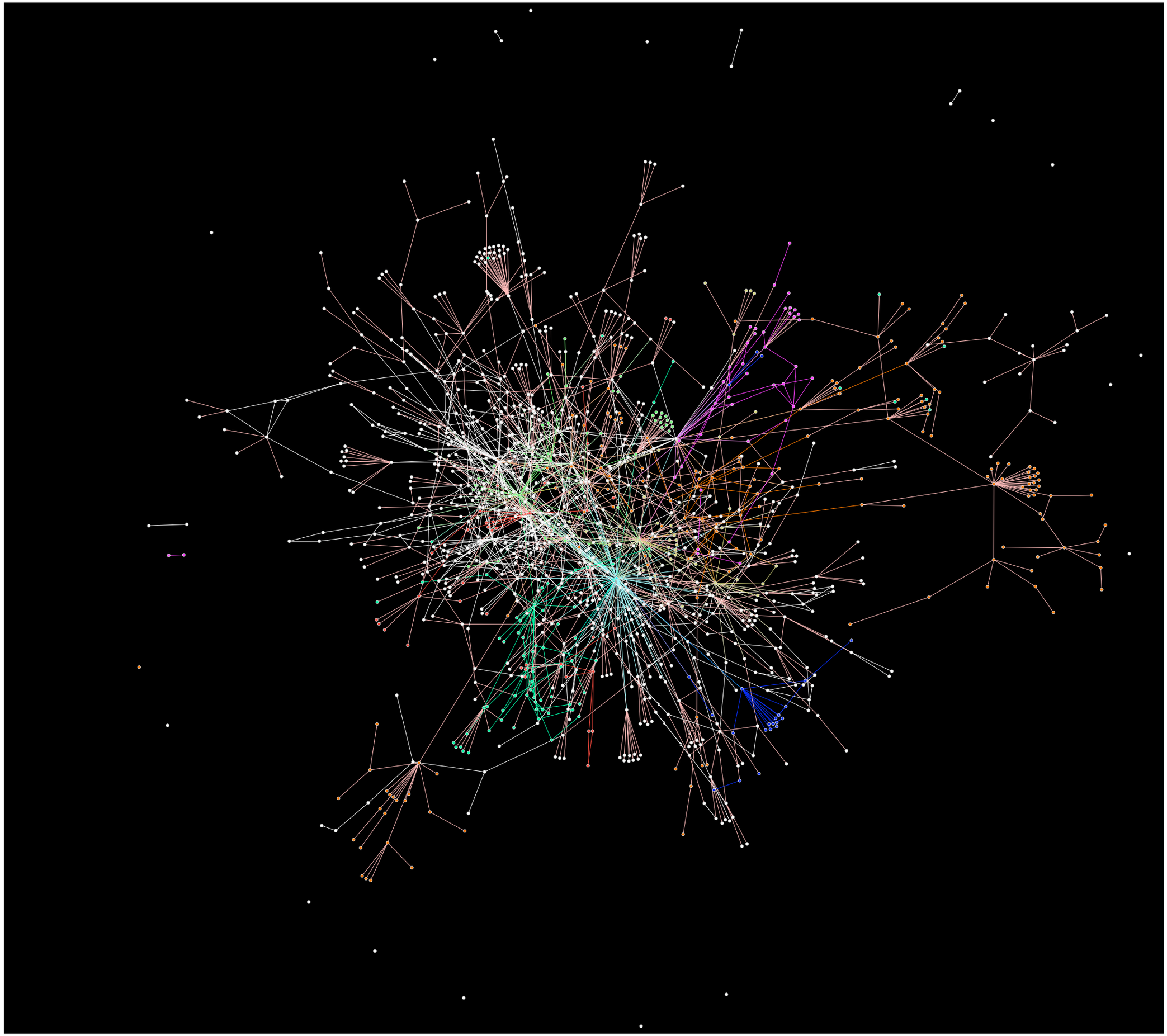
Simplify the development of applications with a **transactional** and **persistent rich domain model**

Simplify the development of applications with a **transactional** and **persistent rich domain model**

(AKA enterprise applications)



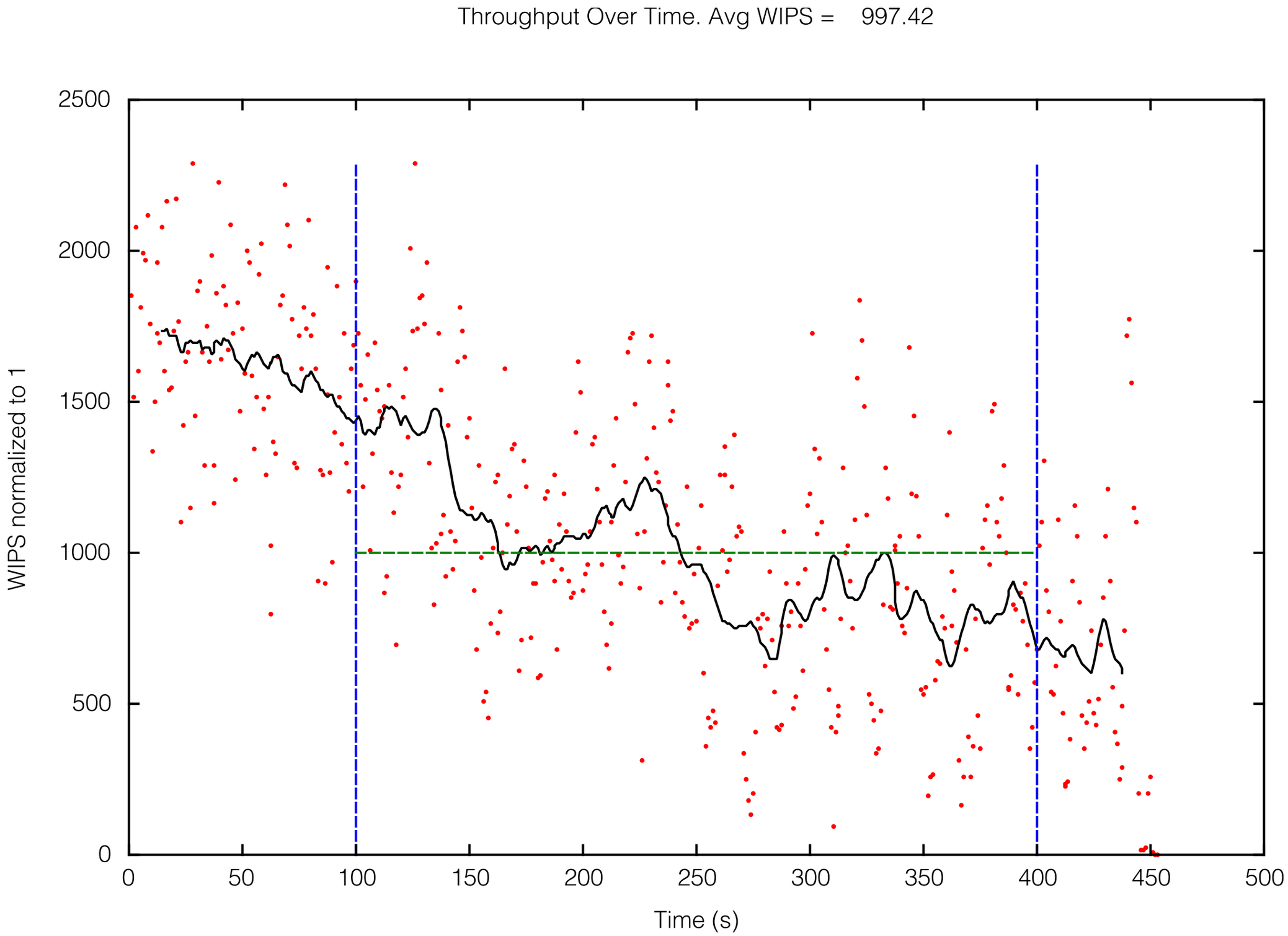




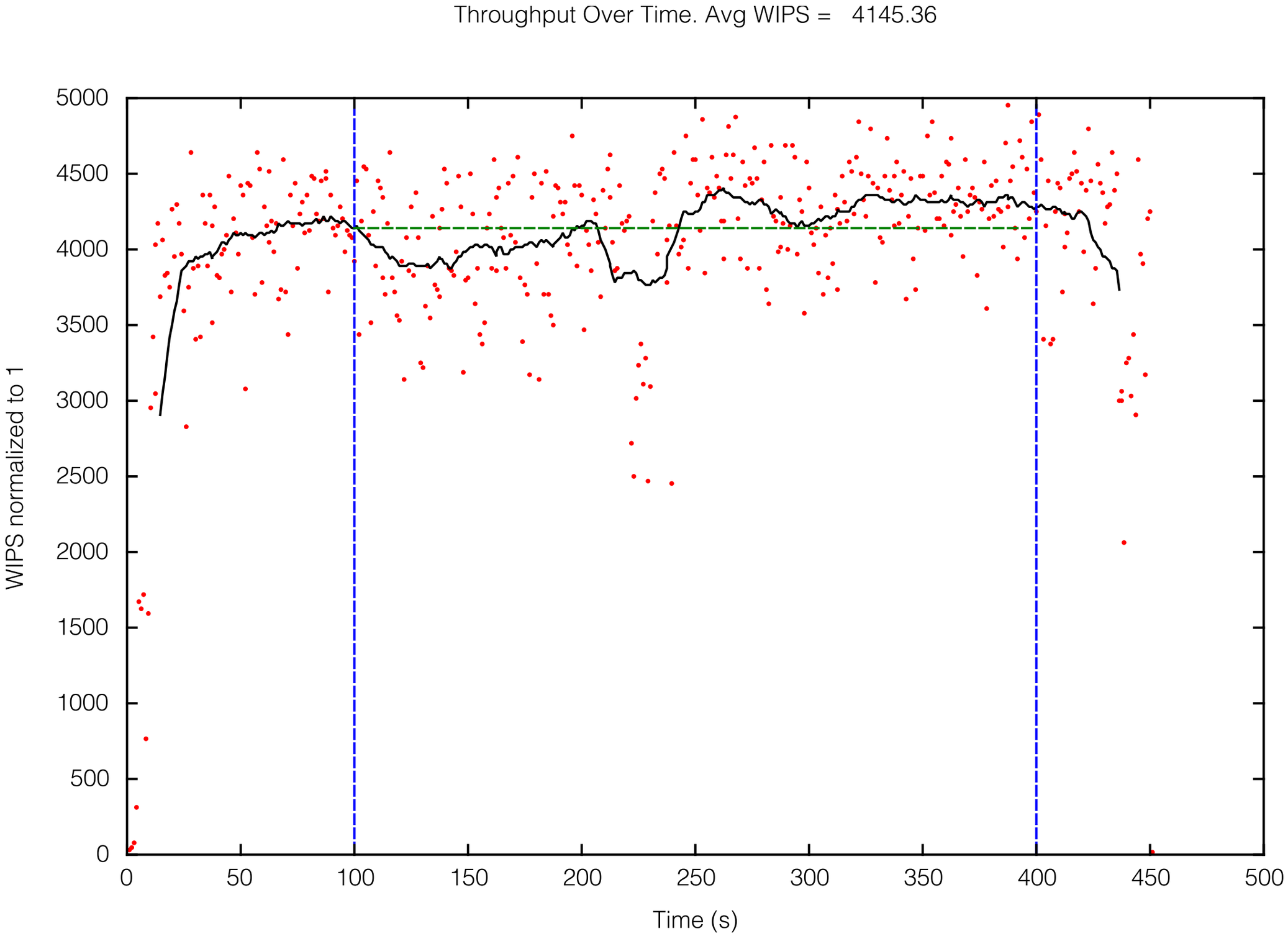
Simpler programming model

Better performance*

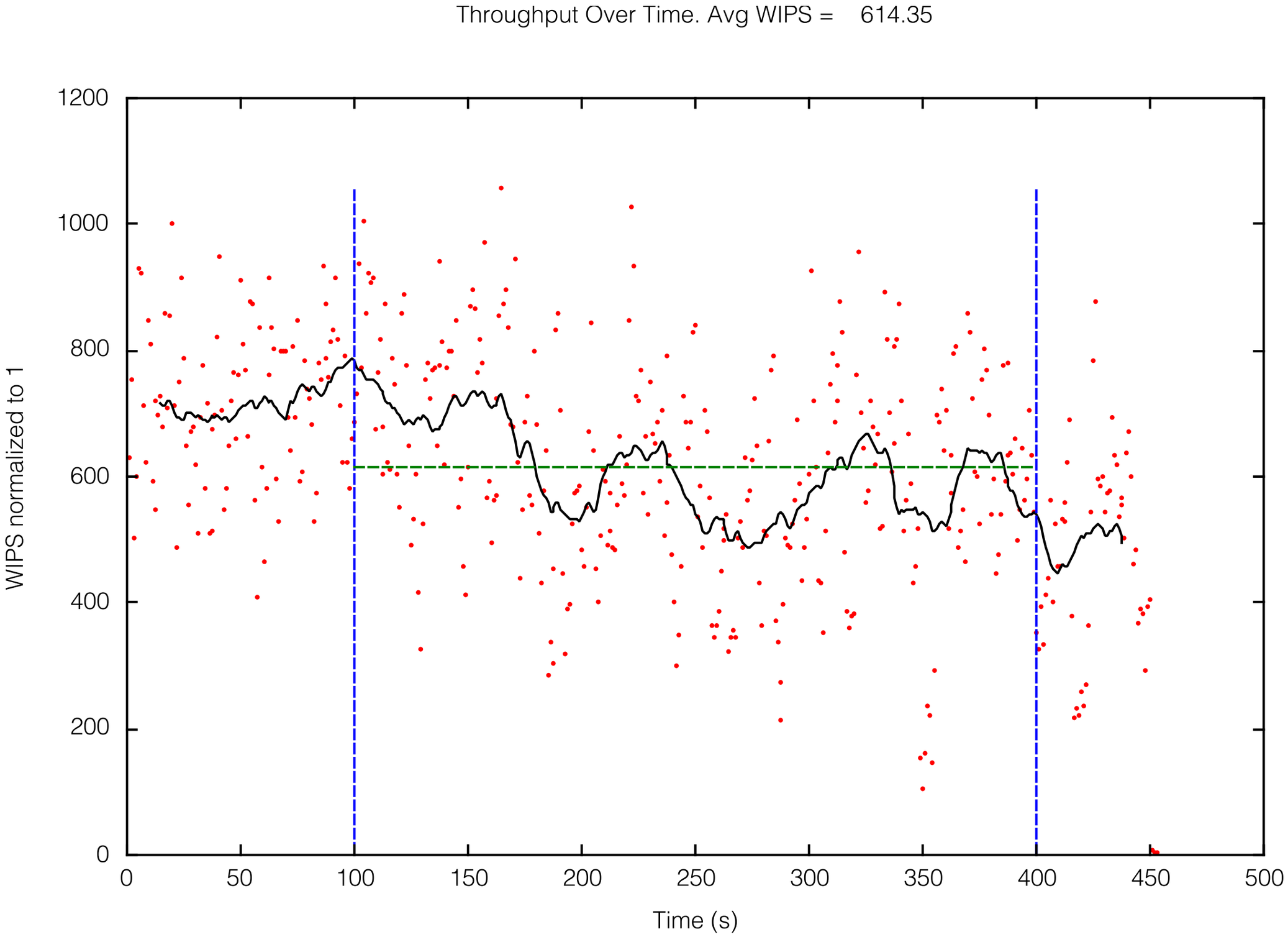
Explores multicores



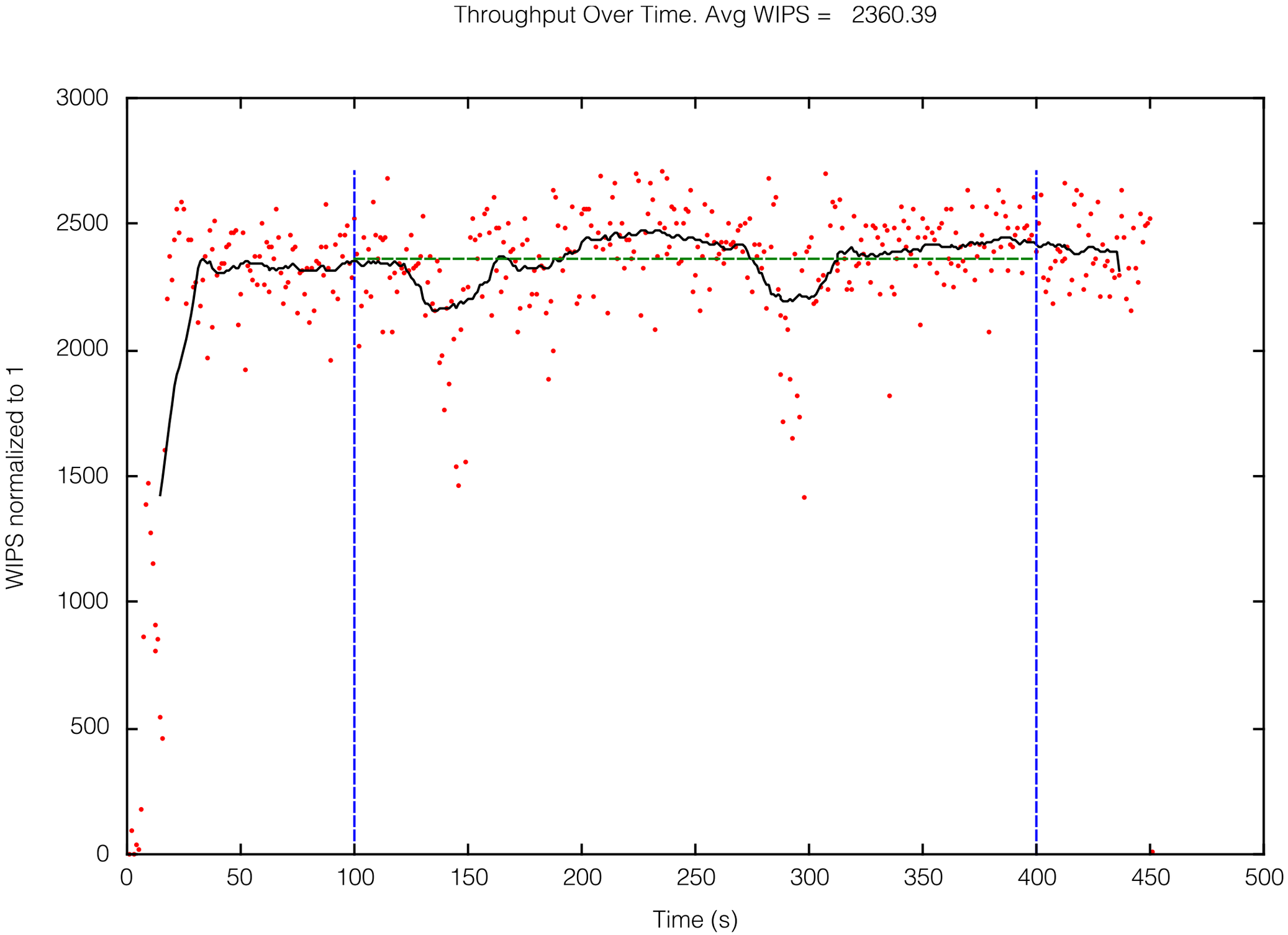
Original JDBC-based (1000 items)



FF-based (1000 items)

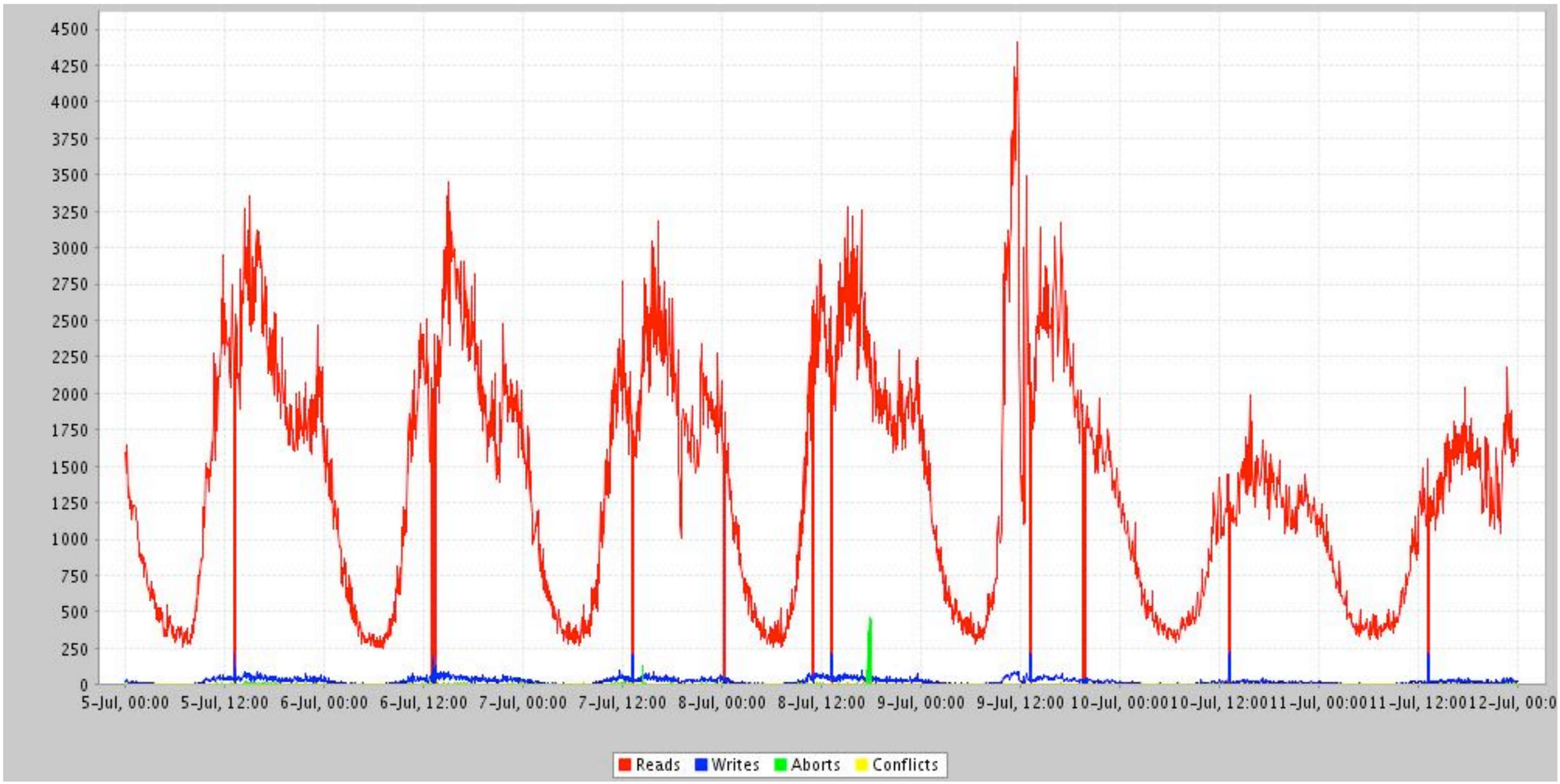


Original JDBC-based (10000 items)

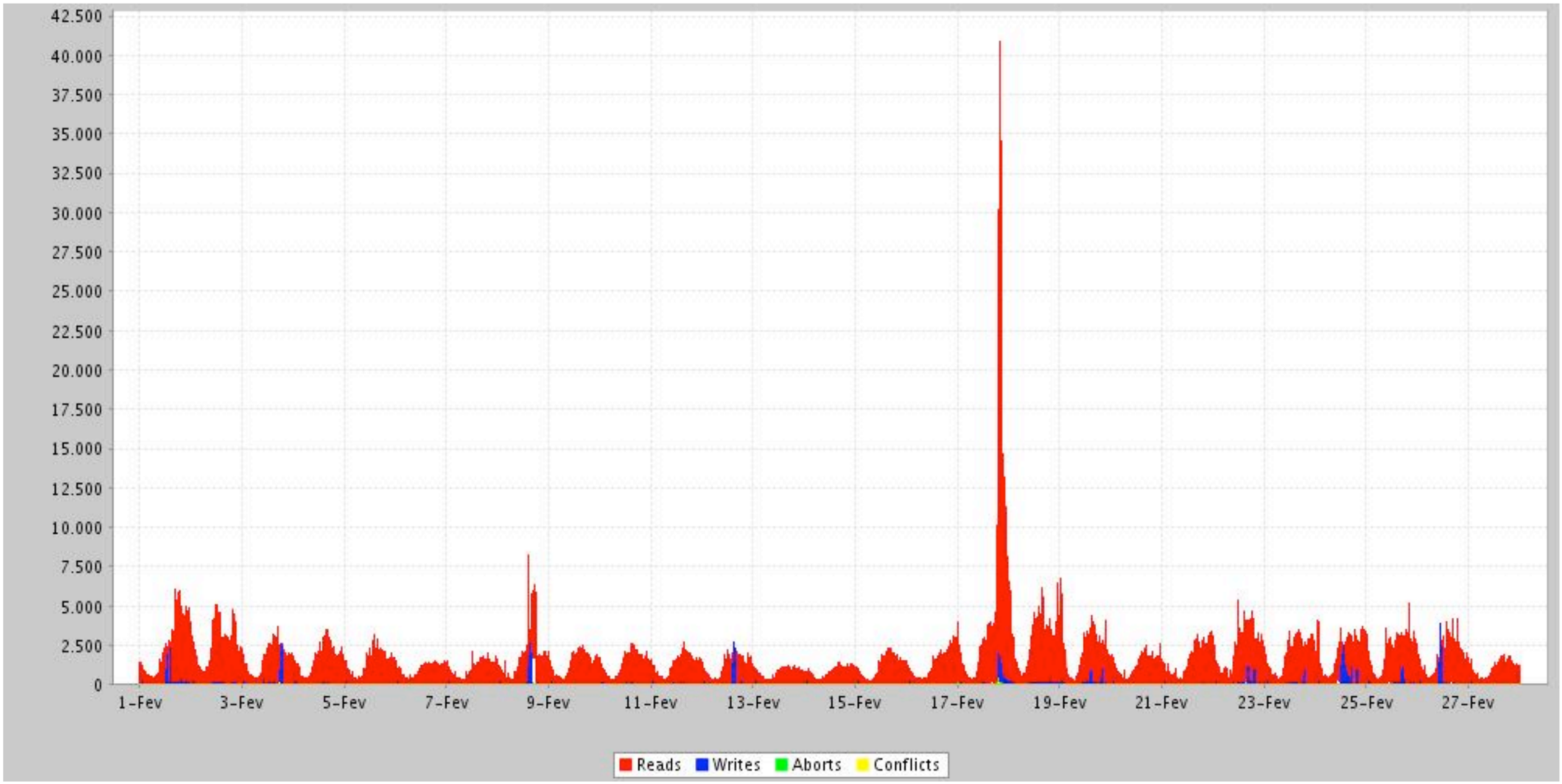


FF-based (10000 items)

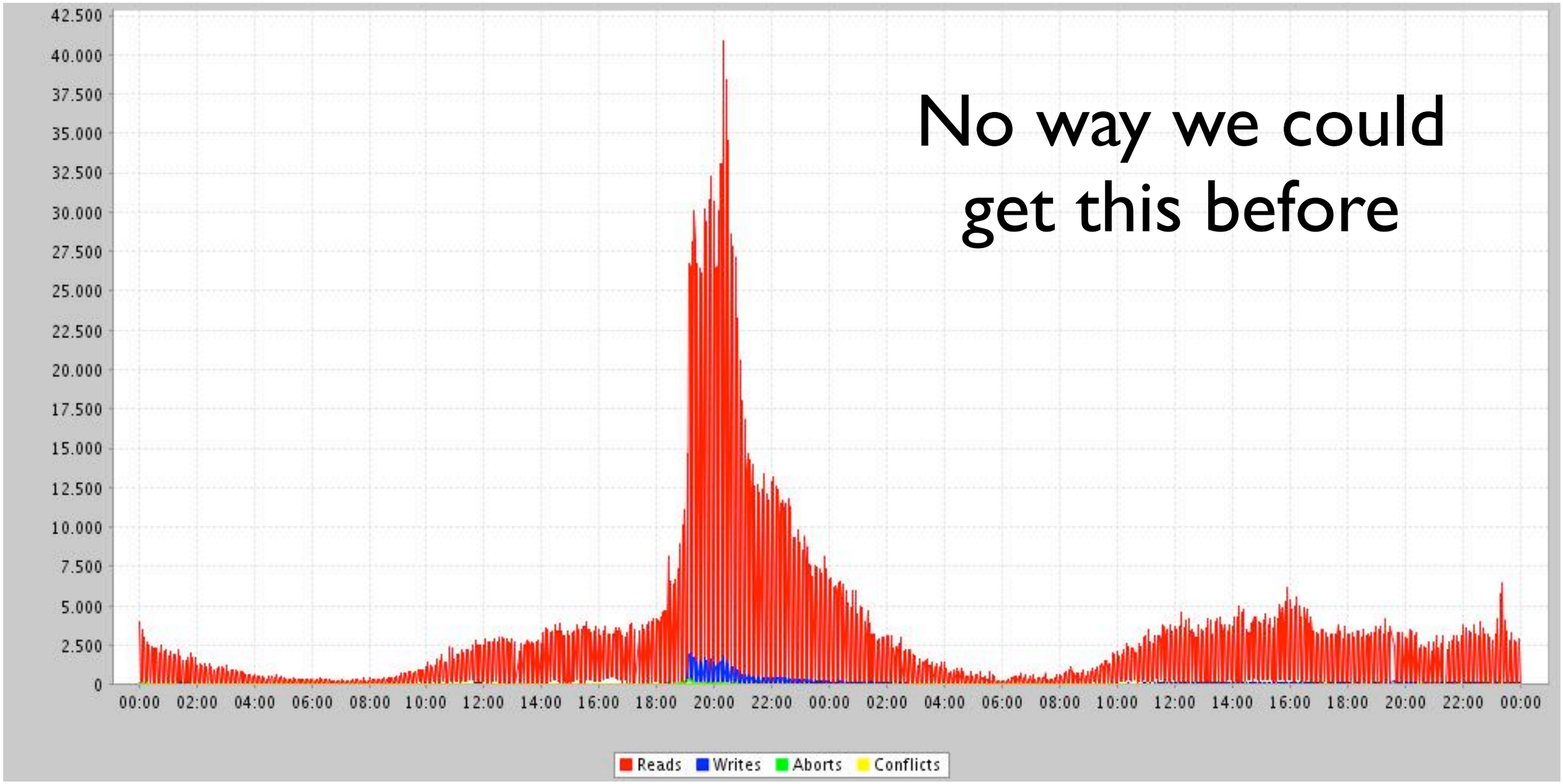
Some Fénix statistics


4500
4250
4000
3750
3500
3250
3000
2750
2500
2250
2000
1750
1500
1250
1000
750
500
250
0
5-Jul, 00:00
5-Jul, 12:00
6-Jul, 00:00
6-Jul, 12:00
7-Jul, 00:00
7-Jul, 12:00
8-Jul, 00:00
8-Jul, 12:00
9-Jul, 00:00
9-Jul, 12:00
10-Jul, 00:00
10-Jul, 12:00
11-Jul, 00:00
11-Jul, 12:00
12-Jul, 00:0
Reads
Writes
Aborts
Conflicts

Normal week



February 2010



No way we could get this before

17th to 19th

Fénix transactional workload

Writes $\sim$ 1%

Conflicts $\sim$ 1%

Some txs $>$ 10.000.000 reads

$$\frac{reads}{writes} > 1.000$$

	Reads	Writes	Conflicts
Normal day	898 802	15 330	9
17th February	3 306 600	116 651	745

END