Selective Optimization of Locks by Runtime Statistics and Just-in-time Compilation

Ray Odaira and Kei Hiraki

Department of Information Science and Technology

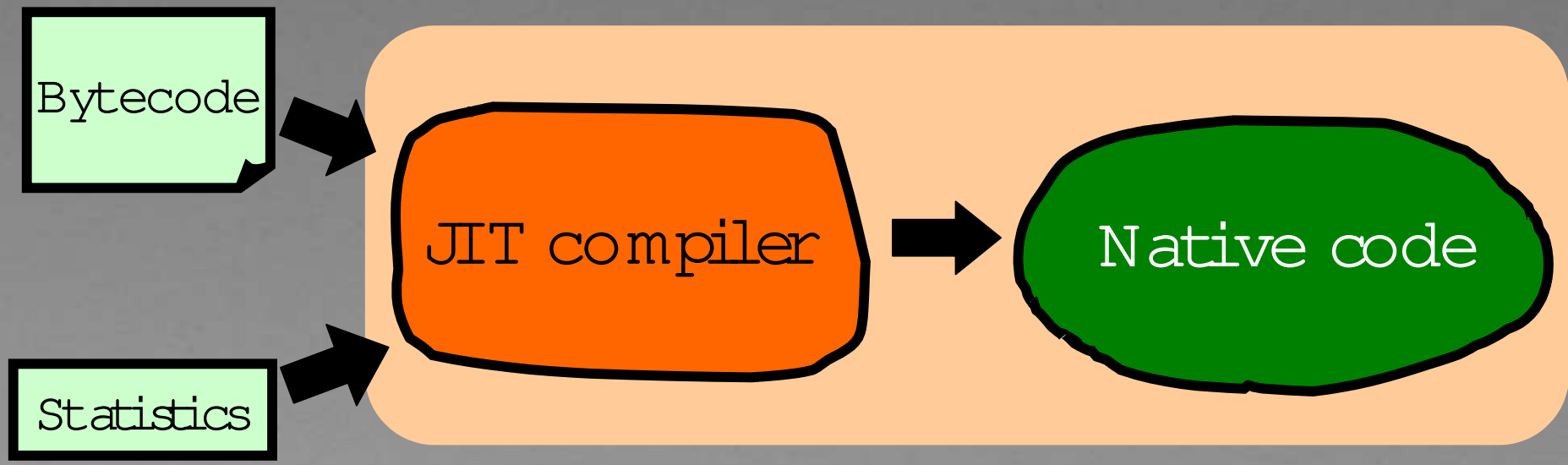
University of Tokyo

{ray, hiraki}@is.s.u-tokyo.ac.jp

Overview

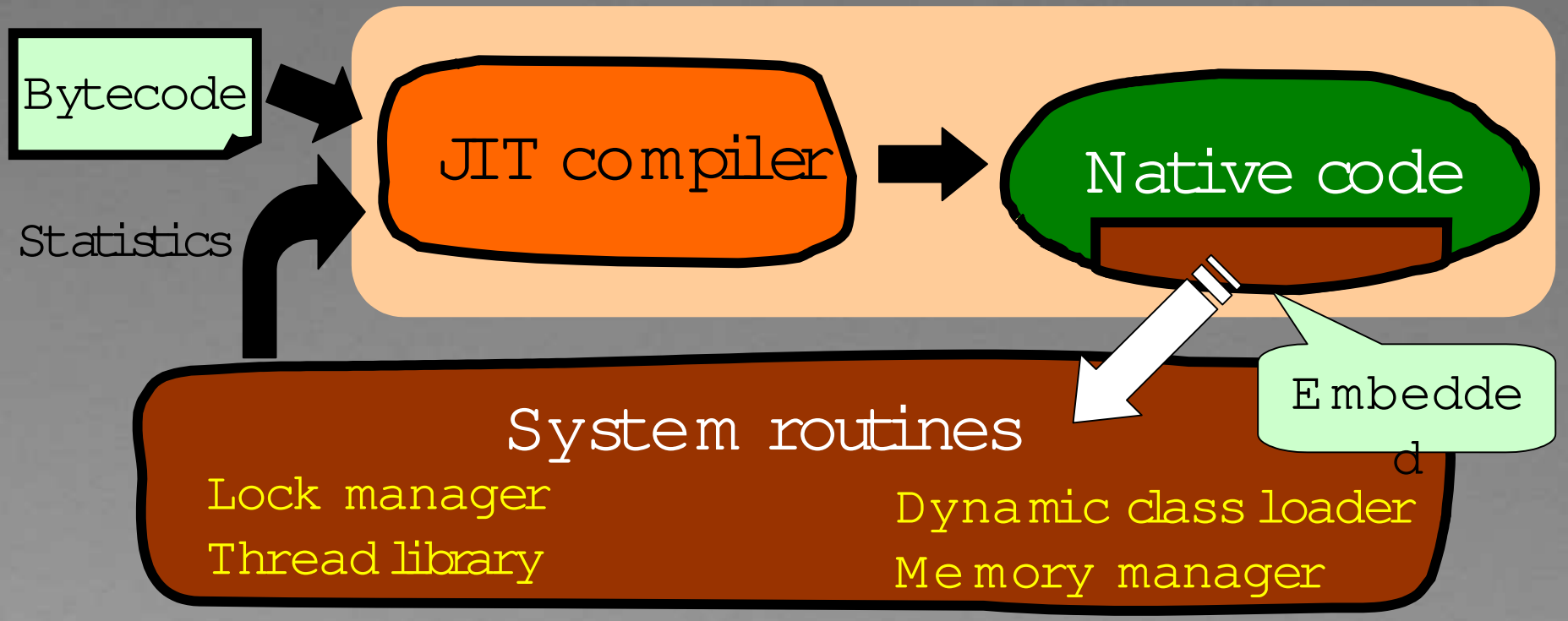
- *SSJIT*
  - Achieves adaptive and dynamic system-level optimization
  - Uses runtime statistics and JIT compiler
  - Focuses on server-side programs
- Apply *SSJIT* to a contention problem of mutex locks in server-side Java
  - *TNP by SSJIT*
- Experimental results show *TNP by SSJIT* can avoid the overhead of *TNP*

Existing JIT Compilers



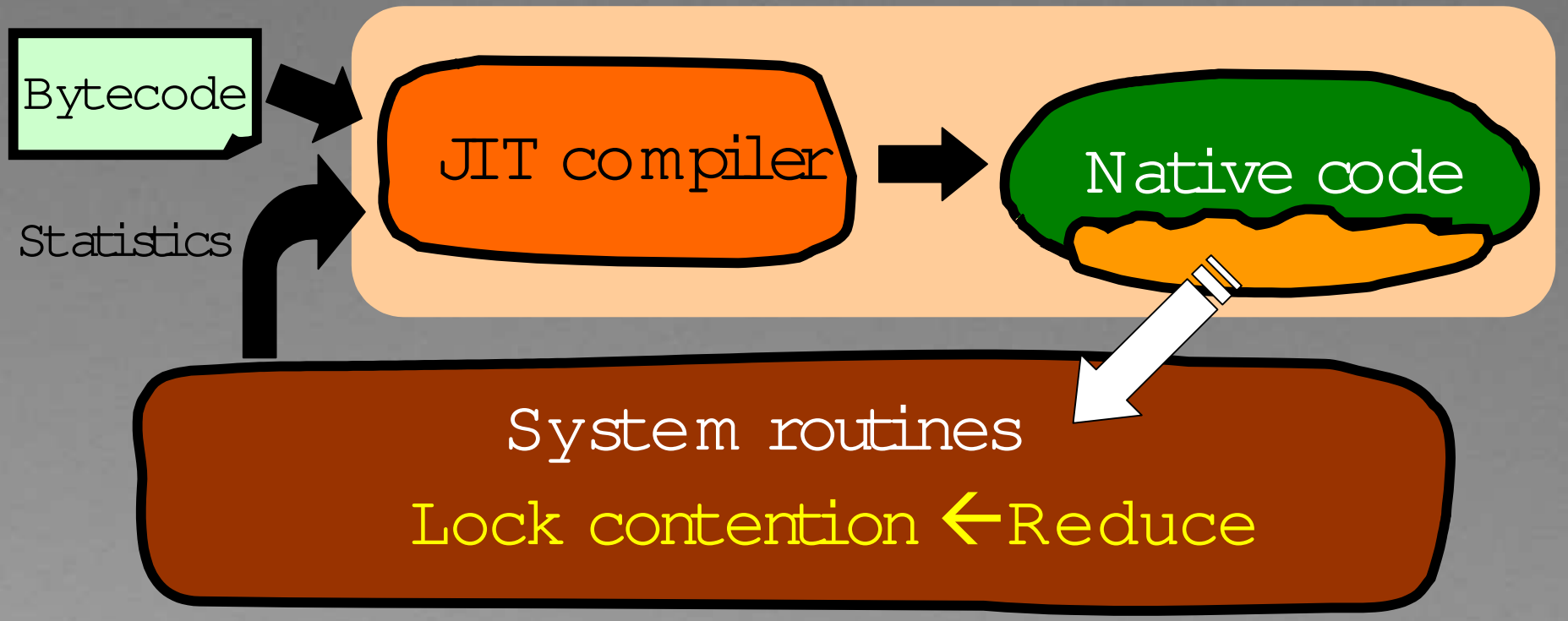
- How to translate bytecode into fast native code
  - Runtime statistics for method inlining [Hoelzle et al., '94]
  - Adaptive Optimization [Arnold et al., '00]

Server-side Programs



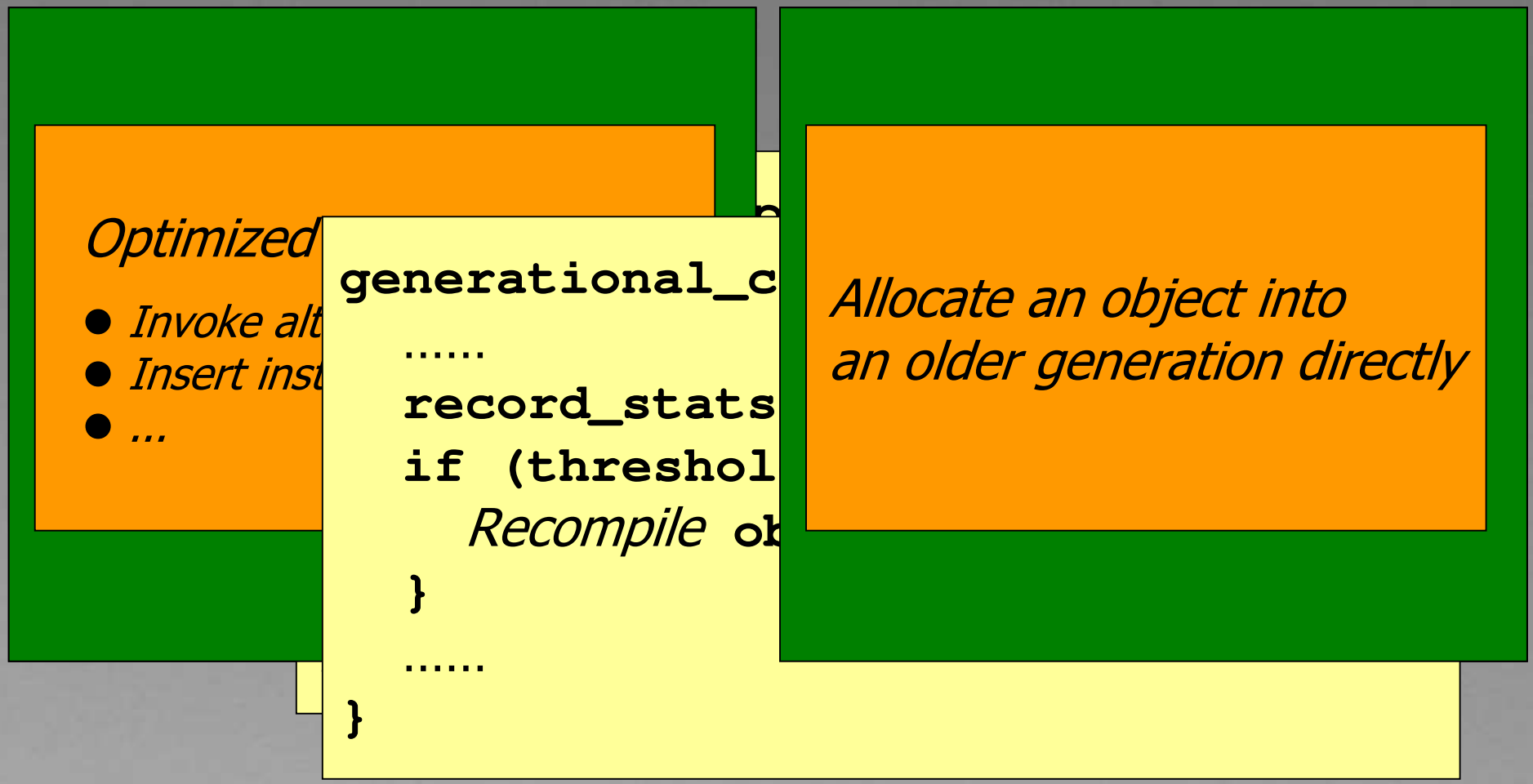
- System routines are executed frequently in server-side programs
  - Cause performance problems
    (Lock contention, GC overhead, etc.)

SSJIT



- *SSJIT (System-level dynamic optimization using runtime Statistics and JIT compiler)*

Methodology

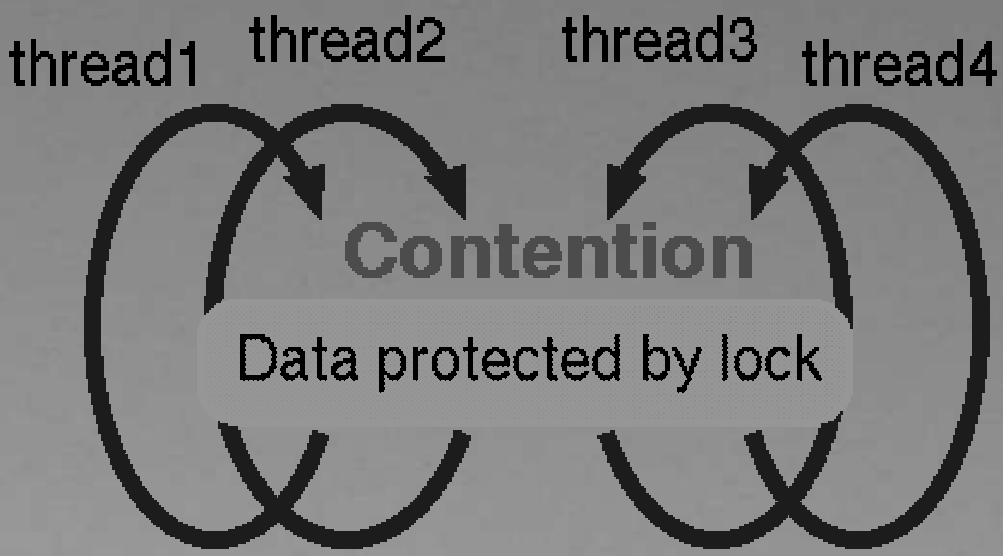


Objectives

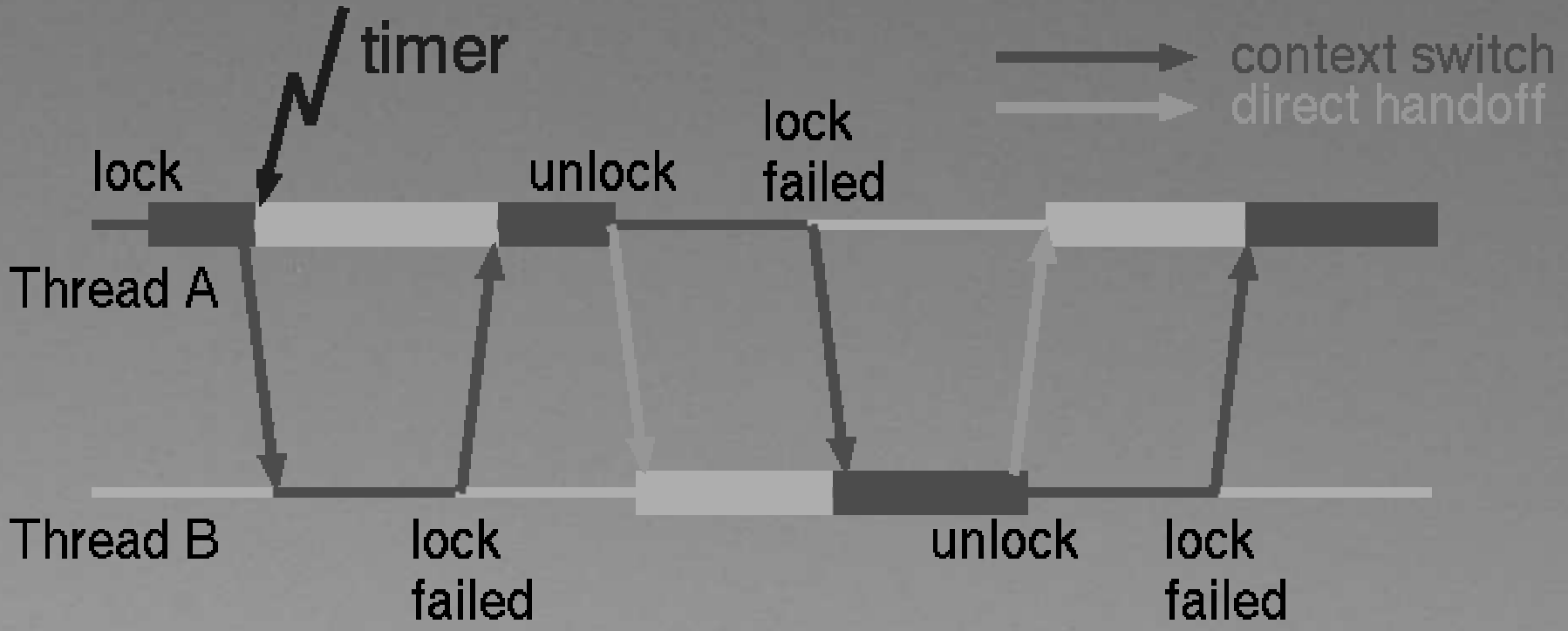
- To achieve adaptive and dynamic system-level optimization
- To avoid lock contention in server-side Java programs

Locks in server-side Java

- Several locks are used to protect data for server management
- Accessed frequently and repeatedly by all processing threads
- Contention causes degradation of throughput and fairness

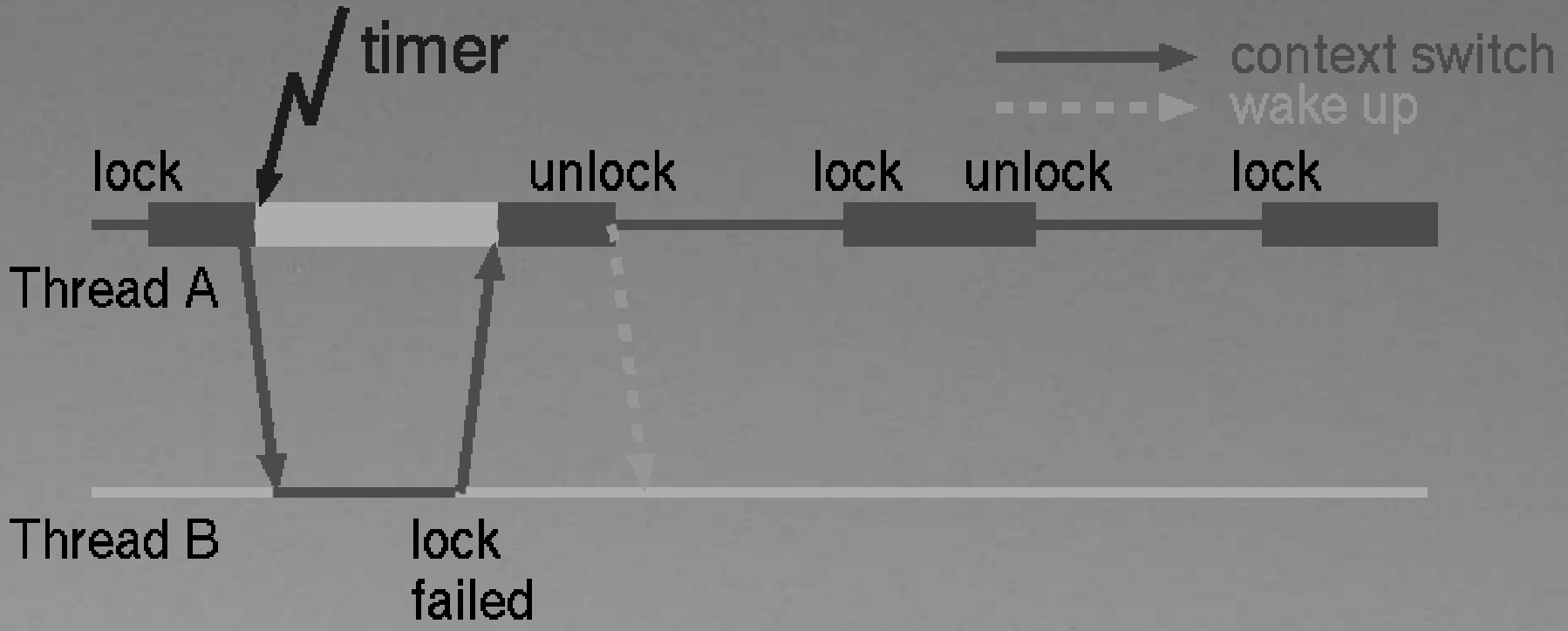


Handover Lock



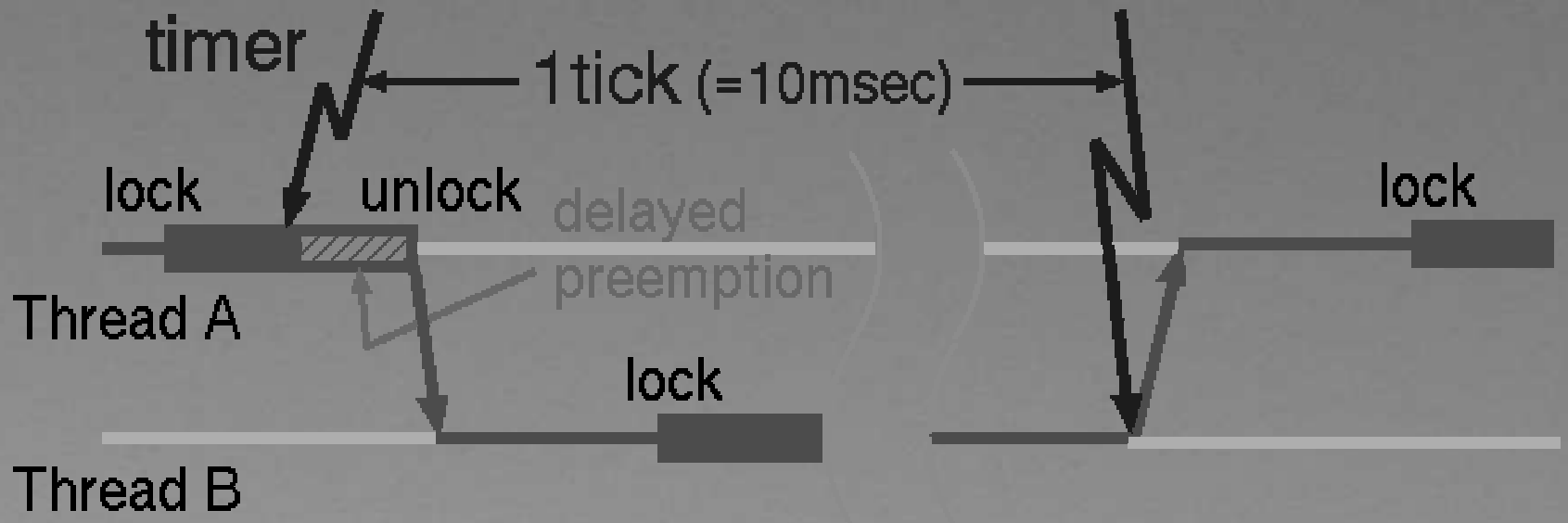
- Hands over the ownership to a waiting thread
- Frequent context switches in one tick
  - Degrades throughput

Renouncement Lock



- Does not handover the ownership, just wake up a waiting thread
- (Virtually) no context switch occurs
  - Degrades fairness of CPU scheduling

Temporally Non-Preemption [Edler '88]



- Delays a timer interruption during critical sections
  - Instrumentation before and after all the critical sections
- Ensures one context switch per one tick
  - Best tradeoff to manage both throughput and fairness
  - 1 tick == Compromise between throughput and fairness

But *TNP* has a problem in Java (1)

- Frequent lock operations in Java
- No contention in most of the locks [David et al., '98][Jong-Deok et al., '99]
- Java programmers cannot specify which locks should be *TNP*
  → *TNP* in all locks

Java language

```
void
synchronized meth() {
  synchronized (obj) {
    ……
  }
}
```

Java bytecode

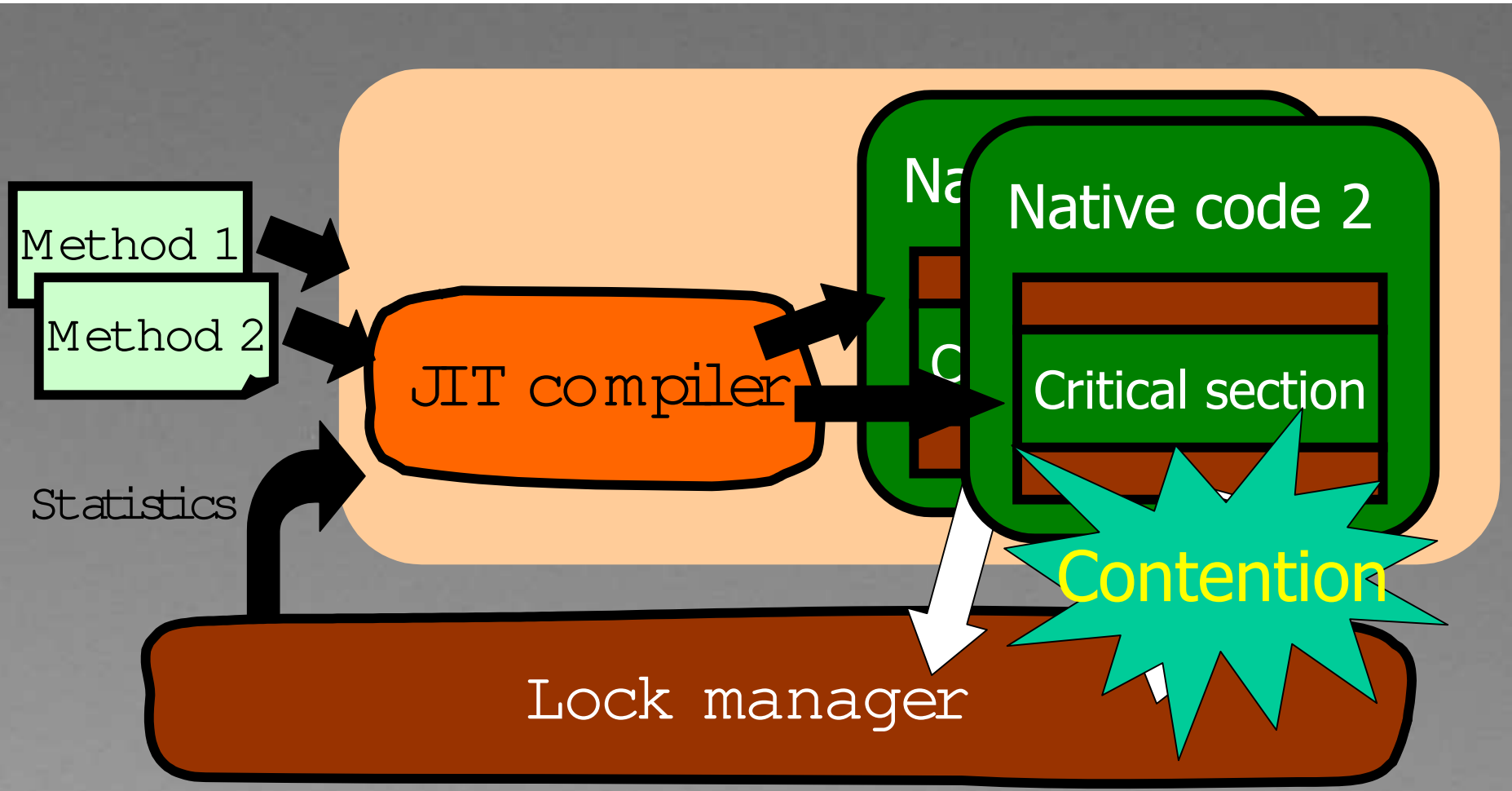
```
aload_0
monitorenter
……
aload_0
monitorexit
```

But *TNP* has a problem in Java (2)

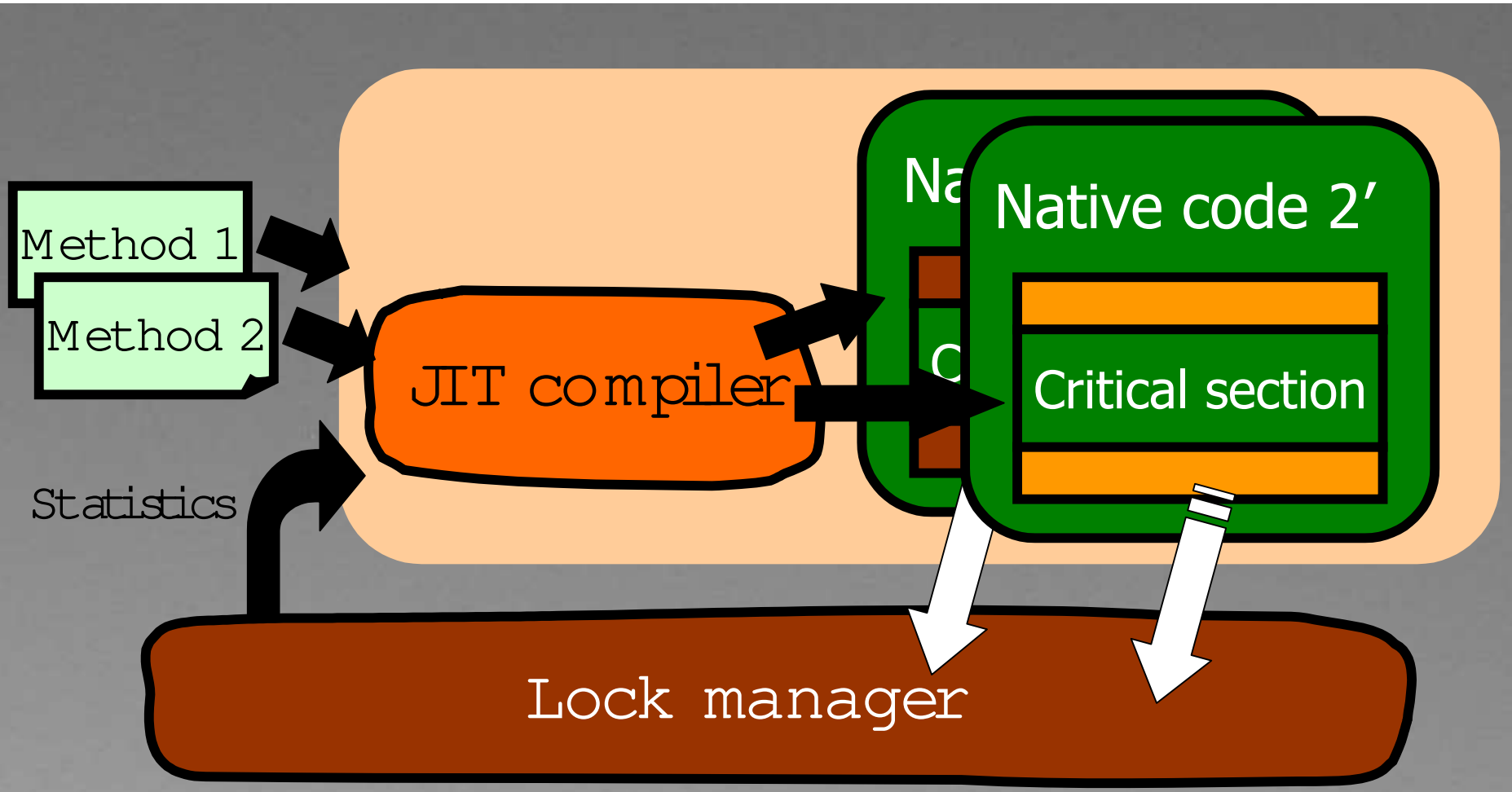
- Unnecessary overhead of *TNP* at the uncontended locks
  - Inserted instrumentation code
- In summary:
  - Handover lock → Degrades throughput
  - Renouncement lock → Degrades fairness
  - *TNP* → Suffers overhead

TNP by SSJIT

- System-level optimization using runtime statistics and a JIT compiler
  - Detects contended locks by runtime statistics
  - Re-compiles the methods containing the contended locks
  - Applies *TNP* to the contended locks



- JIT-compiles each method when it is first invoked
  - Inserts code for gathering runtime statistics
- Detects a contended lock



- Re-compiles the method
- Instruments the lock
  - Forbids preemption before the acquisition
  - Permits preemption after the release

Novel Points of *TNP by SSJIT*

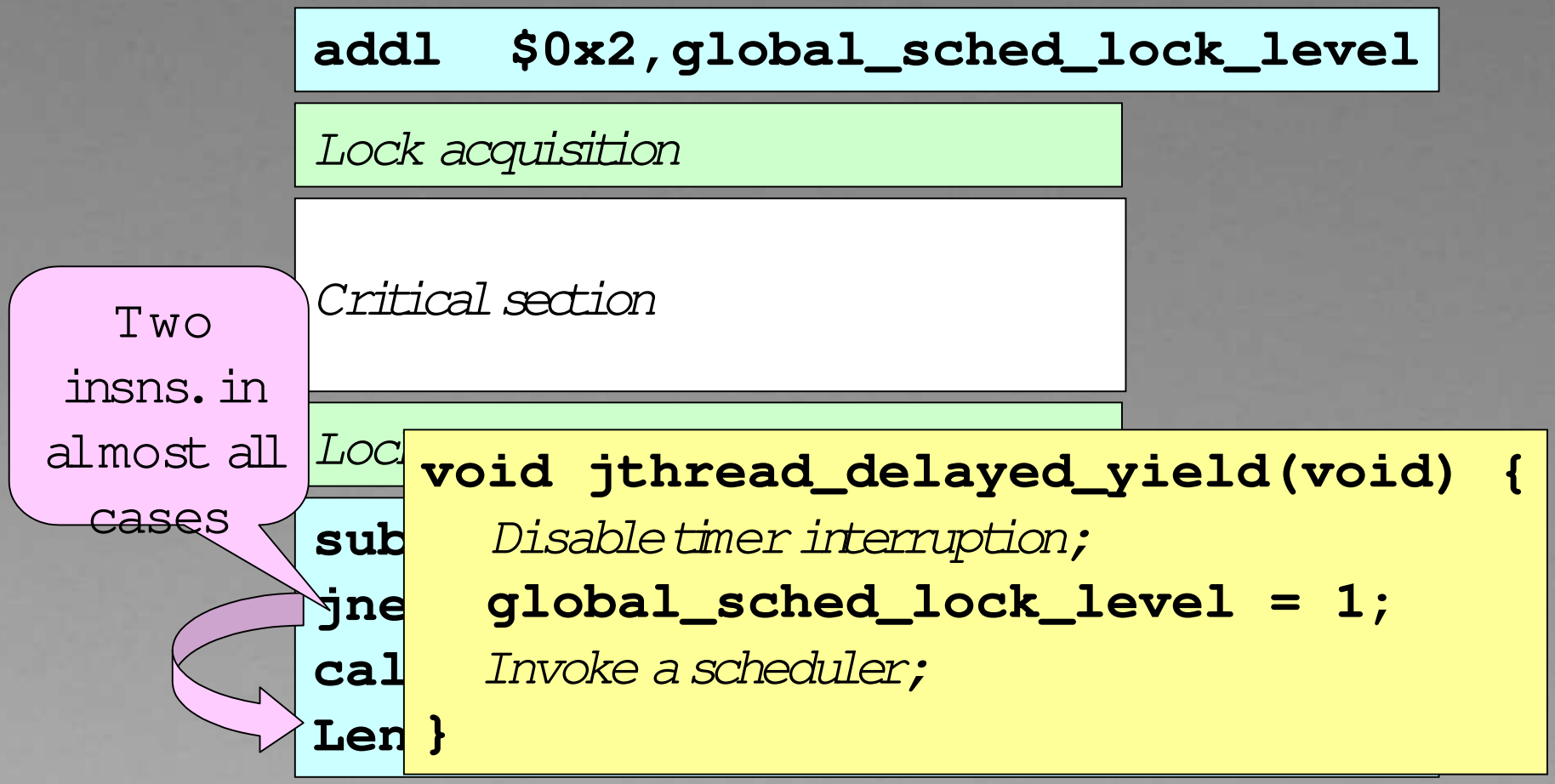
- Restricts *TNP* optimization only to the contended locks
- For non-contented locks
  - No increase in overhead for acquisition and release
- For contended locks
  - Manage both throughput and fairness
- Achieves adaptive and dynamic system-level optimization

Implementation

- Implemented on *Kaffe* 1.0.5
  - User-level thread library: *jthread*
  - JIT compiler: *jit3*
- Modified lock manager routines
  - To gather runtime statistics
- Added analyzing routines to the JIT compiler
  - To find and instrument a pair of acquisition and release

Instrumentation Code

- Implementation on 1 CPU, Intel/x86



Experiments

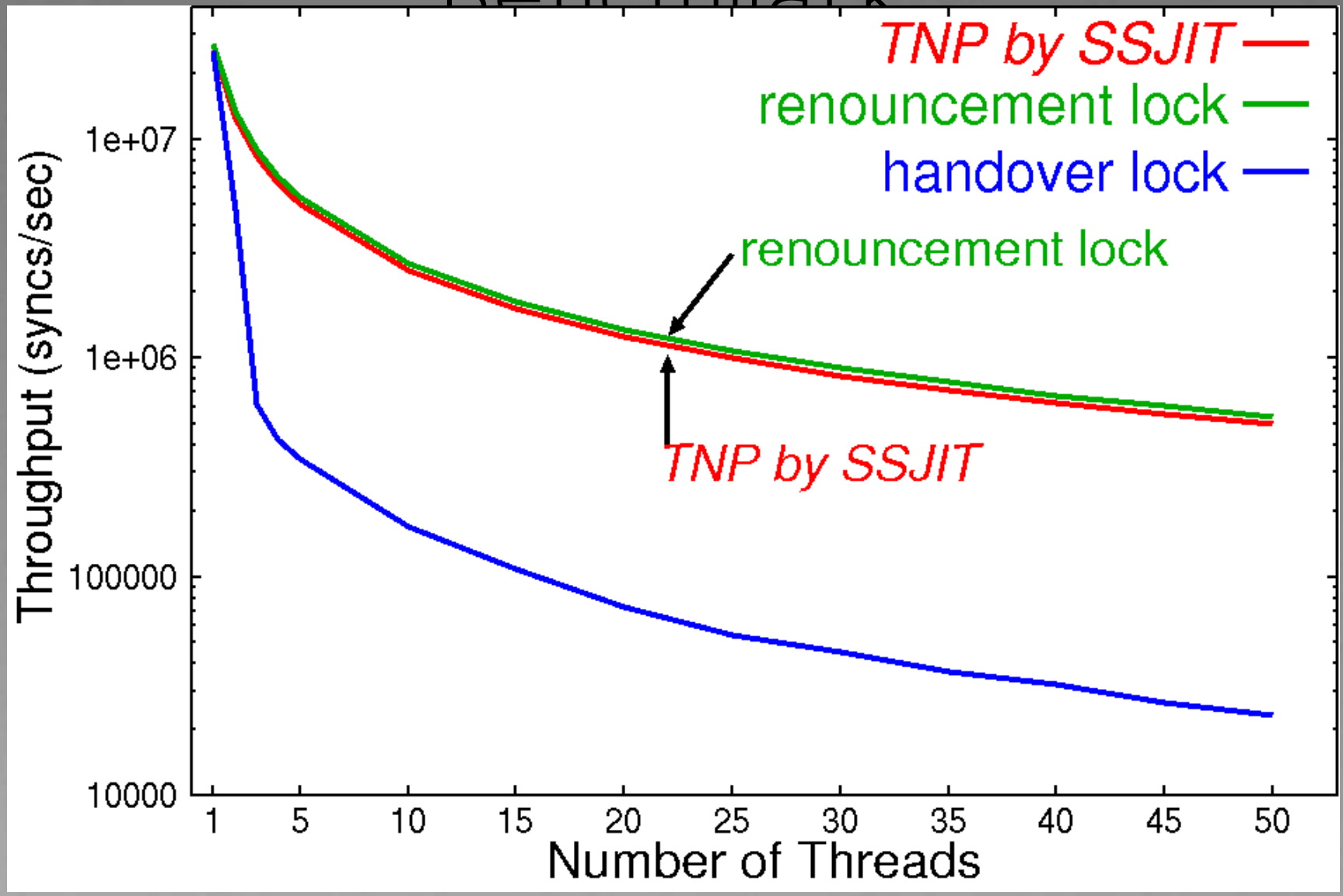
- 6 single-thread programs from SPEC JVM98
- A micro benchmark: *synchmeth*
  - Increment a shared counter inside a loop
- Tomcat 3.2.2
  - With a Servlet program which sends back contents of an HTTP request
- Linux 2.4.17/1.6GHz Athlon MP/512MB RAM

Overhead of *TNP*

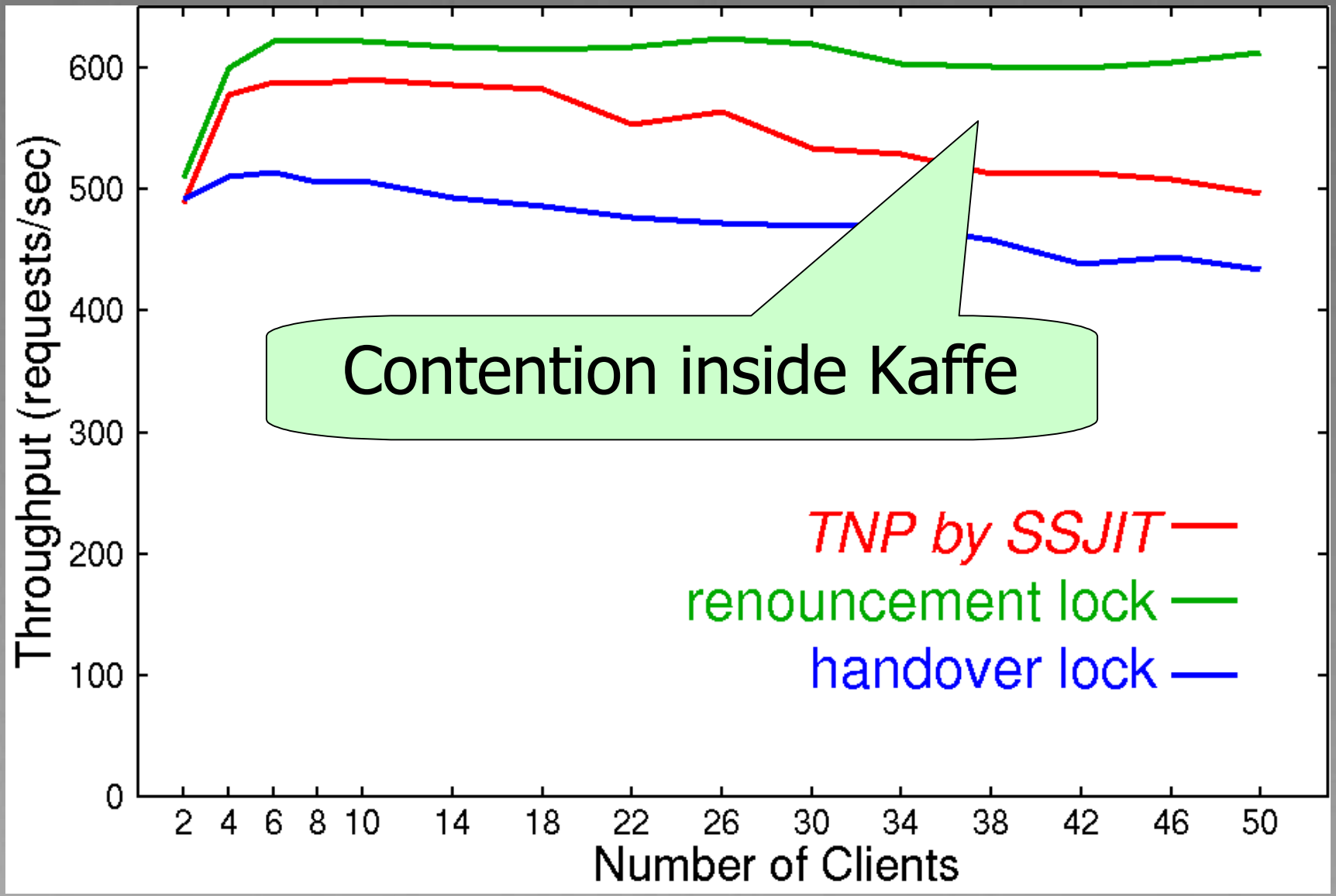
- Single thread means no contention
  - Compare the overhead of original *TNP* with that of *TNP by SSJIT*

SPEC JVM98	A: *TNP by JIT* (sec)	B: *TNP* (sec)	Overhead: (B-A)/A
compress	12.89	12.91	+0.2%
jess	20.01	20.46	+2.2%
db	40.31	41.70	+3.4%
javac	33.15	33.85	+2.1%
mpegaudio	18.88	18.89	+0.1%
jack	28.88	29.45	+1.9%

Throughput of Micro Benchmark



Throughput of Tomcat



Fairness (1)

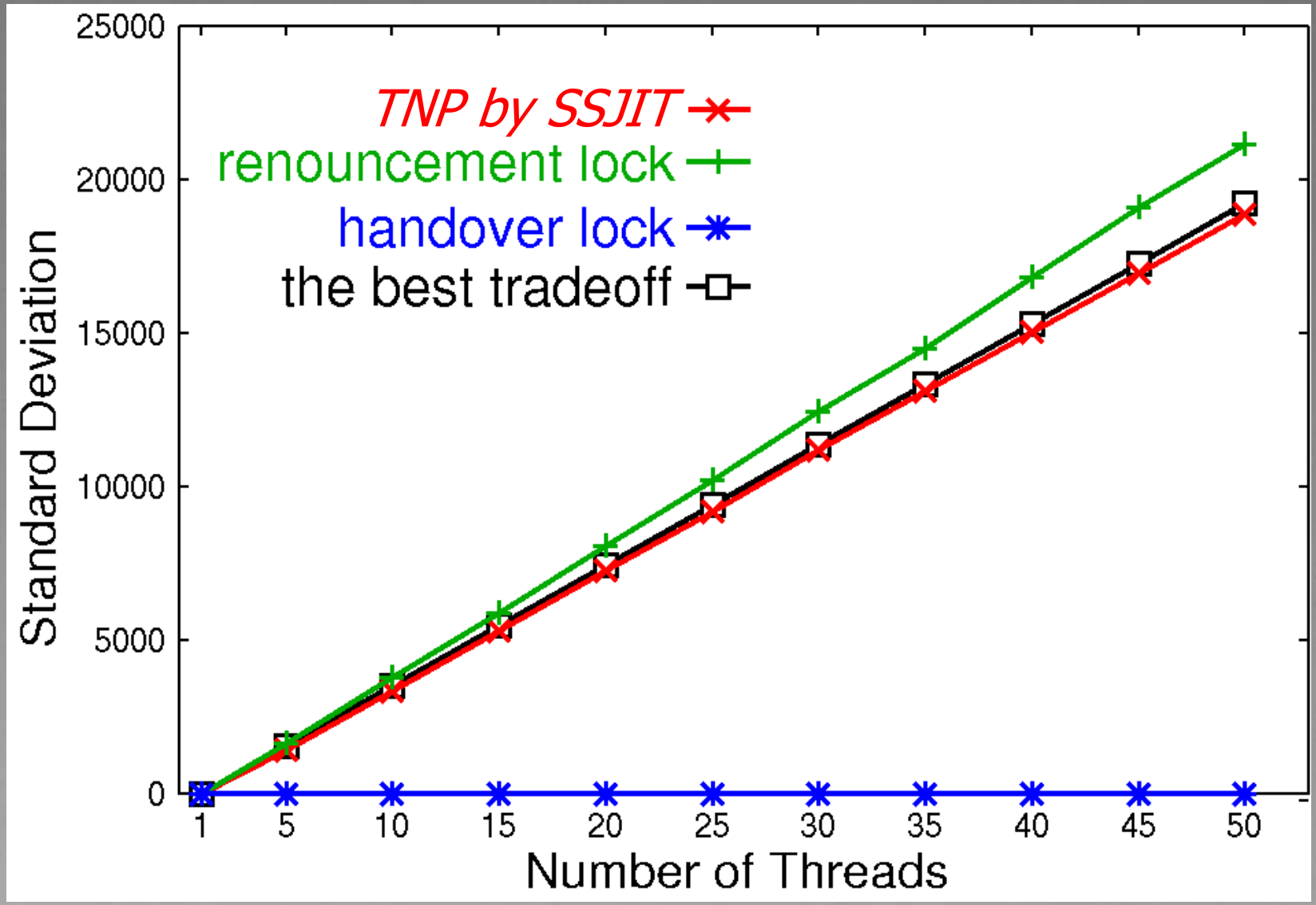
- Dispersion of waiting time to access a shared resource
- Modified the micro benchmark *synchmeth*
  - Dispersion of the difference between the values

Reminder: one context switch per 1tick is the best tradeoff

sqrt(x)×(n−1),
where x #iterations/1tick
and n #threads

thread1	*1, 1, 8, ...*
thread2	*8, 1, 1, ...*
thread3	*1, 9, 1, ...*
thread4	*1, 1, 1, ...*

Fairness (2)


TNP by SSJIT
renouncement lock
handover lock
the best tradeoff
Standard Deviation
Number of Threads
25000
20000
15000
10000
5000
0
1
5
10
15
20
25
30
35
40
45
50

Related Work

- *Run-Time Type Feedback* [Hoelzle et al., '94]
  Adaptive Optimization [Arnold et al., '00]
- *Paradyn* [Zhichen et al.,'99]
  Gather statistics by dynamic instrumentation
- *Temporally Non-Preemption* [Edler et al.,'88],
  Scheduler-Conscious Synchronization [Leonidas et al.,'97]
  Programmers must use special kernel interfaces
- *Three-tier Blocking Lock* [Dimpsey et al., '00]
  Choose spin/yield/block using counters

Conclusions

- Proposed *SSJIT*, a system-level optimization using a JIT compiler and runtime statistics
- *SSJIT* can
  - Manage both throughput and fairness in server-side Java
  - Avoid overhead of unnecessary instrumentation
- Experimental results show the validity of *SSJIT*
- We can apply *SSJIT* to other system-level performance problems