Thin Locks: Featherweight Synchronization for Java

David F. Bacon

Ravi Konuru

Chet Murthy

Mauricio Serrano

IBM T.J. Watson Research Center

Introduction

- It's the same old sad story:
  - ☺ Java has threads and synchronized methods
  - ☹ But synchronization is dog-slow
  - ☠ So synchronization is optional
- Shoot the foot of your choice:
  - ⌛ Get bad performance, or
  - 💣 Get bug-prone code

The Library Dilemma

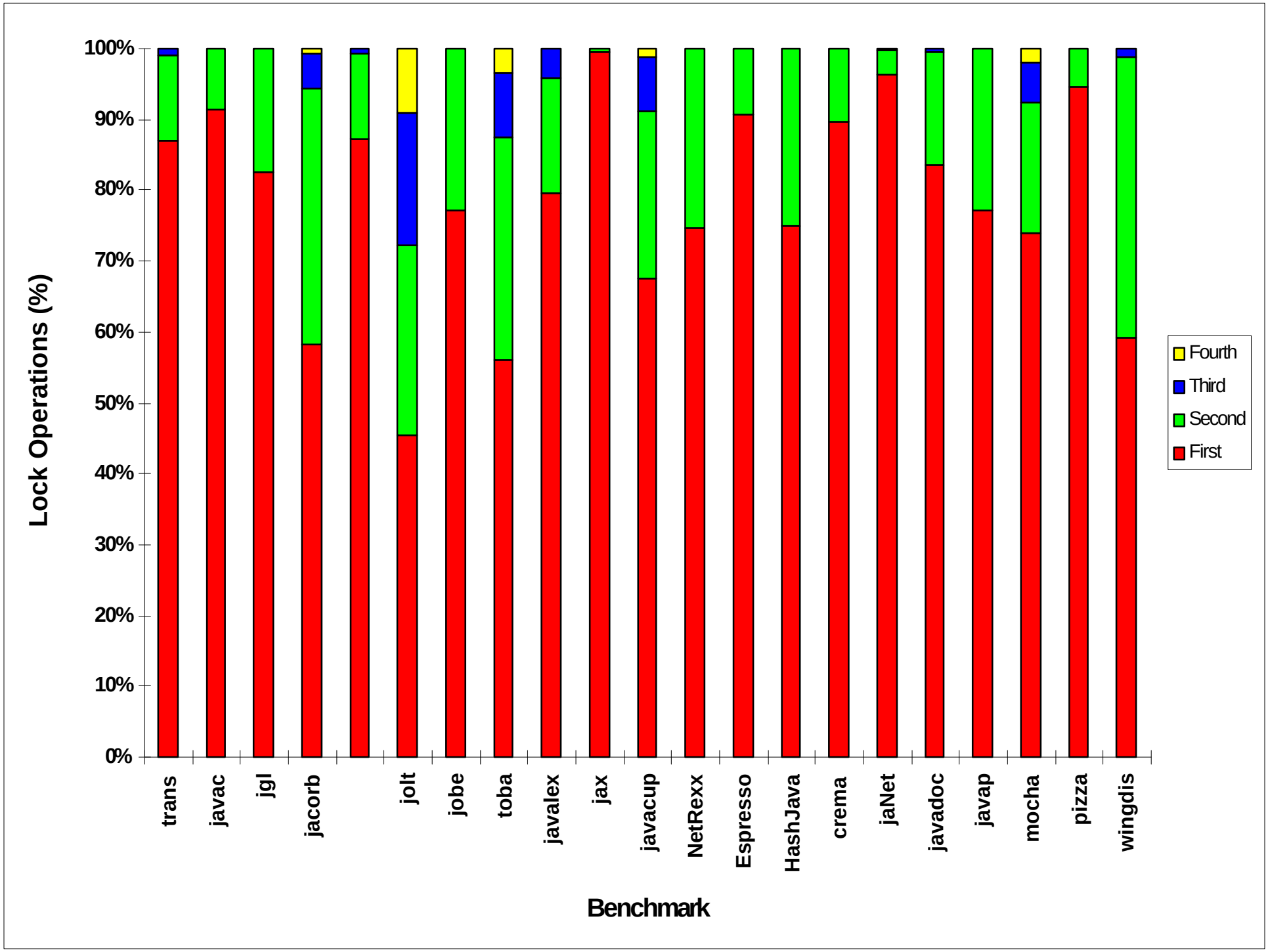
- Libraries must be thread-safe
  - All non-trivial methods are synchronized
  - Library call to set a bit in a bit vector:
    - » ~50 instructions to lock and unlock the object
    - » ~10 instructions method call overhead
    - » ~5 instructions to actually set the bit
- Locking overhead frequently 25-50%
  - even in single-threaded applications!

Java Synchronization Features

- Thread can lock an object repeatedly
  - lock nesting count must be kept
- On exception, thread must release locks
  - call stack implicitly names locked objects
  - therefore, list of locked objects not needed
  - exceptions:
    - » JNI
    - » non-nested **monitorenter**/**monitorexit**

Locking Scenarios by Frequency

- Object is unlocked.
  - We already locked the object a few times
    - We already locked the object a lot of times
      - Object is locked and we are the first to queue up
        - Object is locked and other threads are waiting


Lock Operations (%)
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
trans
javac
jgl
jacorb
jolt
jobe
toba
javalex
jax
javacup
NetRexx
Espresso
HashJava
crema
jaNet
javadoc
javap
mocha
pizza
wingdis
Benchmark
Fourth
Third
Second
First

Unified Locking Design

- Key innovations for most common cases:
  - 24-bit lock in object
  - Hash code compressed to 2 bits
- “Thin locks” implemented as a veneer
  - pre-existing heavy-weight locks for contention
  - simple, portable solution

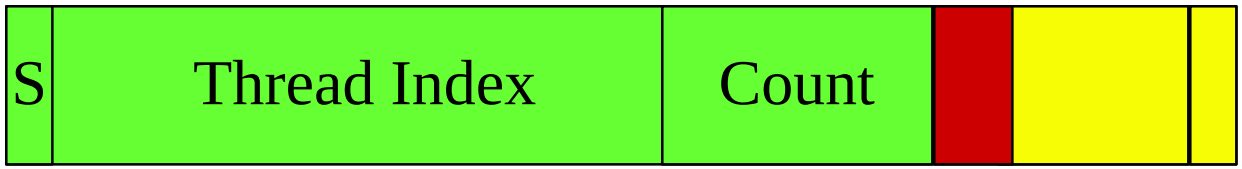
Hash Code Compression

- Handle-based systems use address of object
- IBM's JVM has no handles; objects move
- Apply same idea optimistically:
  - use address of object as its hash code
  - save the address when object is moved by GC
  - 3 states: *unhashed*, *hashed*, *hashed & moved*

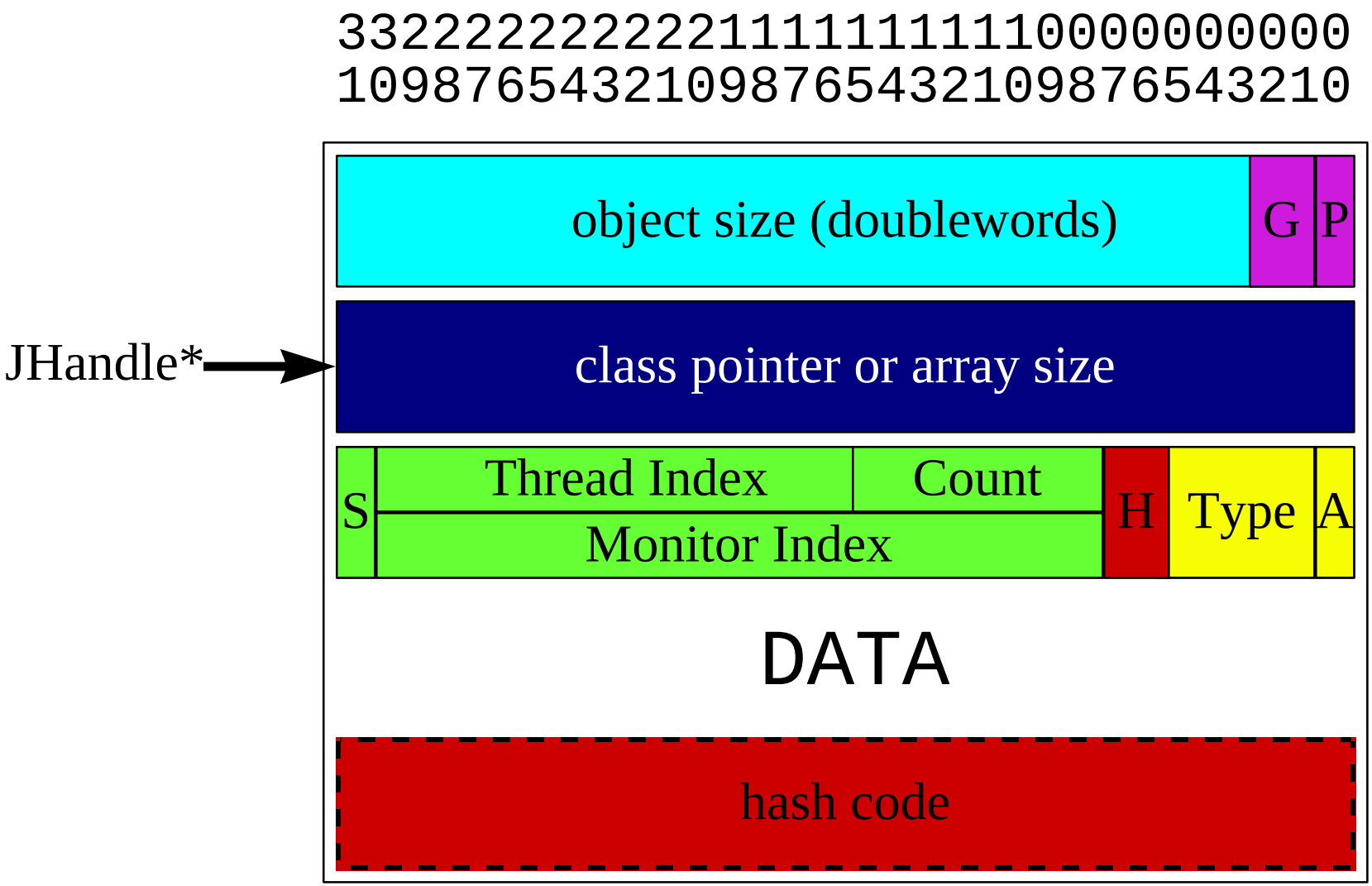
Thin Locks

- 24-bit field in object (using hash code space)
- Contains either
  - Thread index of owner, and lock count, or
  - Monitor index of heavy-weight monitor
- Compare-and-swap used for lock acquisition
- No atomic operations required for
  - unlocking
  - nested locking

Locking

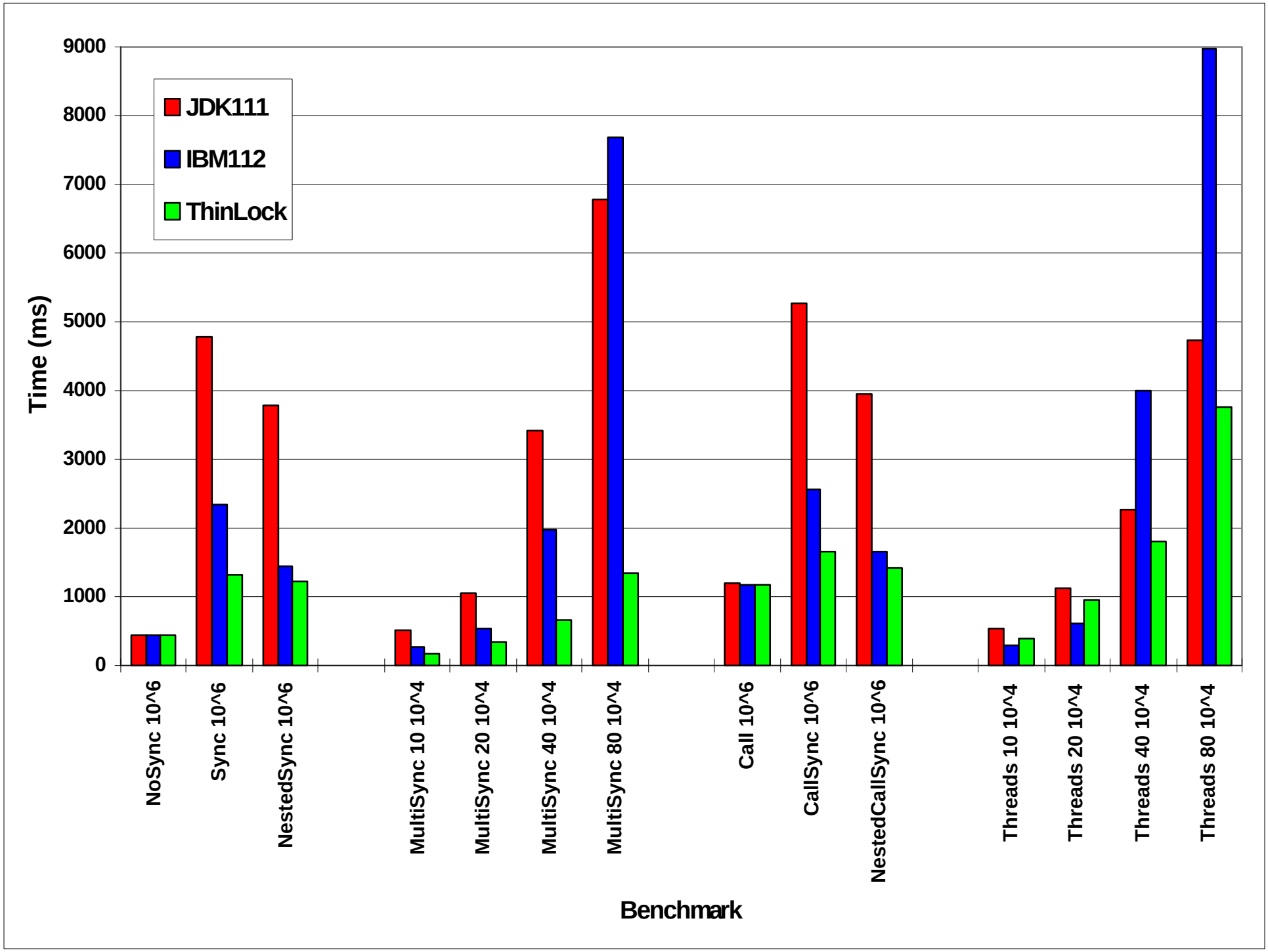


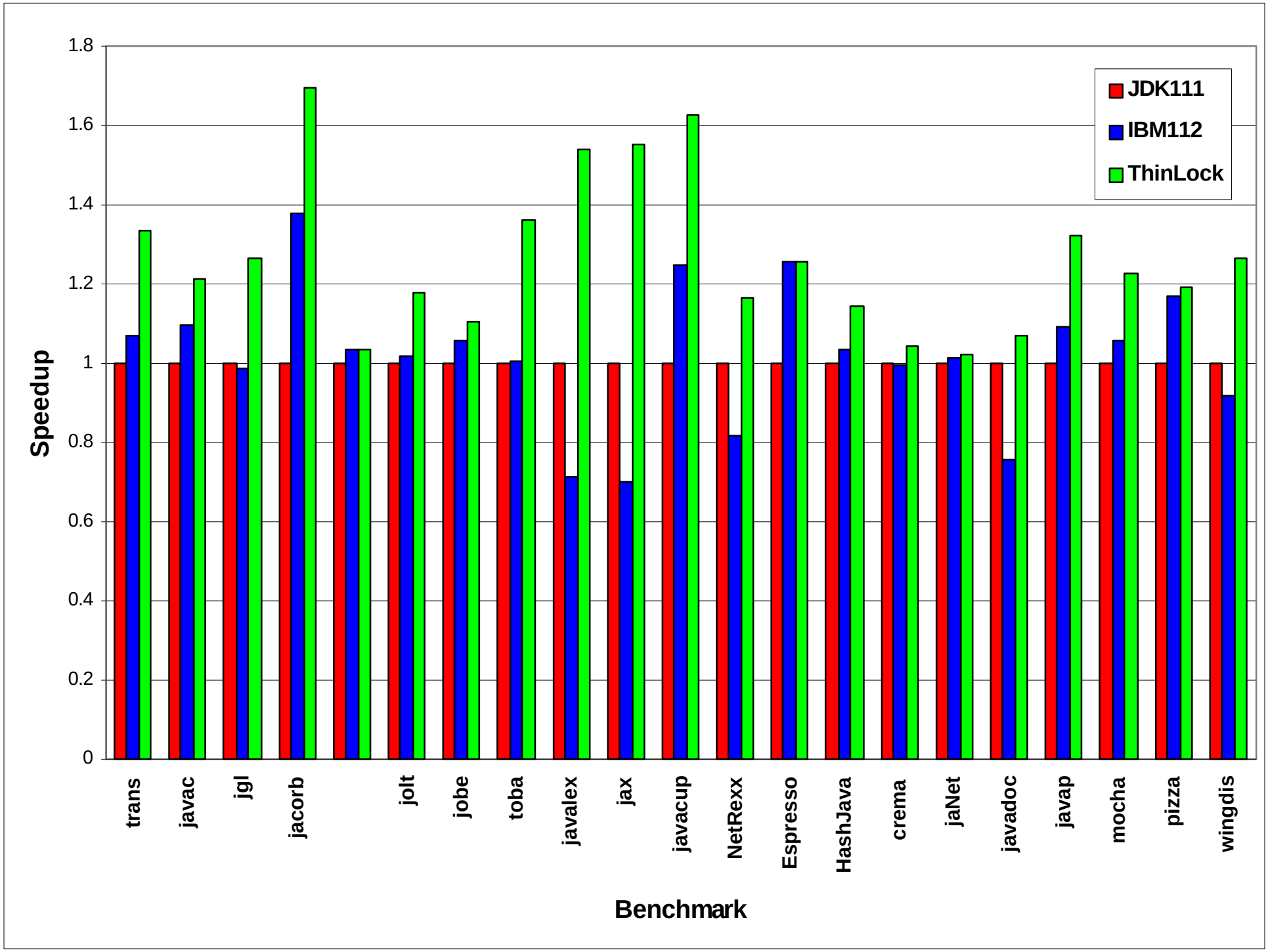

```
atomic_t unlocked = obj->monword & ~LOCK_MASK
atomic_t locked = unlocked | thread;
if (CMP&SWP(obj->monword, unlocked, locked))
  return;
unsigned xored = obj->monword ^ thread;
if (xored < COUNT_MASK) {
  obj->monword += COUNT_INCREMENT;
  return;
}
```

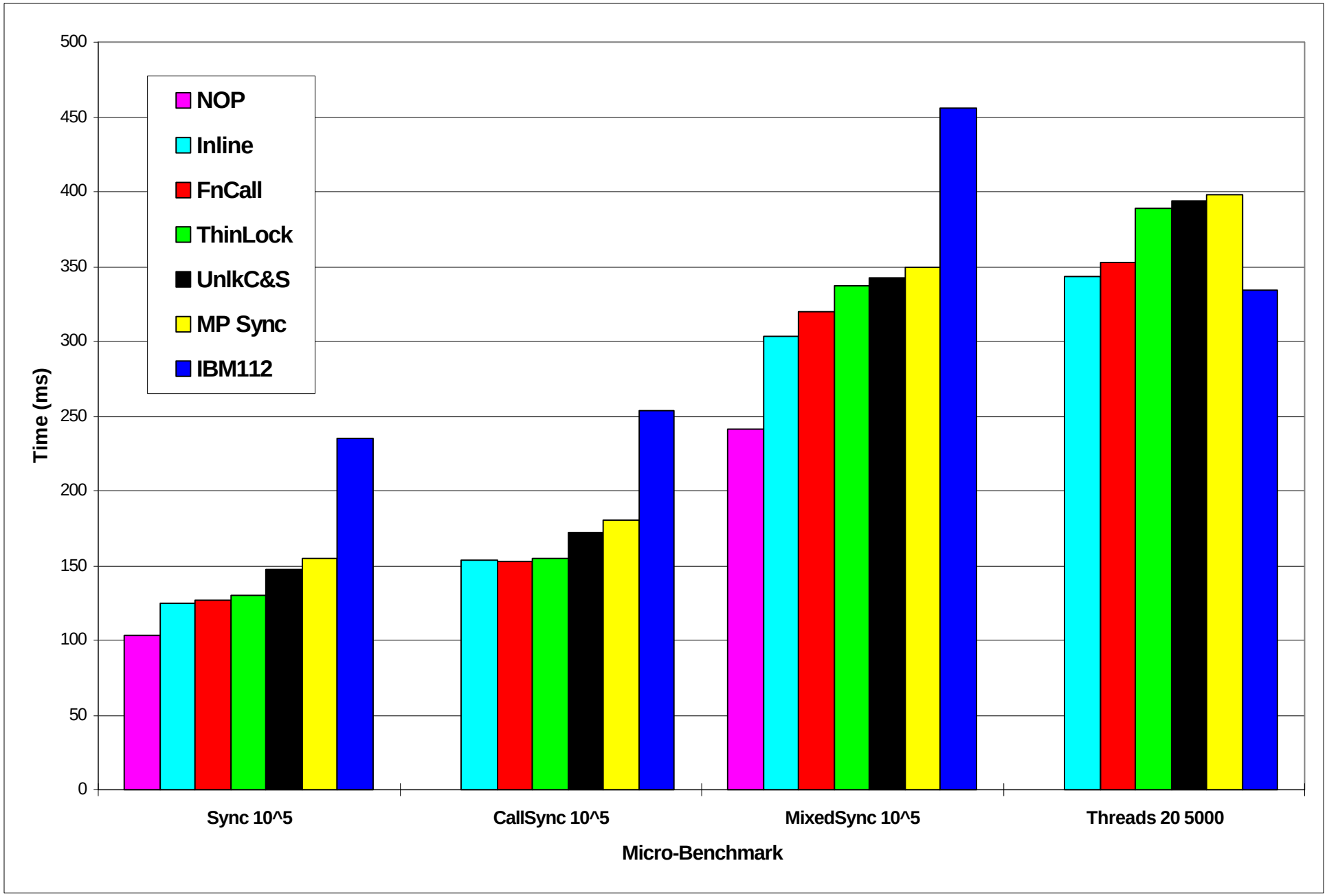


P - Pinned
A - Array

G - Garbage collection (2 bits)
H - Hash code state (2 bits)
S - Monitor Shape Bit


JDK111
IBM112
ThinLock
Time (ms)
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
NoSync 10^6
Sync 10^6
NestedSync 10^6
MultiSync 10 10^4
MultiSync 20 10^4
MultiSync 40 10^4
MultiSync 80 10^4
Call 10^6
CallSync 10^6
NestedCallSync 10^6
Threads 10 10^4
Threads 20 10^4
Threads 40 10^4
Threads 80 10^4
Benchmark


JDK111
IBM112
ThinLock
Speedup
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
trans
javac
jgl
jacorb
jolt
jobe
toba
javalex
jax
javacup
NetRexx
Espresso
HashJava
crema
jaNet
javadoc
javap
mocha
pizza
wingdis
Benchmark


500
450
400
350
300
250
200
150
100
50
0
Time (ms)
NOP
Inline
FnCall
ThinLock
UnlkC&S
MP Sync
IBM112
Sync 10^5
CallSync 10^5
MixedSync 10^5
Threads 20 5000
Micro-Benchmark

Status

- Implemented in IBM JDK 1.1.2
  - ported forward to 1.1.4 and 1.1.6
- Platforms:
  - AIX POWER, PowerPC, PowerPC MP
  - MVS 370/390 MP
  - Windows 95
  - WindowsNT x86
  - IBM Network Computer (NC)

Conclusions

- Java monitors can be efficient
  - 5-10 instructions to lock/unlock an object
  - no increase in object size
- Median speedups of 1.22, maximum 1.7
- Minimal changes to run-time system
  - portable, maintainable solution
- Scalability over a much wider range
  - larger “working sets” of locked objects
  - multiprocessor systems