



IBM Research

TO-Lock: Removing Lock Overhead Using the Owners' Temporal Locality

Takeshi Ogasawara, Hideaki Komatsu, and
Toshio Nakatani
IBM Tokyo Research Laboratory

Outline

- **Background**
- **Our Approach**
- **Experimental results**
- **Conclusions**

Java as a concurrent language

- Programmers can create threads and synchronize them by locking an object.

Example: Synchronizes the current thread and a new thread:

Current thread

```
t = new MyThread();
t.start();
```

```
synchronized(myObject)
```

```
{
```

```
// critical region
```

```
}
```

myObject

New thread

```
// start
```

```
:
```

```
:
```

```
:
```

```
synchronized(myObject)
```

```
{
```

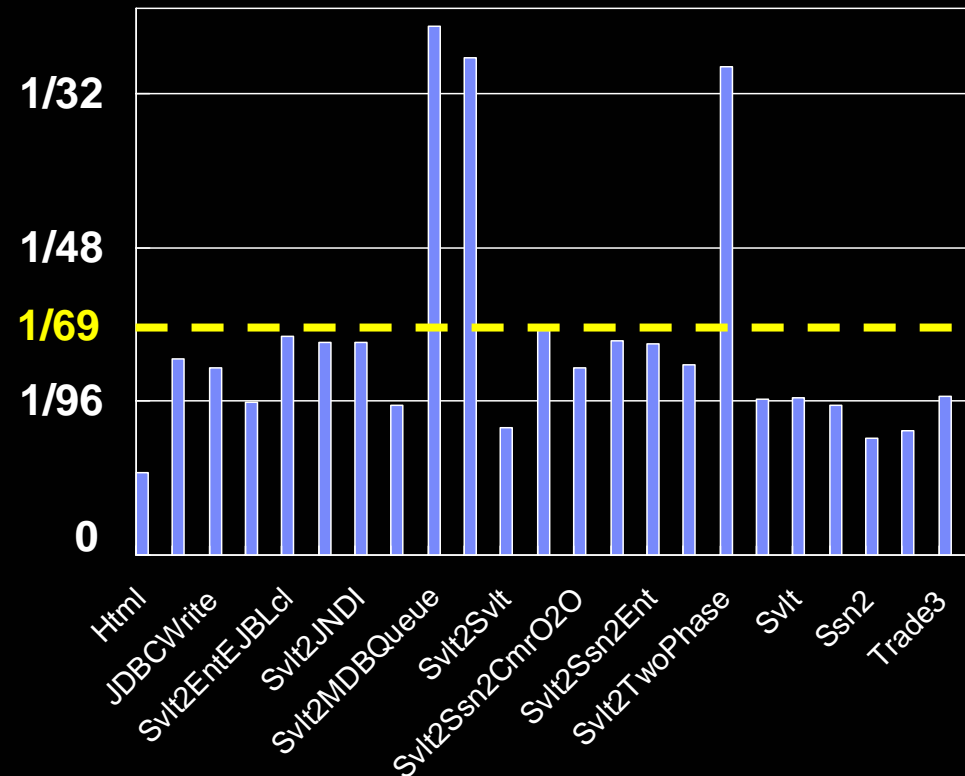
```
// critical region
```

```
}
```

Frequent locks in Java applications

- **We see frequent lock operations on the IBM WebSphere Application Server:**
 - Web-primitive benchmarks
 - Trade3
- **Lock frequency = a lock occurs once per N heap accesses:**
 - Maximum 1/28
 - Average 1/69

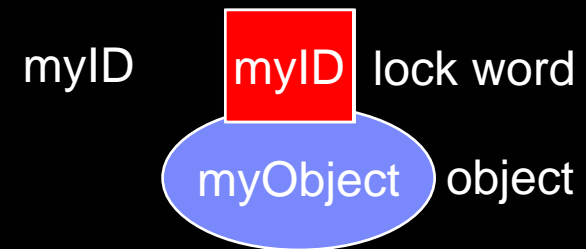
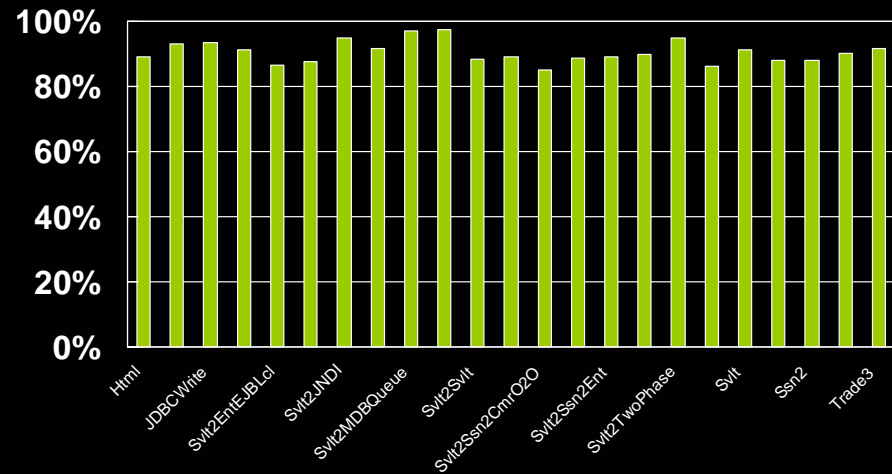
Lock frequency of all the lock operations



Thin lock (Bacon et al., PLDI '98)

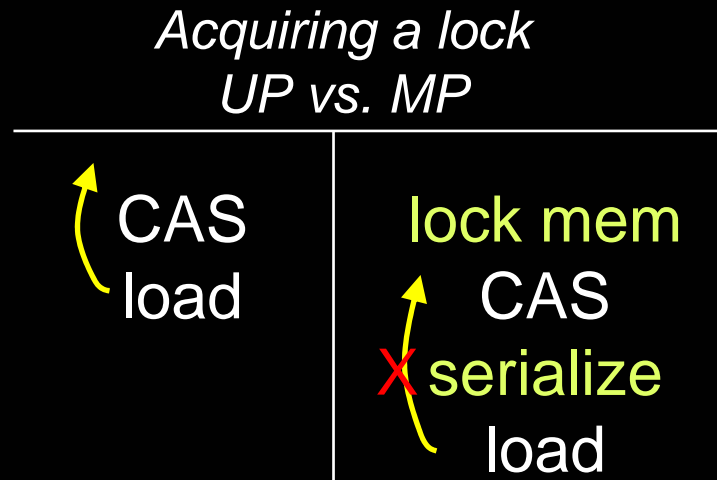
- **Uncontended locks** are the majority of the whole lock operations.
 - More than 90%
- **Thin lock uses a single compare-and-swap (CAS) operation for acquiring an uncontended lock**
 - Efficient on a *UP*
 - **Not efficient** on an *SMP*

Owner locality of all of the lock operations



Why locks are more costly on an SMP

- We must **lock** the cache line during a CAS to atomically set the owner's thread id.
- We must **serialize** the CAS operation and the following operations in a critical region



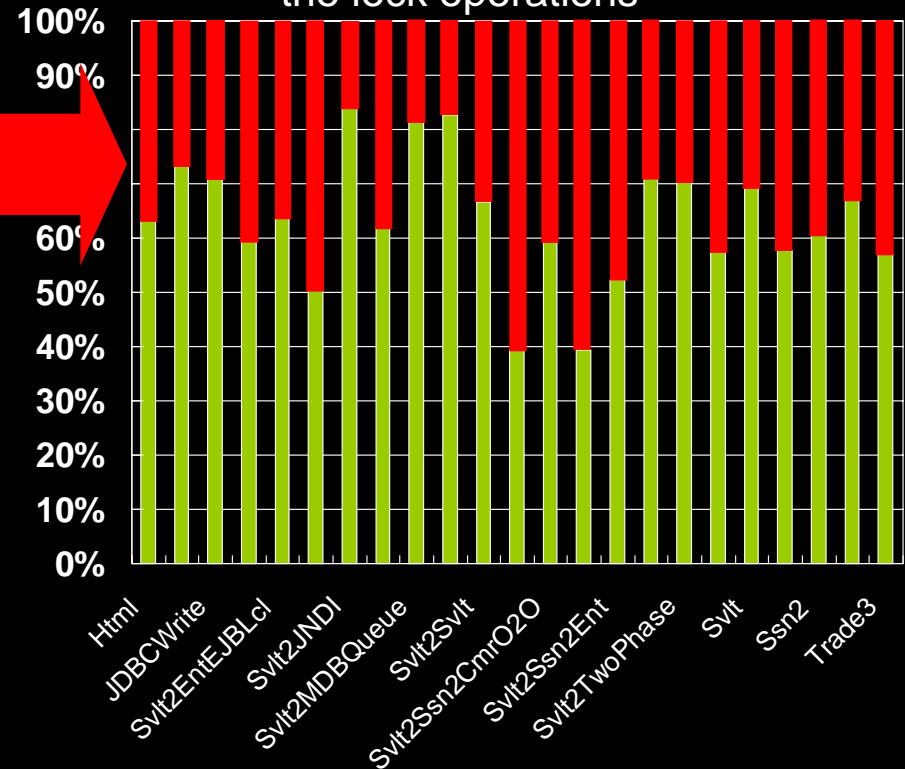
Lock reservation (Kawachiya et al., OOPSLA '02)

- **The first owner thread** tends to continue to acquire the lock for a given object

Opportunity for optimization

- Lock reservation reserves the object for the first owner
 - No cost for the same owner
 - **Needs a thin lock otherwise**
- How often we need a thin lock:
 - Maximum 61%
 - Average 37%

The occurrence of first repetition during all of the lock operations



What is the first repetition?

- **First repetition:** how often *the same thread tends to continue to lock a given object.*

	First repetition
A A A A A A A A	$7/8 = 88\%$
A A A A B B B B	$3/8 = 38\%$
A A B B A A B B	$1/8 = 13\%$

This example shows eight occurrences of lock operations, where the thread, *A* or *B*, owned a given object.

Our approach

- Improves the performance of those locks whose owner may change but continue to hold for a while by exploiting *their owner locality*

What is the owner locality?

- **Owner locality:** *How often the same thread tends to continue to lock a given object until another thread attempts to lock the same object.*

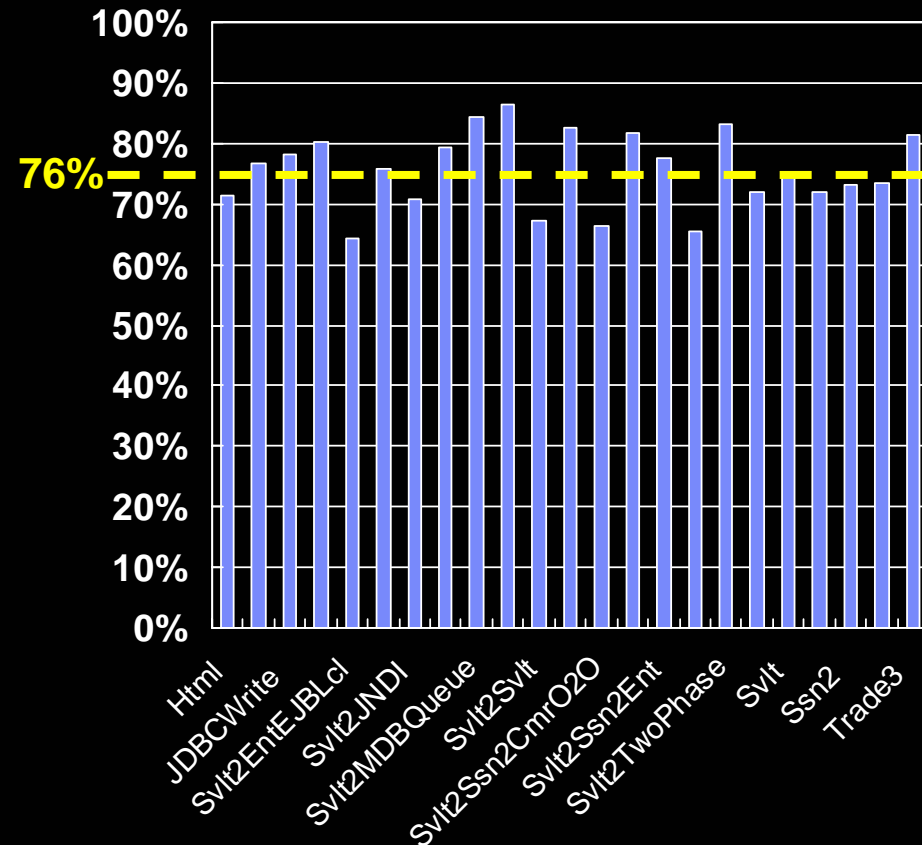
	Owner locality	First repetition
A A A A A A A A	$7/8 = 88\%$	$7/8 = 88\%$
A A A A B B B B	$6/8 = 75\%$	$3/8 = 38\%$
A A B B A A B B	$4/8 = 50\%$	$1/8 = 13\%$

This example shows eight occurrences of lock operations, where the thread, *A* or *B*, owned a given object.

Owner locality of Java applications

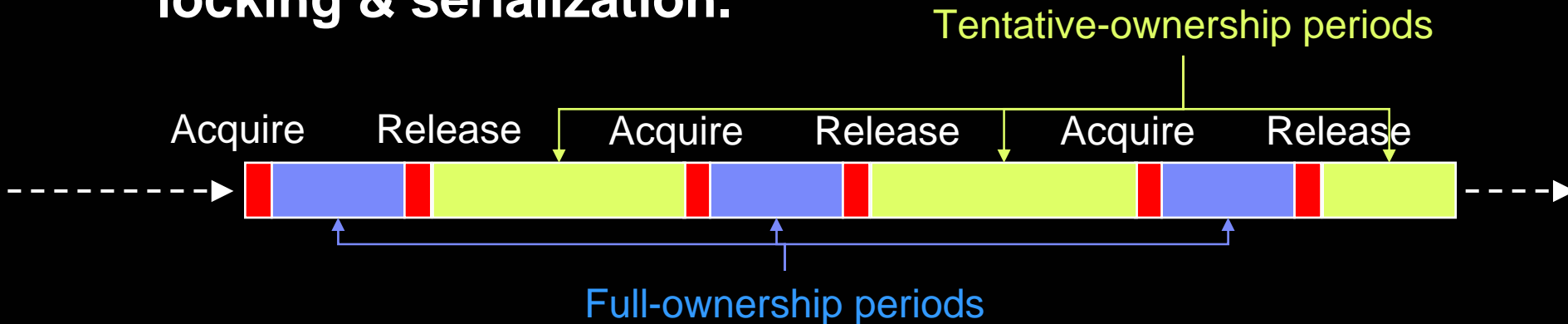
- **Java applications show high owner locality:**
 - Maximum 86%
 - Average 76%

Owner locality of the lock operations owned by multiple owners



How we exploit the owner locality

- **Extend the ownership after the release of the lock**
 - A thread continues to have *tentative ownership* of the object after it releases the lock.
 - The thread establishes *full ownership* when it acquires the lock.
- **Only the new owner pays the costs of memory locking & serialization.**

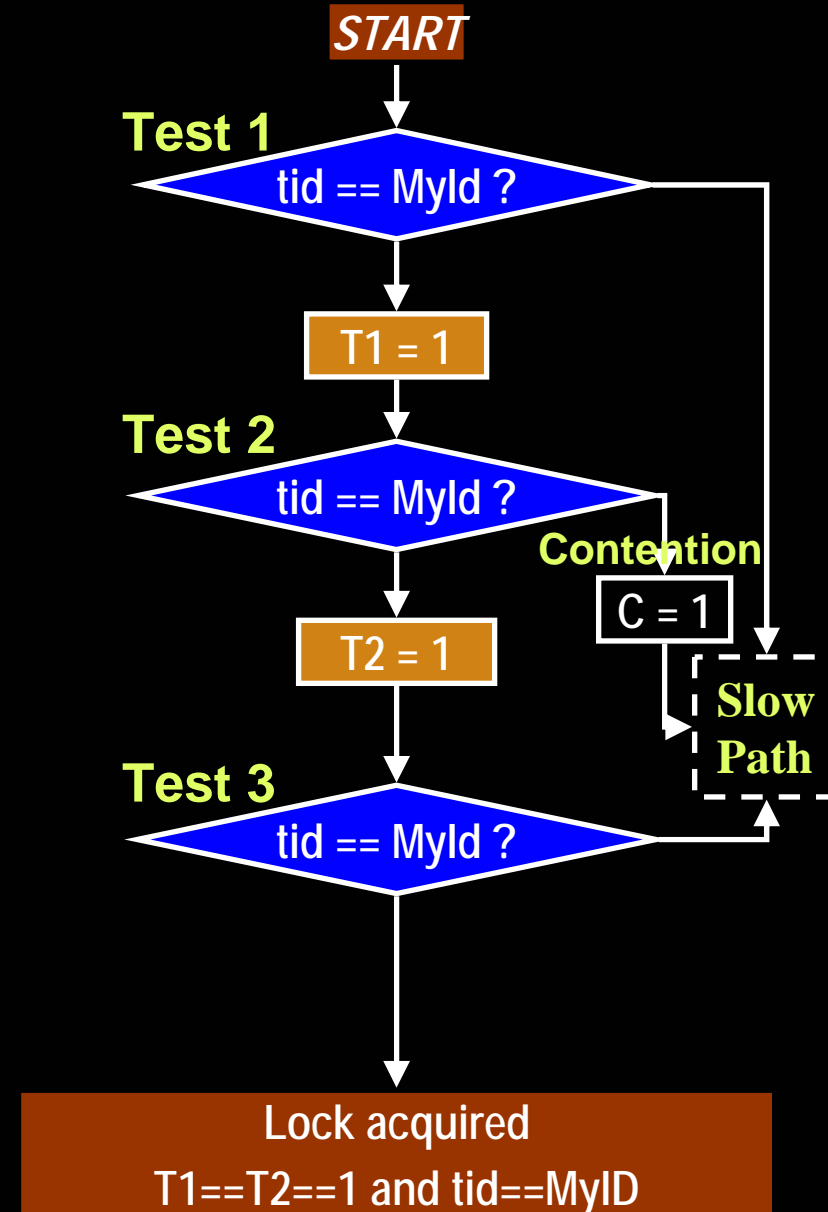


Our algorithm


- Uses a **thread ID field** for the ownership and **three flags** for the status of the lock acquisition
 - In the header of each object
- **Consists of two paths:**
 - **The fast path** – executed while a thread has the tentative ownership of the lock
 - **The slow path** – executed when a thread has no tentative ownership or interrupts the fast path

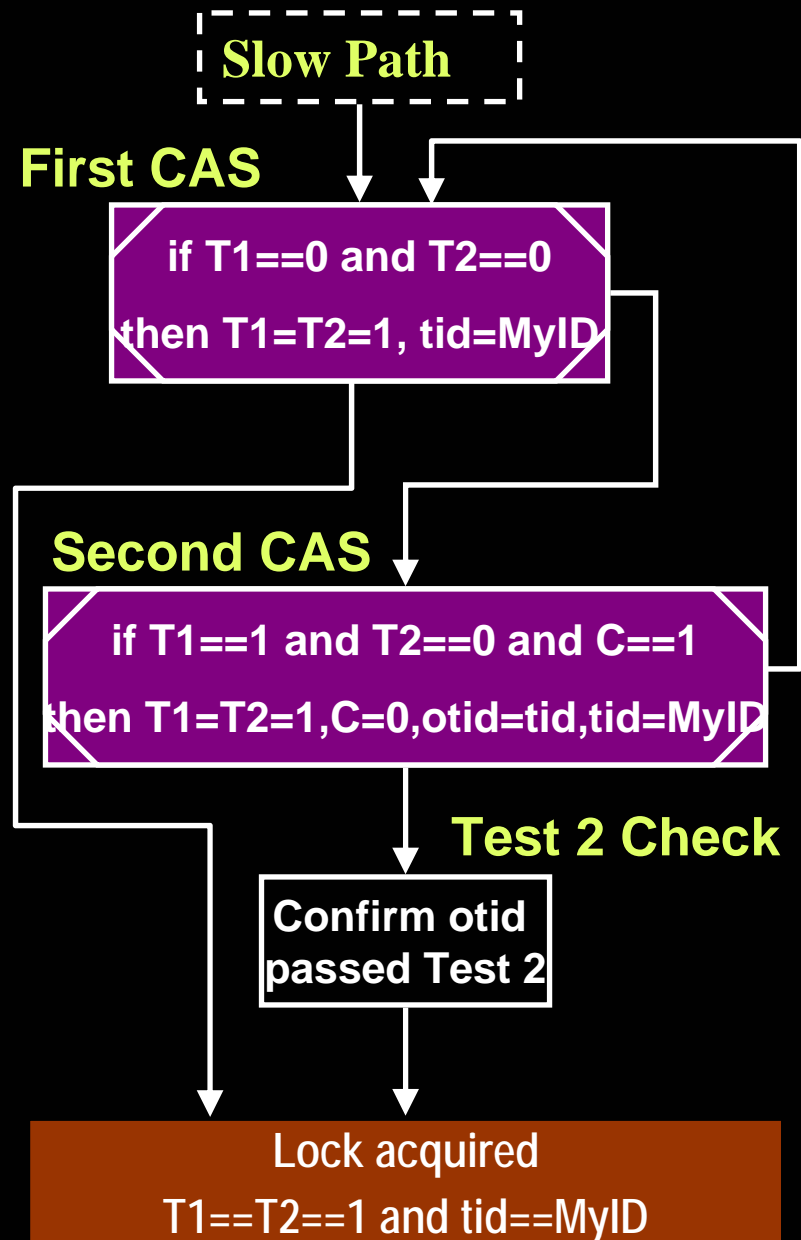
The fast path in our algorithm

- **Non-atomic**
 - A thread can interrupt the fast path and change the ownership.
- **Tests of tentative ownership**
 - The **tid** shows the ownership.
- **Flags T1 and T2 to inhibit the owner change**
- **If all of the tests succeeded, the lock is acquired.**



The slow path in our algorithm

- Compare-and-swap operation (CAS) changes the ownership.
 -  denotes a CAS.
- The **first CAS** acquires the lock when the lock is tentatively owned by another thread but not acquired.
- The **second CAS** detects the contention, clears it, and acquires the lock.

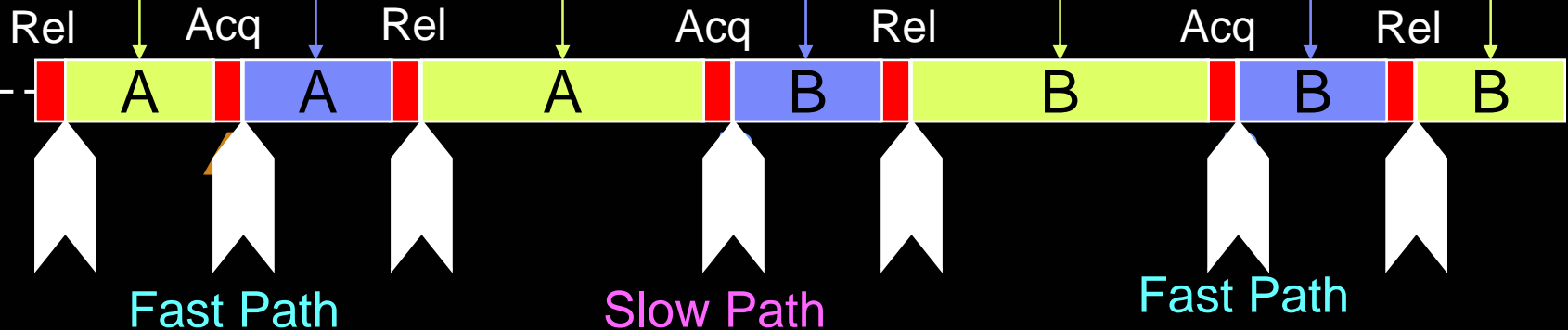


When the fast and slow paths acquire a lock

- **The fast path** establishes full ownership.
- **The slow path** changes ownership.

Tentative-ownership periods

Full-ownership periods



Verification of our algorithm

- **Used a software model checker, called Spin, to verify the following properties of our algorithm:**
 - Mutual exclusion: only one thread acquires the lock
 - Liveness property: if a thread requests a lock, then it will eventually get that lock
- **Spin generates all of the possible cases within the limited number of threads to test the algorithm using the assertions at selected program points.**

Evaluation of our algorithm

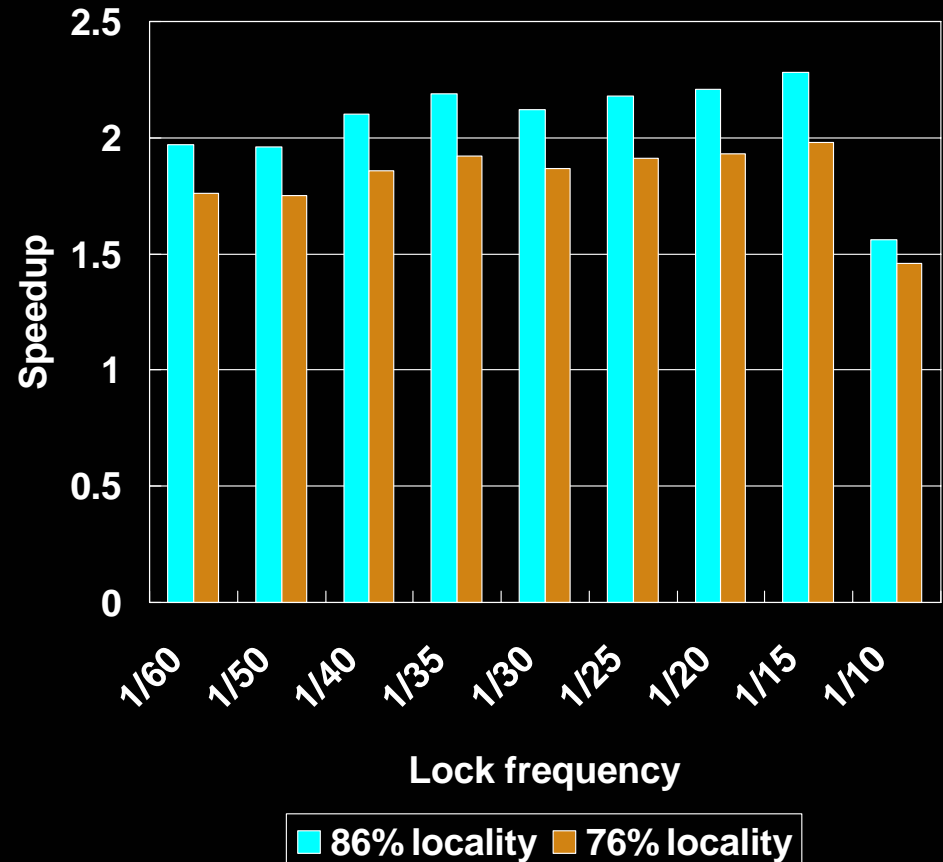
- **Create hand-optimized x86 assembly code to simulate a critical region with different lock frequencies:**
 - Lock frequency of **1/30** indicates that a lock operation occurs once per 30 heap accesses
- **Step 1: measured the performance of the fast path of our algorithm and thin lock on Xeon processors**
 - Assuming 100% owner locality
- **Step 2: estimate the performance for the following two owner localities:**
 - 76% as the average case
 - 86% as the highest case

Assembly code

```
while (true) {  
    lock  
    load  
    load  
    load  
    load  
    store  
    :  
    :  
    unlock  
    load  
    load  
    load  
    load  
    store  
    :  
    :  
}
```

Results

- **Average speedup of our approach over thin lock for uncontended locks**
 - 2.1 x for 86% owner locality
 - 1.8 x for 76% owner locality



Conclusions

- **Our algorithm is efficient for uncontended locks even owned by multiple owners on an SMP**
 - During the extended ownership periods, no memory locking and serializing costs are required.
 - Small memory requirements

Questions?