Simple and Fast Biased Locks

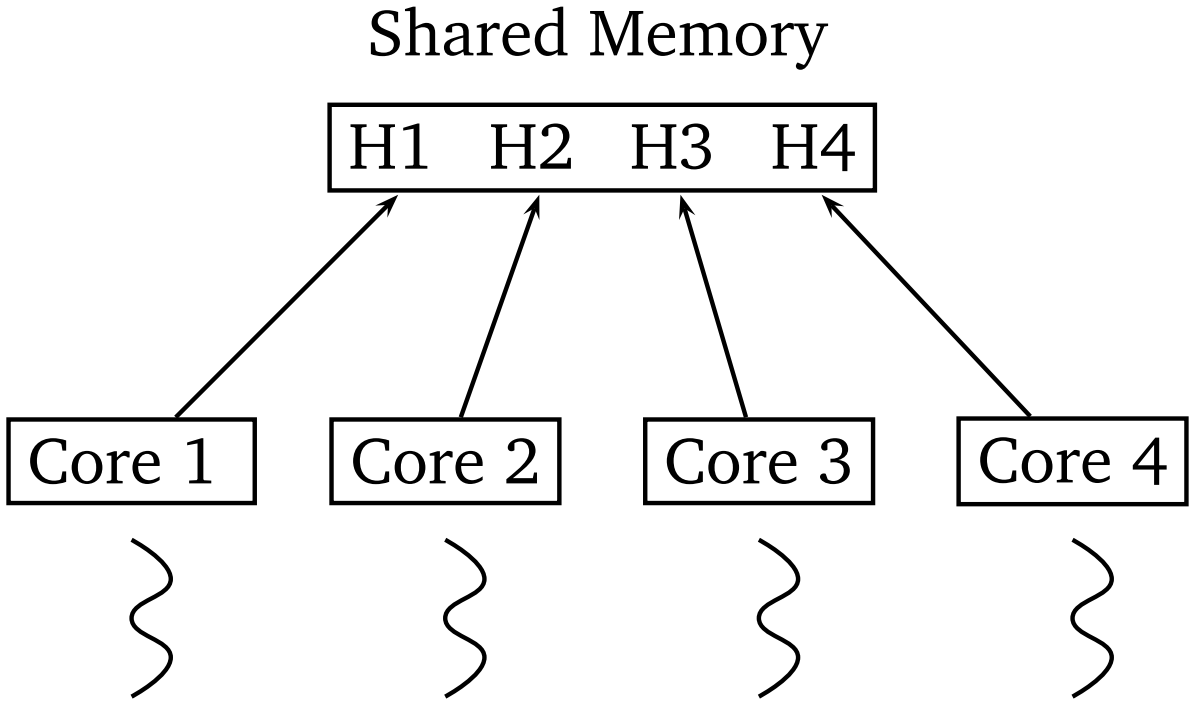
Nalini Vasudevan
Columbia University

Kedar Namjoshi
Bell Laboratories

Stephen Edwards
Columbia University

Motivation

- Packet Analyzer



Locking

```
flag[i] = 1;
while (flag[j] == 1) {}
/* critical section */
flag[i] = 0;
```

Locking

Problematic

```
flag[i] = 1;
while (flag[j] == 1) {}
/* critical section */
flag[i] = 0;
```

Peterson's algorithm

```
flag[i] = 1;
turn = j;
while (flag[j] == 1 && turn == j) {}
/* critical section */
flag[i] = 0;
```

Peterson's algorithm

```
flag[i] = 1;
turn = j;
while (flag[j] == 1 && turn == j) {}
/* critical section */
flag[i] = 0;
```

Still problematic

Peterson's algorithm

```
flag[i] = 1;
turn = j;
fence(); /* force other threads to see flag and turn */
while (flag[j] && turn == j) {} /* spin */
/* critical section */
fence(); /* make visible changes made in critical section */
flag[i] = 0;
```

Peterson's algorithm

```
flag[i] = 1;
turn = j;
fence(); /* force other threads to see flag and turn */
while (flag[j] && turn == j) {} /* spin */
/* critical section */
fence(); /* make visible changes made in critical section */
flag[i] = 0;
```

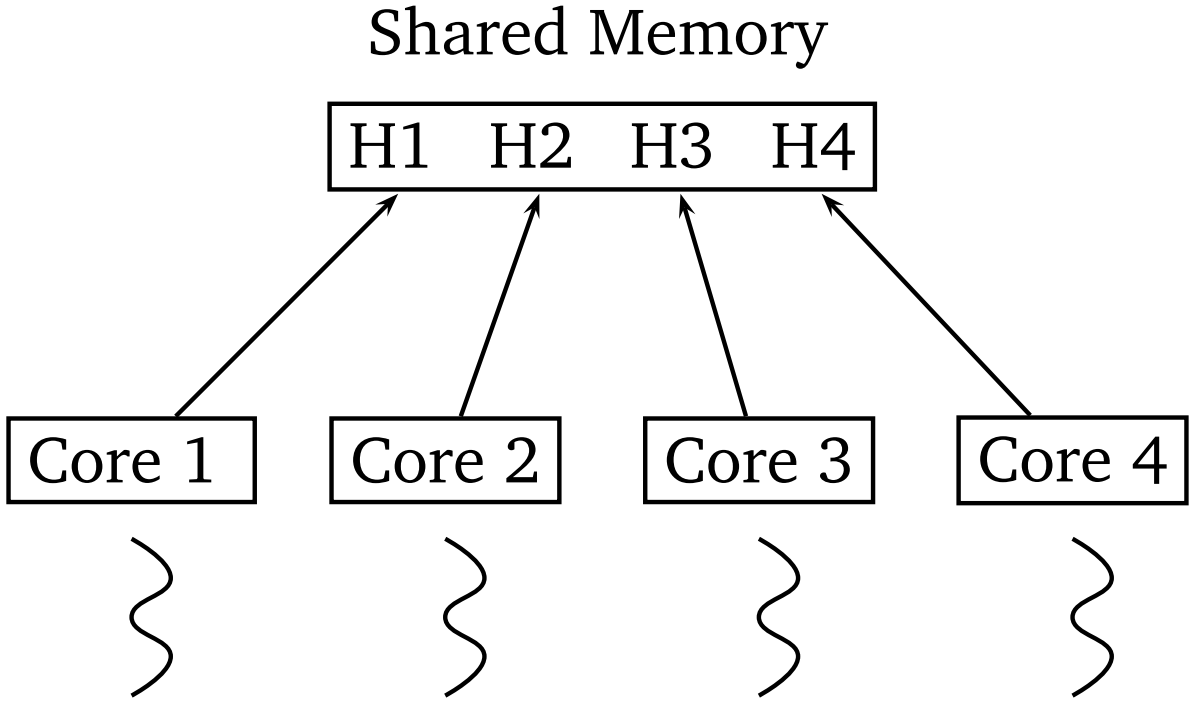
Applies to two processes only

N-process locks

```
bool success;
do {
    while (lck == 1) {} /* wait */
    success = compare_and_swap(&lck, 0, 1);
  } while (!success);
}
/* critical section */
lck = 0;
```

Motivation

- Packet Analyzer



Peterson's algorithm

```
flag[i] = 1;
turn = j;
fence(); /* force other threads to see flag and turn */
while (flag[j] && turn == j) {} /* spin */
/* critical section */
fence(); /* make visible changes made in critical section */
flag[i] = 0;
```

- Two process algorithm

Dominant process lock

- Contends with other processes using Peterson's algorithm

```
peterson_lock();
/* critical section */
peterson_unlock();
```

Non-Dominant process lock

- Contends with the dominant processes using Peterson's algorithm
- Contends with other non-dominant processes using a normal n-process lock.

```
lockN();
peterson_lock();
/* critical section */
peterson_unlock();
unlockN();
```

Biased Lock = 2-lock + n-lock

Locking

Unlocking

```
if (this_thread_id == owner)
   lock2();
else {
   lockN();
   lock2();
}
```

```
if (this_thread_id == owner)
   unlock2();
else {
   unlock2();
   unlockN();
}
```

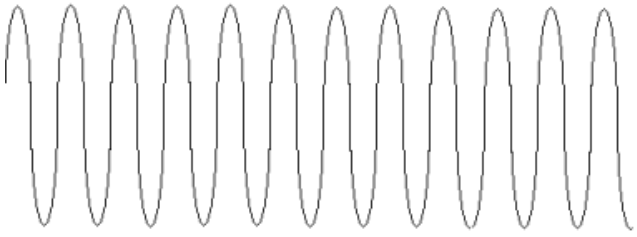
The problem

```
flag[i] = 1;
turn = j;
fence(); /* force other threads to see flag and turn */
while (flag[j] && turn == j) {} /* spin */
/* critical section */
fence(); /* make visible changes made in critical section */
flag[i] = 0;
```

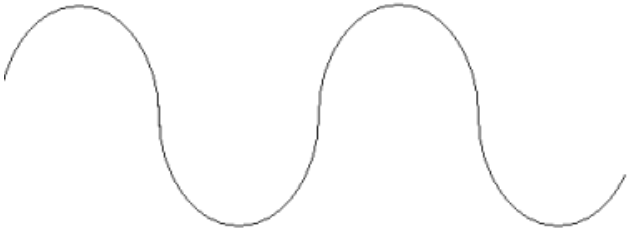
- Need *fences*

Asymmetric locks

- Eliminate fences in the dominant process



Dominant process

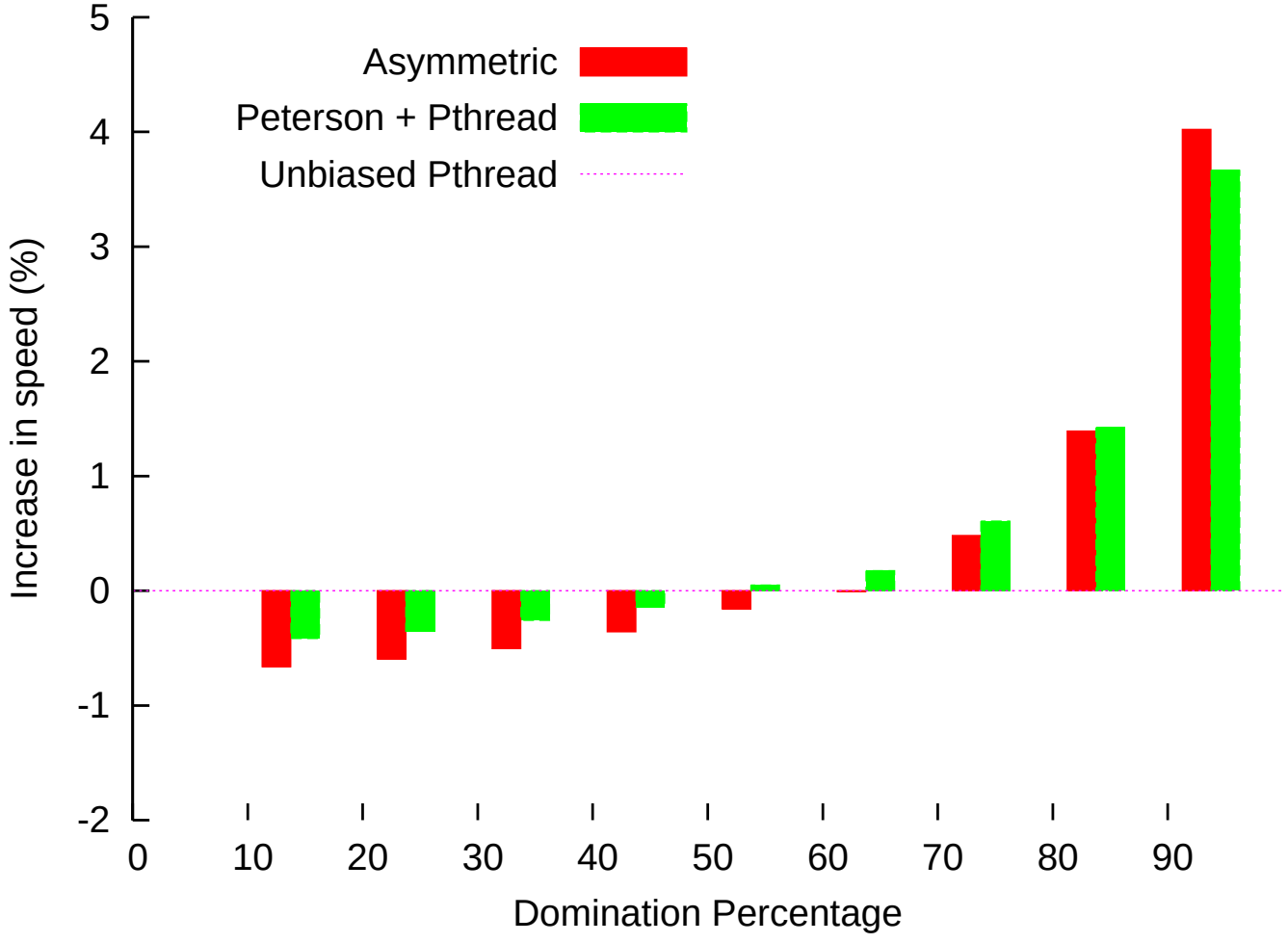


Non-dominant process

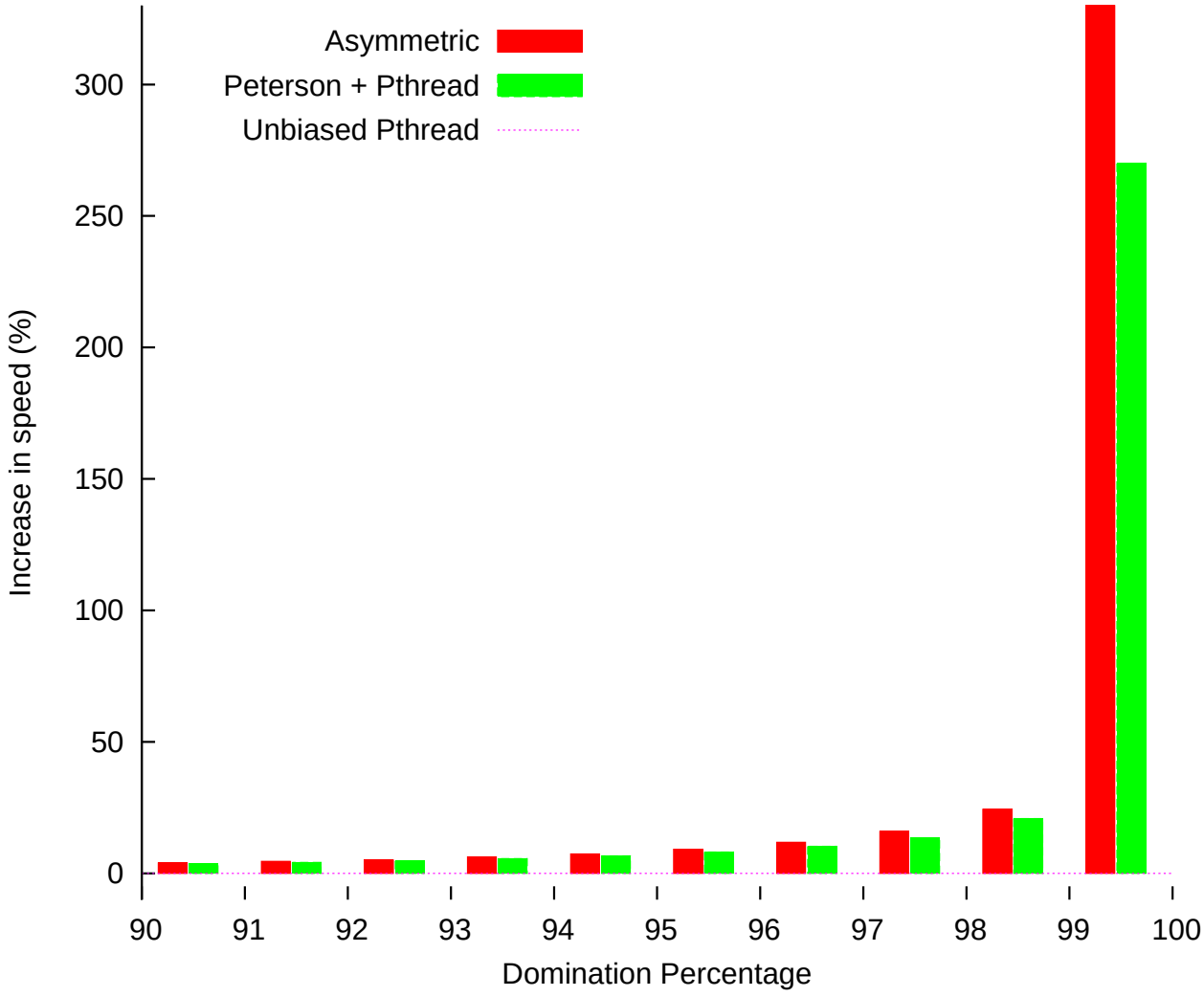
```
while (grant) {} /* wait */
/* critical section */
if (request) {
   request = 0;
   fence();
   grant = 1;
}
```

```
lockN();
request = 1;
while (grant == 0) {} /* wait */
/* critical section */
fence();
grant = 0;
unlockN();
```

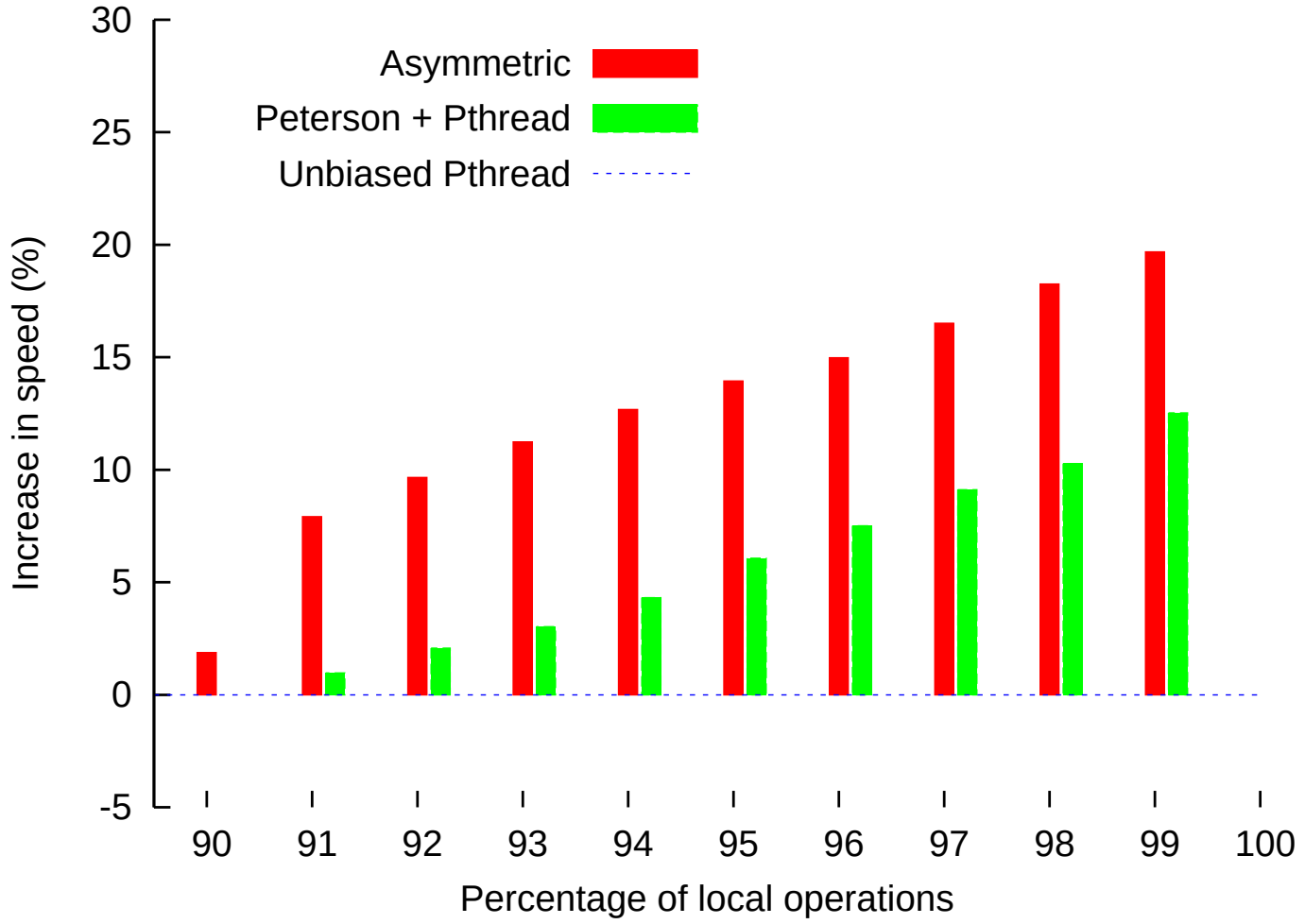
Performance



Performance - Higher domination



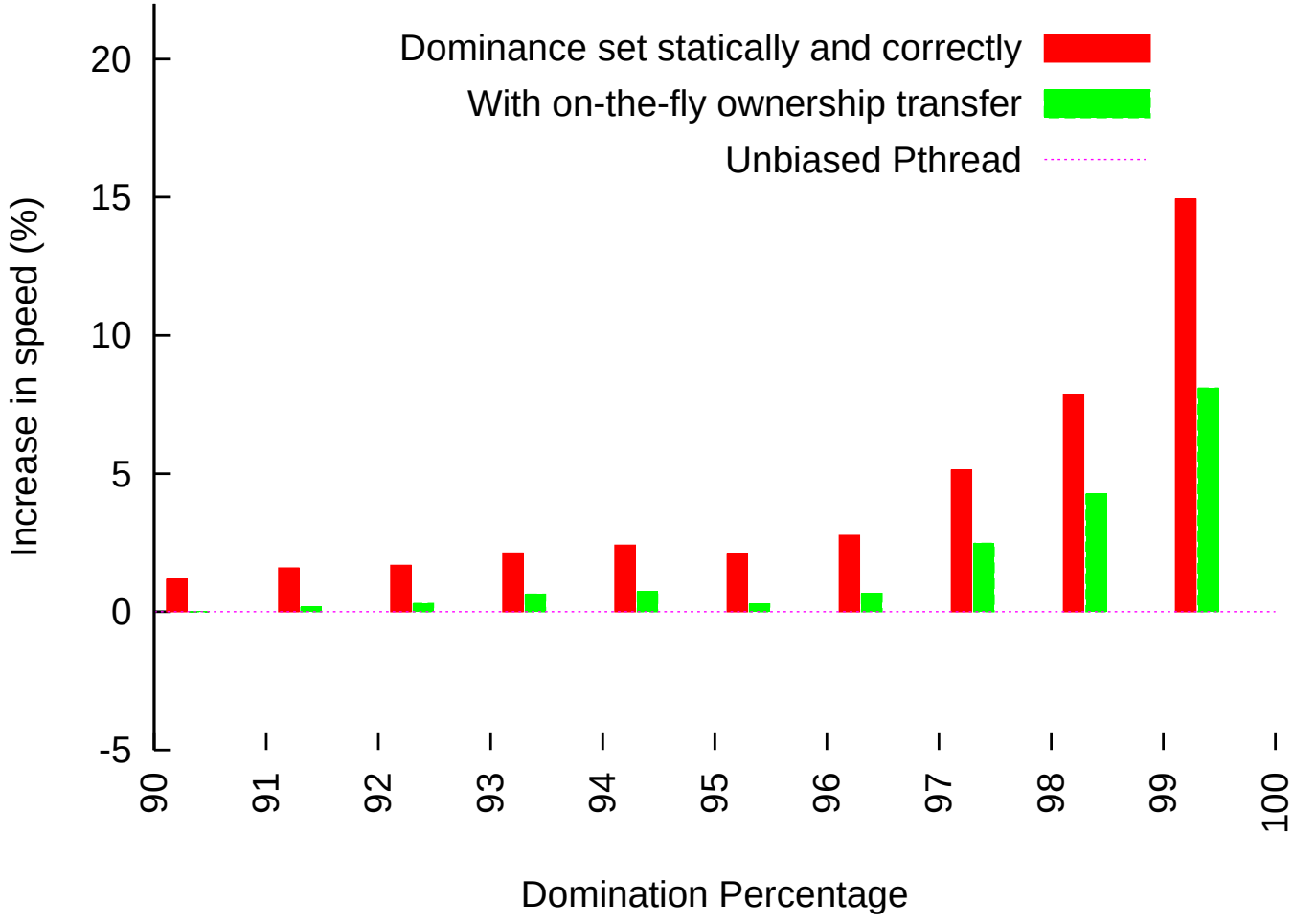
Performance - Packet analyzer



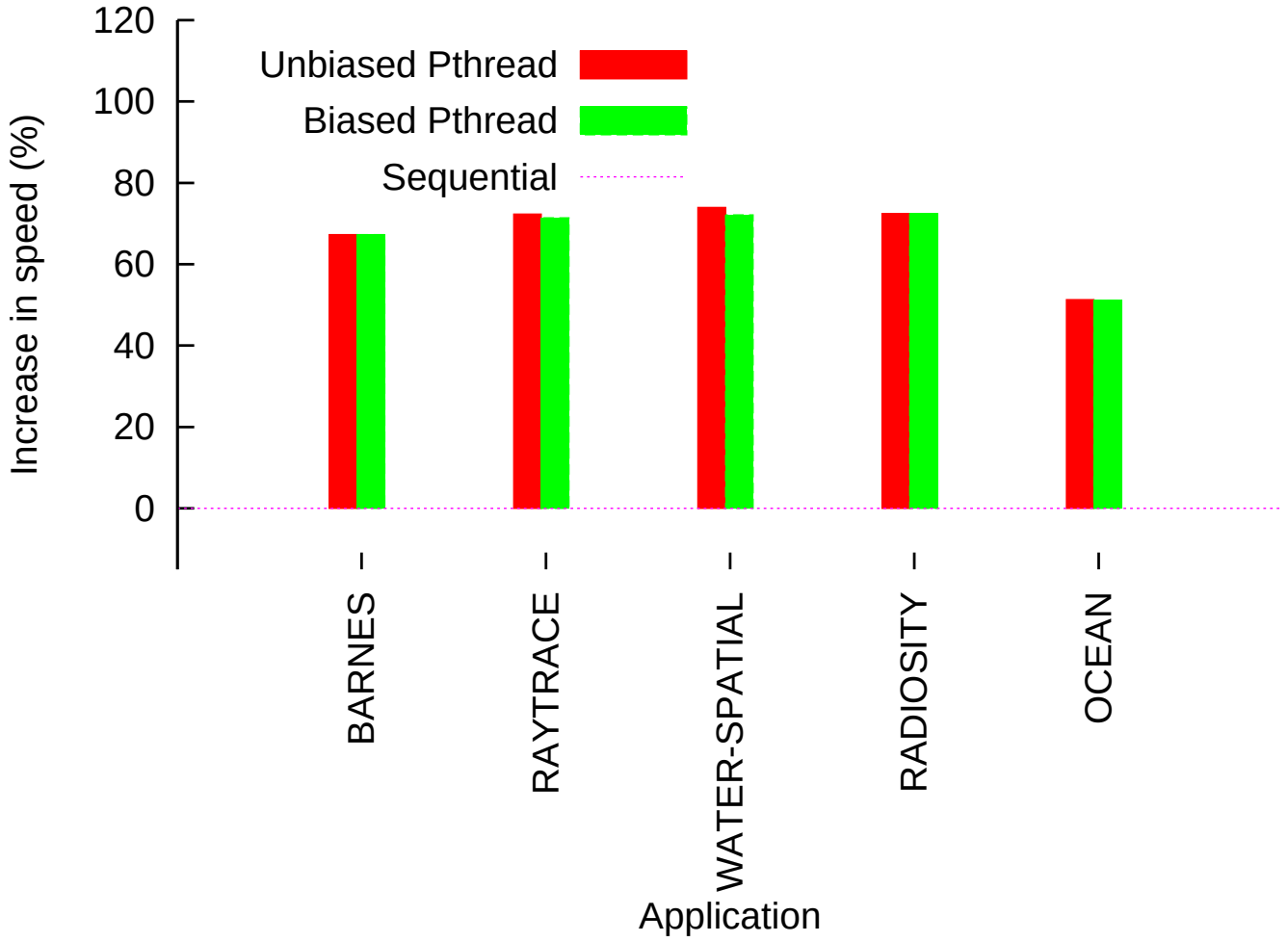
Bias Transfer

- Dynamic scheme for transferring bias
- Based on the frequency
- Only one thread can be declared dominant at any time

Performance - Bias Transfer



Performance - SPLASH



Conclusions

- Simple algorithms for constructing biased locks
- Verified using the SPIN model checker
- Implemented as a library
- Extended it to read-write locks
- Good performance when high dominance
- Future work: different architectures