A lock-less shared memory barrier without atomic operations

Part 1: Theory

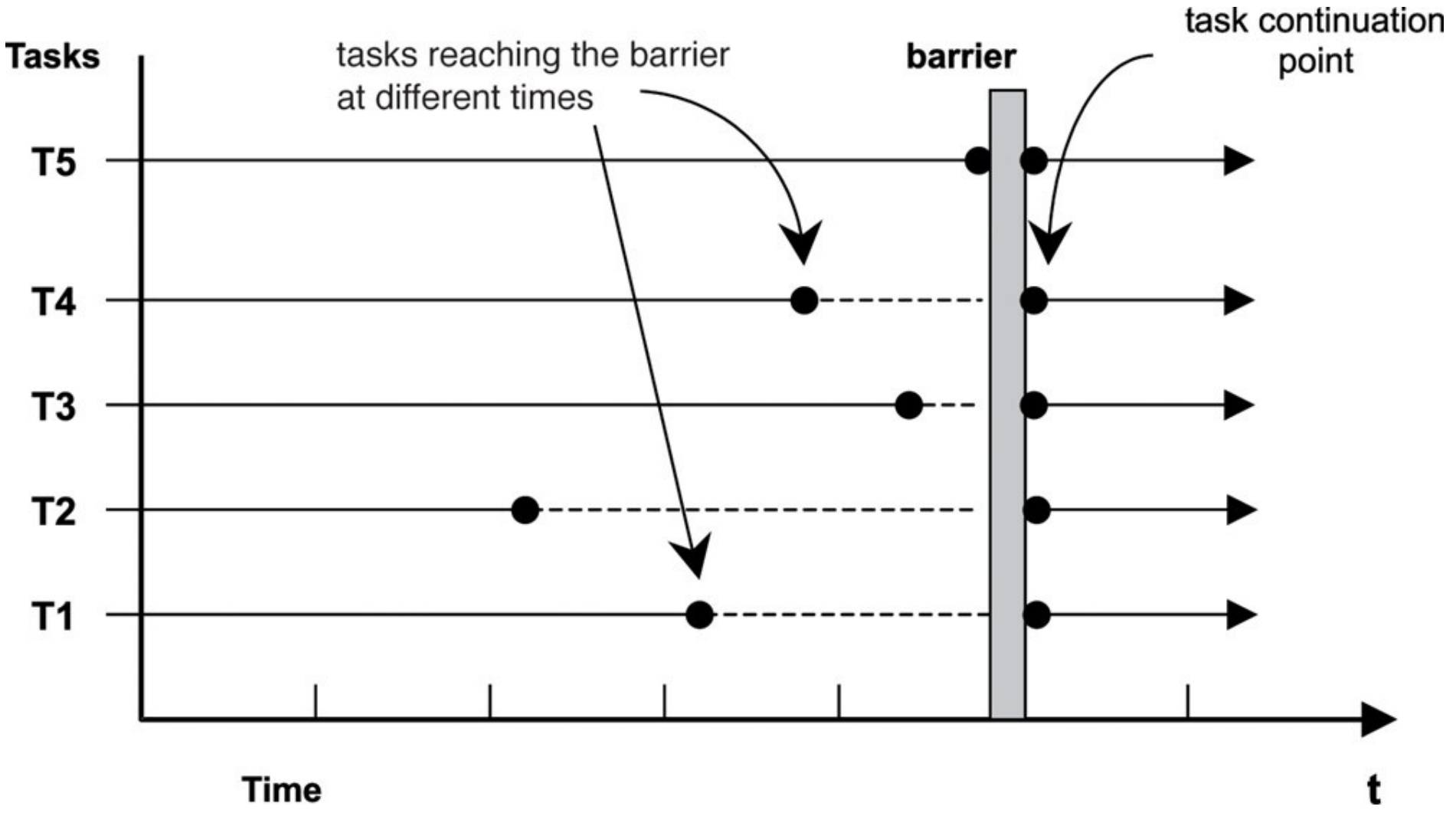
Ronny Brendel

Tutors: Marcus Völp, Sascha Klüppelholz

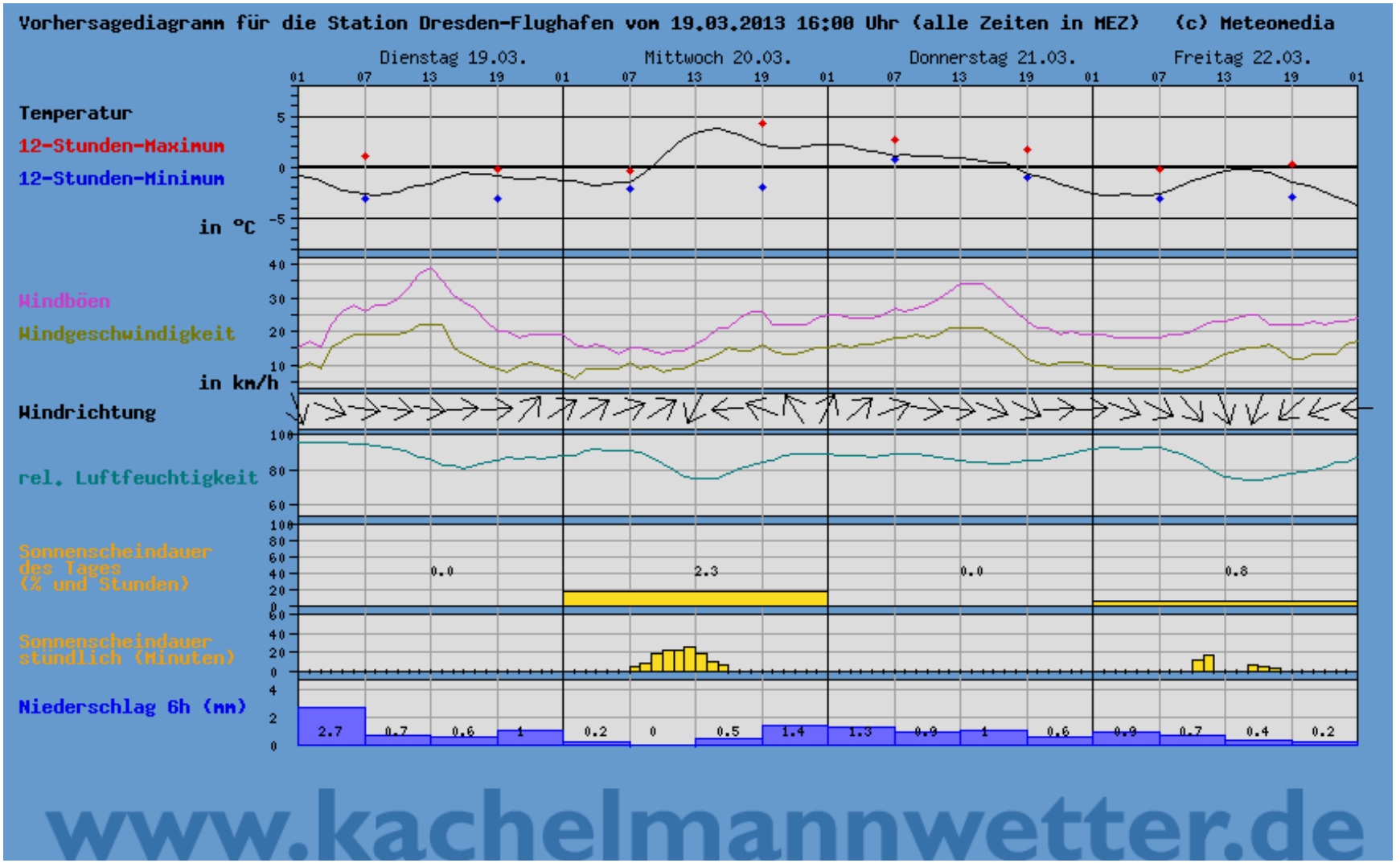
2013-03-22

Table of content

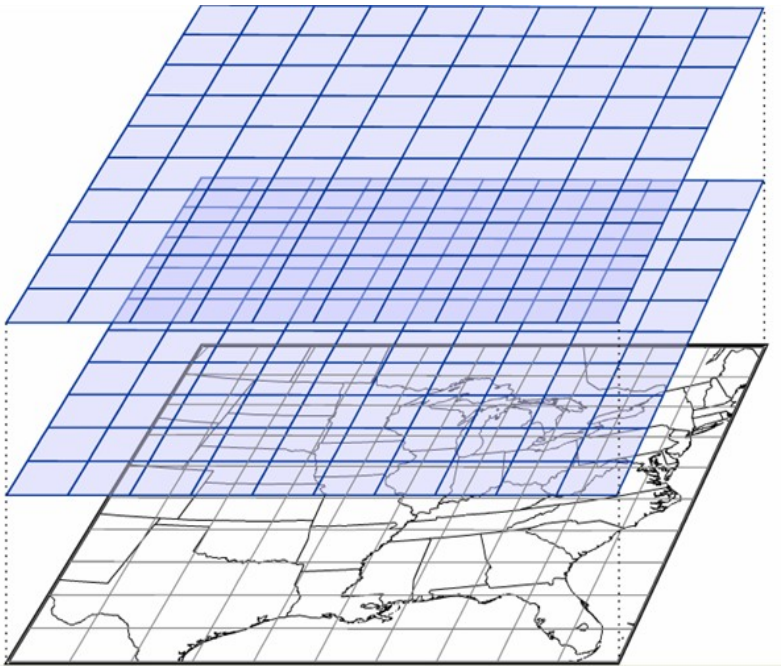
Introduction

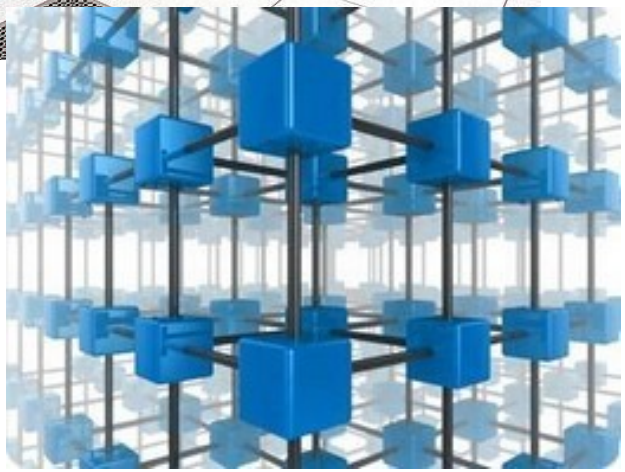


Introduction

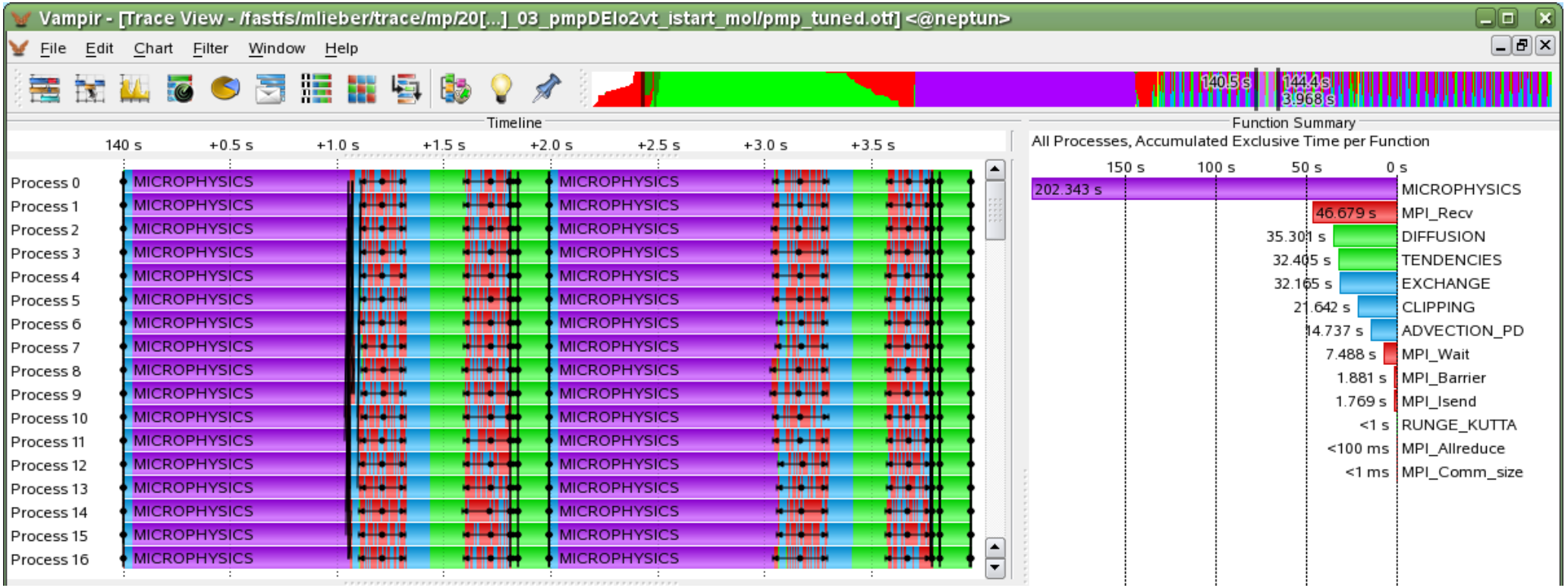


Introduction





Introduction



Introduction

- Usual Implementations include
  - Atomic increment
  - Hierarchical approaches
    - e.g. combining tree

see sources [1]

Mc Guire's idea

- Scalable locking is hard
  - Relies on cache coherency
- Locking is slow

- Revisit existing algorithms/protocols for potential optimization through probabilistic approaches
  - e.g. lock/mutex, barrier

see sources [2] abstract

Mc Guire's idea

```
1  shared: barrier:=1
2  local: me:=primes[index], other:=primes[len-1-index]
3
4  do {
5      if barrier≢0 mod me {
6          barrier := barrier * me
7      }
8  } while barrier≢0 mod productOfAllPrimes
9
10 do {
11     if barrier≡0 mod other {
12         barrier := barrier / other
13     }
14 } while barrier≠1
```

Mc Guire's idea – Problem 1

```
1  shared: barrier:=1
2  local: me:=primes[index], other:=primes[len-1-index]
3
4  do {
5      if barrier≢0 mod me {
6          barrier := barrier * me
7      }
8  } while barrier≢0 mod productOfAllPrimes
9
10 do {
11     if barrier≡0 mod other {
12         barrier := barrier / other
13     }
14 } while barrier≠1
```

Mc Guire's idea – Problem 2

```
4  do {
5      if barrier ≠ 0 mod me {
6          barrier := barrier * me
7      }
8  } while barrier ≠ 0 mod productOfAllP
9
```

Mc Guire's idea – Problem 2

```
4 do {
5     if barrier≠0 mod me {
6         barrier := barrier * me
7     }
8 } while barrier≠0 mod productOfAllP
9
```

time	prime		
	2	3	5
1	read 1	read 1	
2	write 2		
3			read 2
4		write 3	
5	read 3		
6			write 10
7	read 10		
8	write 20		

20 = 2 * 2 * 5

Mc Guire's idea – Problem 3

```
1  shared: barrier:=1
2  local: me:=primes[index], other:=primes[len-1-index]
3
4  do {
5      if barrier≢0 mod me {
6          barrier := barrier * me
7      }
8  } while barrier≢0 mod productOfAllPrimes
9
10 do {
11     if barrier≡0 mod other {
12         barrier := barrier / other
13     }
14 } while barrier≠1
```

The extended protocol

```
1  shared: entry:=0, exit:=0, left:=false
2  local: copy, me:=2^threadIndex, full:=∑_{i=0}^{numThreads-1} 2^i
3
4  if left = false {
5
6      do {
7          copy := entry
8          if copy & me = 0 {
9              copy  := copy | me
10             entry := copy
11         }
12     } while copy ≠ full and left = false
13
14     left := true
15     exit := 0
16
17 } else if left = true {
18
19     do {
20         copy := exit
21         if copy & me = 0 {
22             copy := copy | me
23             exit := copy
24         }
25     } while copy ≠ full and left = true
26
27     left  := false
28     entry := 0
29
30 }
```

Correctness

- Correctness = deadlock freedom + consistency

- Deadlock

  $\neg[\ \Box\Diamond(\text{loc\_i}=14) \wedge \Box\Diamond(\text{loc\_i}=27)\ ]$

- Consistency

  $\neg\Diamond(\ \text{loc\_i}=14\ \&\ \text{loc\_i}=27\ \&\ i\neq j\ )$

Correctness

- Correctness = deadlock freedom + consistency

- Deadlock

  $\neg[\ \Box\Diamond(\text{loc\_i}=14) \wedge \Box\Diamond(\text{loc\_i}=27)\ ]$ ✓

- Consistency

  $\neg\Diamond(\ \text{loc\_i}=14\ \&\ \text{loc\_i}=27\ \&\ i\neq j\ )$ ✓

Correctness

- Intuitive argument for deadlock freedom: If all threads have looped at least once, at least one thread has been successfully committed to entry/exit. Repeat n times. The barrier is complete.

```
do {
    copy := entry
    if copy & me = 0 {
        copy  := copy | me
        entry := copy
    }
} while copy ≠ full and left = false
```

Quantitative analysis

- Determine expensive operations
  - Shared memory access

```
1  shared: entry:=0, exit:=0, left:=false
2  local: copy, me:=2^threadIndex, full:=∑_{i=0}^{numThreads-1} 2^i
3
4  if left = false {
5
6      do {
7          copy := entry
8          if copy & me = 0 {
9              copy  := copy | me
10             entry := copy
11         }
12     } while copy ≠ full and left = false
13
14     left := true
15     exit := 0
```

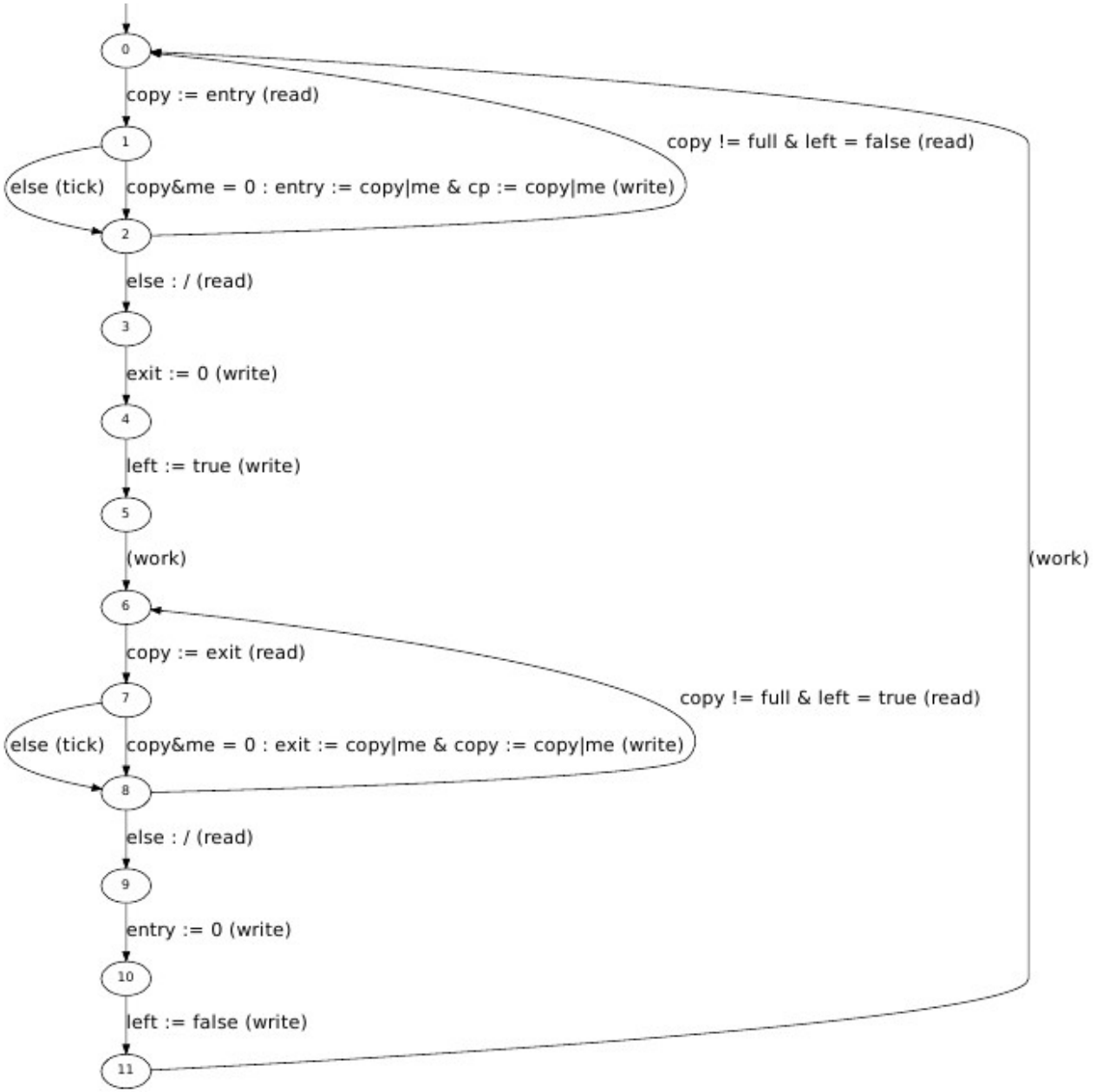
```
17 } else if left = true {
18
19     do {
20         copy := exit
21         if copy & me = 0 {
22             copy := copy | me
23             exit := copy
24         }
25     } while copy ≠ full and left = true
26
27     left  := false
28     entry := 0
29
30 }
```

Quantitative analysis

- Determine expensive operations
  - Shared memory access

```
1  shared: entry:=0, exit:=0, left:=false
2  local: copy, me:=2^threadIndex, full:=∑_{i=0}^{numThreads-1} 2^i
3
4  if left = false {
5
6      do {
7          copy := entry
8          if copy & me = 0 {
9              copy  := copy | me
10             entry := copy
11         }
12     } while copy ≠ full and left = false
13
14     left := true
15     exit := 0
17 } else if left = true {
18
19     do {
20         copy := exit
21         if copy & me = 0 {
22             copy := copy | me
23             exit := copy
24         }
25     } while copy ≠ full and left = true
26
27     left  := false
28     entry := 0
29
30 }
```

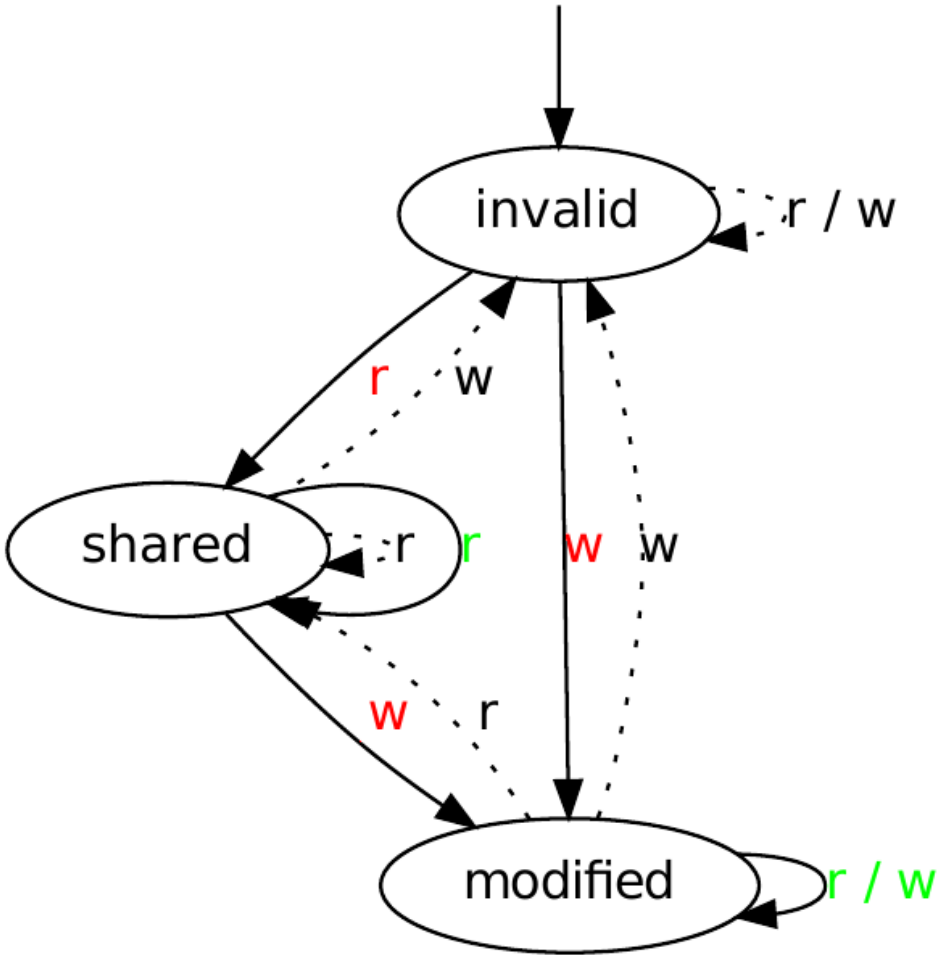
Quantitative analysis



Quantitative analysis

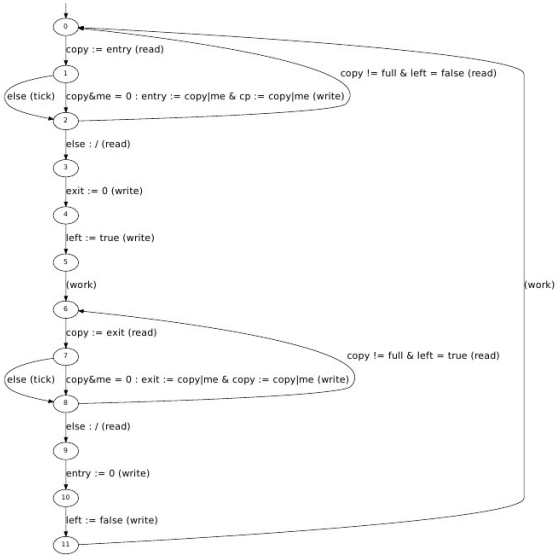
Expensive: red

Cheap: green

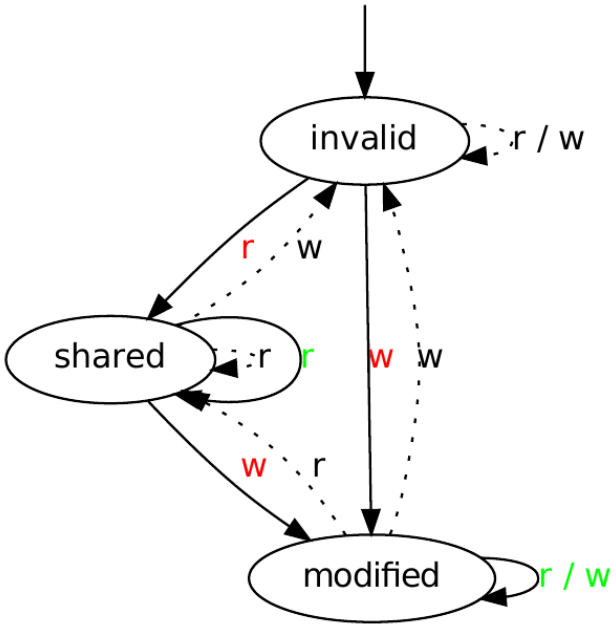


Quantitative analysis

- Per thread:



x



- Interleave all threads
- Too large for four threads already → future work

Quantitative analysis

- (1) How long does it take for one thread to pass the barrier?
- (2) How long does it take for all threads to pass the barrier?
- (3) If one thread has left the barrier, how long does it take until all of them complete the barrier?

Quantitative analysis

- (1) How long does it take for one thread to pass the barrier?

$$P_{=?}\left[\Diamond_{\leq t} \bigvee_{1\leq i\leq n} loc_i \geq 5\right]$$

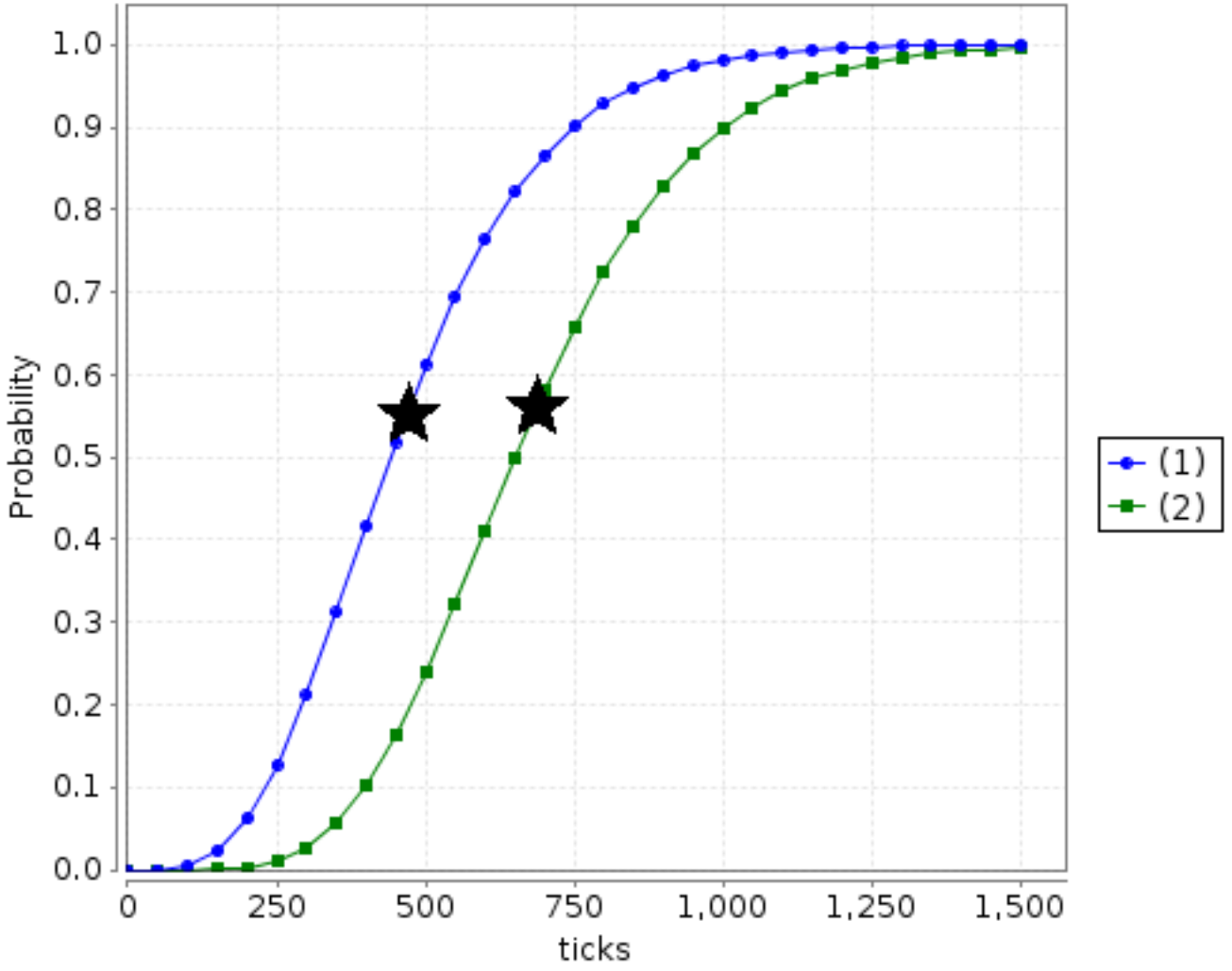
- (2) How long does it take for all threads to pass the barrier?

$$P_{=?}\left[\Diamond_{\leq t} \bigwedge_{1\leq i\leq n} loc_i \geq 5\right]$$

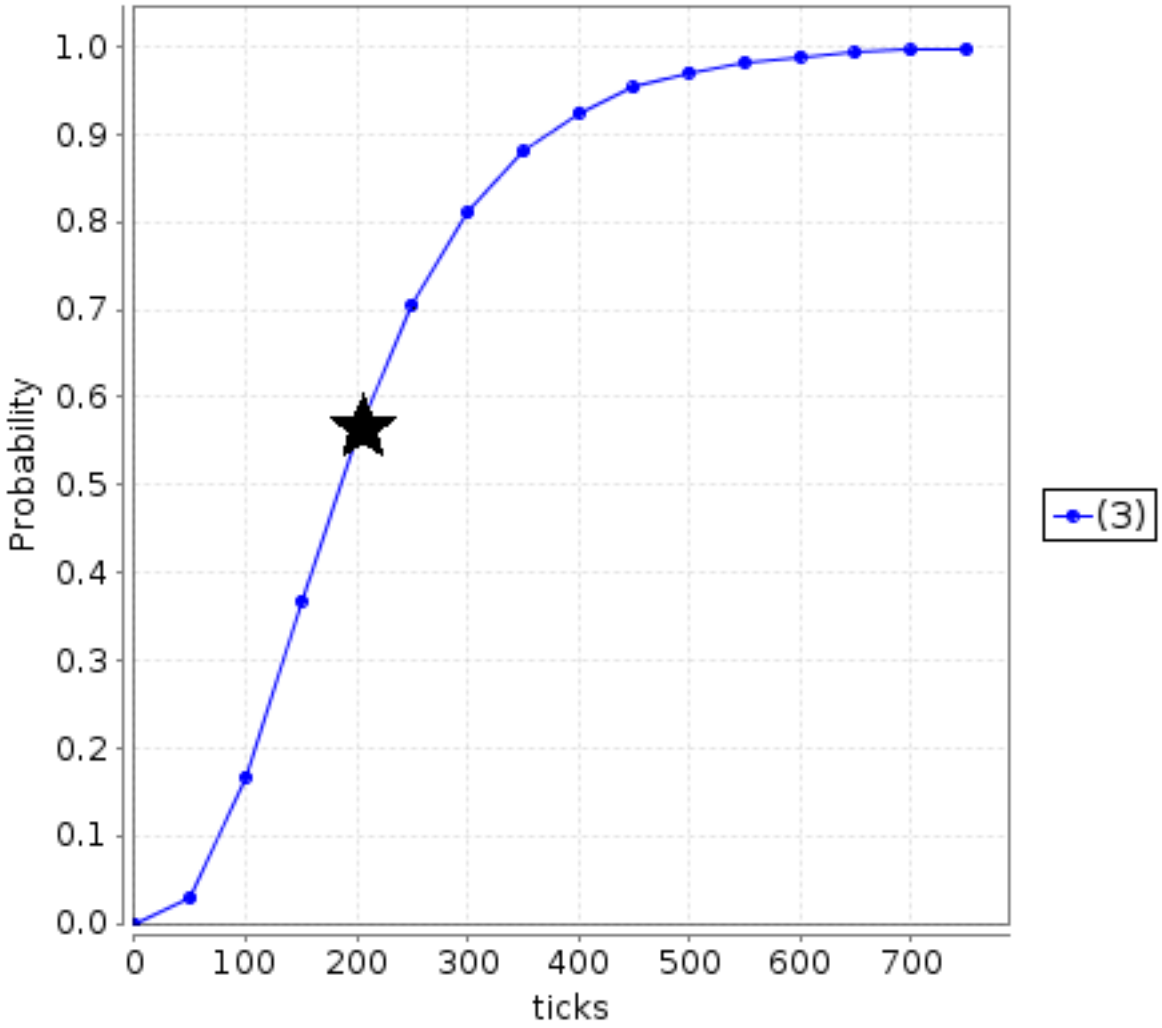
- (3) If one thread has left the barrier, how long does it take until all of them complete the barrier?

$$ClrP_{=?}\left[\Diamond_{\leq t} \bigwedge_{1\leq i\leq n} loc_i \geq 5, \bigvee_{1\leq i\leq n}(loc_i \geq 5 \wedge \bigwedge_{1\leq j\leq n, i\neq j} loc_j < 5)\right]$$

Quantitative analysis



Quantitative analysis



Conclusion

- New barrier protocol
  - + No locks
  - + No atomic operations
  - + Correct
  - Potentially quicker than existing protocols
  - Brief quantitative analysis
- Modeled cache behaviour

Future work

- Model more threads

  → extend quantitative analysis
- Optimize
- Work around the 32 thread limit

- Evaluate potential of the protocol for distributed memory systems

Sources

- [1] Evaluation of publicly available Barrier-Algorithms and Improvement of the Barrier-Operation for large-scale Cluster-Systems with special Attention on InfiniBandTM Networks

  http://archiv.tuchemnitz.de/pub/2005/0073/data/diploma.pdf

- [2] Probabilistic write copy select, Paper,

  In 13th Real-Time Linux Workshop, pages 195–206, Oct. 2011

- [3] PRISM, Website, 13-03-019

  http://www.prismmodelchecker.org

- [4] SPIN, Website, 13-01-08

  http://spinroot.com

- [5] Waiting for Locks How Long Does It Usually Take?, Paper

  http://dx.doi.org/10.1007/978-3-642-32469-7_4

Thank you!

Slides and report are available at
http://automaton2000.com/barrier-slides.pdf
http://automaton2000.com/barrier-report.pdf

A lock-less shared memory barrier without atomic operations

Part 2: Implementation and measurement

Ronny Brendel

Tutors: Marcus Völp, Sascha Klüppelholz

2013-03-22

Table of content

Benchmark

- Straightforward translation to C

```
1  shared: entry:=0, exit:=0, left:=false
2  local: copy, me:=2^threadIndex, full:=∑_{i=0}^{numThreads-1} 2^i
3
4  if left = false {
5
6      do {
7          copy := entry
8          if copy & me = 0 {
9              copy  := copy | me
10             entry := copy
11         }
12     } while copy ≠ full and left = false
13
14     left := true
15     exit := 0
16
17 } else if left = true {
18
19     do {
20         copy := exit
21         if copy & me = 0 {
22             copy := copy | me
23             exit := copy
24         }
25     } while copy ≠ full and left = true
26
27     left  := false
28     entry := 0
29
30 }
```

Setup

- New protocol vs GNU OpenMP, PThreads:
- Spawn n threads and pin them to cores
- Repeatedly visit barrier
- Emulate work period by sleeping
- Measure the whole runtime of the program
- Conducted on: 64 thread / 32 core / 4 socket AMD Opteron 6274 board

Results

- No work period

thread count	OpenMP	PThreads	new
4	3.3s	3.5s	0.2s
8	3.3s	3.5s	0.3s
16	3.2s	3.7s	0.6s
32	5.0s	5.5s	1.4s

Results

- 200,000 cycles / 0.1 millisec work period

thread count	OpenMP	PThreads	new
4	5.1s	5.1s	4.6s
8	5.5s	5.5s	4.8s
16	5.9s	5.9s	5.1s
32	7.0s	7.0s	6.1s

Results

- Drawbacks:
  - Busy waiting
  - False-sharing strains cache, but only ~1% bandwidth is lost
  - There exist mitigation strategies
    - Sleep, mwait, pause [5]

Quantifying cache latency

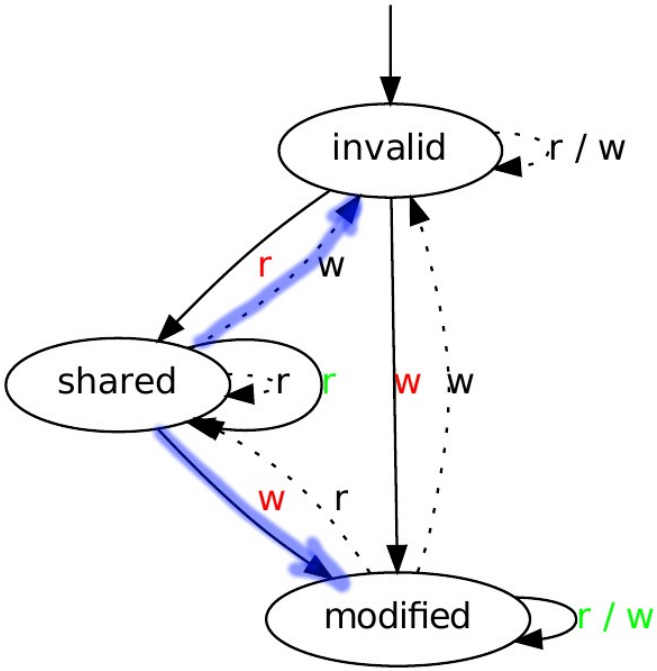
- Question: How does the number of to-be-invalidated cache line copies influence the latency of a shared read/write

- Answer leads to more precise quantitative models of cache-heavy algorithms

Setup

- One shared date
- All threads read the date
- First thread writes and measures the duration

- Conducted on:
  - AMD Opteron 6274 (Bulldozer)
  - Xeon L5630 (Sandybridge)



Results

- Measuring the write duration proves difficult
  - Out-of-order execution
  - Buffered memory operations
  - Measurement duration is very short: 50-400 cycles (30 of which are the measurement itself)
- Results vary between instant and very very long

Results

- Conclusion:
  - AMD: no obvious tendency / wild
  - Intel: pretty much constant
  - CPUs work in mysterious ways: It is hard to analyse and quantify

Results

- Surprising / interesting:
  - AMD: atomic operations duration depends on thread count regardless of what threads do
  - Intel: on one core (2 HT threads) invalidation still takes as much time as multiple cores do (not the case on AMD)
  - Allocating threads sparsely across cores/CPUs increases latency

Conclusion

- New barrier protocol performs well, even in the most straightforward implementation
- Quantifying cache coherency latency is hard
  - Assuming constant latency for shared read/write seems good

Future work

- Optimize the implementation
  - mwait/pause instructions
- Redo benchmarks
- Put more effort into quantifying cache coherency latency

Sources

- [1] GNU C Library, Website, 13-03-19

  http://www.gnu.org/software/libc
- [2] GNU OpenMP, Website, 13-03-19

  http://gcc.gnu.org/projects/gomp
- [3] Message Passing Interface, Website, 13-03-19

  http://mpi-forum.org
- [4] OpenMP, Website, 13-03-19

  http://openmp.org
- [5] Stack Overflow: How does x86 pause [..], Website, 13-03-19

  http://stackoverflow.com/questions/4725676/how-does-x86-pause-instruction-work-in-spinlock-and-can-it-be-used-in-other-sc

Thank you!

Slides and report are available at
http://automaton2000.com/barrier-slides.pdf
http://automaton2000.com/barrier-report.pdf