

Optimizing for Multicores

Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se

Optimizing for the memory system: What is the potential gain?

- Latency difference L1\$ and mem: $\sim 50x$
- Bandwidth difference L1\$ and mem: $\sim 20x$
- Execute from L1\$ instead from mem \implies 50-150x improvement
- At least a factor 2-4x is within reach

Optimizing for cache performance

- Keep the active footprint small
- Use the entire cache line once it has been brought into the cache
- Fetch a cache line prior to its usage
- Let the CPU that already has the data in its cache do the job
- ...



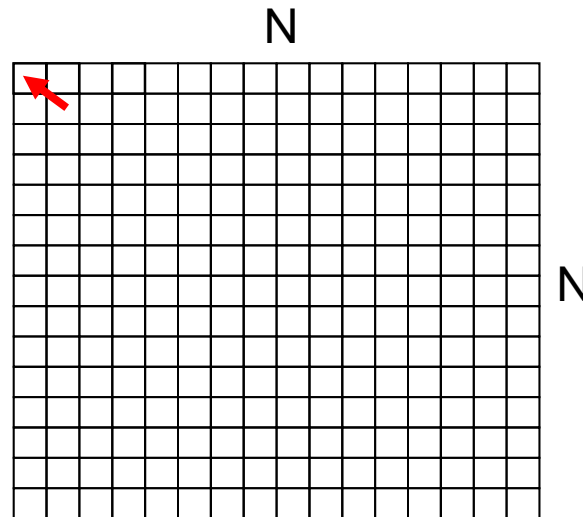
Final cache lingo slide

- **Miss ratio:** What is the likelihood that a memory access will miss in a cache?
- **Miss rate:** D:o per time unit, e.g. per-second, per-1000-instructions
- **Fetch ratio/rate*):** What is the likelihood that a memory access will cause a fetch to the cache [including HW prefetching]
- **Fetch utilization*):** What fraction of a cacheline was used before it got evicted
- **Writeback utilization*):** What fraction of a cacheline written back to memory contains dirty data
- **Communication utilization*):** What fraction of a communicated cacheline is ever used?

*) This is Acumem-ish language

What can go Wrong? A Simple Example...

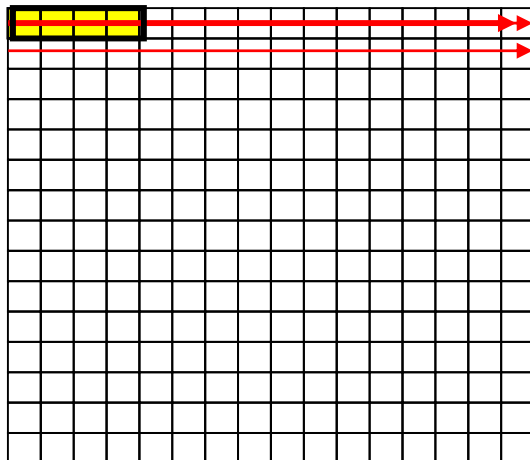
Perform a diagonal copy 10 times



Example: Loop order

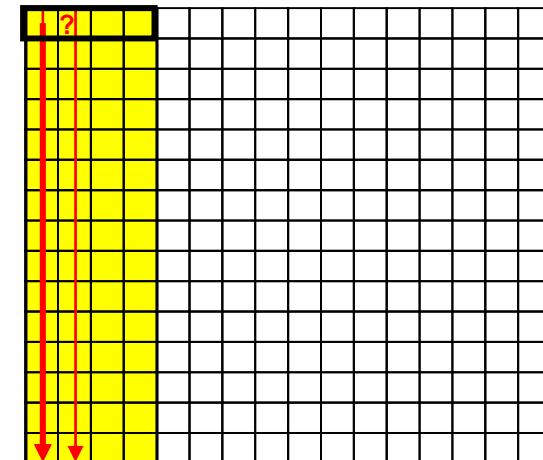
//Optimized Example A

```
for (i=1; i<N; i++) {  
  for (j=1; j<N; j++) {  
    A[i][j]= A[i-1][j-1];  
  }  
}
```

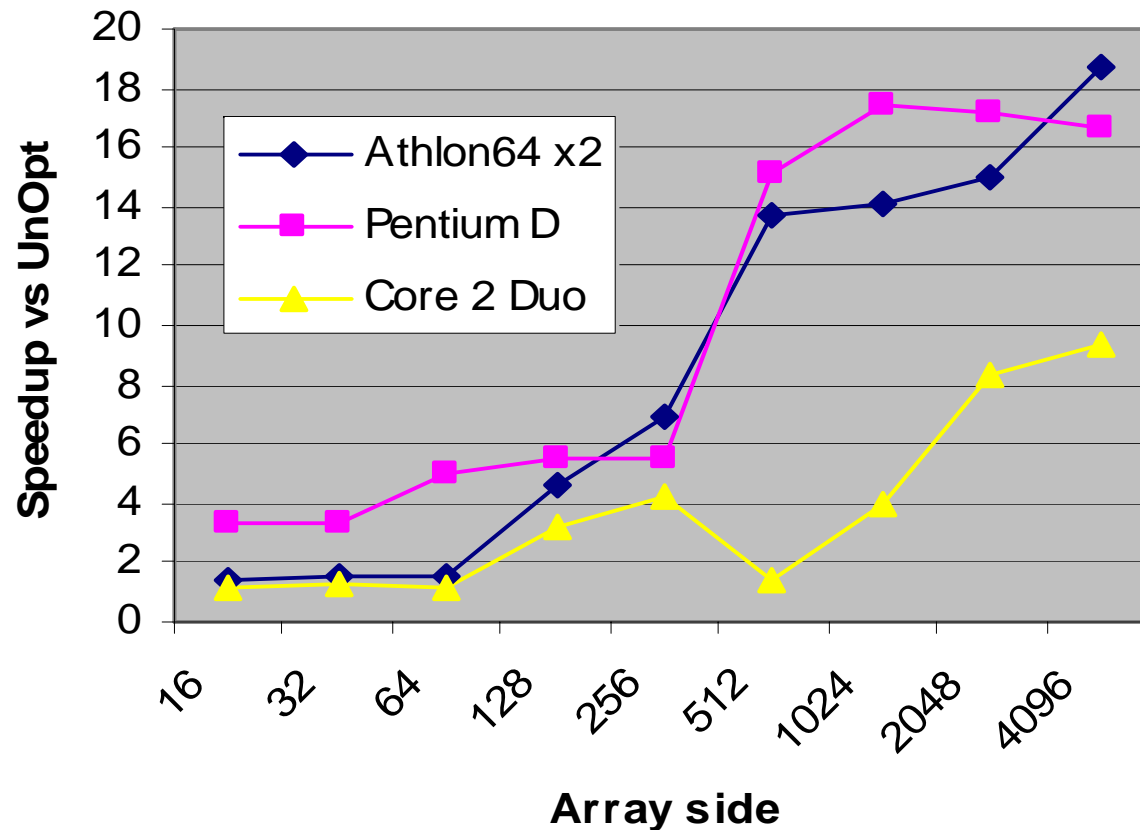


//Unoptimized Example A

```
for (j=1; j<N; j++) {  
  for (i=1; i<N; i++) {  
    A[i][j] = A[i-1][j-1];  
  }  
}
```



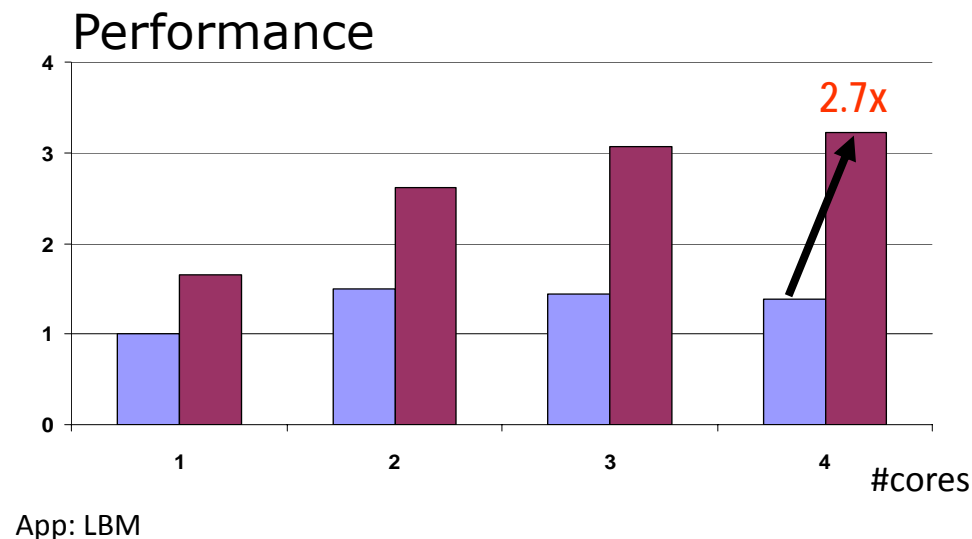
Performance Difference: Loop order



Demo Time!

ThreadSpotter

Example 1: The Same Application Optimized



Demo Time!

LBM:
[Original code](#)

Optimization can be rewarding, but costly...

- ✱ Require expert knowledge about MC and architecture
- ✱ Weeks of wading through performance data

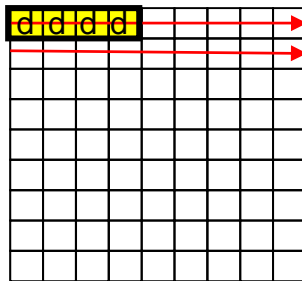
➔ This fix required one line of code to change

OPT 8

Example: Sparse data usage

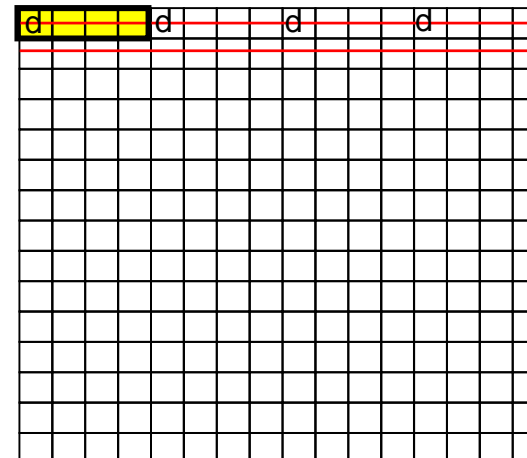
//Optimized Example A

```
for (i=1; i<N; i++) {
    for (j=1; j<N; j++) {
        A_d[i][j]= A_d[i-1][j-1];
    }
}
```



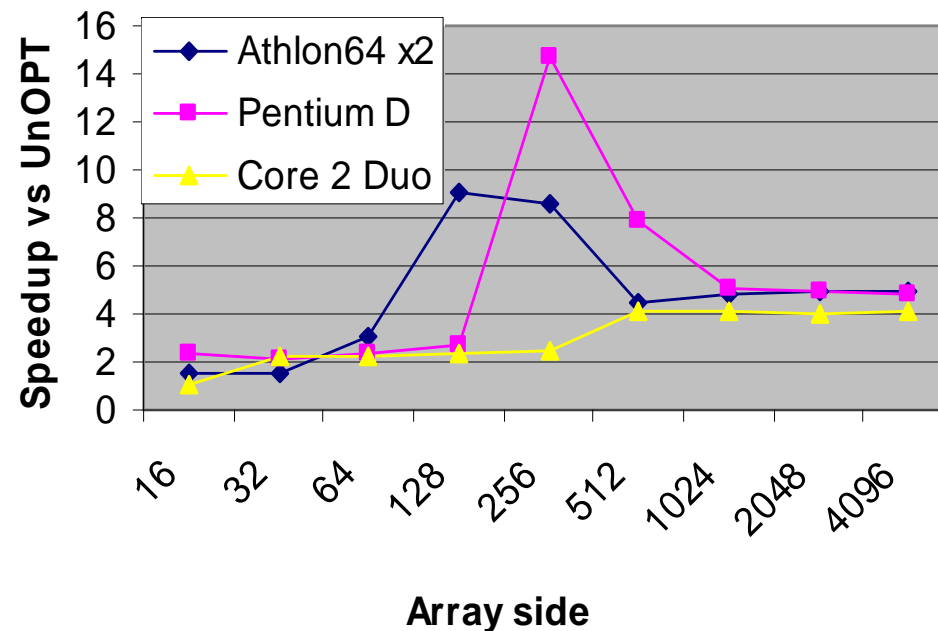
//Unoptimized Example A

```
for (i=1; i<N; i++) {
    for (j=1; j<N; j++) {
        A[i][j].d = A[i-1][j-1].d;
    }
}
```

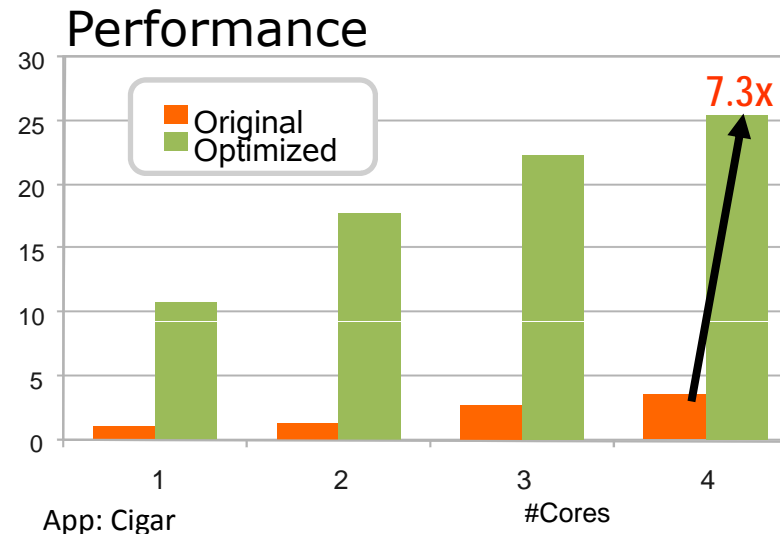


```
struct vec_type
{
    char a;
    char b;
    char c;
    char d;
};
```

Performance Difference: Sparse Data



Example 2: The Same Application Optimized



Looks like a perfect scalable application!
Are we done?

→ Duplicate one data structure

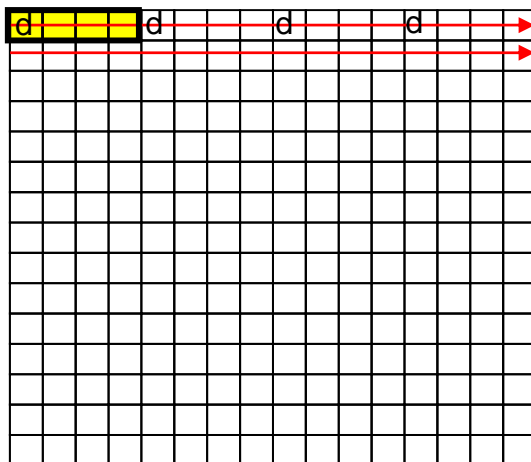
Demo Time!

Cigar
[Original code](#)

Example: Sparse data allocation

```
sparse_rec sparse [HUGE];

for (int j = 0; j < HUGE; j++)
{
    sparse[j].a = 'a'; sparse[j].b = 'b'; sparse[j].c = 'c'; sparse[j].d = 'd'; sparse[j].e = 'e';
    sparse[j].f1 = 1.0; sparse[j].f2 = 1.0; sparse[j].f3 = 1.0; sparse[j].f4 = 1.0; sparse[j].f5 = 1.0;
}
```



```
struct sparse_rec
{
    // size 80B
    char a;
    double f1;
    char b;
    double f2;
    char c;
    double f3;
    char d;
    double f4;
    char e;
    double f5;
};
```

```
struct dense_rec
{
    //size 48B
    double f1;
    double f2;
    double f3;
    double f4;
    double f5;
    char a;
    char b;
    char c;
    char d;
    char e;
};
```

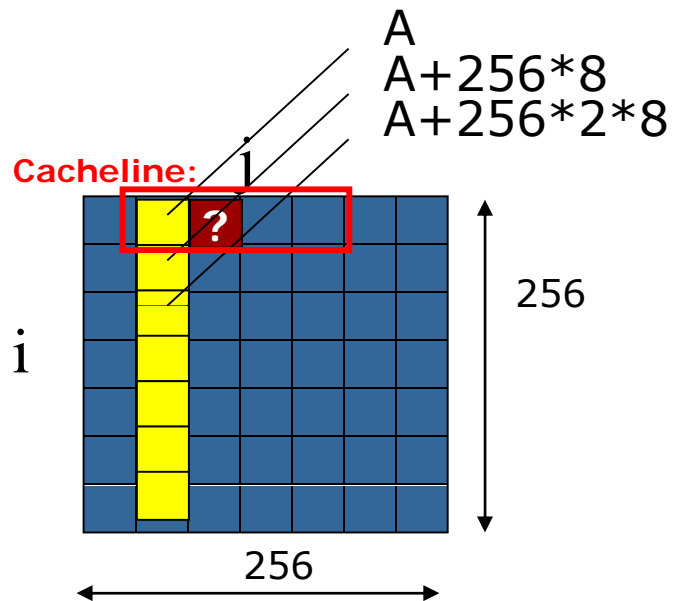
Speedup: ~30%

Loop Merging

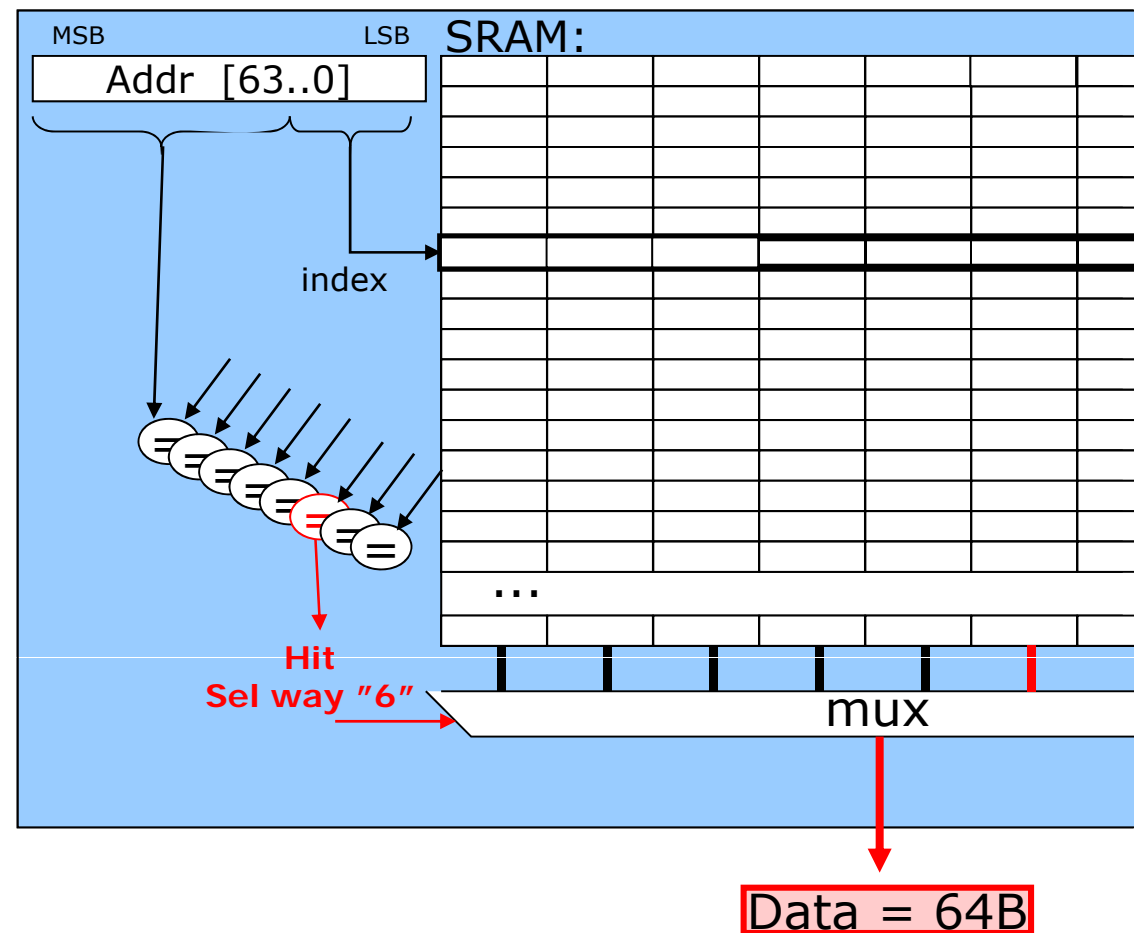
```
/* Unoptimized */
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        a[i][j] = 2 * b[i][j];
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        c[i][j] = K * b[i][j] + d[i][j]/2

/* Optimized */
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        a[i][j] = 2 * b[i][j];
        c[i][j] = K * b[i][j] + d[i][j]/2;
```

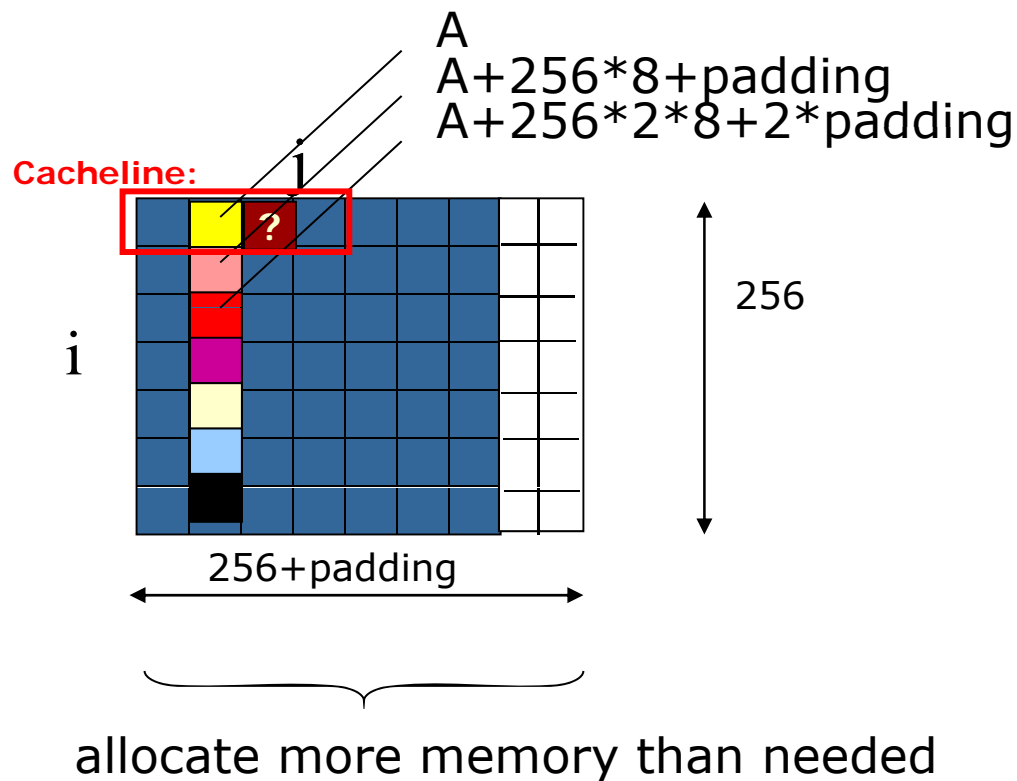
Padding of data structures



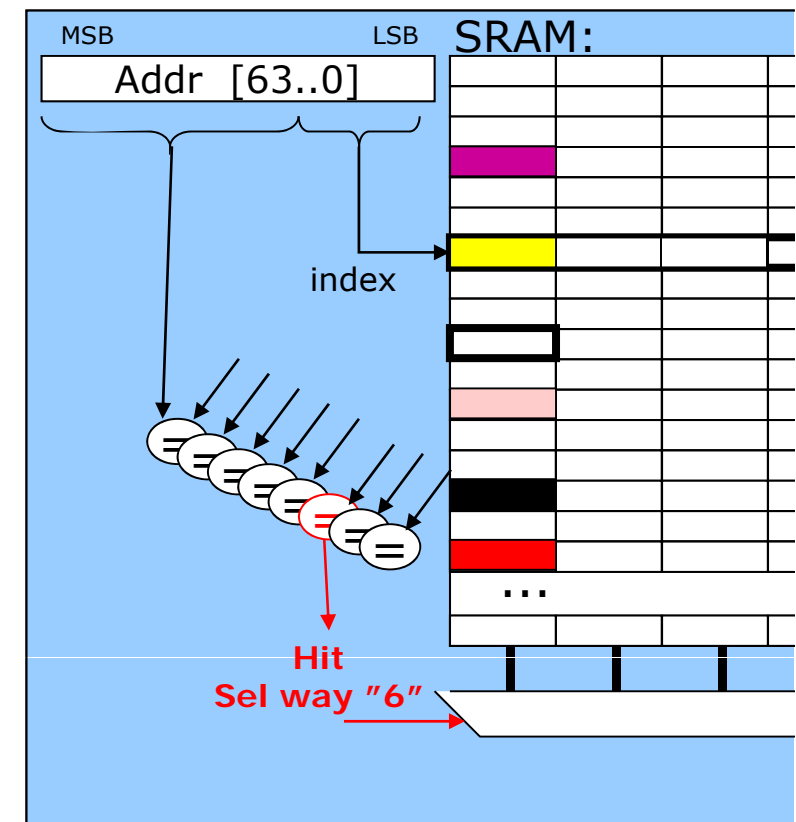
Generic Cache:



Padding of data structures

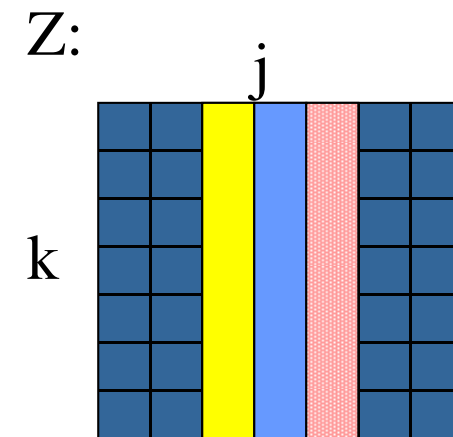
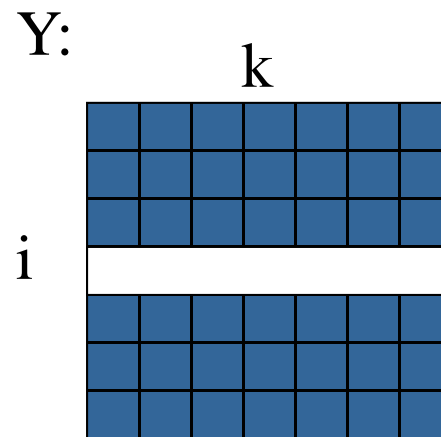
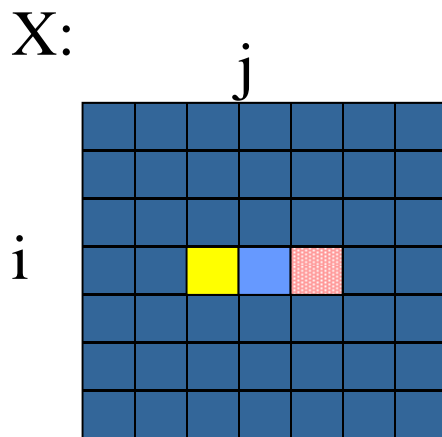


Generic Cache:



Blocking

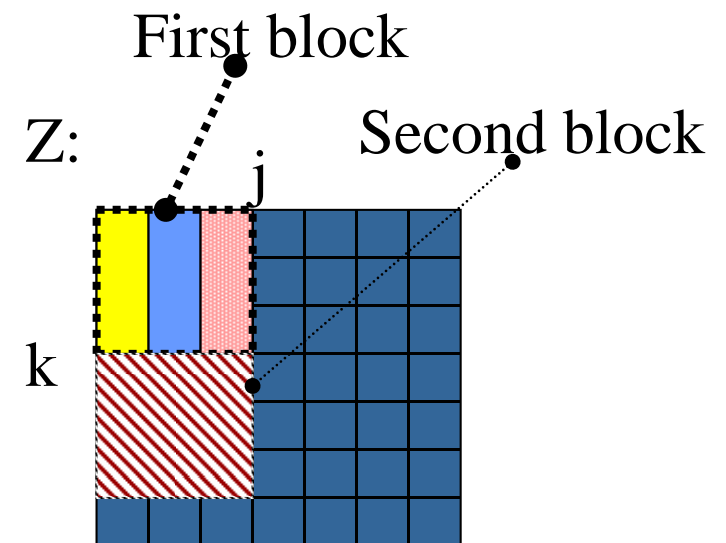
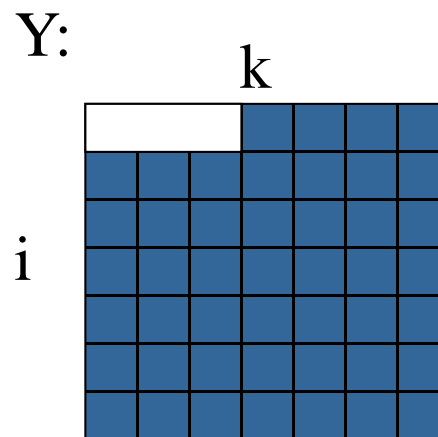
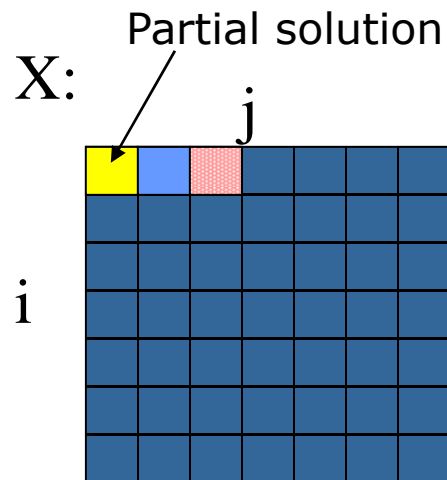
```
/* Unoptimized ARRAY: x = y * z */
for (i = 0; i < N; i = i + 1)
  for (j = 0; j < N; j = j + 1)
    {r = 0;
     for (k = 0; k < N; k = k + 1)
       r = r + y[i][k] * z[k][j];
     x[i][j] = r;
    };
```





Blocking

```
/* Optimized ARRAY: X = Y * Z */  
for (jj = 0; jj < N; jj = jj + B)  
for (kk = 0; kk < N; kk = kk + B)  
for (i = 0; i < N; i = i + 1)  
    for (j = jj; j < min(jj+B,N); j = j + 1)  
        {r = 0;  
        for (k = kk; k < min(kk+B,N); k = k + 1)  
            r = r + y[i][k] * z[k][j];  
        x[i][j] += r;  
        };
```



Blocking: the Movie!

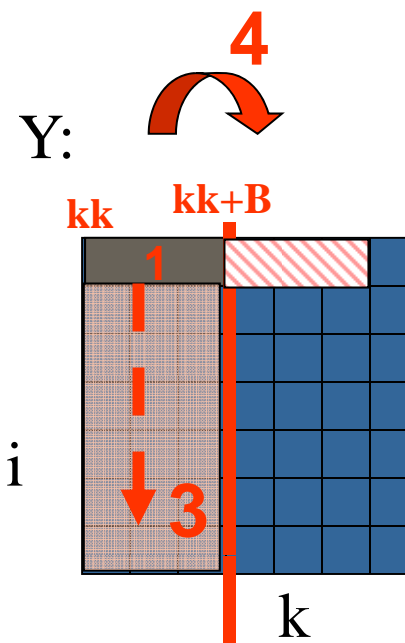
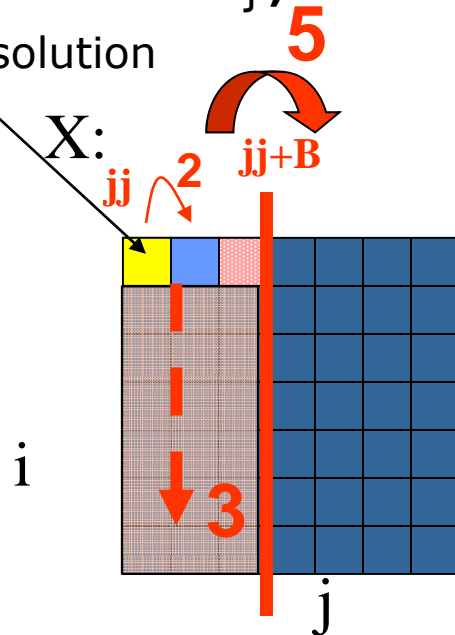
```

/* Optimized   ARRAY: X = Y * Z */
for (jj = 0; jj < N; jj = jj + B)           /* Loop 5 */
for (kk = 0; kk < N; kk = kk + B)           /* Loop 4 */
for (i = 0; i < N; i = i + 1)               /* Loop 3 */
    for (j = jj; j < min(jj+B,N); j = j + 1) /* Loop 2 */
        {r = 0;
        for (k = kk; k < min(kk+B,N); k = k + 1) /* Loop 1 */
            r = r + y[i][k] * z[k][j];
        x[i][j] += r;
    };

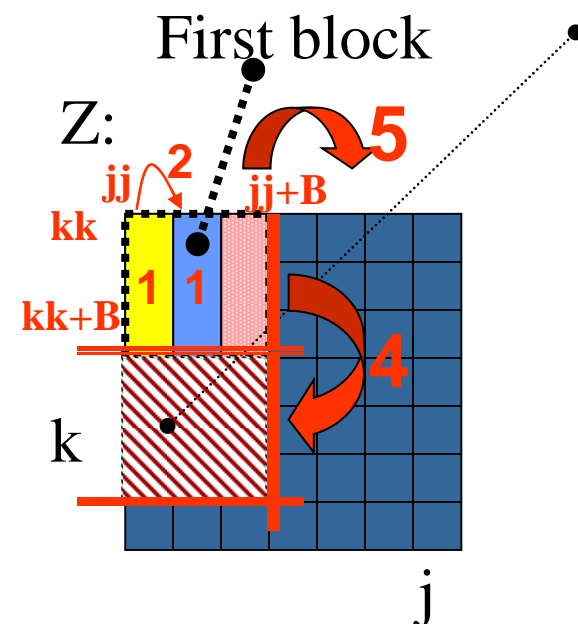
```

Second block

Partial solution



Second block





SW Prefetching

```
/* Unoptimized */  
for (j = 0; j < N; j++)  
    for (i = 0; i < N; i++)  
        x[j][i] = 2 * x[j][i];
```

```
/* Optimized */  
for (j = 0; j < N; j++)  
    for (i = 0; i < N; i++)  
        PREFETCH x[j+1][i]  
        x[j][i] = 2 * x[j][i];
```

(Typically, the HW prefetcher will successfully prefetch sequential streams)

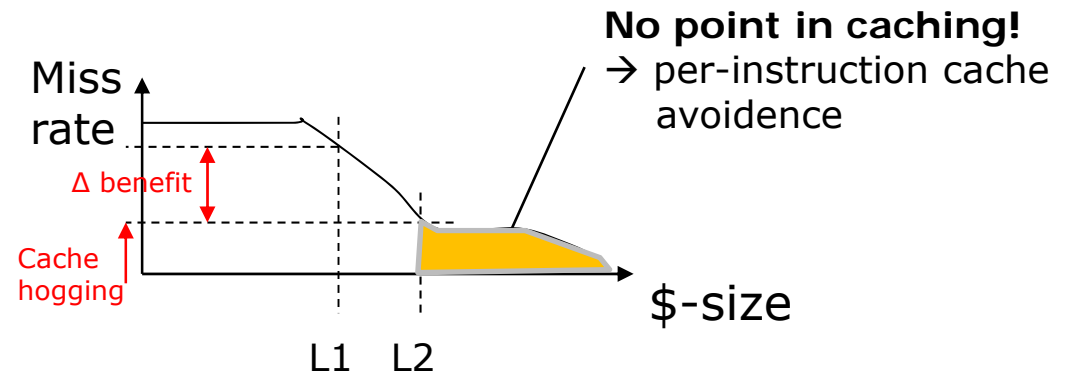
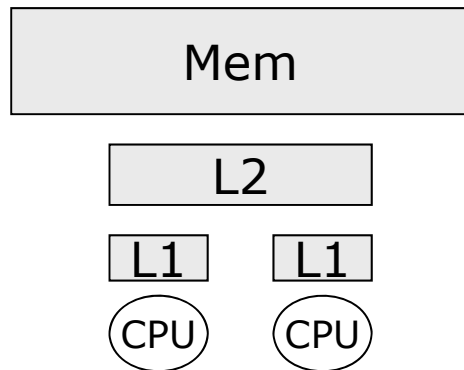
Cache Waste

```
/* Unoptimized */
for (s = 0; s < ITERATIONS; s++){
    for (j = 0; j < HUGE; j++)
        x[j] = x[j+1];          /* will hog the cache but not benefit*/
    for (i = 0; i < SMALLER_THAN_CACHE; i++)
        y[i] = y[i+1];          /* will be evicted between usages */
}

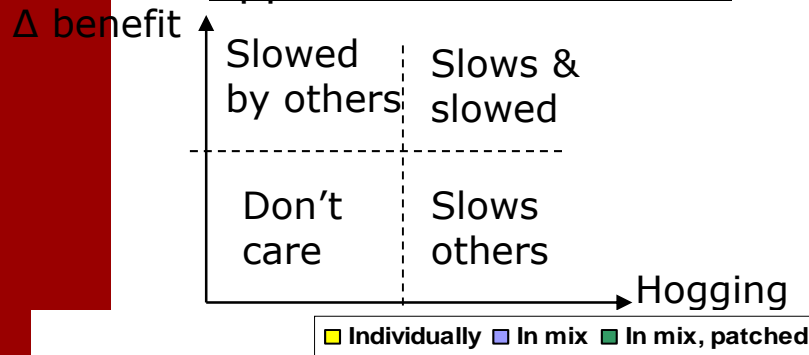
/* Optimized */
for (s = 0; s < ITERATIONS; s++){
    for (j = 0; j < HUGE; j++) {
        PREFETCH_NT x[j+1] /* will be installed in L1, but not L3 (AMD) */
        x[j] = x[j+1];
    }
    for (i = 0; i < SMALLER_THAN_CACHE; i++)
        y[i] = y[i+1];          /* will always hit in the cache*/
}
```

→ Also important for single-threaded applications if they are co-scheduled and share cache with other applications.

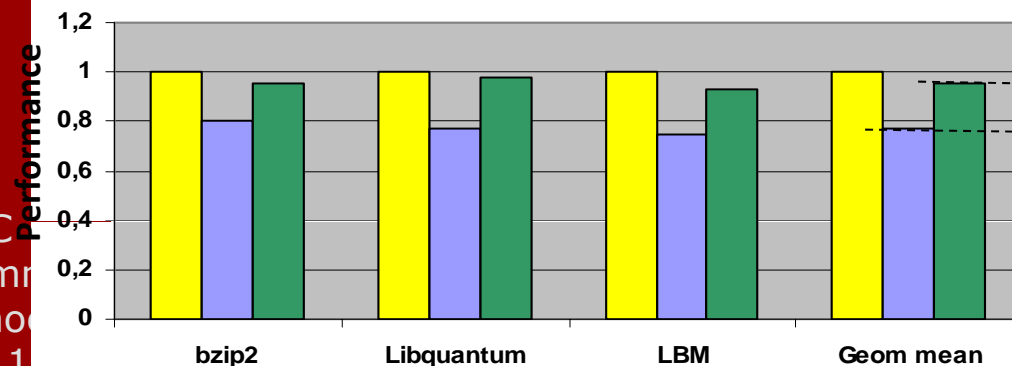
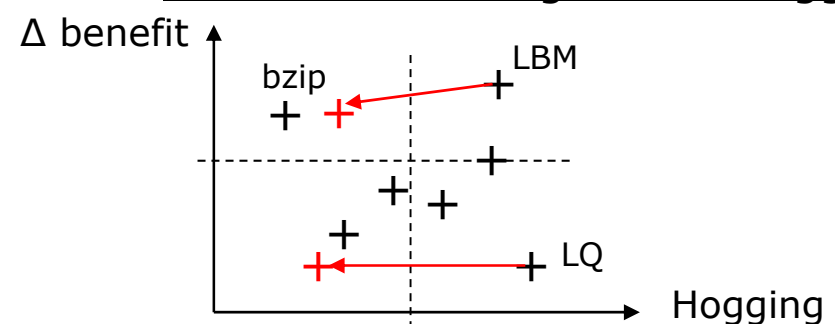
Categorize and avoiding cache waste



Application classification

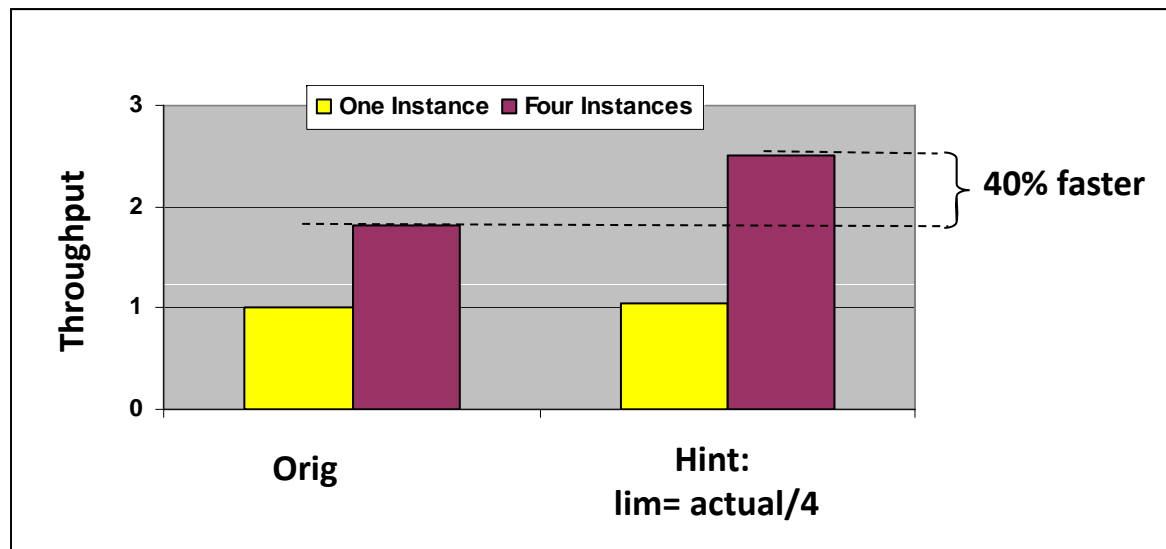
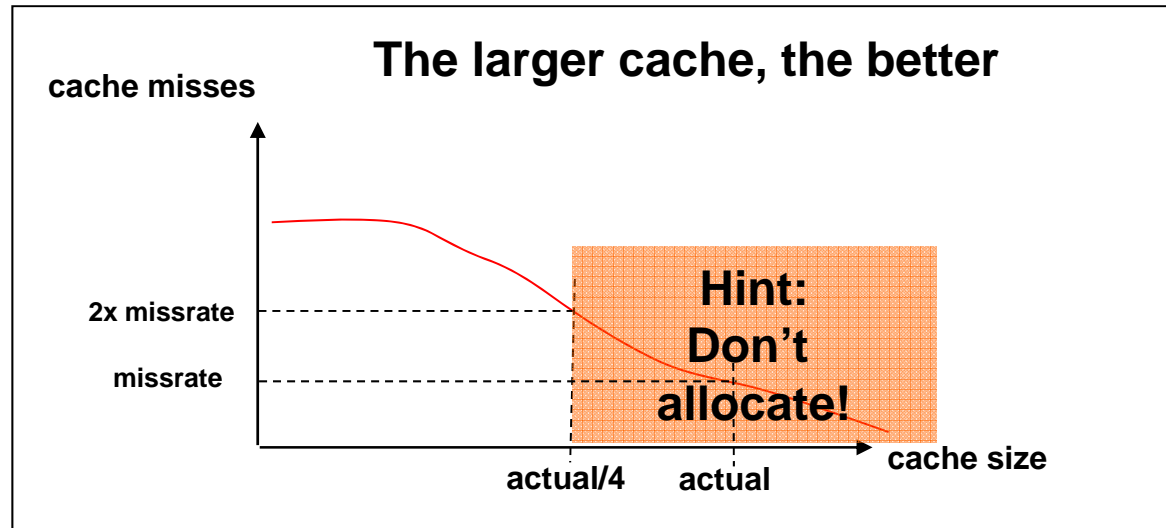


Automatic "taming" of the hoggers



Andreas Sandberg, David Eklov and Erik Hagersten. Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses, In Proceedings of Supercomputing (SC), New Orleans, LA, USA, November 2010.

Example: Hints to avoid cache pollution (non-temporal prefetches)





Coherence traffic

ORIG:

Thread 0:

```
int a, total;  
spawn_child()  
for (int i; i < HUGE; i++) {  
    /* do some work */  
    a++;  
}  
join()  
total = a;
```

Child:

```
for (int i; i < HUGE; i++) {  
    /* do some work*/  
    a++;  
}
```

OPT:

Thread 0:

```
int a, total;  
spawn_child()  
for (int i; i < HUGE; i++) {  
    /* do some work */  
    a++;  
}  
join()  
total += a;
```

Child:

```
int b;  
for (int i; i < HUGE; i++) {  
    /* do some work */  
    b++;  
}  
total += b;
```



False sharing

ORIG:

Thread 0:

```
int a, b;  
spawn_child()  
for (int i; i < HUGE; i++) {  
    ...  
    a++;  
}  
join()  
total = a;
```

Child:

```
for (int i; i < HUGE; i++) {  
    ...  
    b++;  
}
```

OPT:

Thread 0:

```
int a;  
spawn_child()  
for (int i; i < HUGE; i++) {  
    ...  
    a++;  
}  
join()  
total += a;
```

Child:

```
int b;  
for (int i; i < HUGE; i++) {  
    ...  
    total += b;  
}
```


Coherence Utilization

```
struct vec_type
{
    int a;
    int b;
    int c;
    int d;
    int e;
    int f;
};
```

x[0]	x[12]	x[
abcdef	abcdef	ab

ORIG:

Thread 0:

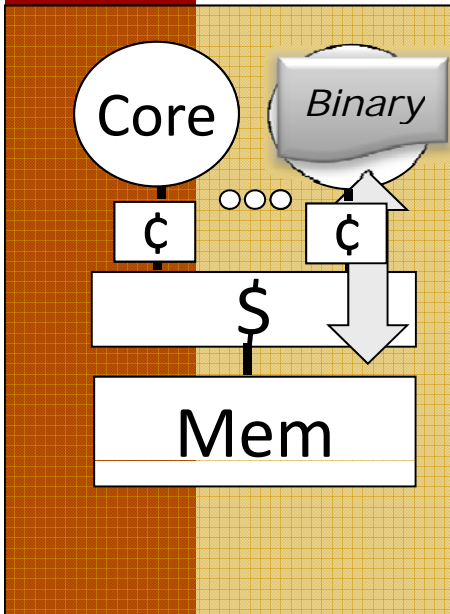
```
vec_type x[HUGE];
for (int i; i < HUGE; i++) {
    ...
    x[i].a++;
}
spawn_child()
...
join() ←
```

Child (Thread 1)

```
for (int i; i < HUGE; i++) {
    y[i] = x[i].a;
}
```



1. Optimize for Cache/Memory



1st Order MC Performance Problems

- Limited cache capacity
- Deep cache hierarchy
- Slow DRAM latency
- Low DRAM bandwidth

Issues to deal with:

- Data allocation schema
- Temporal data re-use
- Spatial data usage
- Cache utilization
- HW/SW prefetching issues

...



2. Remove Cache Waste

1st Order MC Performance Problems

- Additional multicore issues:
 - Even less cache resources per application
 - Sharing of cache resources
 - Wasted cache usage

Issues to deal with

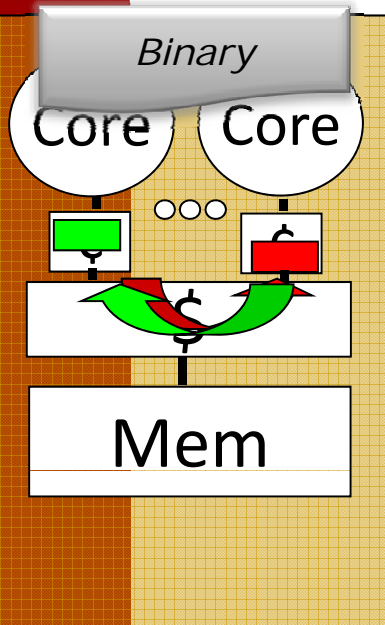
All previous issues and:

- Cache sharing effects
- Cache pollution side-effects
- Optimal usage of cache resources

...



3. Optimize Parallel Code

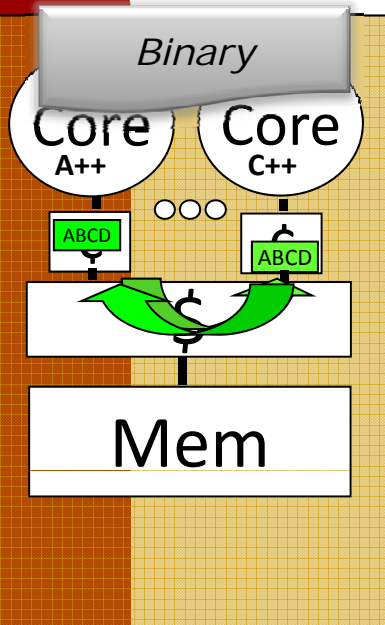


1st Order MC Performance Problems

- Thread interaction
 - Coherence traffic
 - Producer/consumer sharing
 - False sharing ...



3. Optimize Parallel Code



1st Order MC Performance Problems

- Thread interaction
 - Coherence traffic
 - Producer/consumer sharing
 - False sharing ...

Issues to deal with

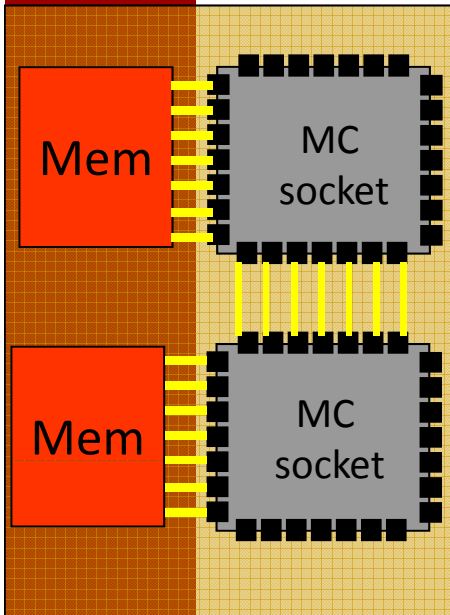
All previous issues and:

- Allocation/division of shared data
- Inter-thread data dependence
- Inter-thread cache sharing
- Communication patterns

...



4. Place Threads/Memory



1st Order MC Performance Problems

- Core2 Quad, Magny Cours, Multisocket
- Non-uniform memory access (NUMA)
- Non-uniform communication cost (NUCA)
- Higher costs everywhere...

Issues to deal with:

All previous issues become more important, plus

- Job scheduling issues
- Thread to core binding issues
- Memory placement issues

Some performance tools

Free licenses

- Oprofile
- GNU: gprof
- AMD: code analyst
- Google performance tools
- Virtual Inst: High Productivity Supercomputing (<http://www.vi-hps.org/tools/>)

Not free

- Intel: Vtune and many more
- ThreadSpotter (of course☺)
- HP: Multicore toolkit (some free, some not)

- Uppsala Programming for Multicore Architecture Center
- 62 MSEK grant / 10 years [\$9M/10y]
+ related additional grants at UU = 130MSEK
- Research areas:

Erik: * Performance modeling

* New parallel algorithms

* Scheduling of threads and resources

* Testing & verification

* Language technology

* MC in wireless and sensors



Multi-threaded Case Study: Gauss-Seidel on Multicores

From Wallin et al, ICS 2006

Criteria for HPC Algorithms

■ Past:

- ✱ Minimize communication
- ✱ Maximize scalability (1000s of CPUs)

■ Optimize for Multicore chip:

- ✱ On-chip communication is “for free”
- ✱ Scalability is limited to ~ 10 threads
- ✱ The caches are tiny
- ✱ Memory bandwidth is the bottleneck

➔ Data locality is key!

Selected HPC Wire Articles

More Than 16 Cores May Well Be Pointless

Sandia Labs, Dec 07 2008

Up Against the Memory Wall

"Never mind the cores. Just hand over the cache"

Michael Feltman, Dec 11 2008

HPC@Intel: When to Say No to Parallelism

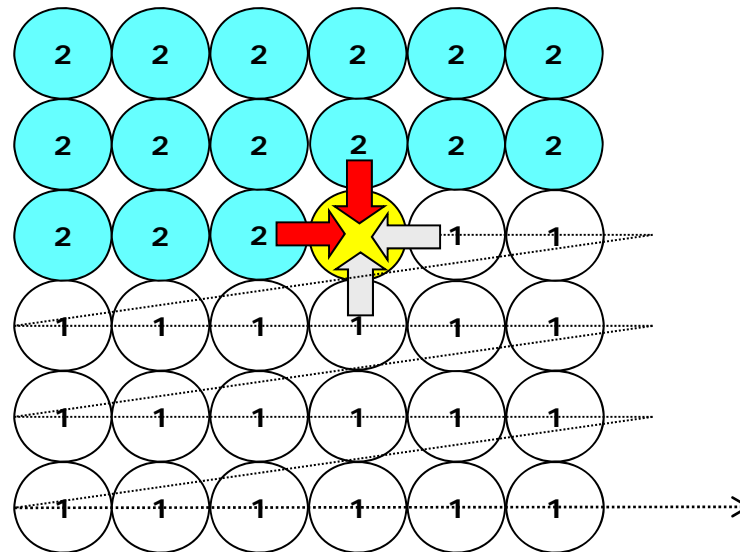
Sanjiv Shah, Intel. January 14 2009

Finding a Door in the Memory Wall

Erik Hagersten, Acumem. Feb-April 2009

Example: Gauss Seidel

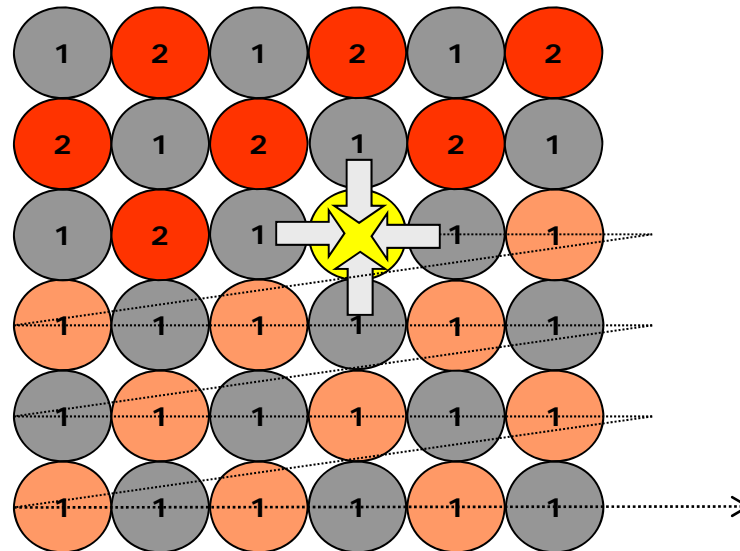
Mission: “Maximize the parallelism and minimize the inter-thread communication”



LOOP:
 UPDATE ALL POINTS
 IF (convergence_test)
 <done>

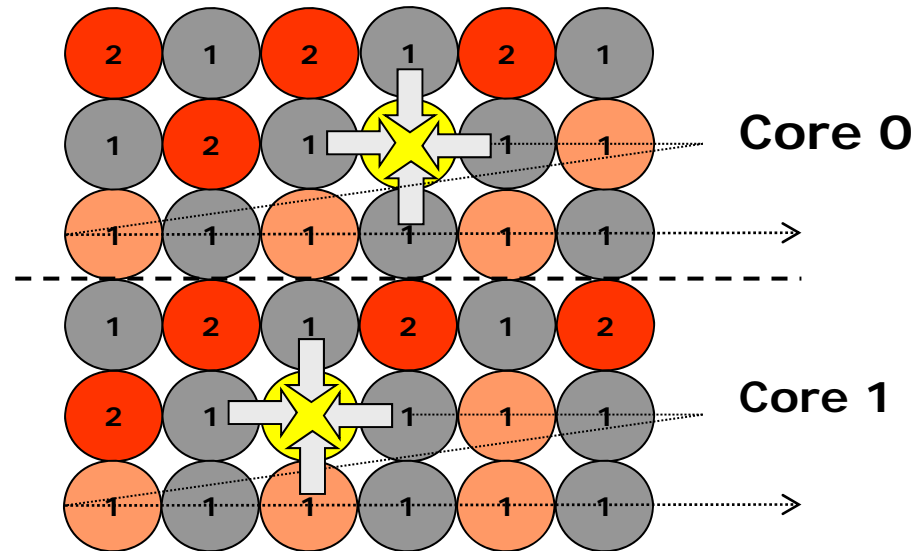
(Longer explanation: [Finding a Door in the Memory Wall](#) @ HPCWire)

State-of-the-art: Removing Dependence: Red/Black



LOOP:
UPDATE ALL **RED** POINTS
UPDATE ALL **BLACK** POINTS
IF (convergence_test)
 <done>

State-of-the-art: Red/Black, Parallelism = $N^2/2$



LOOP:

IN PARALLEL: UPDATE ALL **RED** POINTS

<barrier>

IN PARALLEL: UPDATE ALL **BLACK** POINTS

<barrier>

IF (convergence_test)

<done>

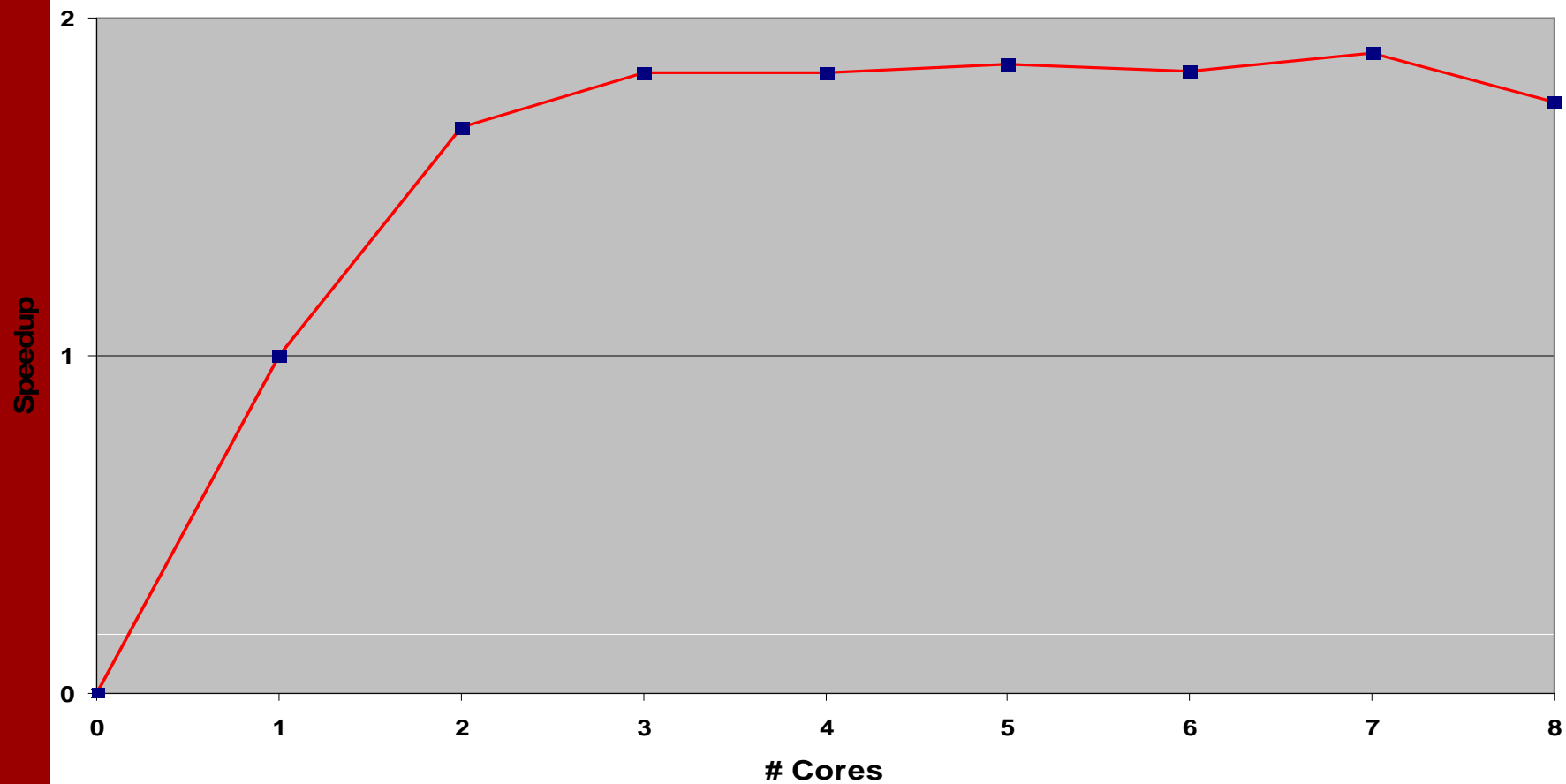
Limited communication ☺

$N^2/2$ parallelism ☺

Done!

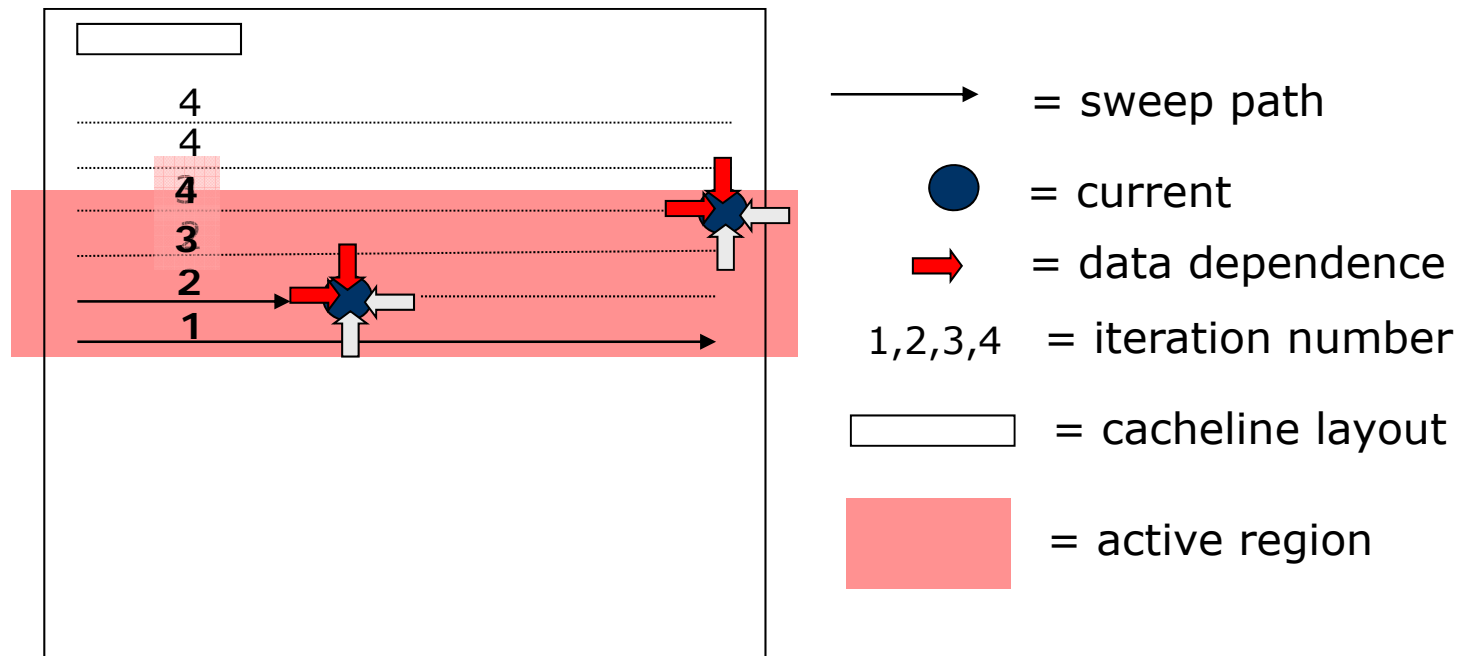
OPT Only one problem...

Only One Problem: Performance



Back to the drawing board: Temporal blocking for seq. code

Communication is “for free” and moderate parallelism is OK
Priority 1: limit bandwidth needs!



LOOP:

LOOP:

UPDATE ALL POINTS IN ACTIVE REGION

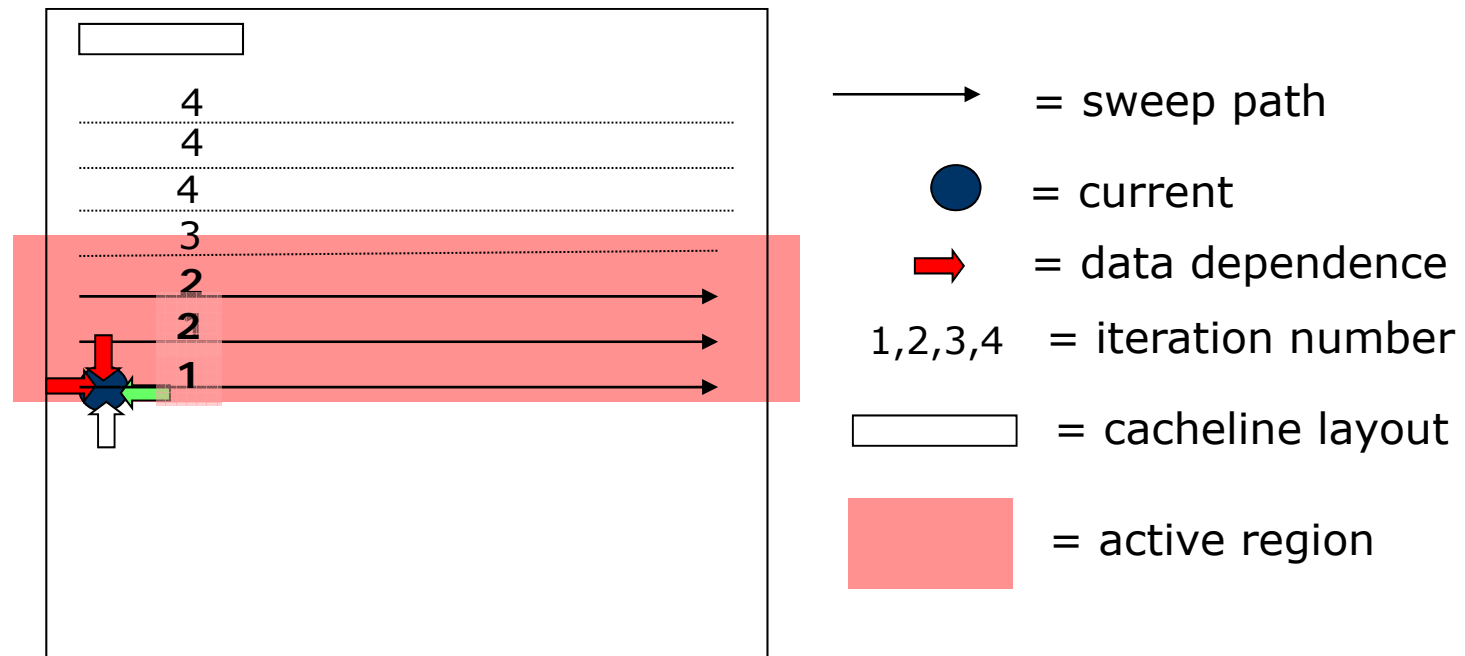
SLIDE DOWN THE REGION

IF (convergence_test)

<done>

Back to the drawing board: Temporal blocking for seq. code

Communication is “for free” and moderate parallelism is OK
Priority 1: limit bandwidth need!



LOOP:

LOOP:

UPDATE ALL POINTS IN ACTIVE REGION

SLIDE DOWN THE REGION

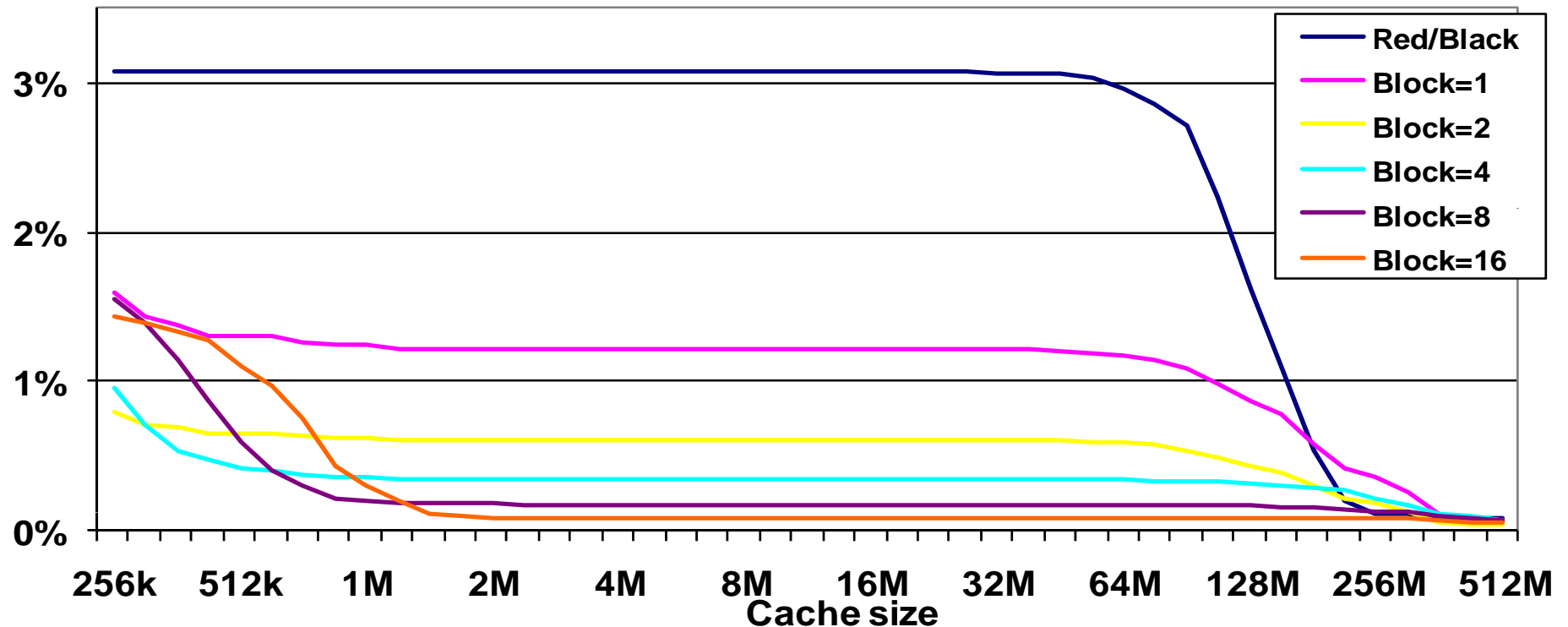
IF (convergence_test)

<done>

4 iterations in
one sweep!

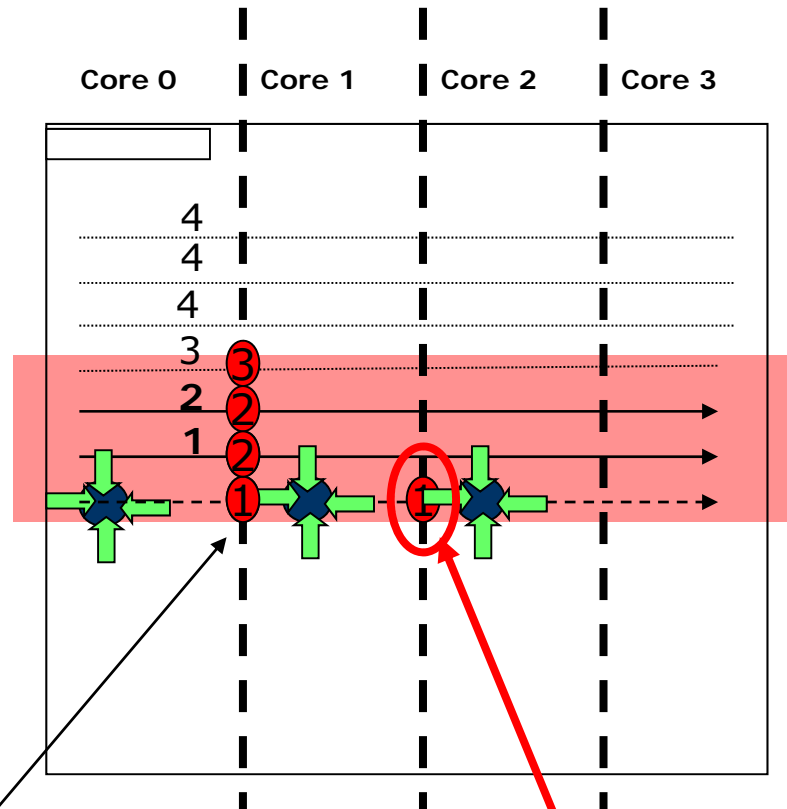
DRAM_traffic(cache_size)

Fetch Rate,
i.e, fraction of mem_ops generating DRAM traffic





G-S, temp block Parallelism = N

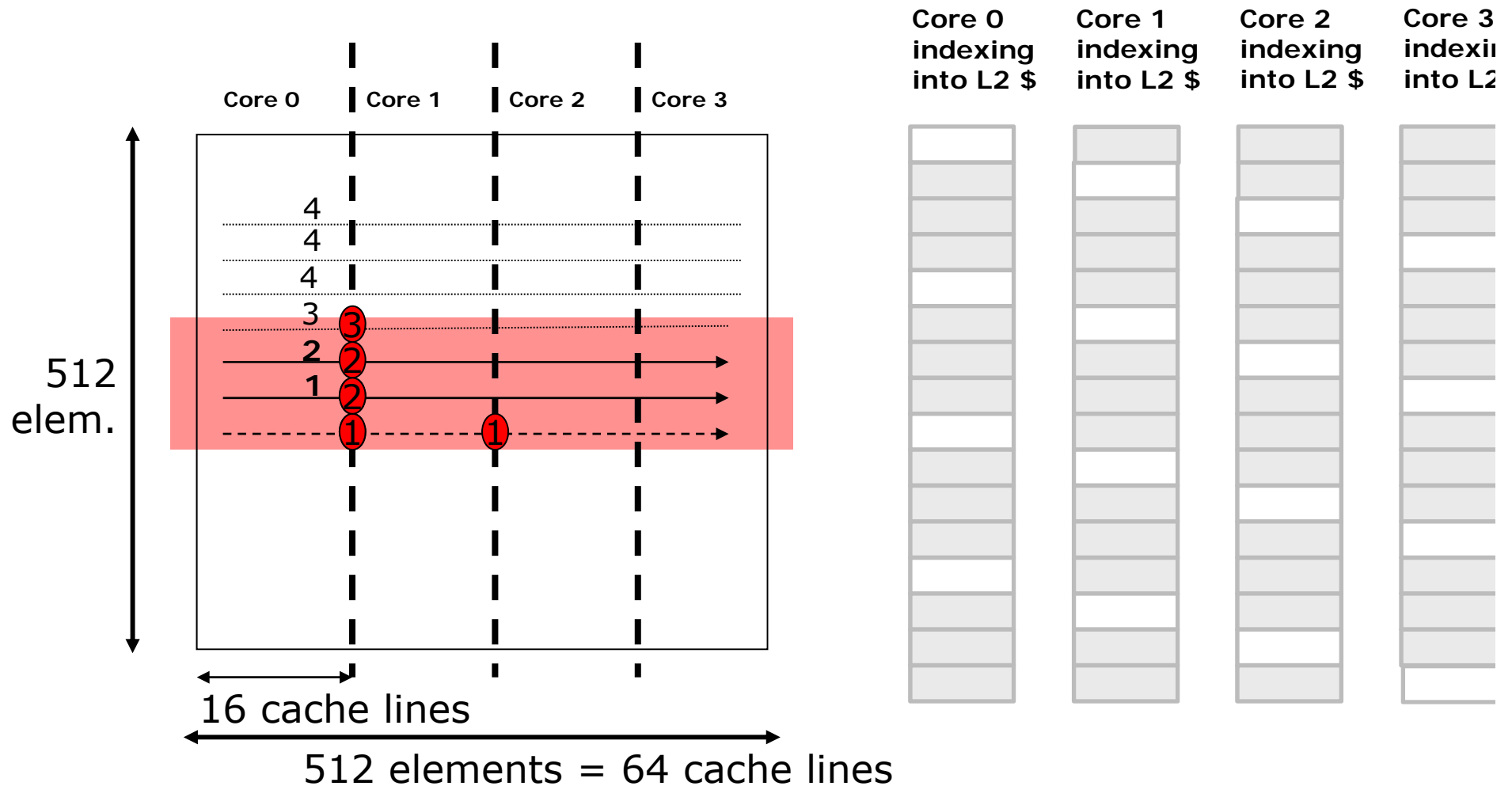


- = sweep path
- = current
- = data dependence
- 1,2,3,4 = iteration number
- = cacheline layout
- = active region
- ① = sync flag iteration no

Synchronization
flags

Wait until "lefty" is done:
Lots of communication ☹
• Producer/Consumer Flag
• Sharing of data values
Only N-fold parallelism ☹

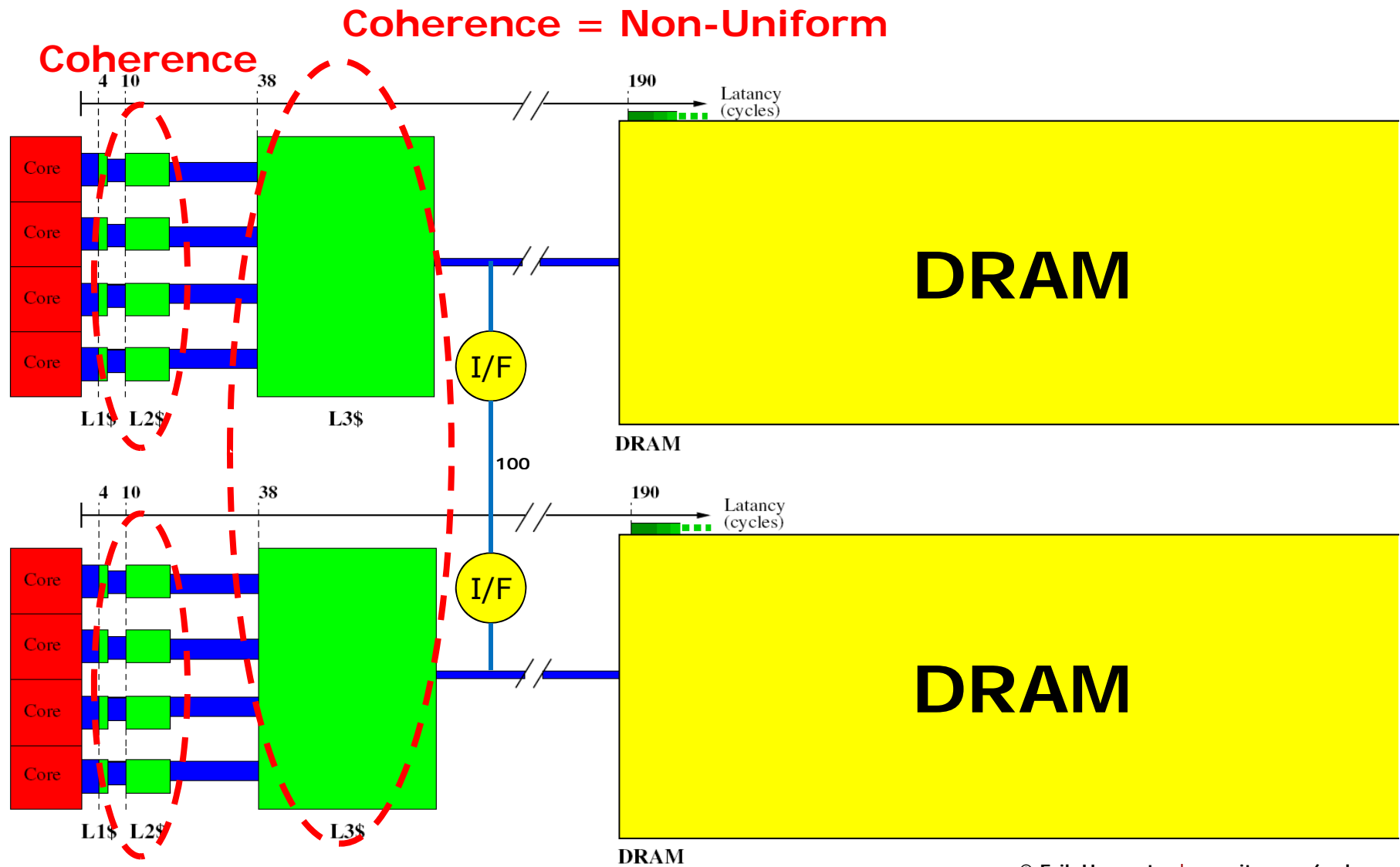
Problems we ran into 1 (2)



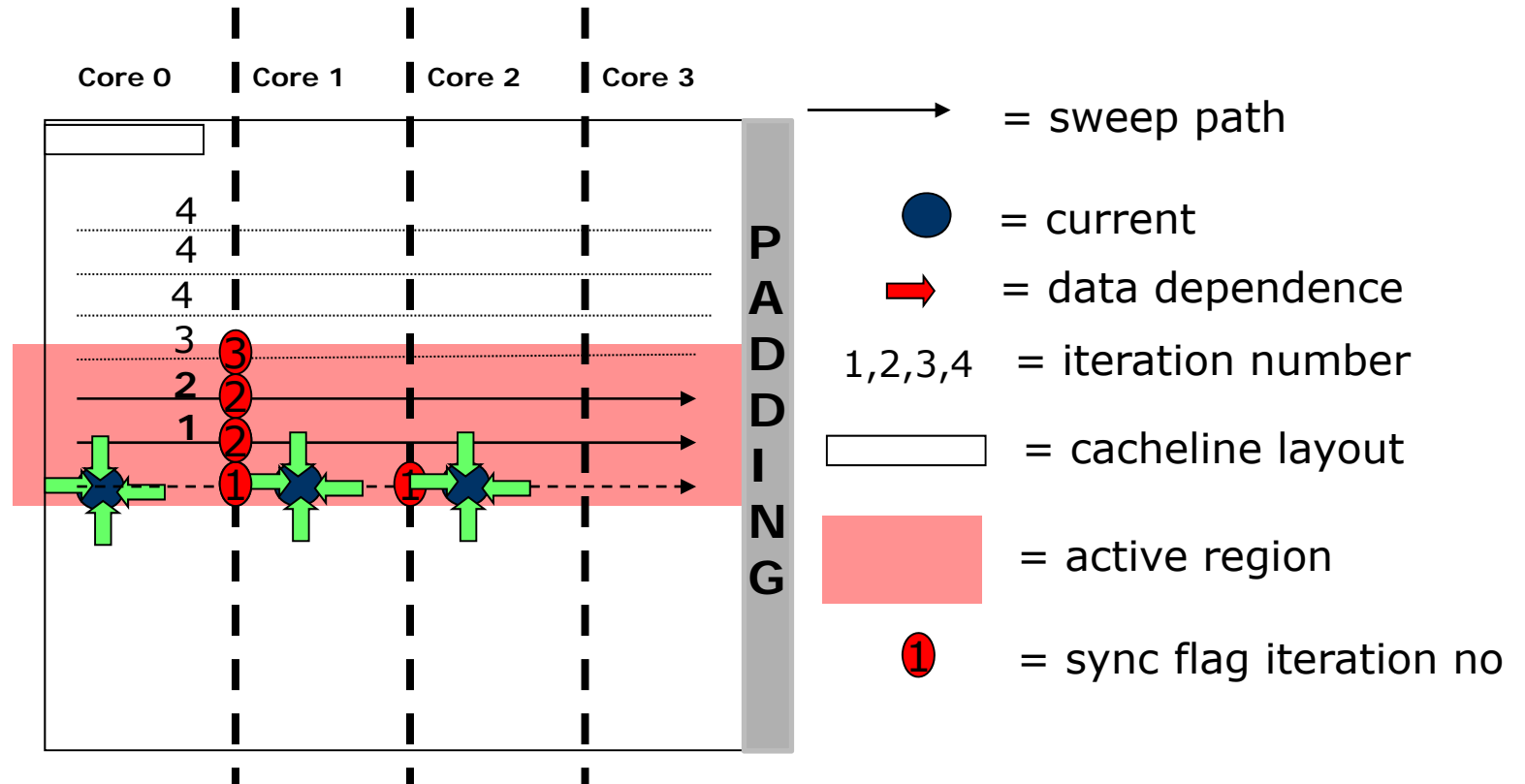
Problems we ran into 2 (2)

- We had a loop nesting problem that the compiler optimized away
- ... sometimes

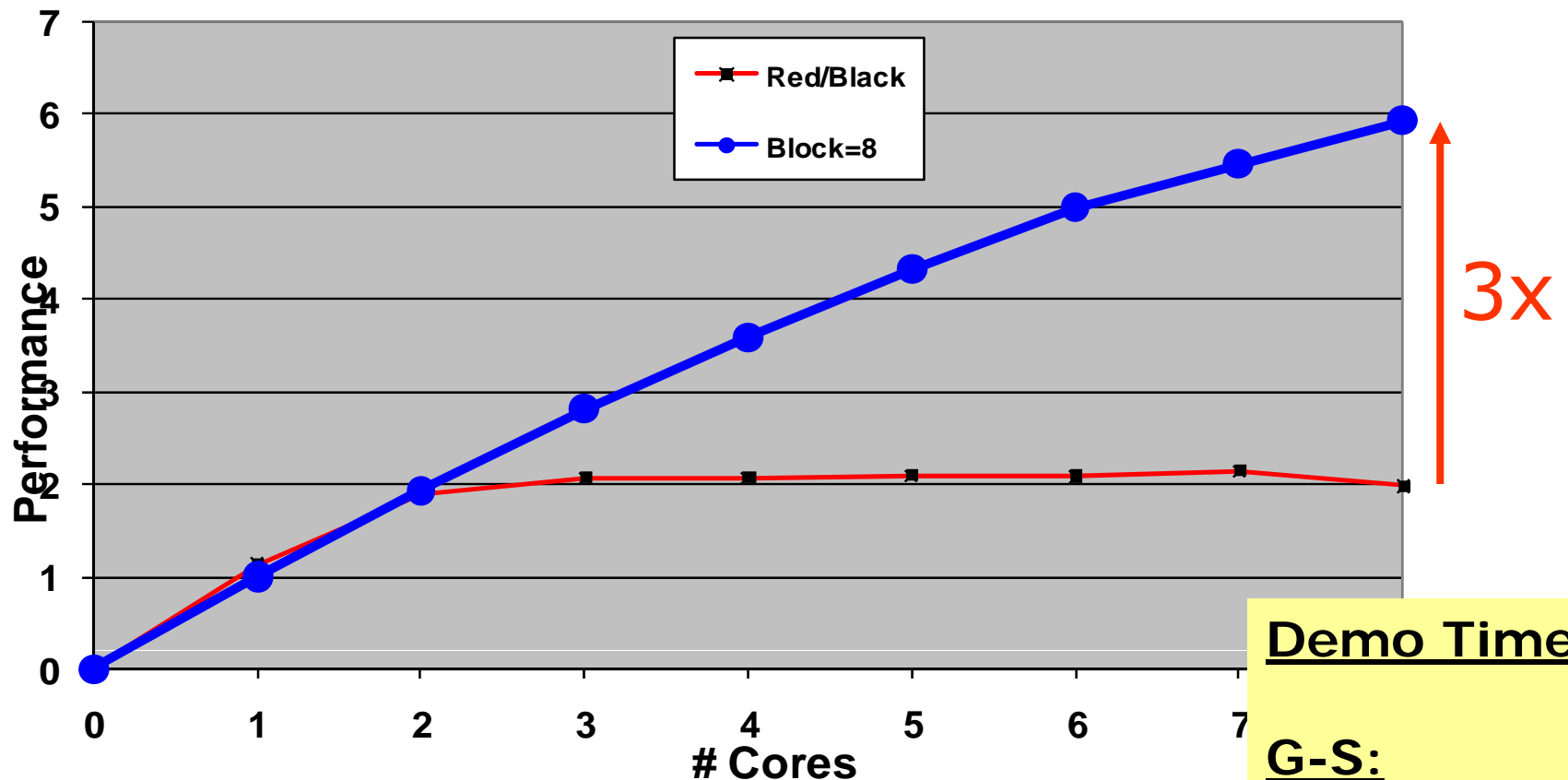
Running on a Multisocket



Example: G-S, temp blocking



Lessons Learned: Optimize cache usage **BEFORE** parallelizing



Demo Time!

G-S:
[Original code](#)
[Optimized](#)

[Wallin, Löf, Holmgren, Hagersten @ ICS 2006]

OPT 48



Fooling Amdahl

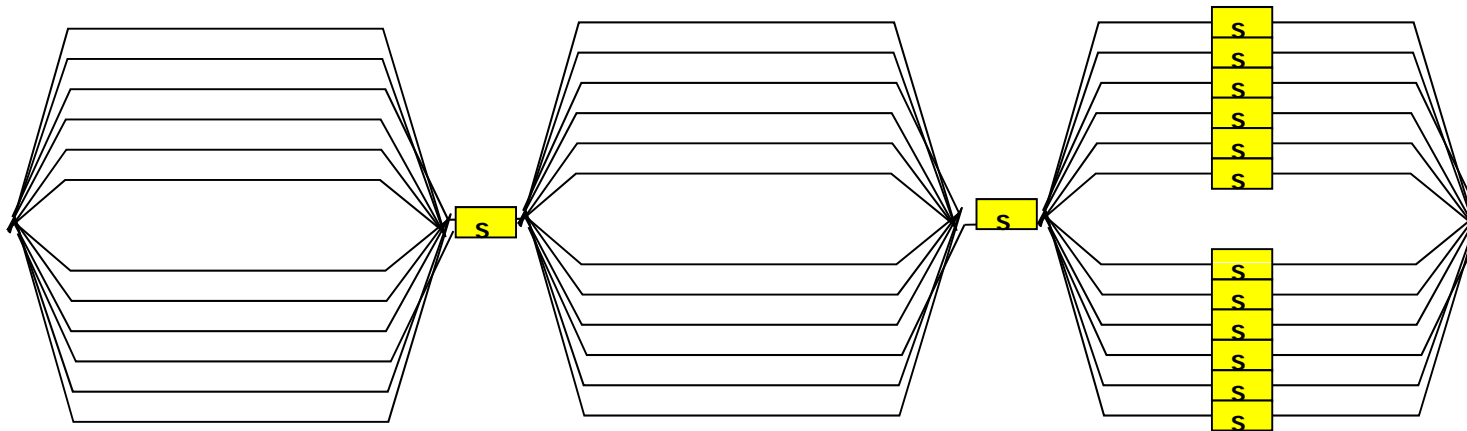
-- being unfair is not fair --

Erik Hagersten
Uppsala University
Sweden

Amdahl rears his ugly head again

Multicore/manycore:

- ✱ Many simple cores clocked moderat freq.
- ✱ But: application have serial segments...



- ✱ and/or serial critical sections
- ✱ 1000s of cores will not make much difference

Need to introduce synchronization

- Locking primitives are needed to ensure that only one process can be in the critical section:

```
LOCK(lock_variable) /* wait for your turn */  
    A = A + 1;  
UNLOCK(lock_variable) /* release the lock*/
```



Critical Section

Atomic Instruction to Acquire a Lock

Atomic example: test&set "TAS" (SPARC: LDSTB)

- The value at Mem(lock_addr) loaded into the specified register
- Constant "1" atomically stored into Mem(lock_addr) (SPARC: "FF")
- Software can determine if won (i.e., set changed the value from 0 to 1)
- Other constants could be used instead of 1 and 0

Looks like a store instruction to the caches/memory system

Implementation:

1. Get an exclusive copy of the cache line
2. Make the atomic modification to the cached copy

Other read-modify-write primitives can be used too

- Swap (SWAP): atomically swap the value of REG with Mem(lock_addr)
- Compare&swap (CAS): SWAP if Mem(lock_addr) == REG2

Optimistic Test&Set Lock "spinlock"

```
proc lock(lock_variable) {  
    while true {  
        if (TAS[lock_variable] == 0) break; /* bang on the lock once, done if TAS==0 */  
        while(lock_variable != 0) {} /* spin locally in your cache until "0" observed */  
    }  
}  
  
proc unlock(lock_variable) {  
    lock_variable = 0  
}
```

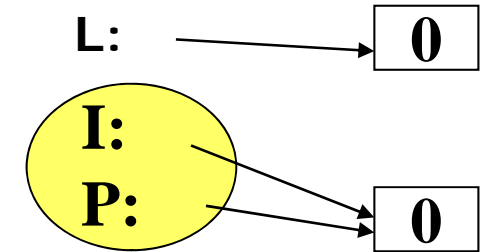
Lots of traffic at lock handover!
→ CS cost grows with #threads
→ Amdahl gets mad

Queue-based lock example: Ticket

```
proc lock(lstruct) {  
    int my_num;  
    my_num := INC(lstruct.ticket)    /* get your unique number*/  
    while(my_num != lstruct.now-serving) {} /* wait here for your turn */  
}  
  
proc unlock(lstruct) {  
    lstruct.now-serving++             /* next in line please */  
}
```

Only one thread gets excited at lock handover
→ Less traffic at lock handover!
→ ~ Constant CS cost

Queue-based lock: CLH-lock



"Initially, each process owns one global cell, pointed to by private *I and *P
Another global cell is pointed to by global *L "lock variable"

- 1) Initialize the *I flag to busy (= "1")
- 2) Atomically, make *L point to "our" cell and make "our" *P point where *L's cell
- 3) Wait until *P points to a "0"

```
proc lock(int **L, **I, **P)
{
    **I =1;                /*initialized "our" cell as "busy"*/
    atomic_swap { *P =*L; *L=*P}
                        /* P now stores a pointer to the cell L pointed to*/
                        /* L now stores a pointer to our cell */
    while (**P != 0){} }/* keep spinning until prev owner releases lock*/
```

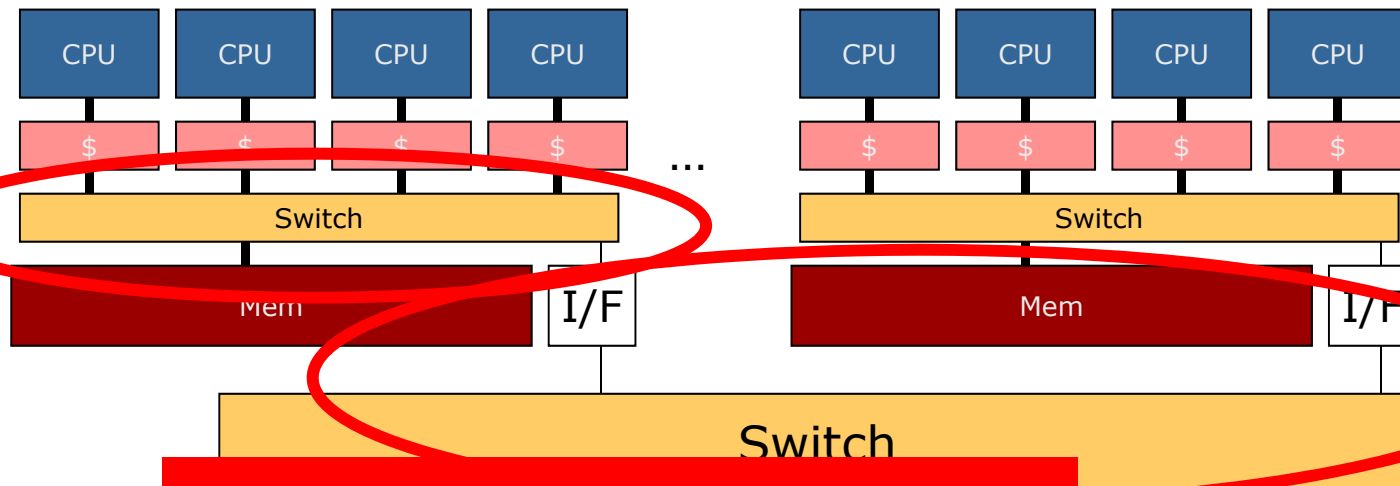
```
proc unlock(int **I, **P)
{
    **I =0;                /* release the lock */
    *I =*P; }              /* next time *I to reuse the previous guy's cell*/
```

NUMA:



**NUCA:
Non-uniform
Comm Arch.**

Snoop

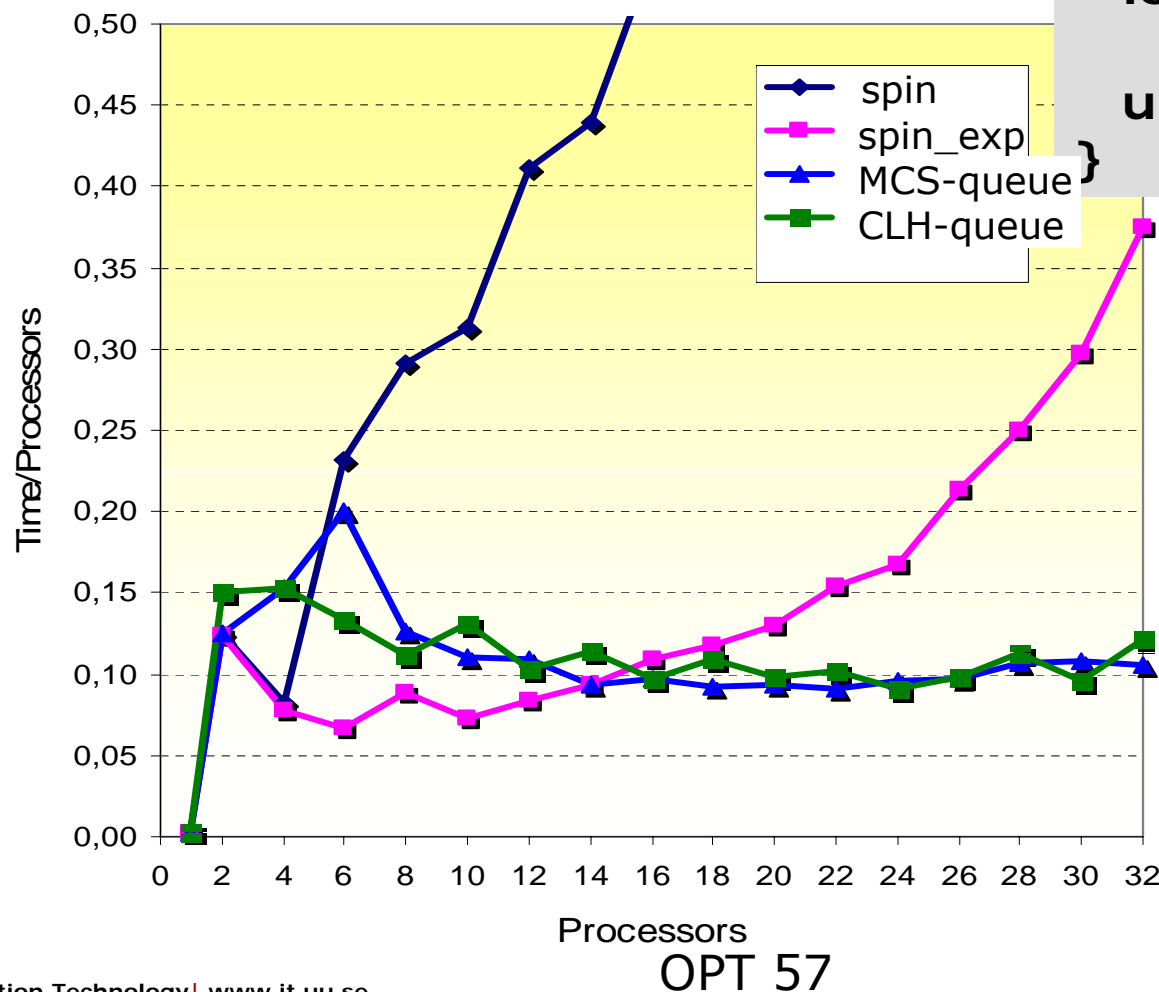


**Directory-latency = 6x snoop
i.e., roughly CMP NUCA-ness**

Trad. chart over lock performance on a hierarchical NUMA (round robin scheduling)

Benchmark:

```
for i = 1 to 10000 {
  lock(AL)
  A := A + 1;
  unlock(AL)
}
```

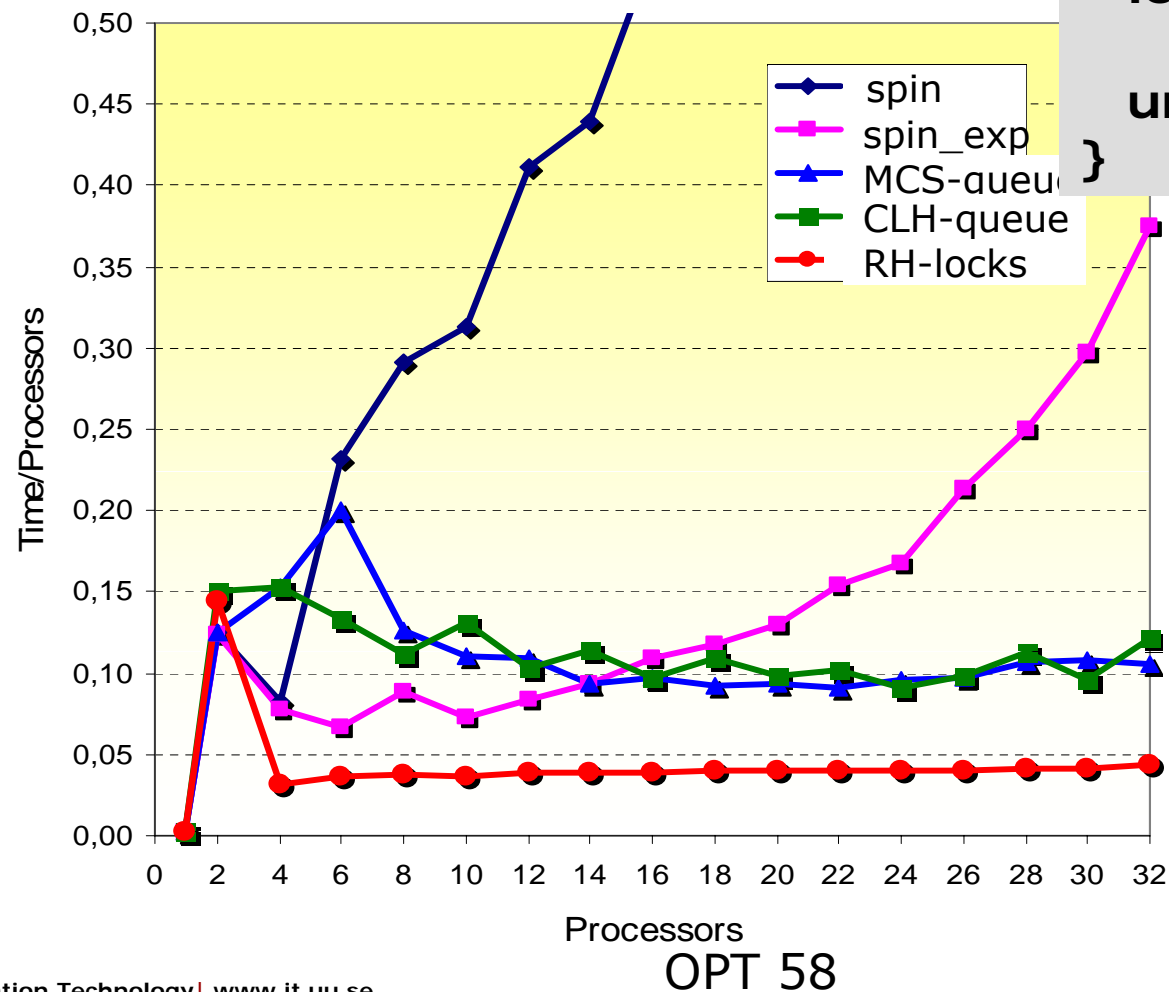




Introducing RH locks

Benchmark:

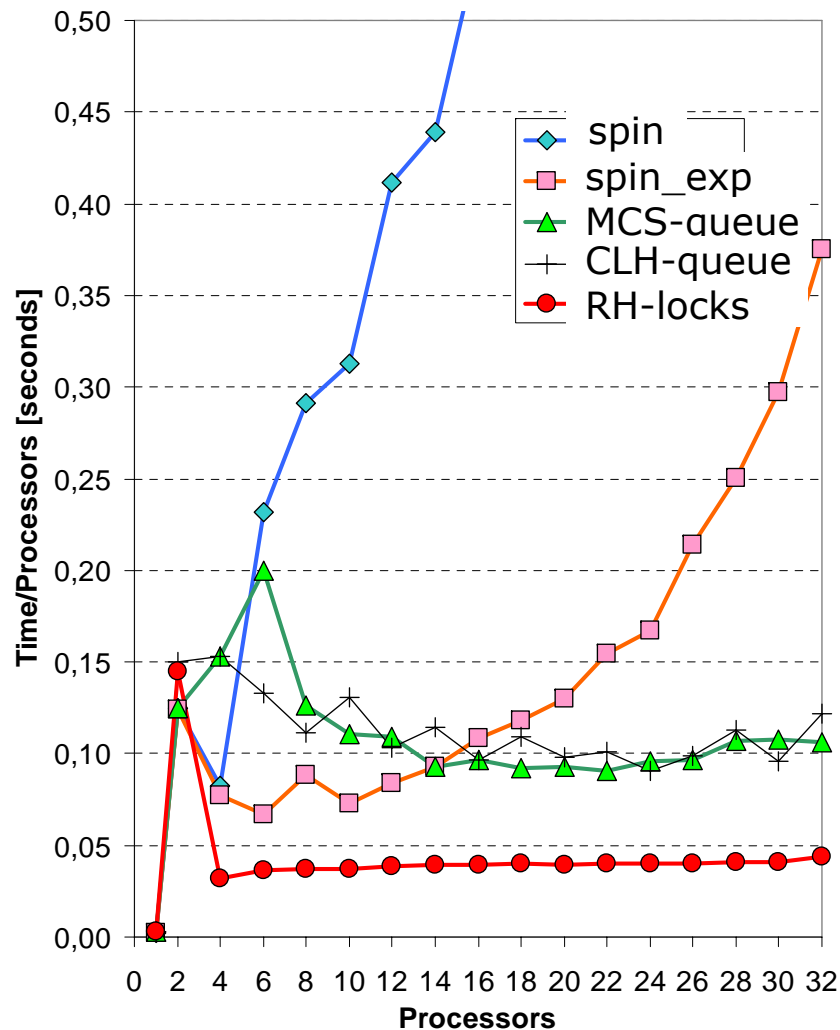
```
for i = 1 to 10000 {  
  lock(AL)  
  A := A + 1;  
  unlock(AL)  
}
```



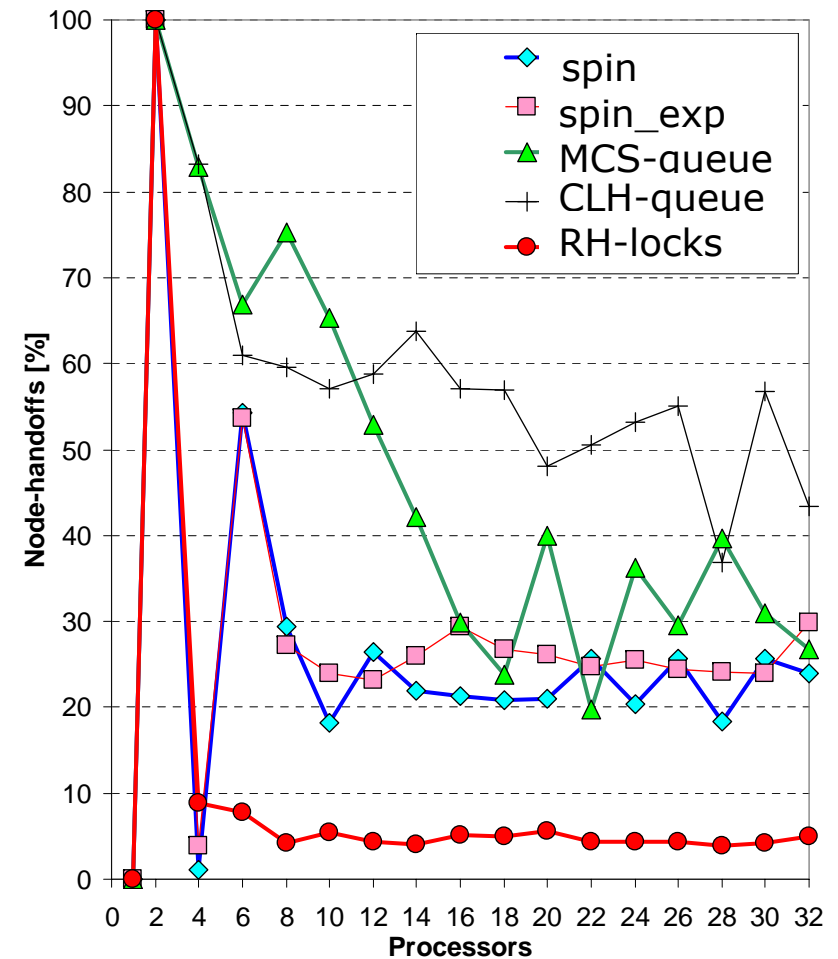


RH locks: encourages unfairness

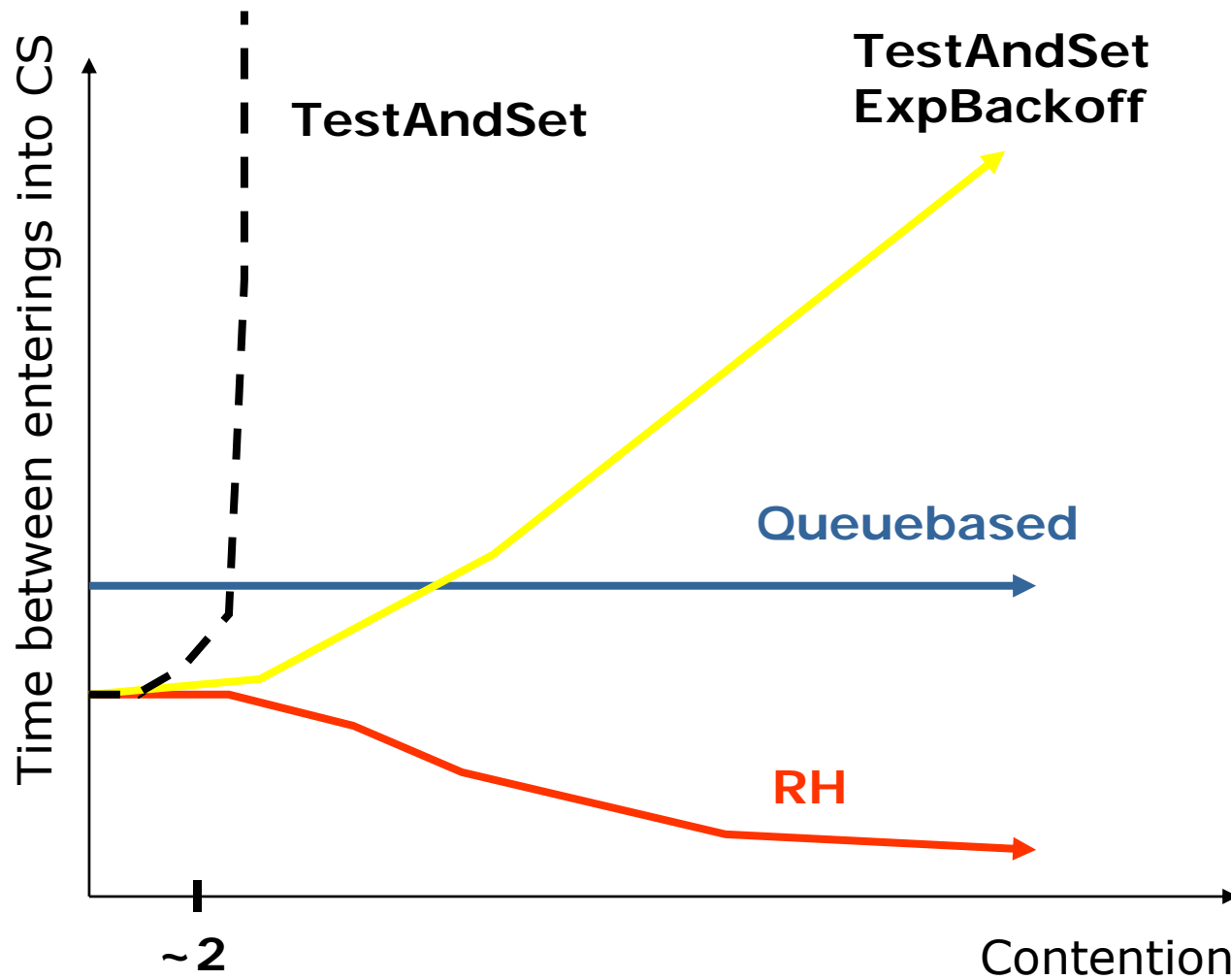
Time per lock handover



Node migration (%)



Performance under contention



Ex: Splash Raytrace Application Speedup

HBO@HPCA 2003
RH@SC 2002

