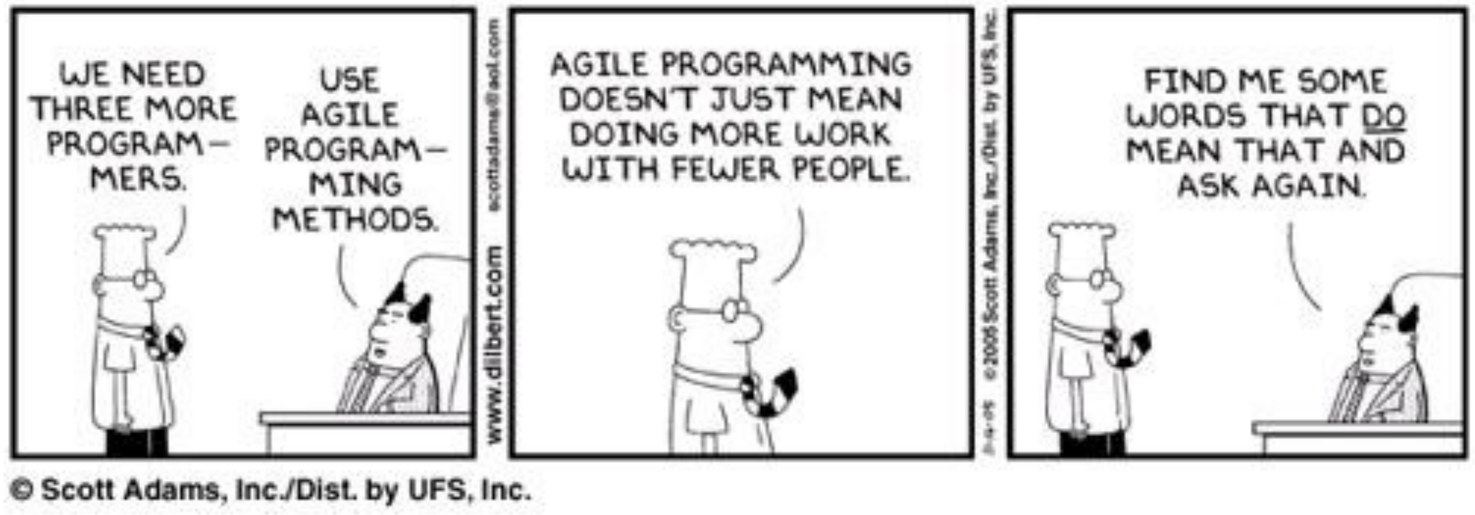
CS302

Optimizing Software


WE NEED THREE MORE PROGRAM-MERS.
USE AGILE PROGRAM-MING METHODS.
AGILE PROGRAMMING DOESN'T JUST MEAN DOING MORE WORK WITH FEWER PEOPLE.
FIND ME SOME WORDS THAT DO MEAN THAT AND ASK AGAIN.
www.dilbert.com
scottadams@aol.com
© 2005 Scott Adams, Inc./Dist. by UFS, Inc.
© Scott Adams, Inc./Dist. by UFS, Inc.

Spring 2025

Arkaprava Basu & Babak Falsafi

parsa.epfl.ch/course-info/cs302

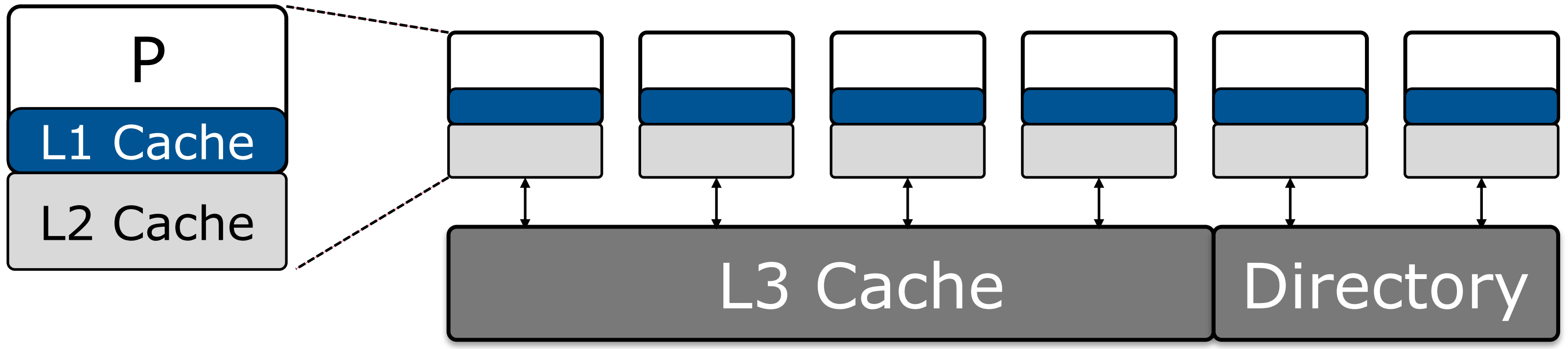
Where are We?

M	T	W	T	F
17-Feb	18-Feb	19-Feb	20-Feb	21-Feb
24-Feb	25-Feb	26-Feb	27-Feb	28-Feb
	4-Mar	5-Mar	6-Mar	7-Mar
10-Mar	11-Mar	12-Mar	13-Mar	14-Mar
17-Mar	18-Mar	19-Mar	20-Mar	21-Mar
24-Mar	25-Mar	26-Mar	27-Mar	28-Mar
31-Mar	1-Apr	2-Apr	3-Apr	4-Apr
7-Apr	8-Apr	9-Apr	10-Apr	11-Apr
14-Apr	15-Apr	16-Apr	17-Apr	18-Apr
21-Apr	22-Apr	23-Apr	24-Apr	25-Apr
28-Apr	29-Apr	30-Apr	1-May	2-May
5-May	6-May	7-May	8-May	9-May
12-May	13-May	14-May	15-May	16-May
19-May	20-May	21-May	22-May	23-May
26-May	27-May	28-May	29-May	30-May

- Software Optimizations
  - Locality & Memory access
  - Scheduling & work distribution
- Thursday
  - Lecture: SIMD and vector instructions
  - Exercise session: Performance debugging
- Next Tuesday:
  - Message Passing

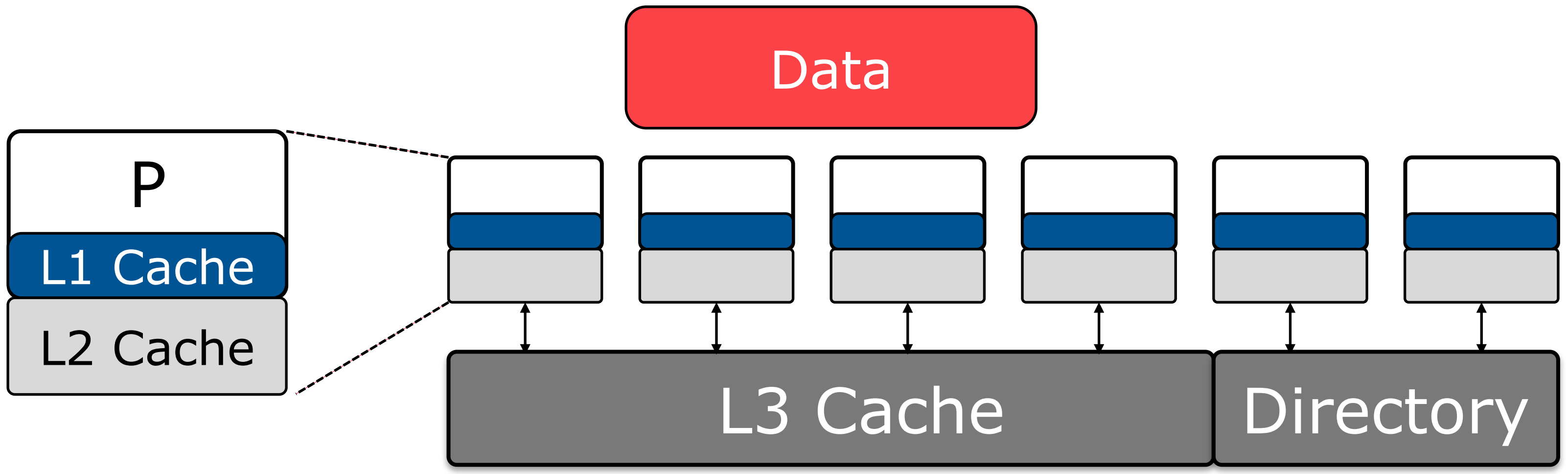
Reminder: Multiprocessor Caching

- Modern CPUs have a hierarchy of caches, that are kept coherent transparently
  - Eases programmability
  - … But, can hurt performance if we’re not careful!
  - e.g., Data going back/forth between two caches



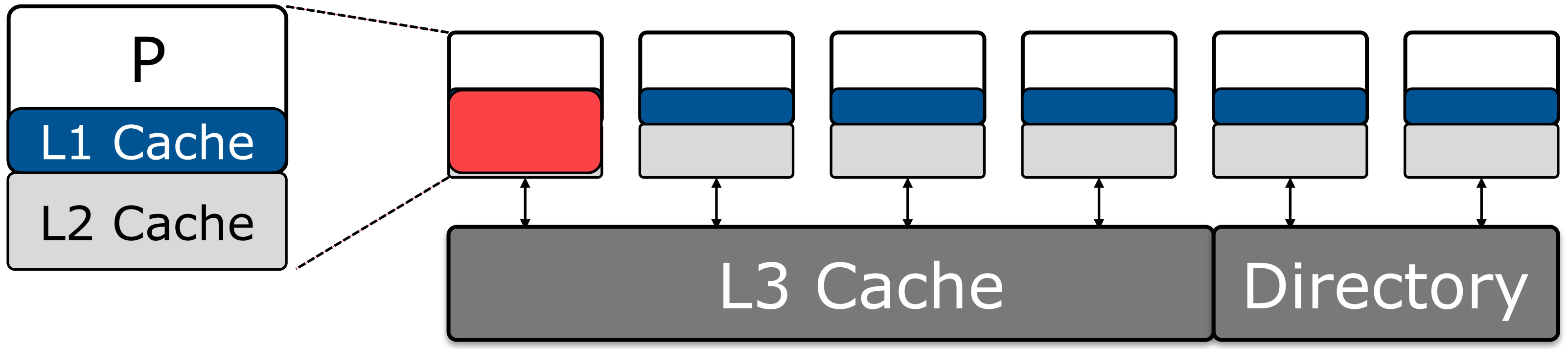
Reminder: Multiprocessor Caching

- Modern CPUs have a hierarchy of caches, that are kept coherent transparently
  - Eases programmability
  - … But, can hurt performance if we're not careful!
  - e.g., Data going back/forth between two caches



Reminder: Multiprocessor Caching

- Modern CPUs have a hierarchy of caches, that are kept coherent transparently
  - Eases programmability
  - … But, can hurt performance if we're not careful!
  - e.g., Data going back/forth between two caches



Latency Numbers Every Computer Scientist Should Know (× 1 billion)

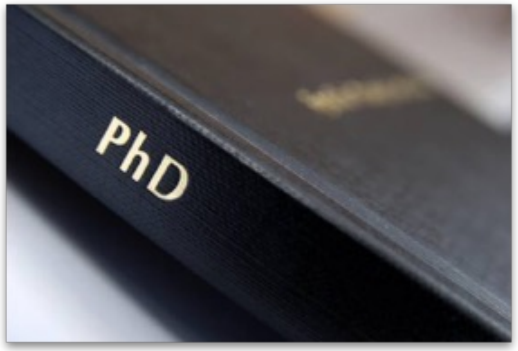
branch mispredict / div() / sqrt() – 5s

Mutex lock/unlock 25s

Main memory reference – 100s

Read 1 MB sequentially from main memory – 3 days

Read 1 MB sequentially from disk – 1 year



Network roundtrip EU ⇄ US – 4.8 years

This Lecture

- First, how to improve program performance
  - **Given** that we know size & organization of the cache
  - It’s **all** about maximizing fraction of cache hits
- Second, parallel scheduling optimizations

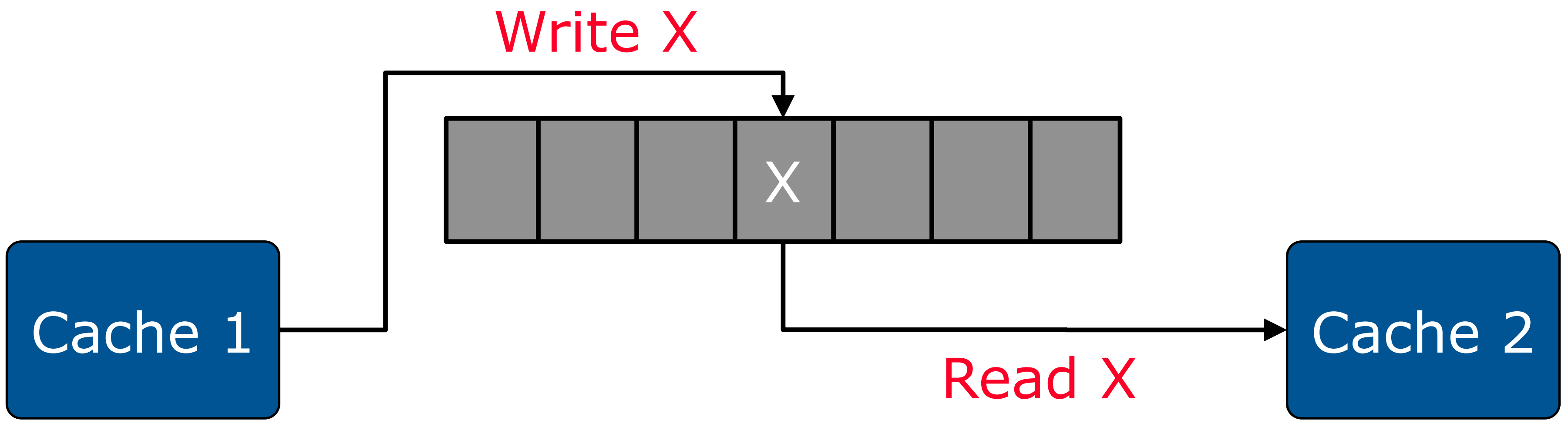
Hierarchy Level	Latency (cycles)	Latency (ns)
L1	2-3	1-3
L2	6-10	5-10
LLC	25-50	15-30
Memory	100-200	50+

Review: 4C Cache Miss Model

- Compulsory: Access data for the first time
  - Need to actually go get the data from memory
- Capacity: When the cache is not big enough
  - e.g., Our cache is 32kB, we access a 1MB array.
- Conflict: When two addresses map to the same set and way of the cache, evicting one of them
  - e.g., Evenly striding over an array
- **Coherence**: When a block is removed due to coherence messages from another core
- Reminder: cache block == cache line
  - In the rest of this lecture we may go back and forth

Coherence Misses

- Coherence adds extra misses due to transfers
  - When two cores access the same cache line
- Types of coherence misses
  - True sharing → e.g., producer/consumer comm.



Coherence Misses

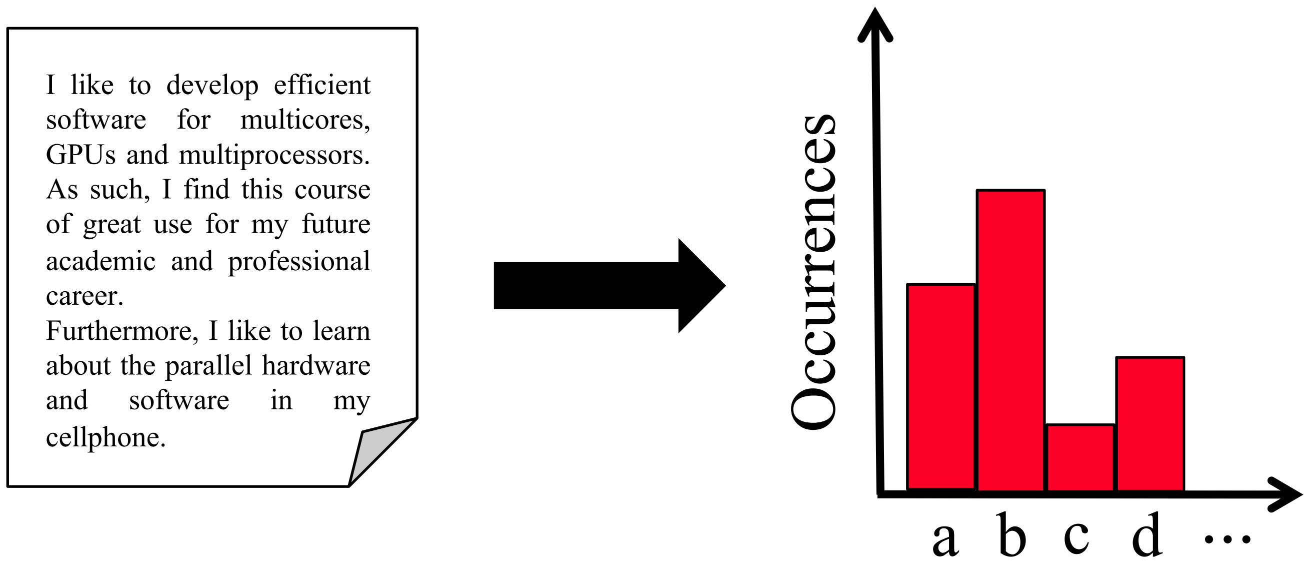
- Coherence adds extra misses due to transfers
  - When two cores access the same cache line

- Types of coherence misses
  - True sharing → Producer/consumer communication when processors read and write the same variable
  - False sharing → Processors updating different data which are placed **in the same cache block**

- We will address each issue with examples

Histogram Example

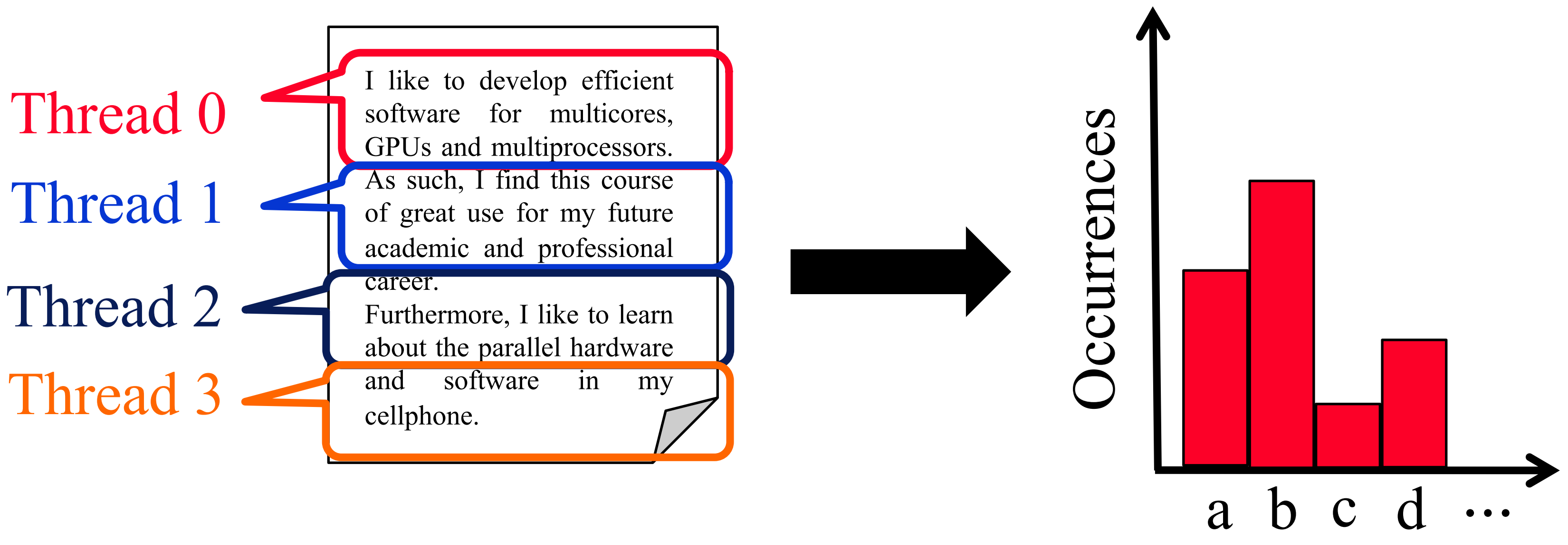
- Count num. of ASCII characters in an input text
- Input: ASCII characters in array format
- Output: Histogram w. each bucket, # occurrences



I like to develop efficient software for multicores, GPUs and multiprocessors. As such, I find this course of great use for my future academic and professional career.
Furthermore, I like to learn about the parallel hardware and software in my cellphone.

Histogram Example

- Simple work division
  - Partition the text array across threads
  - OpenMP's `parallel for` does this automatically


Thread 0
Thread 1
Thread 2
Thread 3
I like to develop efficient software for multicores, GPUs and multiprocessors.
As such, I find this course of great use for my future academic and professional career.
Furthermore, I like to learn about the parallel hardware
and software in my cellphone.
Occurrences
a b c d …

Histogram Example #1

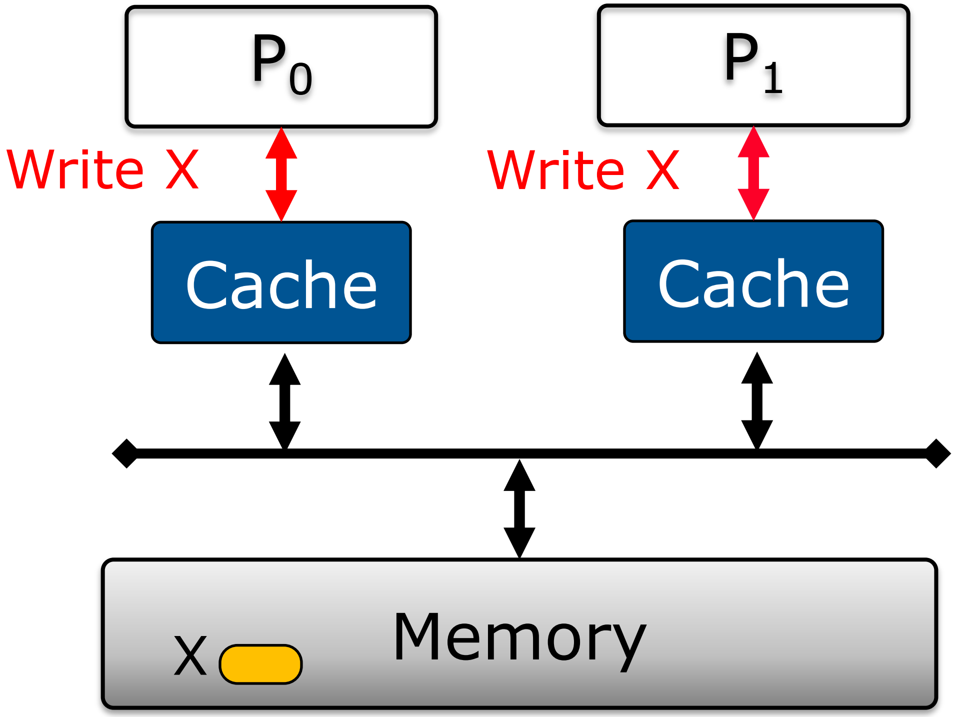
```
int histogram[N_BUCKETS];
int in_data[INPUT_SIZE];

#pragma omp parallel for shared (in_data, histogram)
for(int i = 0; i < INPUT_SIZE;i++) {
   unsigned bucket = calc_hist(in_data[i]);

   #pragma omp critical
   histogram[bucket]++;
}
```

- Characters are mapped to buckets by `calc_hist`
- **Input** array divided between threads (by OMP)
- Protect the **output** array with a critical section

Histogram: True Sharing

- Assume `histogram` is mapped to cache block X
- True sharing when processors modify the histogram:

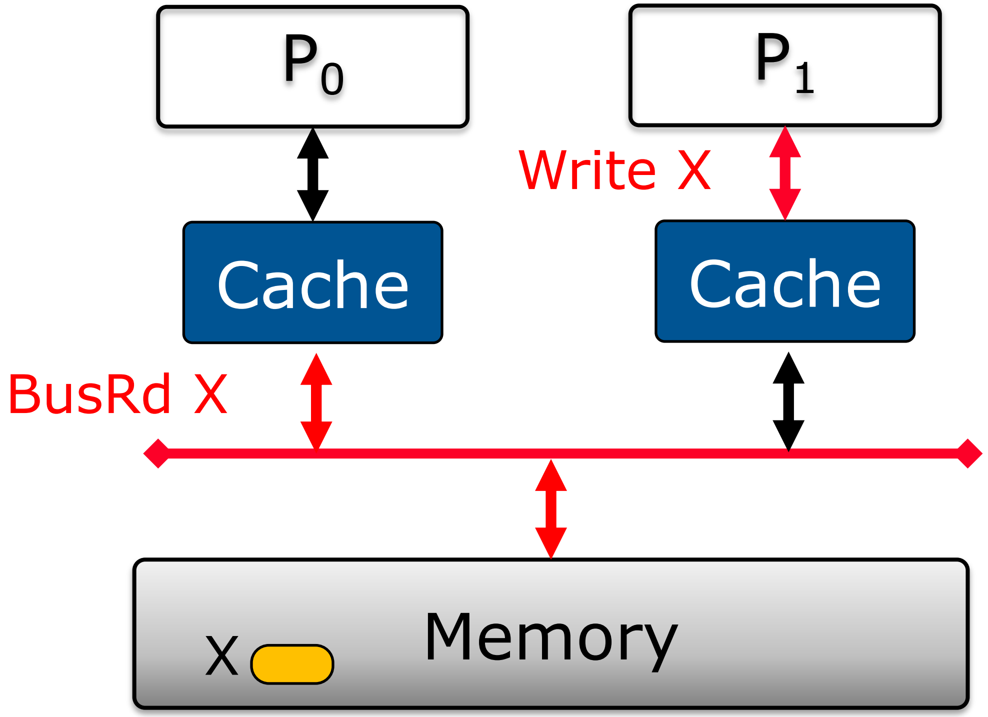
```
histogram['a']++;
```



Histogram: True Sharing

- Assume `histogram` is mapped to cache block X
- True sharing when processors modify the histogram:

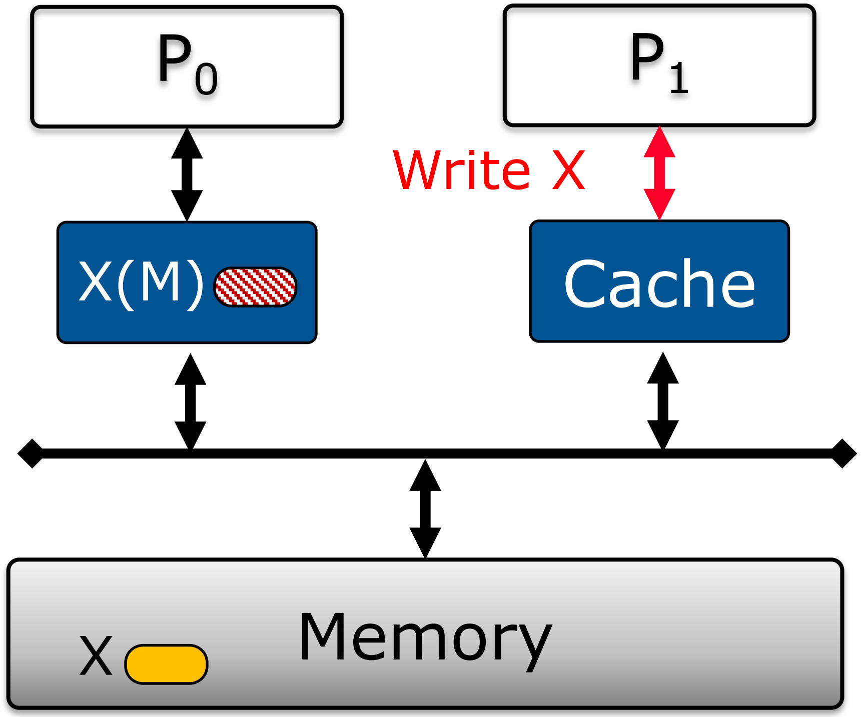
```
histogram['a']++;
```



Histogram: True Sharing

- Assume `histogram` is mapped to cache block X
- True sharing when processors modify the histogram:

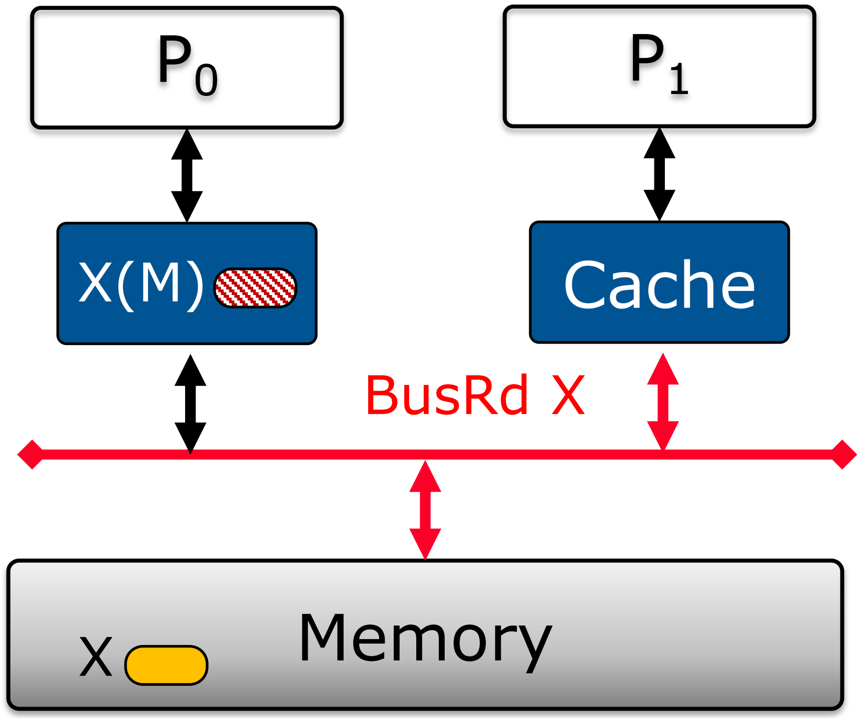
```
histogram['a']++;
```



Histogram: True Sharing

- Assume `histogram` is mapped to cache block X
- True sharing when processors modify the histogram:

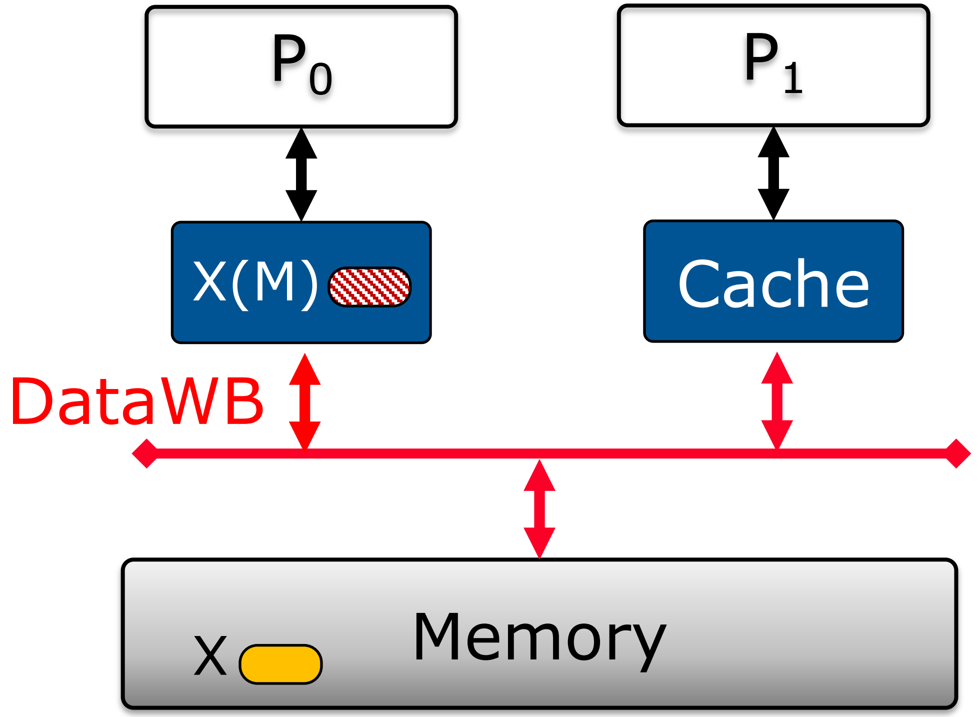
```
histogram['a']++;
```



Histogram: True Sharing

- Assume `histogram` is mapped to cache block X
- True sharing when processors modify the histogram:

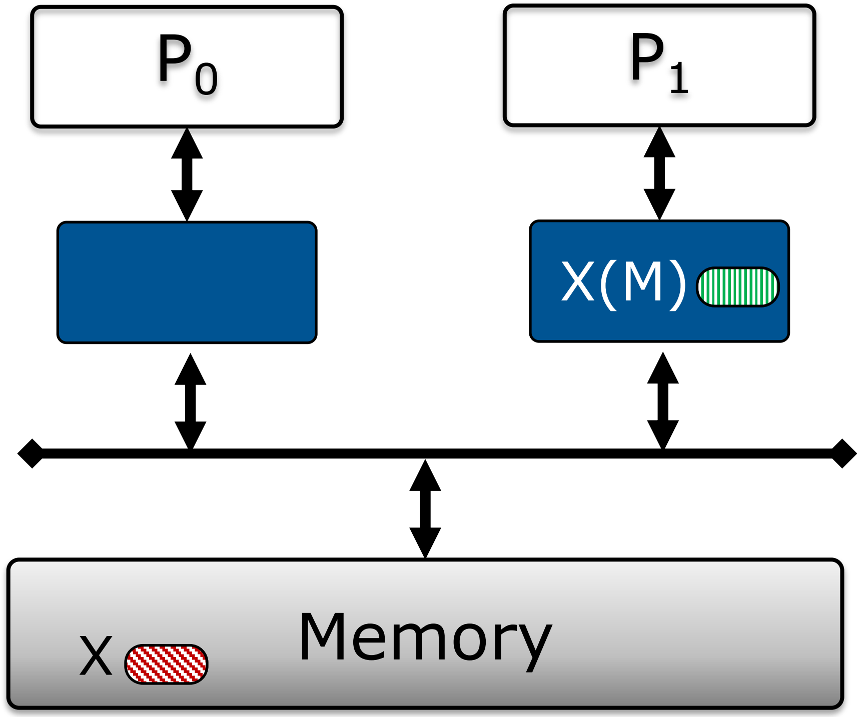
```
histogram['a']++;
```



Histogram: True Sharing

- Assume `histogram` is mapped to cache block X
- True sharing when processors modify the histogram:

```
histogram['a']++;
```



Exercise: Reducing True Sharing

- Given that we have to preserve correct value…
- How can we eliminate the coherence activity from repeatedly accessing the `histogram` array?

Reducing True Sharing: Example #2

```
int histogram[N_BUCKETS];
int tmp_hist[N_THREADS][N_BUCKETS];
int in_data[INPUT_SIZE];

#pragma omp parallel for shared (in_data, tmp_hist)
for(int i = 0; i < INPUT_SIZE;i++) {
   int tid = omp_get_thread_num();
   unsigned bucket = calc_hist(in_data[i]);
   tmp_hist[tid][bucket]++;
}
```

- Partition the data structures
  - Threads update their own `tmp_hist` and merge later
  - Requires another loop (next slide)

Reducing True Sharing: Example #2

```
int histogram[N_BUCKETS];
int tmp_hist[N_THREADS][N_BUCKETS];
int in_data[INPUT_SIZE];
#pragma omp parallel for shared (in_data, tmp_hist)
{...}

// Merge loop
#pragma omp parallel shared (tmp_hist, histogram)
for(int i = 0; i < N_BUCKETS; i++) {
   int tid = omp_get_thread_num();
   #pragma omp critical
   histogram[i] += tmp_hist[tid][i];
}
```

- Merge loop
  - Does **not** use `parallel for`, all threads need to add all buckets up
  - Note both loops declare `tmp_hist` as **shared**

Difference in Performance

- Example #1:
  - Critical section (lock/unlock) **`INPUT_SIZE`** times
  - Read/Write **`histogram[bucket]`** **`INPUT_SIZE`** times

- Example #2:
  - Read/Write **`tmp_hist[][bucket]`** **`INPUT_SIZE`** times
  - Critical section (lock/unlock) **`N_BUCKET *`** # of threads
  - Read/Write **`histogram[bucket]`** **`N_BUCKET *`** # of threads

- If **`INPUT_SIZE >> N_BUCKET *`** # of threads, we eliminate a lot of critical sections & true sharing
- Can we do better?

Reducing True Sharing: Example #3

```
int histogram[N_BUCKETS];
int tmp_hist[N_THREADS][N_BUCKETS];
int in_data[INPUT_SIZE];
#pragma omp parallel for shared (in_data, tmp_hist)
{...}
// Merge loop
#pragma omp parallel for shared (tmp_hist, histogram)
for(int i = 0; i < N_BUCKETS; i++) {
    int tid;
    for (tid = 0; tid < omp_get_num_threads(); tid++) {
        histogram[i] += tmp_hist[tid][i];
     }
}
```

- Merge loop
  - Divides the buckets by the threads
  - Eliminates all critical sections, but every thread accesses a fraction of everyone's histogram

True Sharing Summary

- True sharing is inherent to the algorithm
  - Can only optimize it
  - Can not eliminate it

- In general, approaches for optimization:
  1. Divide up input data in advance
  2. Privatize results when possible, reduce communication

Coherence Misses Continued

- Coherence adds extra misses due to transfers
  - When two cores access the same cache line

- Types of coherence misses
  - False sharing → Processors updating different data which happen to be **in the same cache block**

Counting Odd Numbers Example

```
int counters[NTHREADS];
int in_data[INPUT_SIZE];

#pragma omp parallel for shared (in_data,counters)
for(int i = 0; i < INPUT_SIZE;i++) {
   int tid = omp_get_thread_num();

   if( in_data[i] % 2 ) {
      counters[tid]++;
   }
}
```

- **No** true sharing, threads update own counter
  - Therefore, no need to use `#pragma omp critical` as we did previously
  - Assume `counters[]` stored in Block X

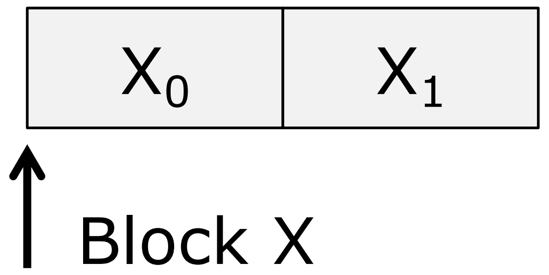
Problem: False Sharing

Assumptions:

- Two processors, two words per cache block
- Block X is initially shared by both processors

Notation:

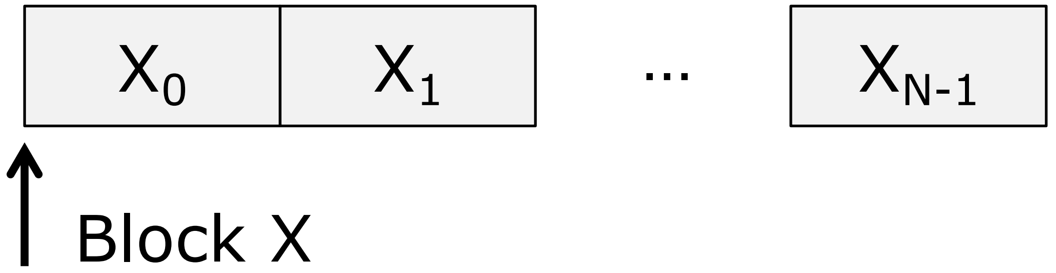
- X_0 and X_1: least and most significant words of block X



Problem: False Sharing

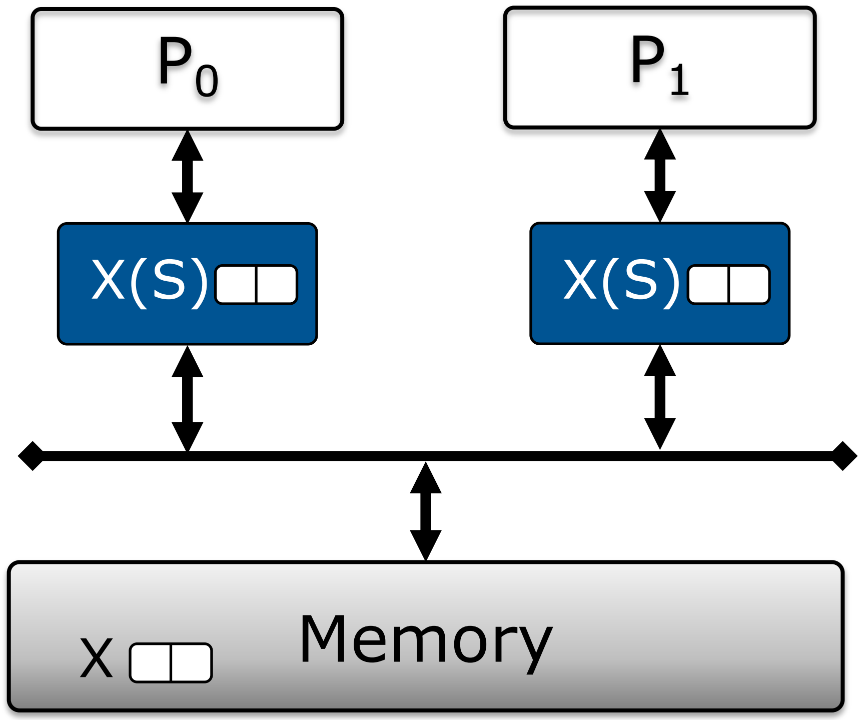
- Thread-private counters reside in same blocks!
  - Below, X_0-X_{N-1} represent counters for N threads

- As cores update them, coherence messages repeatedly sent around the chip



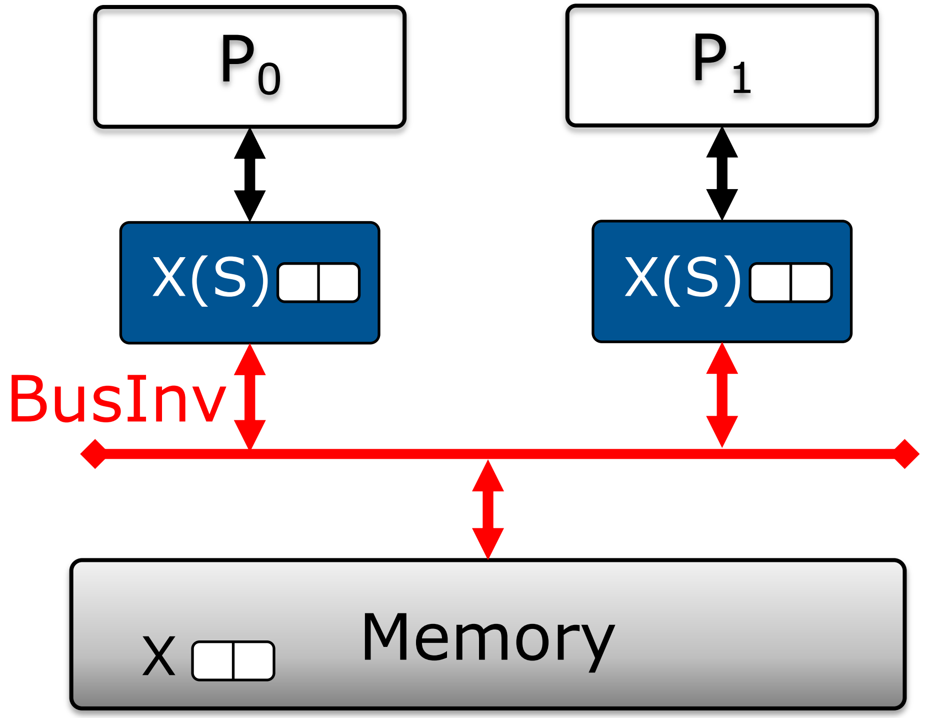
False Sharing in Hardware

1. P_0 Read $X_0 \rightarrow$ **E**
2. P_1 Read $X_1 \rightarrow$ **S**



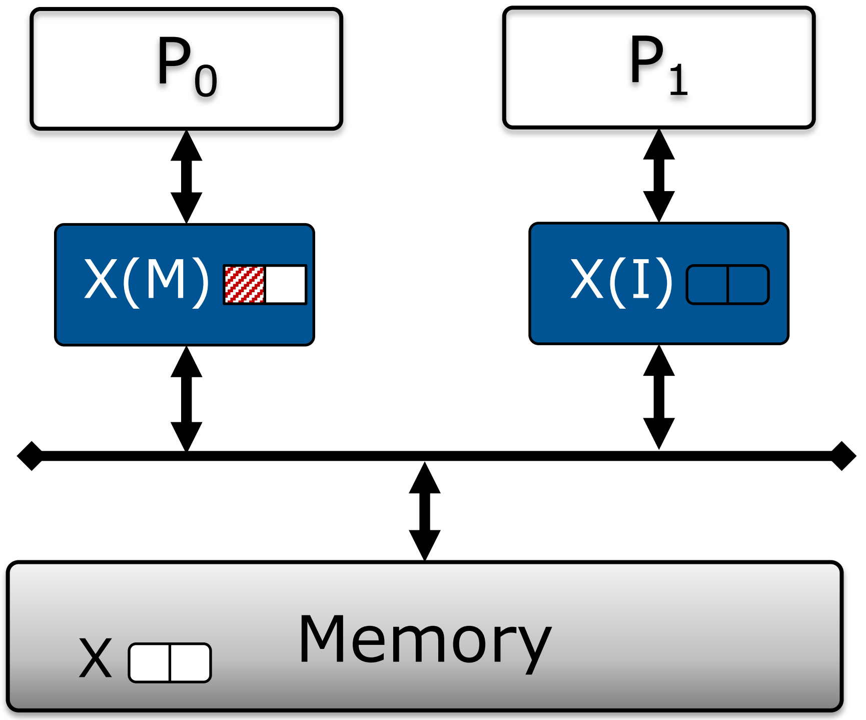
False Sharing in Hardware

1. P_0 Read X_0 → **E**
2. P_1 Read X_1 → **S**
3. P_0 Write X_0
   a) Need to invalidate A in P_1
   b) Send BusInv, P_1 → **I**



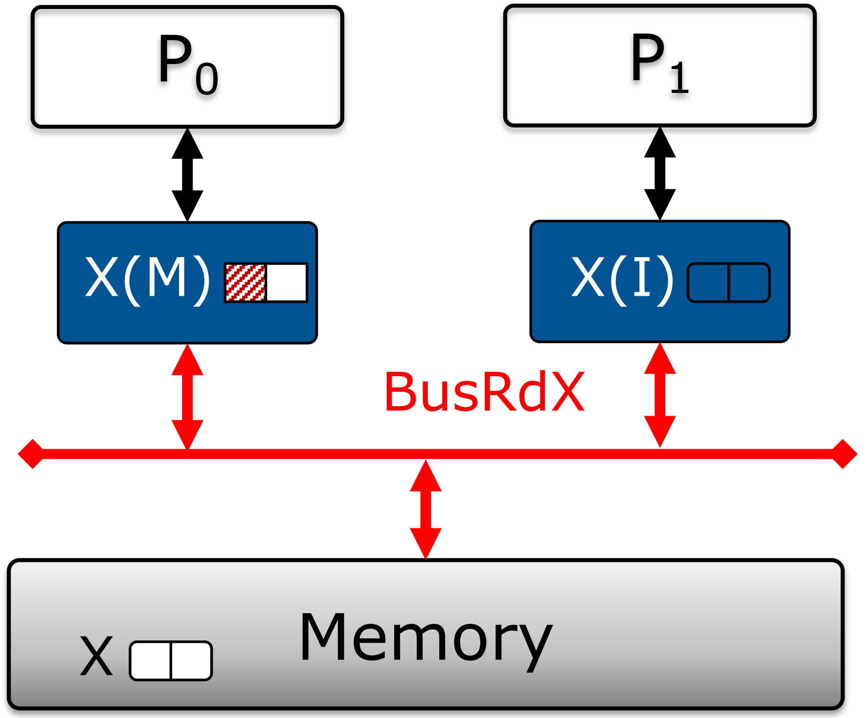
False Sharing in Hardware

1. P_0 Read $X_0 \rightarrow$ **E**
2. P_1 Read $X_1 \rightarrow$ **S**
3. P_0 Write $X_0 \rightarrow$ **M**



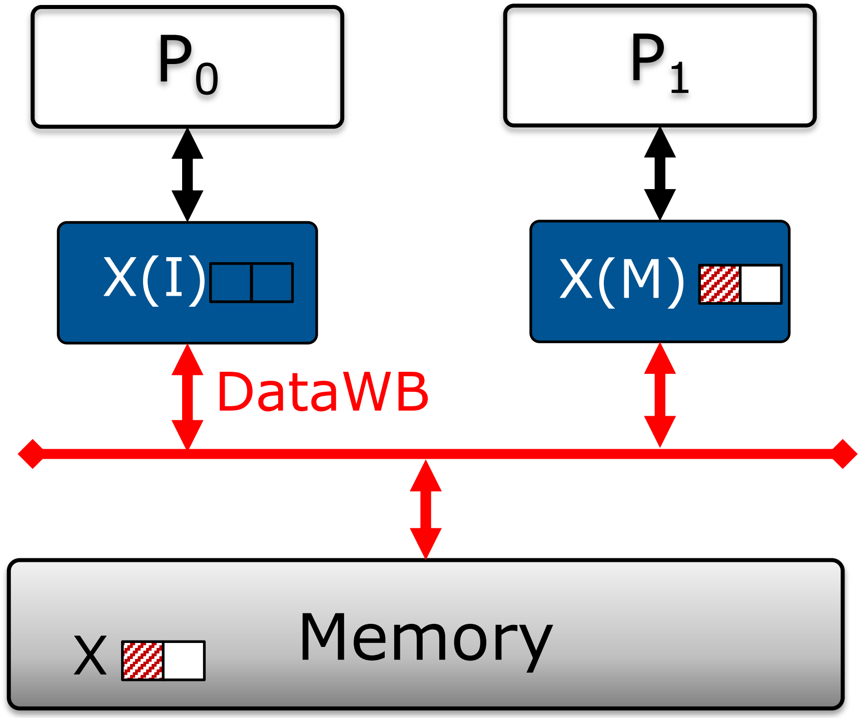
False Sharing in Hardware

1. P_0 Read $X_0 \rightarrow$ **E**
2. P_1 Read $X_1 \rightarrow$ **S**
3. P_0 Write $X_0 \rightarrow$ **M**
4. P_1 Write X_1
   a) Need to invalidate X in P_0
   b) Send BusRdX, $P_0 \rightarrow$ **I**



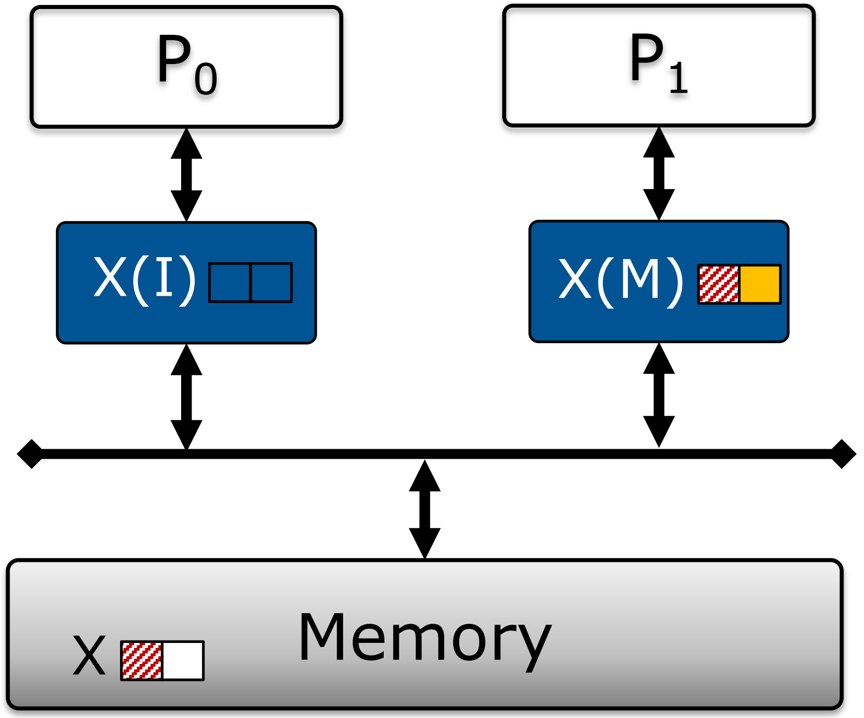
False Sharing in Hardware

1. P_0 Read $X_0 \rightarrow$ **E**
2. P_1 Read $X_1 \rightarrow$ **S**
3. P_0 Write $X_0 \rightarrow$ **M**
4. P_1 Write X_1
   a) Need to invalidate X in P_0
   b) Send BusRdX, $P_0 \rightarrow$ **I**
   c) P_0 sends data to P_1 & Mem



False Sharing in Hardware

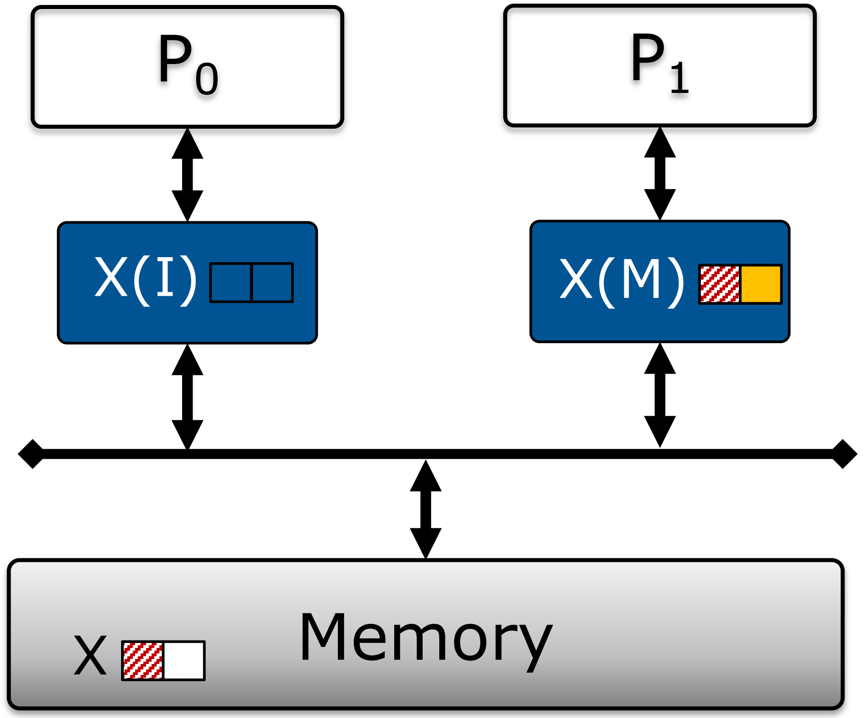
1. P_0 Read $X_0 \rightarrow$ **E**
2. P_1 Read $X_1 \rightarrow$ **S**
3. P_0 Write $X_0 \rightarrow$ **M**
4. P_1 Write $X_1 \rightarrow$ **M**



False Sharing in Hardware

1. P_0 Read $X_0 \rightarrow$ **E**
2. P_1 Read $X_1 \rightarrow$ **S**
3. P_0 Write $X_0 \rightarrow$ **M**
4. P_1 Write $X_1 \rightarrow$ **M**

- Block A continues bouncing between caches!
- Excess invalidations, and memory updates



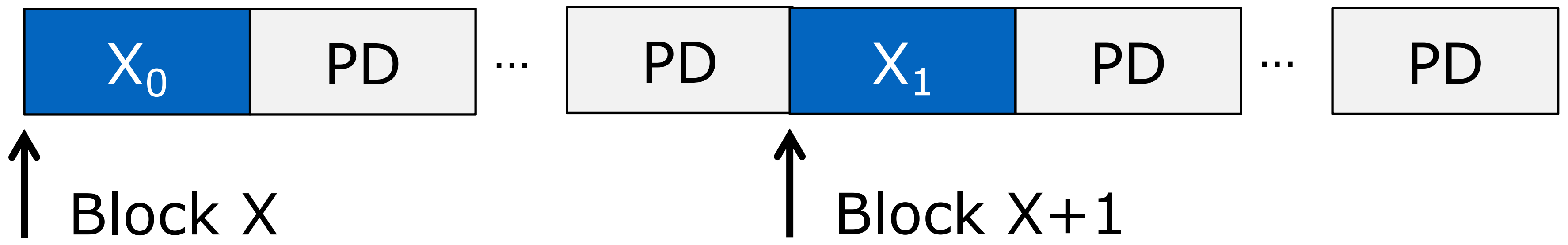
False Sharing Solution: Data Padding

```
typedef struct {
    unsigned _c;
    unsigned _padding[ BLK_SIZE/ UNSIG_SIZE - 1];
} PaddedCounter;
PaddedCounter counters[NTHREADS];

#pragma omp parallel for
{
    ...
    counters[tid]._c++;
}
```

- Move the counters into different cache blocks
  - Introduce “excess” variables that are never used

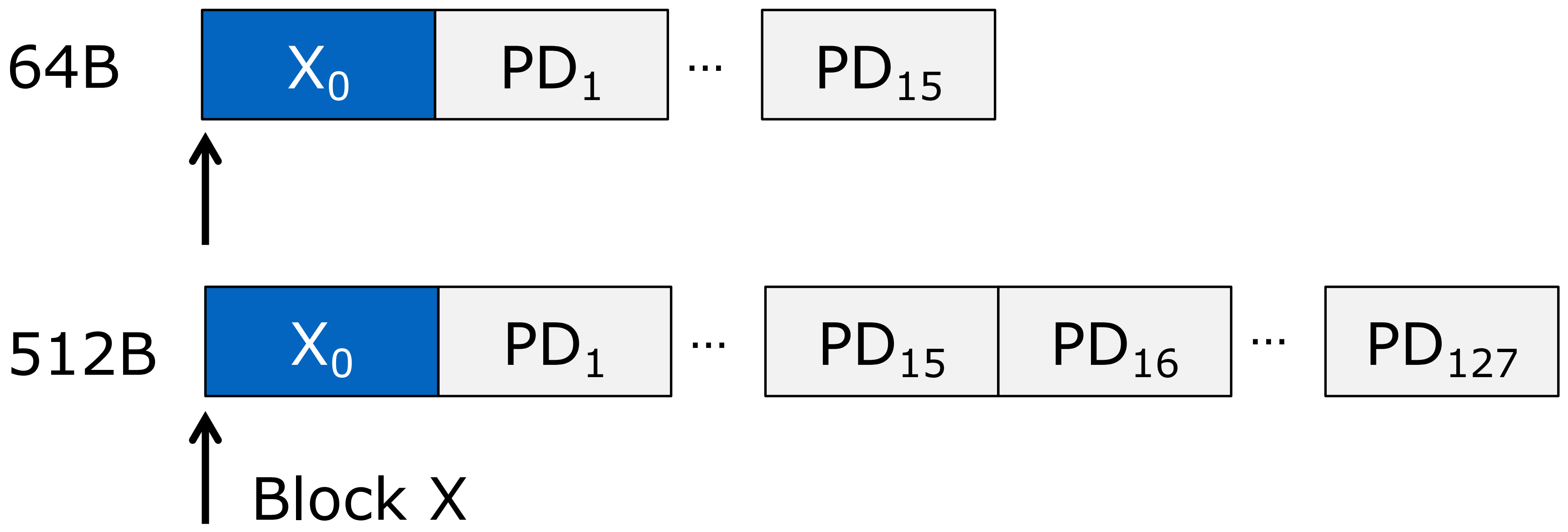
Data Padding Layout

- Assuming aligned data padding:
  - P represents the miscellaneous padded data
  - Now we have one counter per cache block
  - No coherence messages during parallel loop

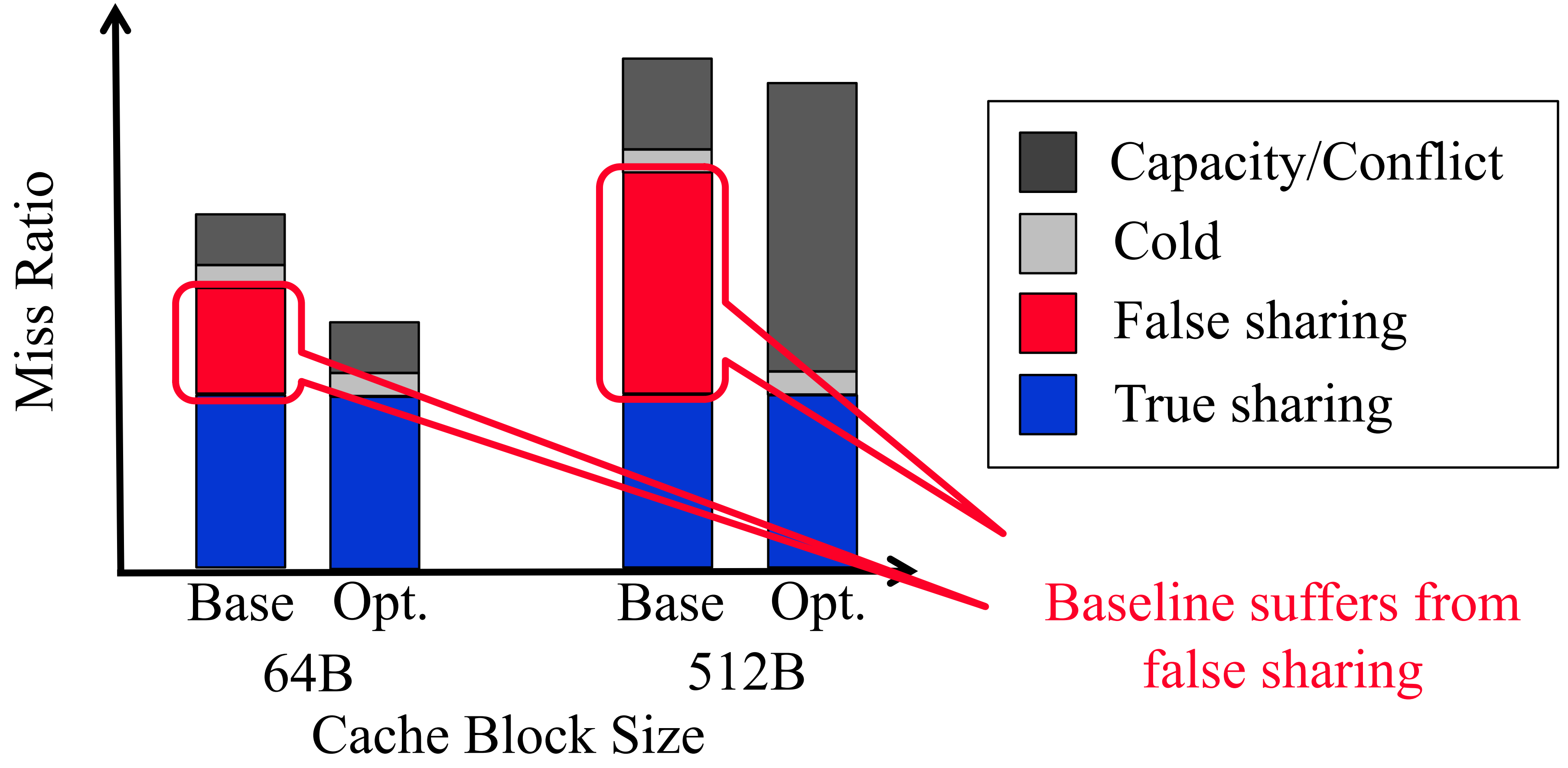


Software Tradeoffs: Data Padding

- Behavior depends on underlying hardware
- For example, how does data padding perform with different cache block sizes?
  - Assuming 4B counters

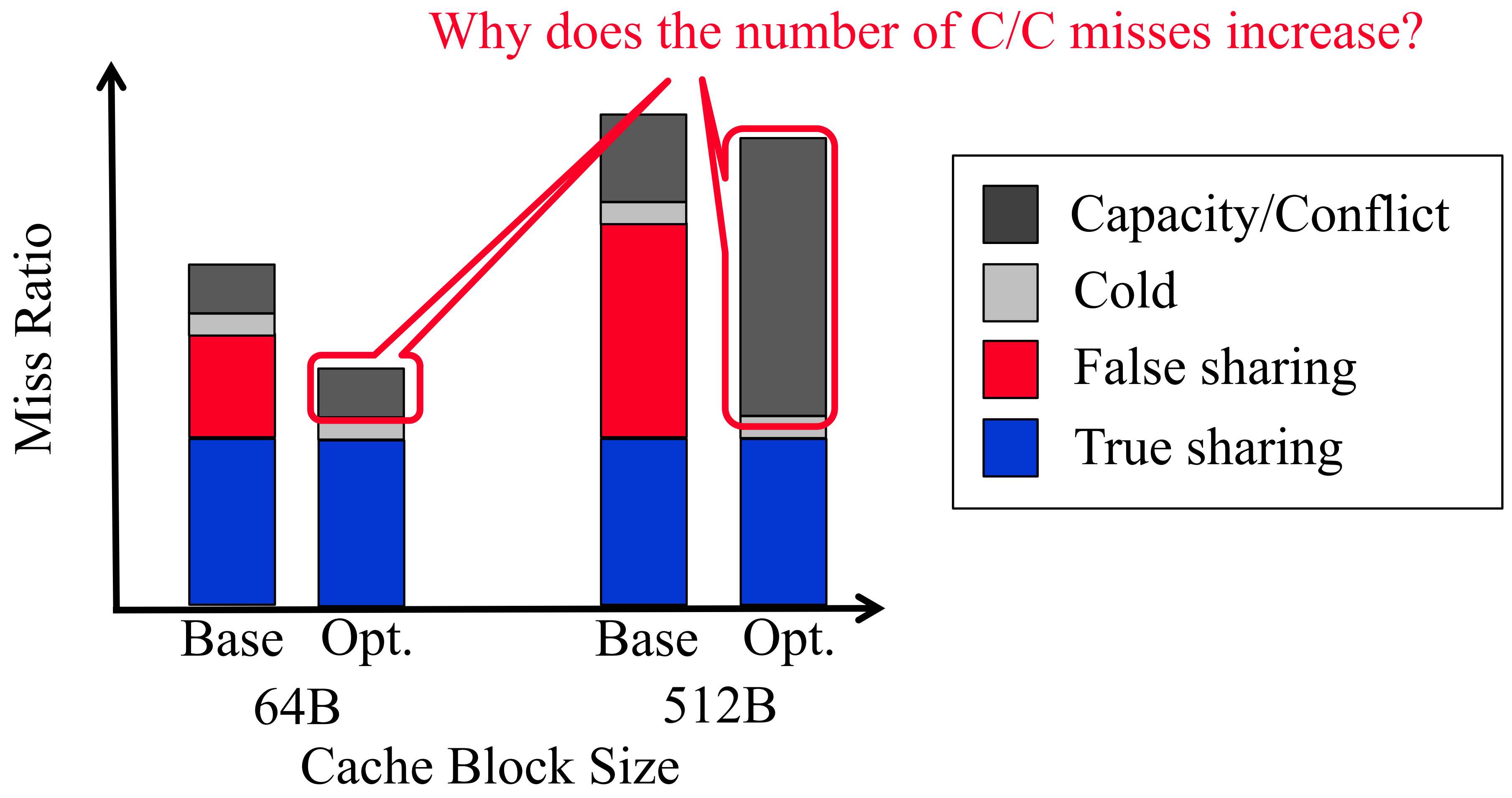


Software Tradeoffs: Data Padding



Baseline suffers from false sharing

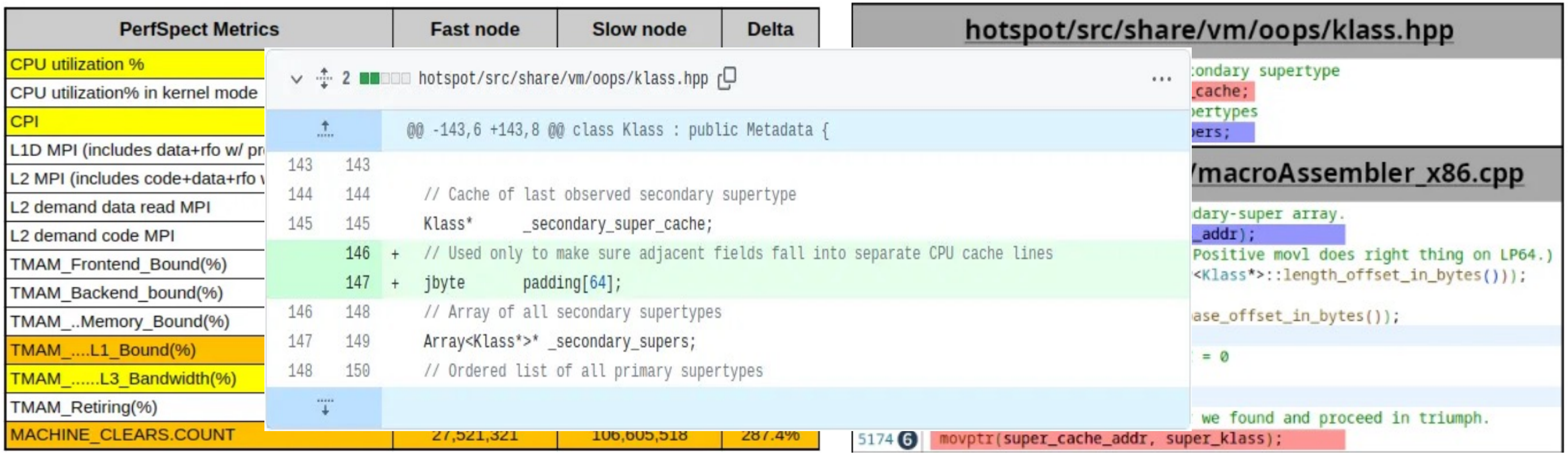
Software Tradeoffs: Data Padding



Software Tradeoffs: Data Padding

- Application performance depends on the underlying hardware
- Number of coherence misses affected by:
  - Cache size
  - Number of processors
  - Cache block size

False Sharing in Netflix's JAVA Microservice



PerfSpect Metrics	Fast node	Slow node	Delta
CPU utilization %			
CPU utilization% in kernel mode			
CPI			
L1D MPI (includes data+rfo w/ pr			
L2 MPI (includes code+data+rfo v			
L2 demand data read MPI			
L2 demand code MPI			
TMAM_Frontend_Bound(%)			
TMAM_Backend_bound(%)			
TMAM_..Memory_Bound(%)			
TMAM_....L1_Bound(%)			
TMAM_......L3_Bandwidth(%)			
TMAM_Retiring(%)			
MACHINE_CLEARS.COUNT	27,521,321	106,605,518	287.4%

```
2 hotspot/src/share/vm/oops/klass.hpp

@@ -143,6 +143,8 @@ class Klass : public Metadata {

     // Cache of last observed secondary supertype
     Klass*      _secondary_super_cache;
   + // Used only to make sure adjacent fields fall into separate CPU cache lines
   + jbyte       padding[64];
     // Array of all secondary supertypes
     Array<Klass*>* _secondary_supers;
     // Ordered list of all primary supertypes
```

hotspot/src/share/vm/oops/klass.hpp

```
condary supertype
cache;
pertypes
pers;
```

/macroAssembler_x86.cpp

```
dary-super array.
_addr);
Positive movl does right thing on LP64.)
<Klass*>::length_offset_in_bytes()));

ase_offset_in_bytes());

 = 0

 we found and proceed in triumph.
5174 6  movptr(super_cache_addr, super_klass);
```

- In a blog post, Netflix reported how false sharing had been degrading the performance of one of their services by 3x !

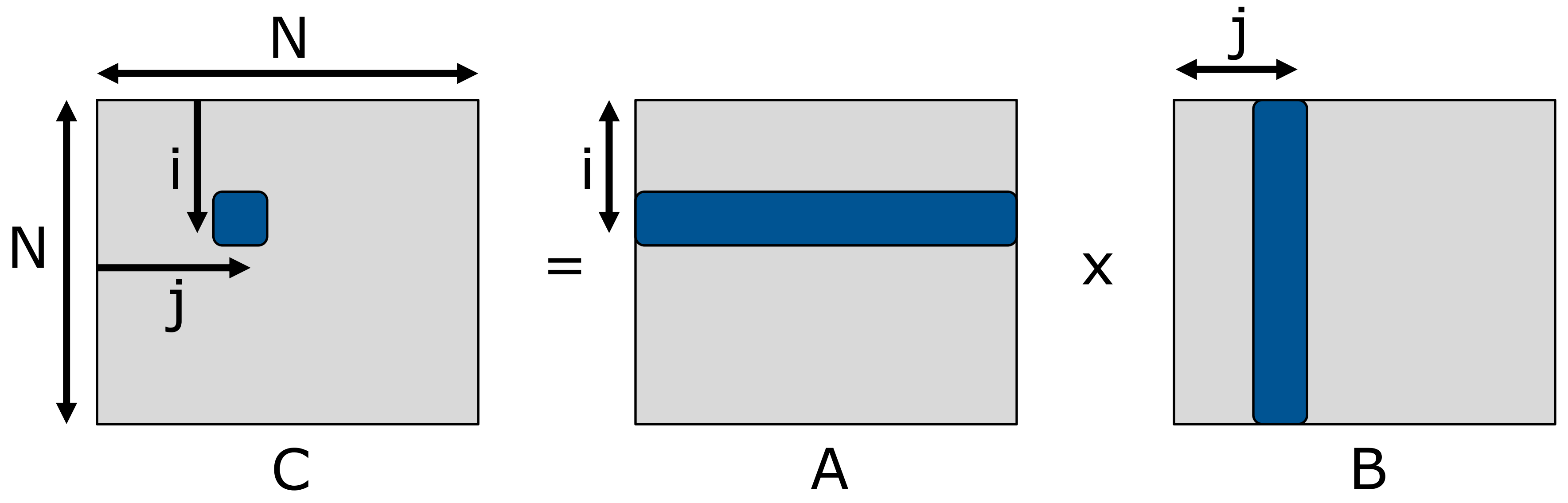
Source: https://netflixtechblog.com/seeing-through-hardware-counters-a-journey-to-threefold-performance-increase-2721924a2822

The Locality Principle

- Originally came from the first VM systems in 60’s
  - **Problem**: Bad page replacement policies led to swapping and unusable machines
  - **Solution**: Design memory to prioritize the “working set” of the currently executing applications

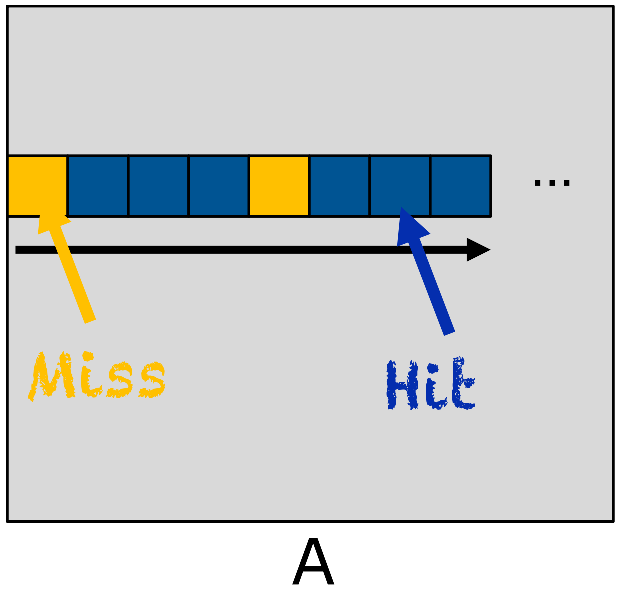
- A cache operates on the same principle
  - Keep frequently accessed data closer to the CPU
  - For maximum performance, make sure we are using it effectively

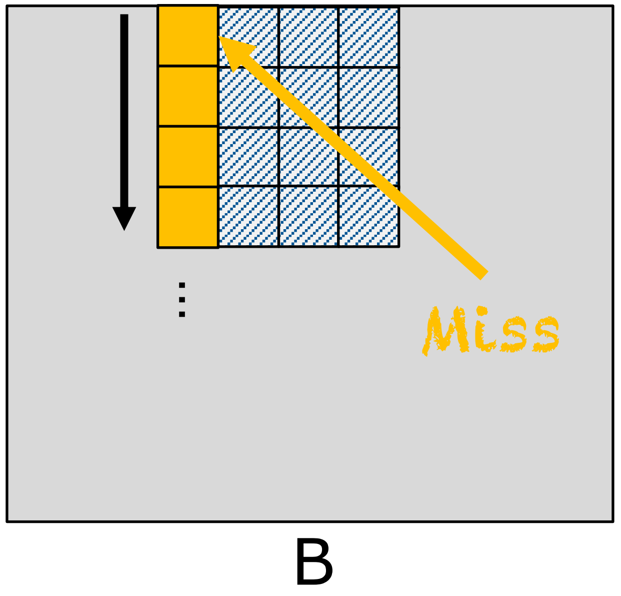
Example: Matrix Multiplication



- Output element (i,j) = Row i * Column j
- Assume N is large (~ 10^5), 4-byte elements

Matrix Multiplication Locality



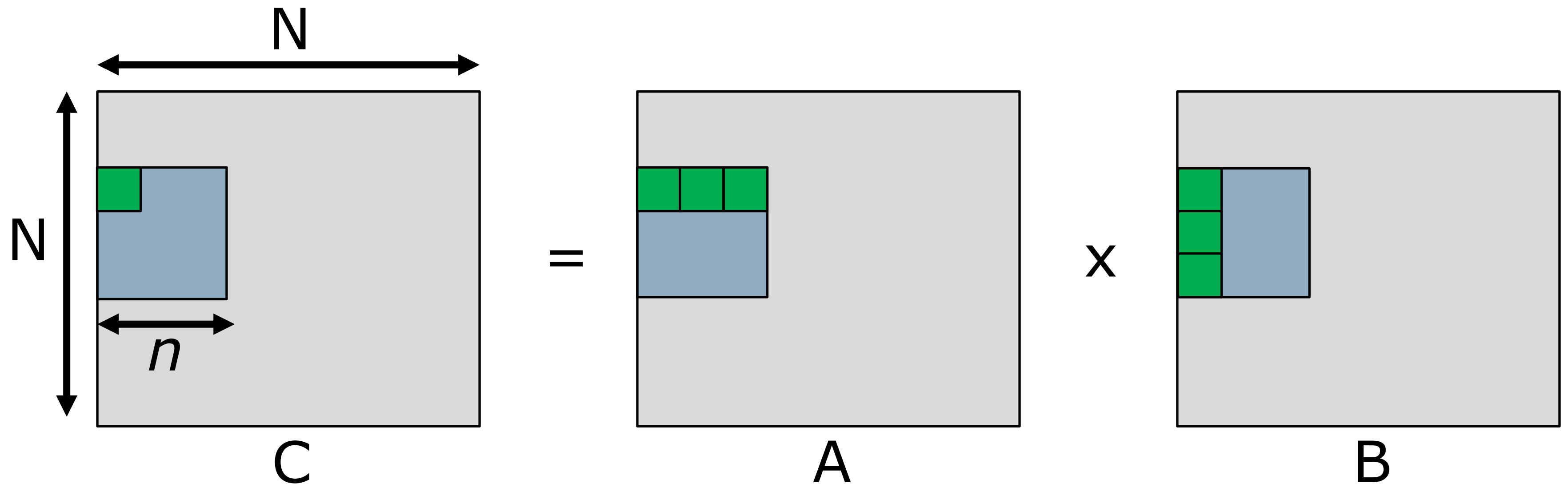


- For each element in `C`:
  - A array, 1 miss for every $\frac{Cache\ line\ size}{4}$ elements
  - B array, miss on every element

Matrix Multiplication Locality

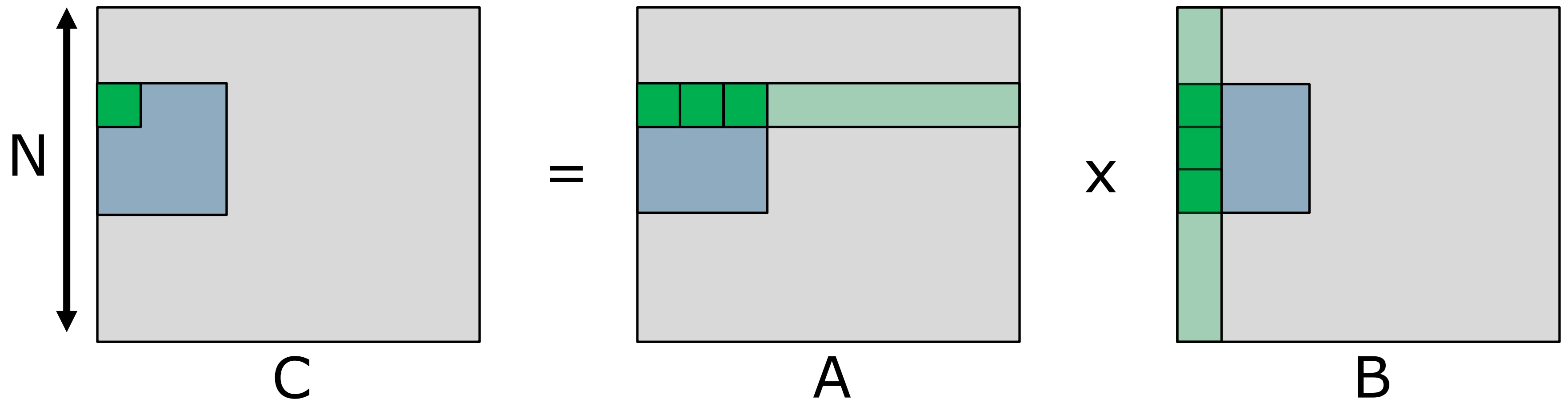
- **Problem:** Arrays are too large to fit in caches!
  - How many misses for every element in C (N^2 of them)?
  - Cache misses ~ $\underbrace{1}_{\text{Array C}} + \underbrace{N\left[\frac{4}{\mathit{Cache\ line\ size}}\right]}_{\text{Array A}} + \underbrace{N}_{\text{Array B}}$
  - Most misses come from A&B (C doesn't count)
- For the multiplication as a whole:
  - Assume generalized element size
  - Roughly $(N^3(1+c) + N^2)$ which is $O(N^3)$ misses
  - *c* is $\frac{\mathit{element\ size\ in\ bytes}}{\mathit{Cache\ line\ size}}$

Blocking For Locality



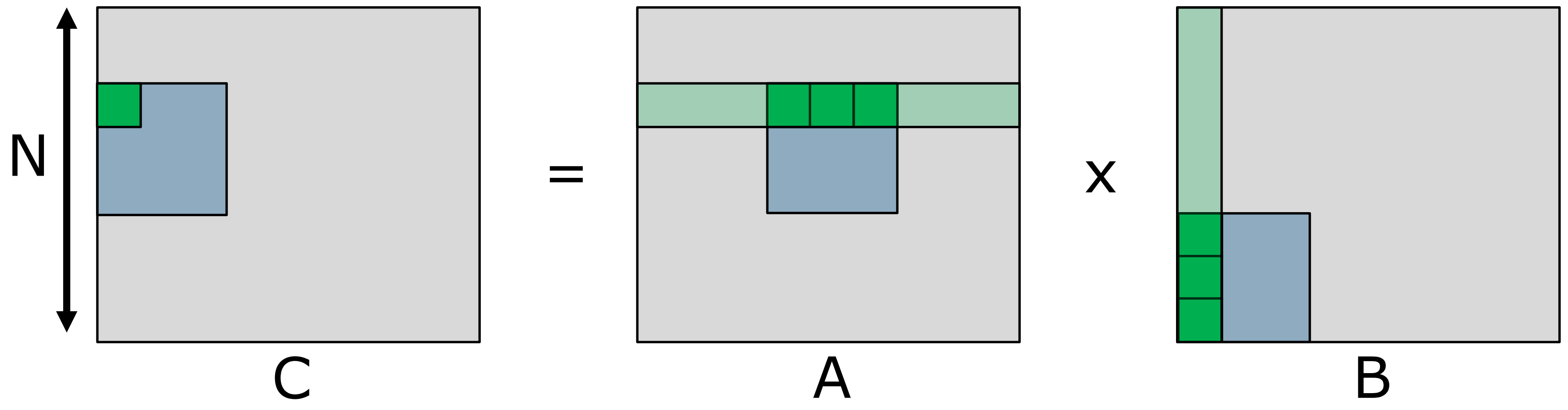
- Divide the matrix into sub-matrices, small enough to fit in the cache
  - Reuse them on every “block iteration”
  - Sub-matrix size: n^2

Blocking Correctness



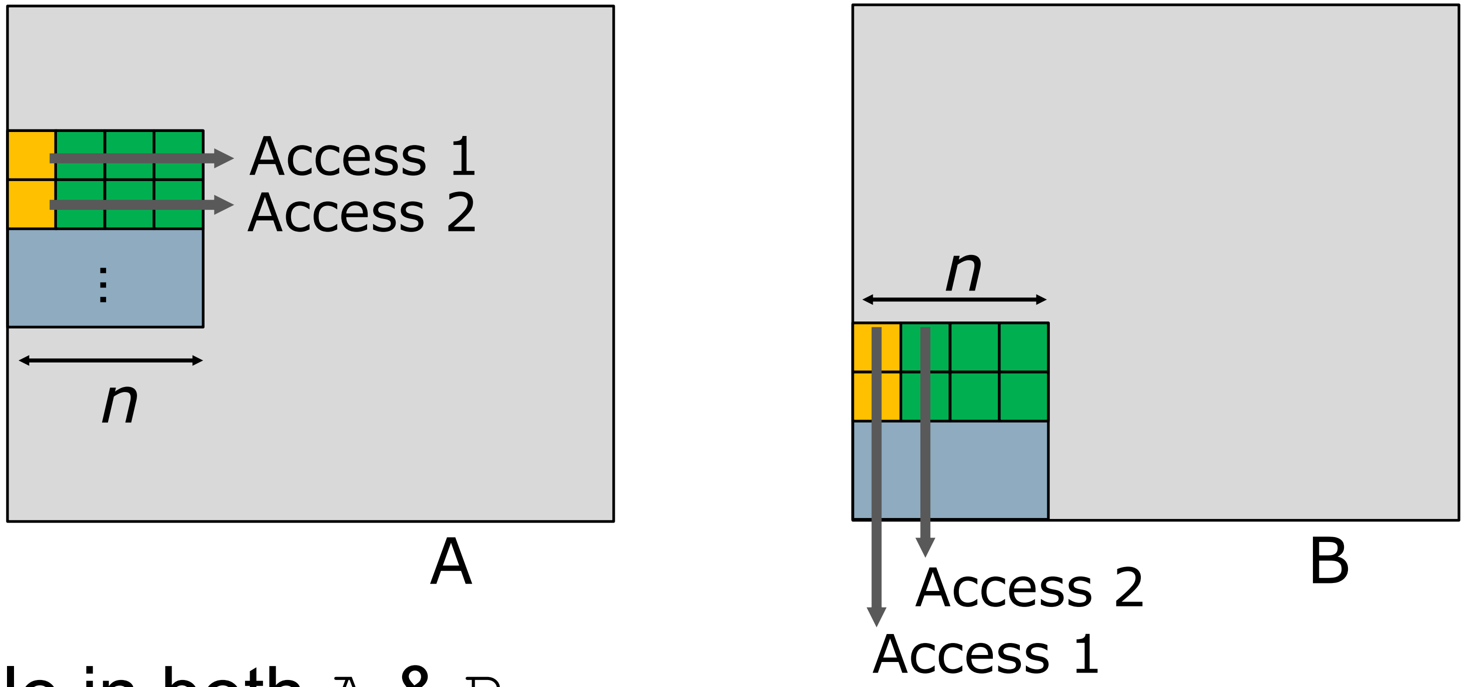
- Output sums in `C` gradually accumulate
  - `A` and `B` tiles slide over the large arrays
  - e.g., The green element in `C` will eventually get all of row `i` and column `j` as the tiles move

Blocking Correctness



- Output sums in `C` gradually accumulate
  - `A` and `B` tiles slide over the large arrays
  - e.g., The green element in `C` will eventually get all of row `i` and column `j` as the tiles move

Blocking Locality



- For each tile in both `A` & `B`
  - `A`: $\left\lceil \frac{4 * n}{Cache\ line\ size} \right\rceil$ misses every **new** row due to spatial locality
  - `B`: **same** as `A`, similar spatial locality in next column
- C block remains in cache (no new misses)

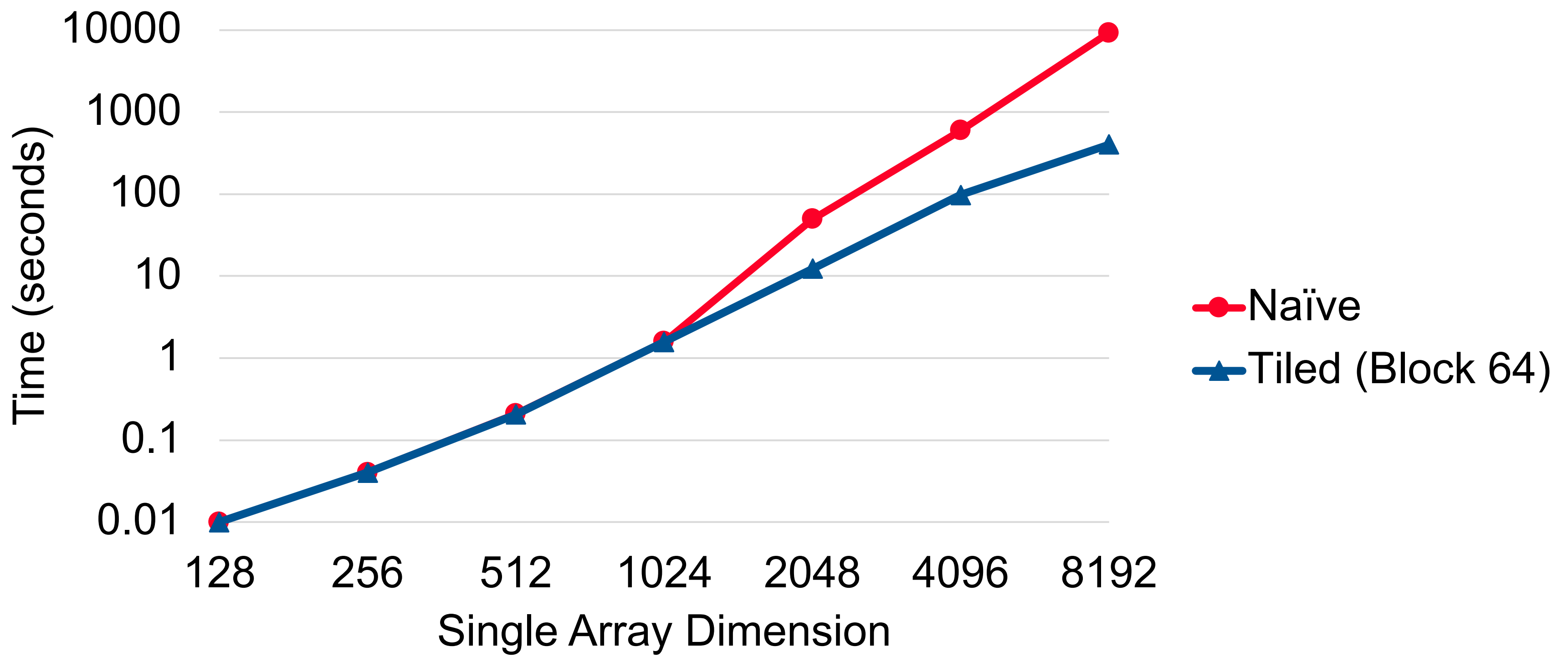
Blocking Locality

- For the whole array, $\left(\frac{N}{n}\right)^2$ tiles are created
  - Each has 2*$\left\lceil\frac{4 * n}{Cache\ line\ size}\right\rceil$ misses
  - Total cache misses: $O(N^2)$

- Miss results collected from `cachegrind`
  - N = 512

Algorithm	Predicted Misses	Observed Misses
Naïve	~151M	~168M
Blocked	524k	650k

Blocking Performance

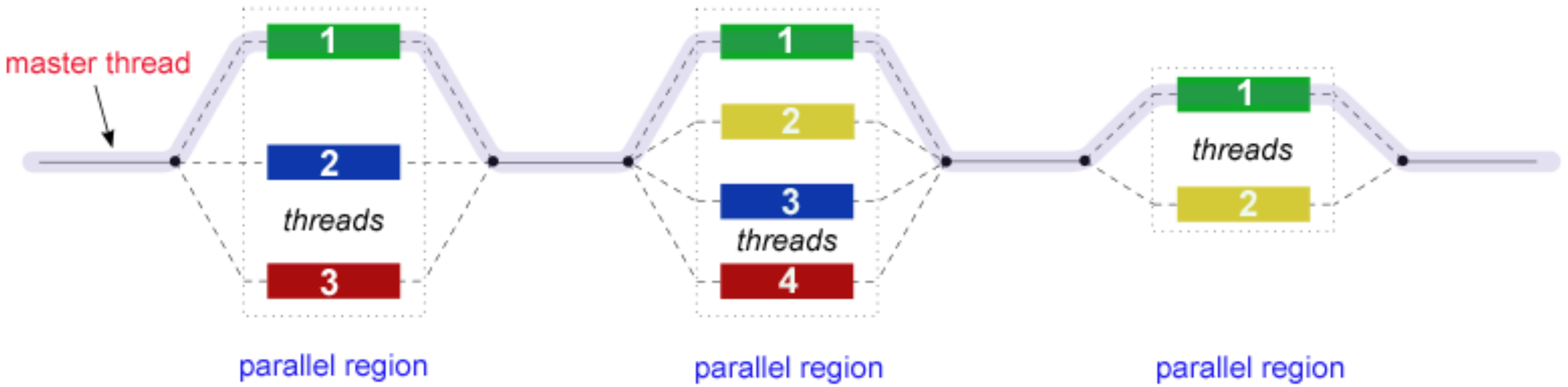


- Using `cachegrind`, miss rate of Naïve ~ 33%
- Tiled miss rate 1.7%, 22.5x speedup for N = 8k

Optimizations Part II:
Scheduling & Work Distribution

Review: OpenMP Execution Model

- Fork/join model
  - Initially only the master thread is active
    - Master thread executes until a parallel region is encountered
  - Fork: master thread creates a team of parallel threads
    - Statements in parallel region are executed in parallel
  - Join: team threads sync & terminate at the end of parallel region
    - Master thread continues executing sequentially



Fork/Joins Come with an Overhead

```
#pragma omp parallel for
for (i = 0; i < n; i++)
```

- What if `n` turns out to be small (e.g., 100)?

Fork/Joins Come with an Overhead

```
#pragma omp parallel for
for (i = 0; i < n; i++)
```

- What if `n` turns out to be small (e.g., 100)?
- If loop has too few iterations:
  - Fork/join overhead >= savings from parallel execution
  - There is no need to parallelize the loop
- The `if` clause:
  - instructs compiler to insert code that checks whether loop should be executed in parallel
  - Parallelize the loop only if it's worth it

```
#pragma omp parallel for if(n > 5000)
```

Fork/Joins Come with an Overhead

```
for (i=1; i<n; i++)
  #pragma omp parallel for
  for(j=0; j<m; j++)
    a[i][j]=2*a[i-1][j];
```

- What's wrong with this code?

Fork/Joins Come with an Overhead

```
for (i=1; i<n; i++)
  #pragma omp parallel for
  for(j=0; j<m; j++)
    a[i][j]=2*a[i-1][j];
```

- What's wrong with this code?
  - The inner loop does fork/join every iteration (n times)
- Excess fork/joins lowers performance
- Inverting loops helps performance:

```
#pragma omp parallel for private(i)
for (j=0; j<m; j++)
  for(i=1; i<n; i++)
    a[i][j]=2*a[i-1][j];
```

Fork/Joins Come with an Overhead

- What's wrong with this code?
  - Excess fork/joins
  - Each loop does fork/join

```
#pragma omp parallel for
for (. . .)
#pragma omp parallel for
for (. . .)
#pragma omp parallel for
for (. . .)
```

Fork/Joins Come with an Overhead

- What's wrong with this code?
  - Excess fork/joins
  - Each loop does fork/join

```
#pragma omp parallel for
for (. . .)
#pragma omp parallel for
for (. . .)
#pragma omp parallel for
for (. . .)
```

- Maximize parallel regions
  - Avoids excess fork/joins

```
#pragma omp parallel
{
  #pragma omp for
  for (. . .)
  #pragma omp for
  for (. . .)
  #pragma omp for
  for (. . .)
}
```

`for` Loop Itself Has an Overhead

```
#pragma omp parallel for
for (i=0; i<n; i++)
  a[i] = a[i] + 10;
```

Compile $\Longrightarrow$

```
loop: ld   r2, addr[r1]
      add  r2, r2, 10
      st   addr[r1], r2
      add  r1, r1, 1
      bne  r1, r3, loop
```

`for` Loop Itself Has an Overhead

```
#pragma omp parallel for
for (i=0; i<n; i++)
  a[i] = a[i] + 10;
```

Compile

```
loop: ld   r2, addr[r1]
      add  r2, r2, 10
      st   addr[r1], r2
      add  r1, r1, 1
      bne  r1, r3, loop
```

Loop Overhead

`for` Loop Itself Has an Overhead

```
#pragma omp parallel for
for (i=0; i<n; i++)
  a[i] = a[i] + 10;
```

Compile $\Rightarrow$

```
loop: ld   r2, addr[r1]
      add  r2, r2, 10
      st   addr[r1], r2
      add  r1, r1, 1
      bne  r1, r3, loop
```

Loop Overhead

- Loop unrolling
  - Perform multiple loop iterations in one
  - Reduce loop overhead
  - Compilers do this for simple patterns

```
#pragma omp parallel for
for (i=0; i<n; i+=4){
  a[i]   = a[i]   + 10;
  a[i+1] = a[i+1] + 10;
  a[i+2] = a[i+2] + 10;
  a[i+3] = a[i+3] + 10;
}
```

Other Loop Optimizations

- Loop fusion
  - Combine two back-to-back loops with exactly the same iteration pattern (e.g., **for** (`i=0; i<n; i++`))
  - Reduces loop overhead
- Loop fission
  - Split a big loop into smaller loops
  - Can improve L1 data and instruction cache miss rate
- Compilers can do these for simple loop bodies

Reminder: Division of Work – Load Balancing

Schedule clause determines how loop iterations are divided among thread team:

`schedule(<type>[,<chunk> ])`

- **static([chunk])** divides iterations statically between threads
  - Each thread receives [chunk] iterations, rounding as necessary to account for all iterations
  - Default **[chunk]** is **ceil(# iterations / # threads)**

- **dynamic([chunk])** allocates **[chunk]** iterations per thread, allocating an additional **[chunk]** iterations when a thread finishes
  - Forms a logical work queue, consisting of all loop iterations
  - Default **[chunk]** is 1

- **guided([chunk])** allocates dynamically, but **[chunk]** is exponentially reduced with each allocation

Static vs. Dynamic vs. Guided Scheduling

- Static:
  - Low overhead
  - May exhibit high workload imbalance
- Dynamic:
  - High overhead
  - Can reduce workload imbalance
- Guided:
  - Less overhead than dynamic
  - Comparable to dynamic in reducing imbalance

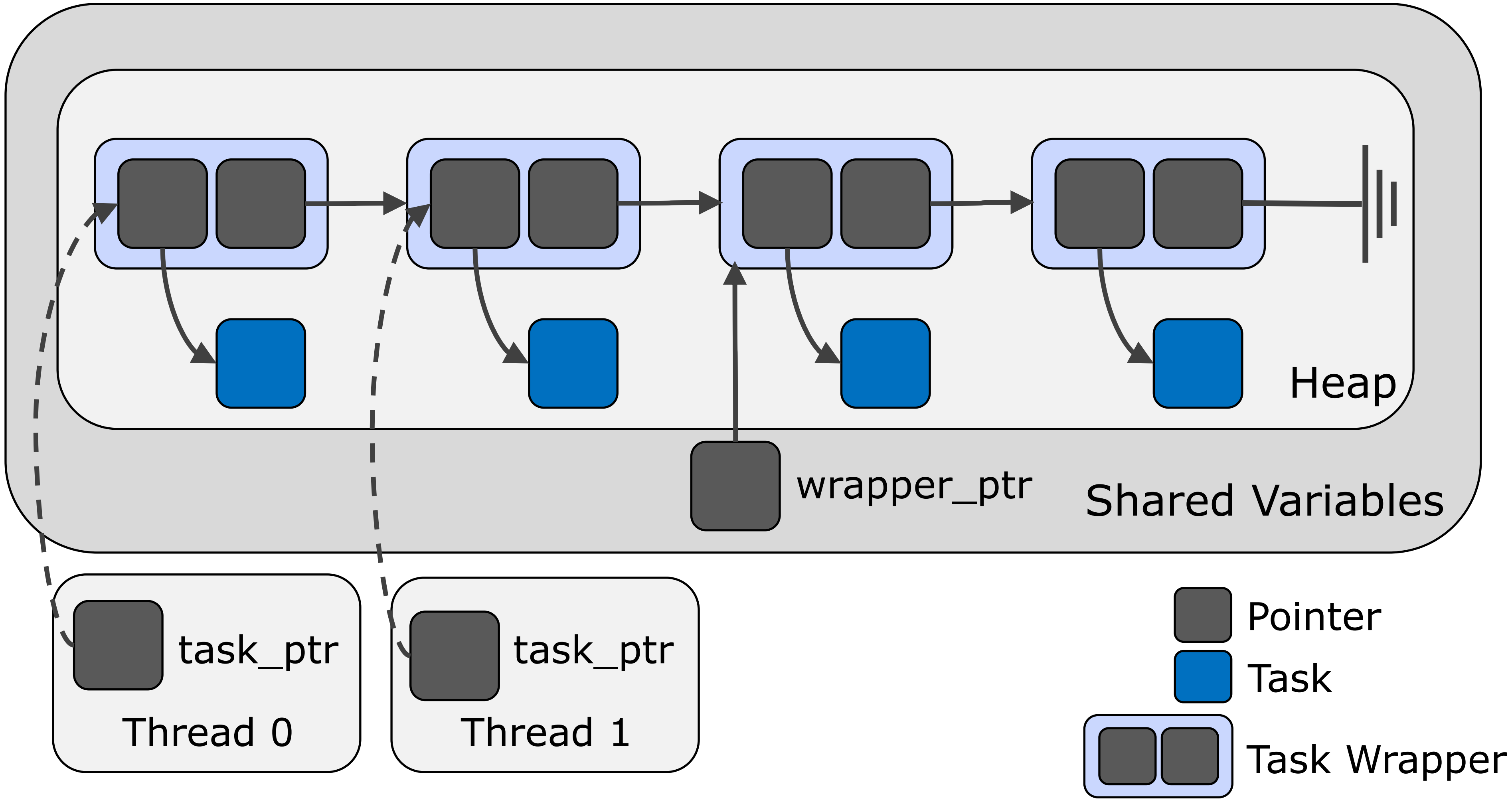
More General Work Scheduling

- Dynamic work scheduling using a task queue
  - Processing items from a list of tasks to do (task queue)
- As long as there is a task in the queue:
  - Each thread picks up a task from the queue
  - Processes it (might lead to adding new tasks to the queue)
- The program stops when there is no more task to process

More General Work Scheduling

- Model the queue as a linked-list of elements
- Each element is a task-wrapper (`wrapper_struct`)
- Task-wrapper includes:
  - A pointer to the task (which is `task_struct`)
  - A pointer to the next element

More General Work Scheduling



Task Queue: Sequential Code (1/2)

```
int main (int argc, char *argv[]){
  wrapper_struct *wrapper_ptr;
  task_struct *task_ptr;
  ...
  task_ptr = get_next_task(&wrapper_ptr);
  while (task_ptr != NULL) {
    complete_task(task_ptr);
    task_ptr = get_next_task(&wrapper_ptr);
  }
...
}
```

Task Queue: Sequential Code (2/2)

```
task_struct *get_next_task(wrapper_struct **wrapper_ptr){
  task_struct *next_task;

  if (*wrapper_ptr == NULL) next_task = NULL;
  else {
    next_task = (*wrapper_ptr)->task;
    *wrapper_ptr = (*wrapper_ptr)->next;
  }
  return next_task;
}
```

Task Queue: Parallelization Strategy

- Every thread should repeat (until no more tasks):
  - Taking next task from the queue
  - Completing the task
- Ensure no two threads take the same task
  - Must declare a critical section

Task Queue: Parallel Code (1/2)

```
int main (int argc, char *argv[]){
  wrapper_struct *wrapper_ptr;
  task_struct *task_ptr;
  ...
  #pragma omp parallel private(task_ptr)
  {
    task_ptr = get_next_task(&wrapper_ptr);
    while (task_ptr != NULL) {
      complete_task(task_ptr);
      task_ptr = get_next_task(&wrapper_ptr);
    }
  }
...
}
```

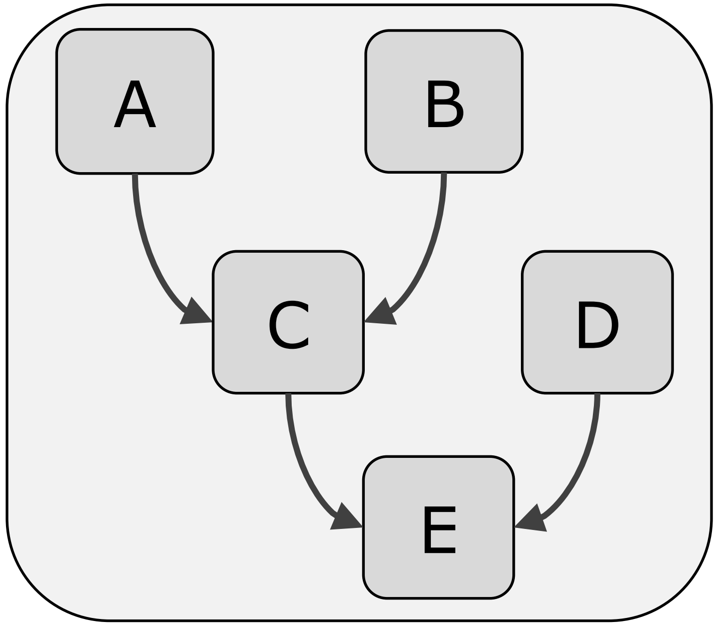
Task Queue: Parallel Code (2/2)

```
task_struct *get_next_task(wrapper_struct **wrapper_ptr){
  task_struct *next_task;
  #pragma omp critical
  {
    if (*wrapper_ptr == NULL) next_task = NULL;
    else {
      next_task = (*job_ptr)->task;
      *wrapper_ptr = (*wrapper_ptr)->next;
    }
  }
  return next_task;
}
```

Functional (or Task) Parallelism

- Performing distinct computations (tasks) at the same time

```
a = A();
b = B();
c = C(a, b);
d = D();
e = E(c, d);
printf (”Result=%d\n", e);
```



- Independent tasks can be executed in parallel
  - E.g., A, B, and D

Functional (or Task) Parallelism

- `parallel sections` Pragma
  - Precedes a block of *k* blocks of code
  - Each block is preceded by a `section` pragma
  - Blocks may be executed concurrently by *k* threads

```
#pragma omp parallel sections
{
   #pragma omp section /*Optional*/
   a = A();
   #pragma omp section
   b = B();
   #pragma omp section
   d = D();
}
c = C(a, b);
e = E(c, d);
printf (”Result=%d\n", e);
```

Summary

- Writing **fast** parallel programs is not easy
- Access pattern & locality makes a big difference
  - Remove false sharing
  - Use blocking to increase locality
- Load balancing is important
  - Dynamic work distribution works better
- Maximize parallel regions in your code
- Unroll loops to reduce the loop overhead
- Exploit functional parallelism when there are several independent tasks