Memory Technology

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

Main memory characteristics

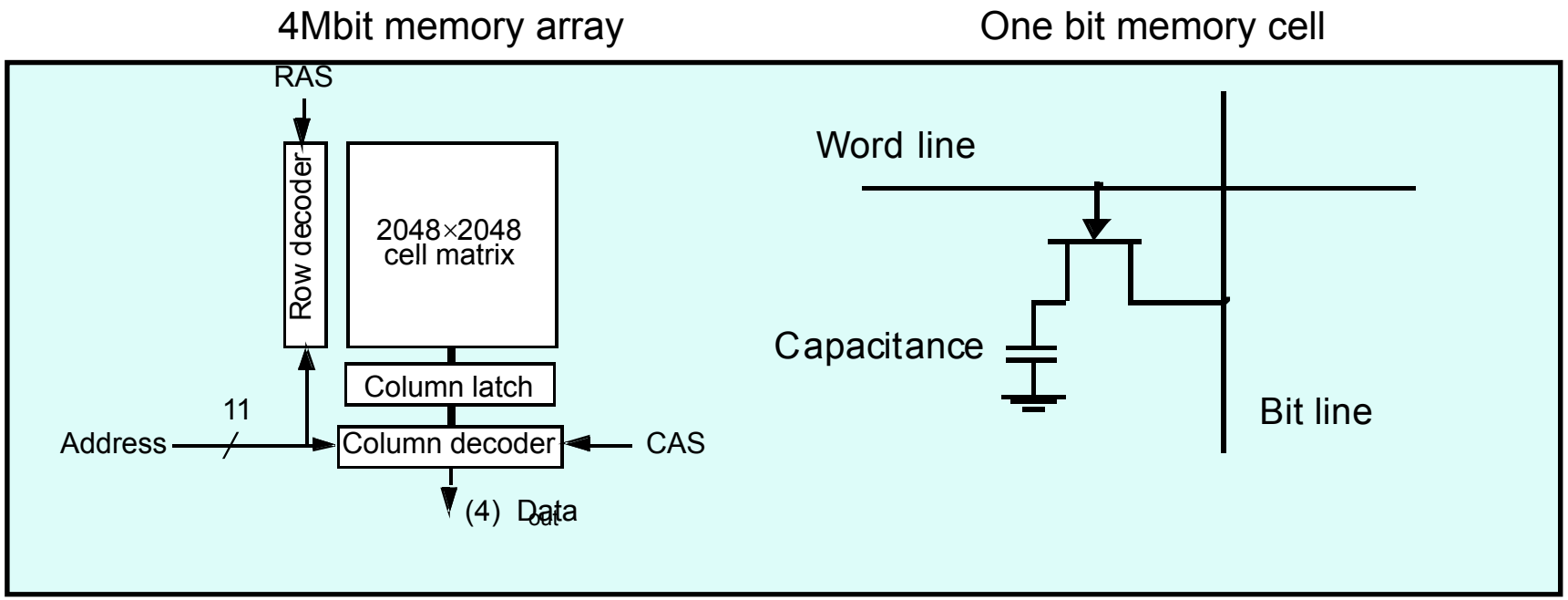
UPPSALA UNIVERSITET

Performance of main memory (from 3rd Ed… faster today)

- *Access time*: time between address is latched and data is available (~50ns)
- *Cycle time*: time between requests (~100 ns)
- *Total access time:* from ld to REG valid (~150ns)

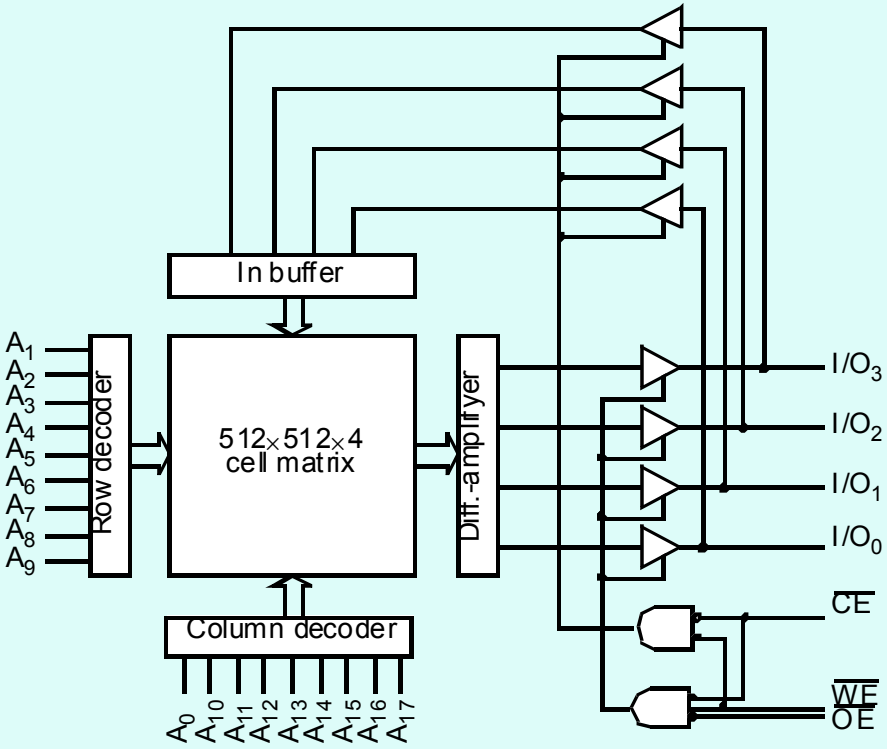
- Main memory is built from ***DRAM***: Dynamic RAM
- 1 transistor/bit ==> more error prune and slow
- Refresh and precharge
- Cache memory is built from ***SRAM***: Static RAM
  - about 4-6 transistors/bit

DRAM organization



- The address is multiplexed Row/Address Strobe (RAS/CAS)
- "Thin" organizations (between x16 and x1) to decrease pin load
- Refresh of memory cells decreases bandwidth
- Bit-error rate creates a need for error-correction (ECC)

SRAM organization



- Address is typically not multiplexed
- Each cell consists of about 4-6 transistors
- Wider organization (x18 or x36), typically few chips
- Often parity protected (ECC becoming more common)

Error Detection and Correction

Error-correction and detection

- E.g., 64 bit data protected by 8 bits of ECC
  - Protects DRAM and high-availability SRAM applications
  - Double bit error detection (”crash and burn”)
  - Chip kill detection (all bits of one chip stuck at all-1 or all-0)
  - Single bit correction
  - Need “memory scrubbing” in order to get good coverage

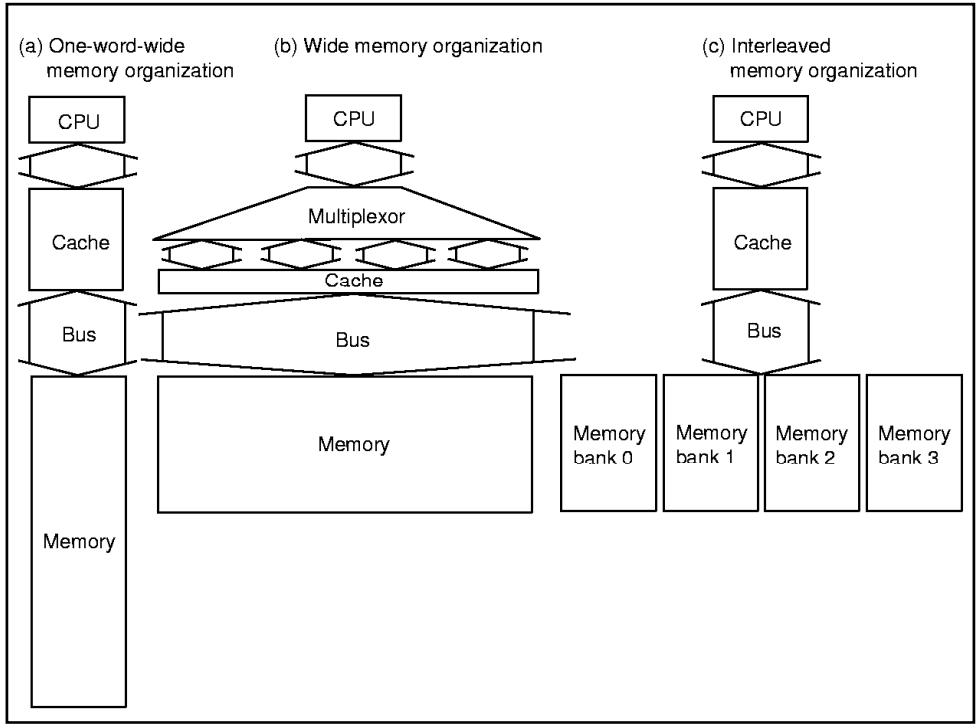
Parity

- E.g., 8 bit data protected by 1 bit parity
  - Protects SRAM and data paths
  - Single-bit ”crash and burn” detection
  - Not sufficient for large SRAMs today!!

Correcting the Error

- Correction on the fly by hardware
  - no performance-glitch
  - great for cycle-level redundancy
  - fixes the problem for now…
- Trap to software
  - correct the data value and write back to memory
- Memory scrubber
  - kernel process that periodically touches all of memory

Improving main memory performance

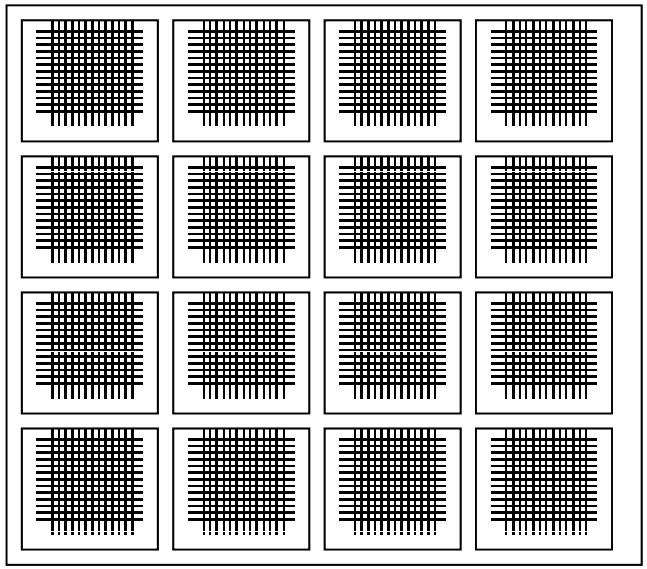


- Page-mode => faster access within a small distance
- Improves bandwidth per pin -- not time to critical word
- Single wide bank improves access time to the complete CL
- Multiple banks improves bandwidth

Newer kind of DRAM…

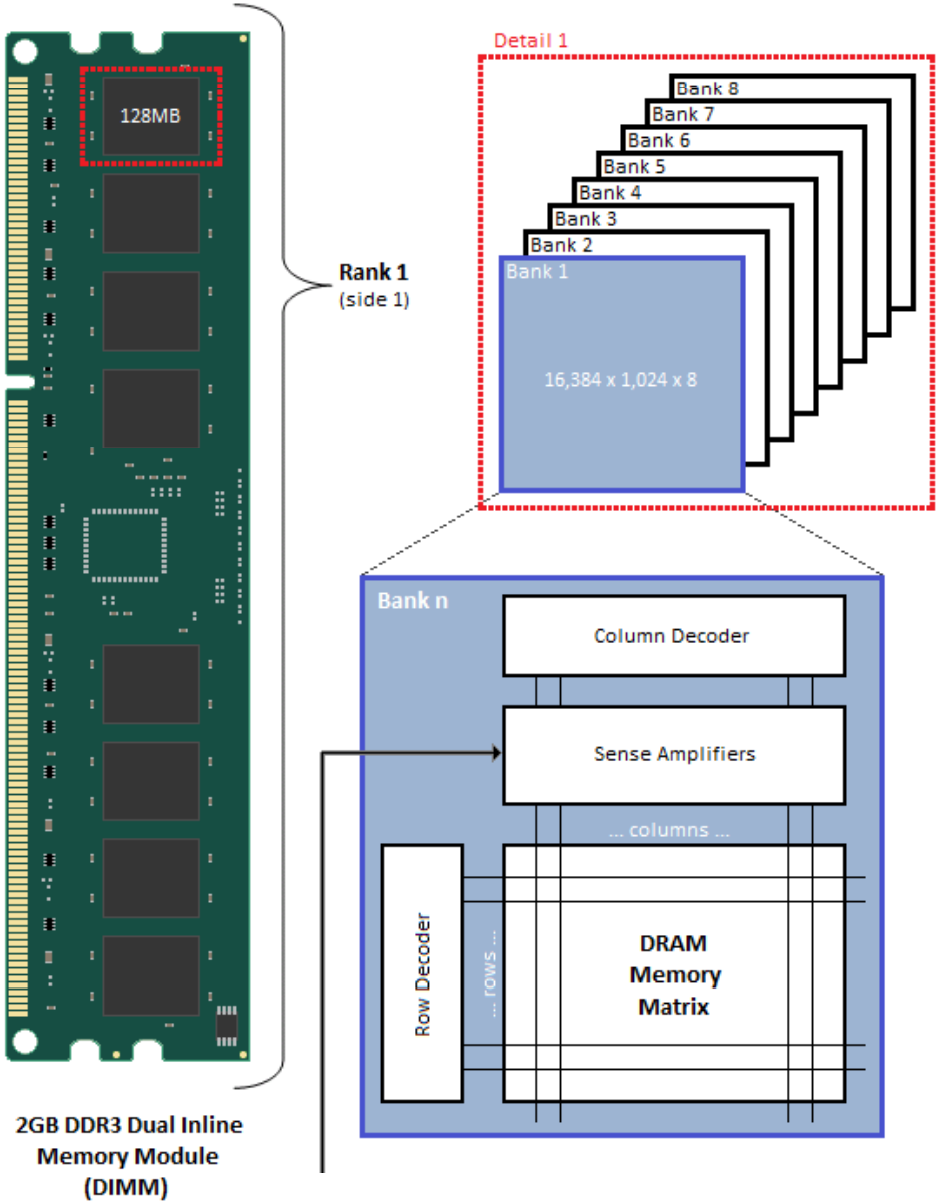
- SDRAM (5-1-1-1 @100 MHz)
  - Mem controller provides strobe for next seq. access
- DDR-DRAM (5-½-½-½)
  - Transfer data on both edges
- RAMBUS
  - Fast unidirectional circular bus
  - Split transaction addr/data
  - Each DRAM devices implements RAS/CAS/refresh… internally
- CPU and DRAM on the same chip?? (IMEM)…

Newer DRAMs …
(Several DRAM arrays on a die)



Name	Clock rate (MHz)	BW (GB/s per DIMM)
DDR-260	133	2,1
DDR-300	150	2,4
DDR2-533	266	4,3
DDR2-800	400	6,4
DDR3-1066	533	8,5
DDR3-1600	800	12,8

Modern DRAM (1)



From AnandTech:
Everything You Always Wanted to Know About SDRAM: But Were Afraid to Ask
http://www.anandtech.com/show/3851/everything-you-always-wanted-to-know-about-sdram-memory-but-were-afraid-to-ask

Timing "page hit"

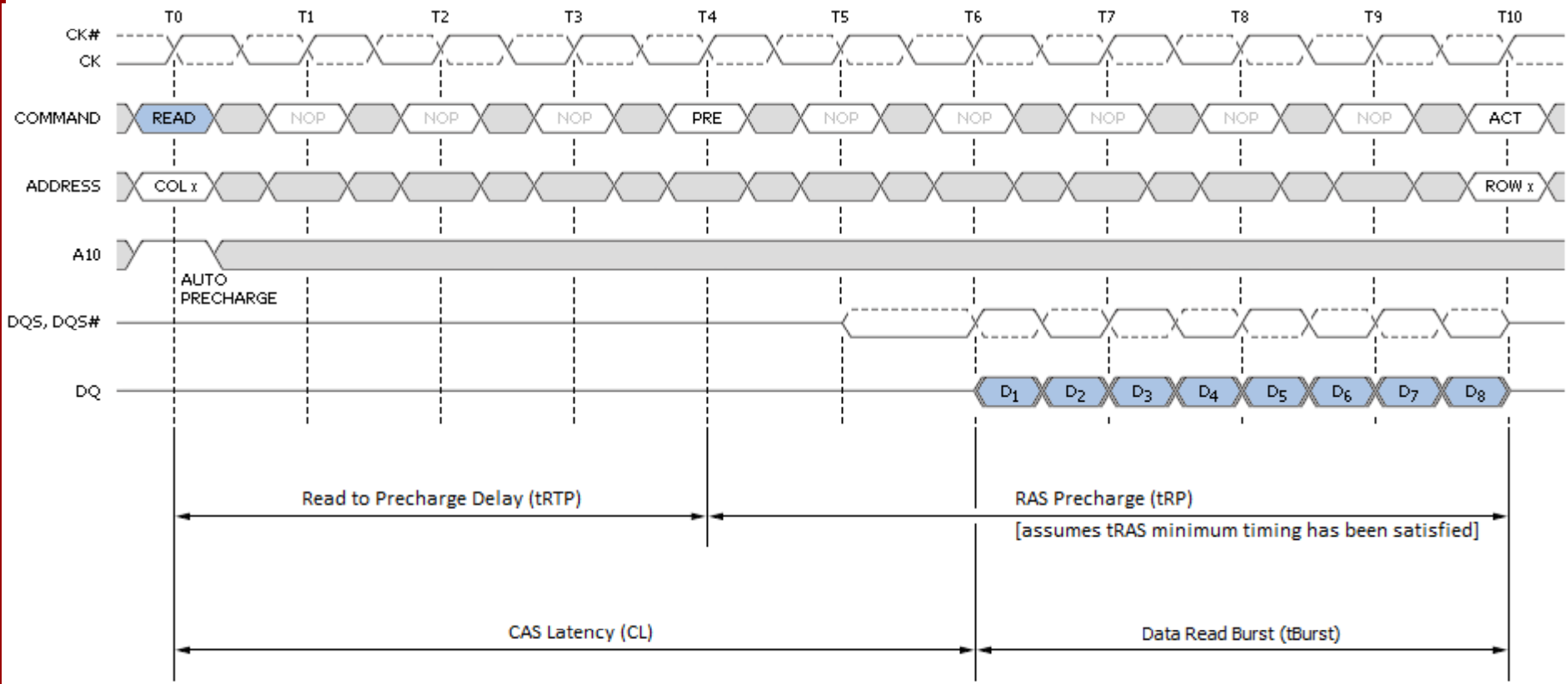


Figure 6. Page-hit timing (with precharge and subsequent bank access)

From AnandTech:
Everything You Always Wanted to Know About SDRAM: But Were Afraid to Ask
http://www.anandtech.com/show/3851/everything-you-always-wanted-to-know-about-sdram-memory-but-were-afraid-to-ask

Timing "page miss"

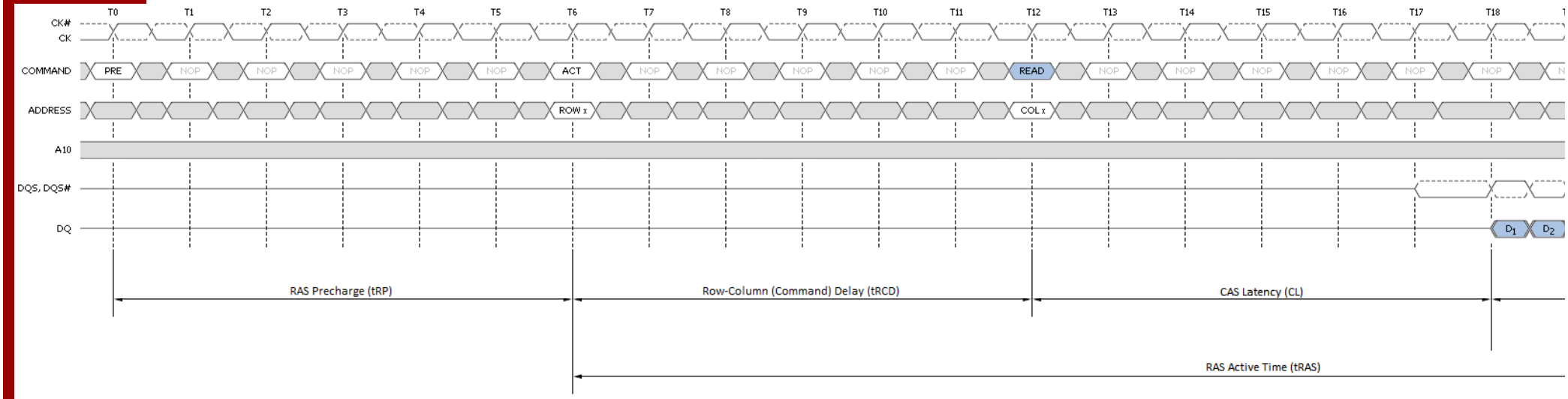
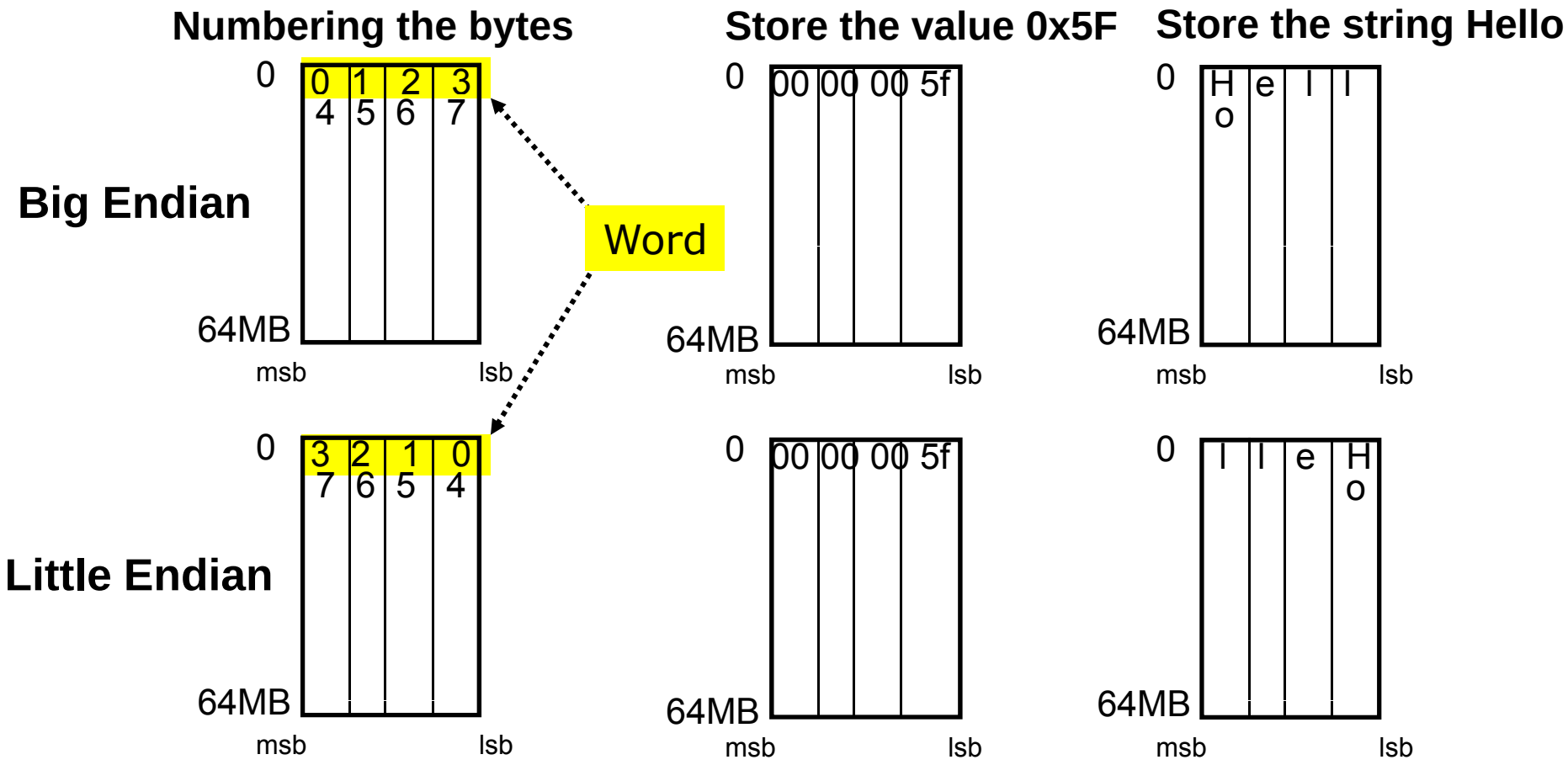


Figure 8. Page-miss timing

From AnandTech:
Everything You Always Wanted to Know About SDRAM: But Were Afraid to Ask
http://www.anandtech.com/show/3851/everything-you-always-wanted-to-know-about-sdram-memory-but-were-afraid-to-ask

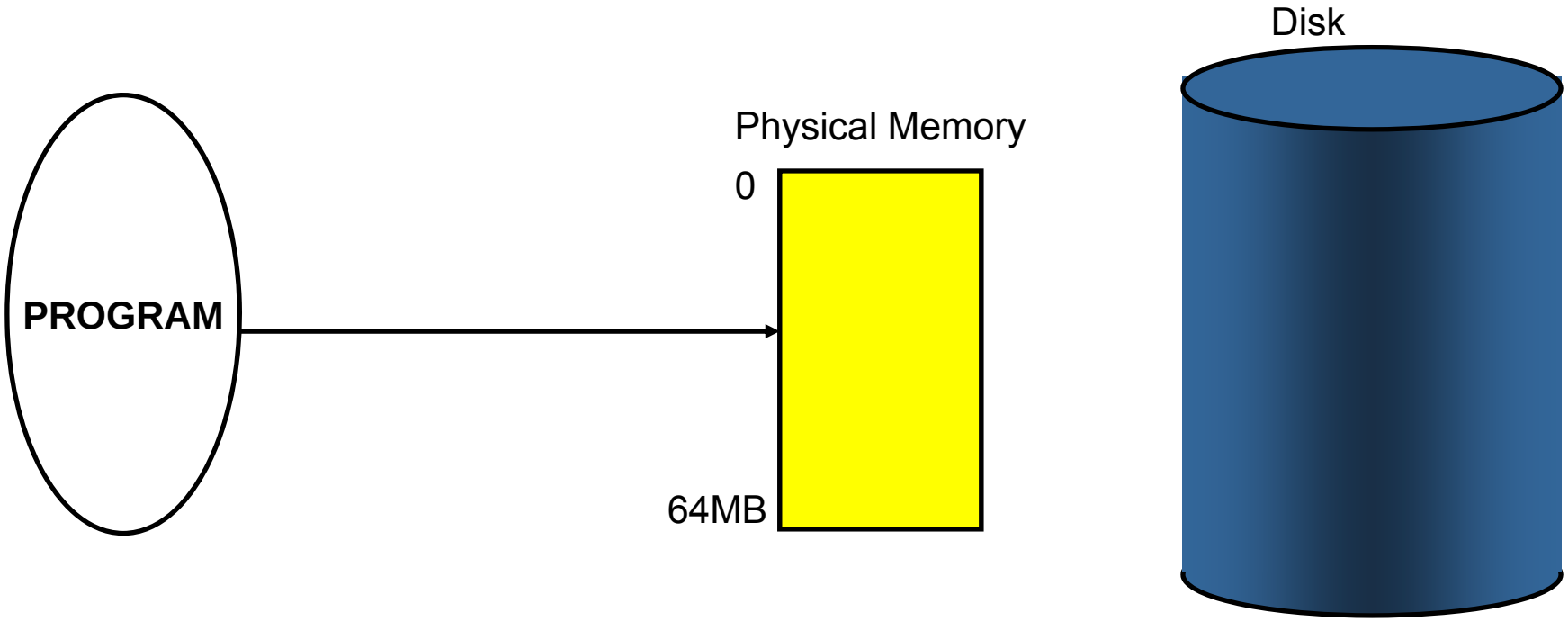
The Endian Mess



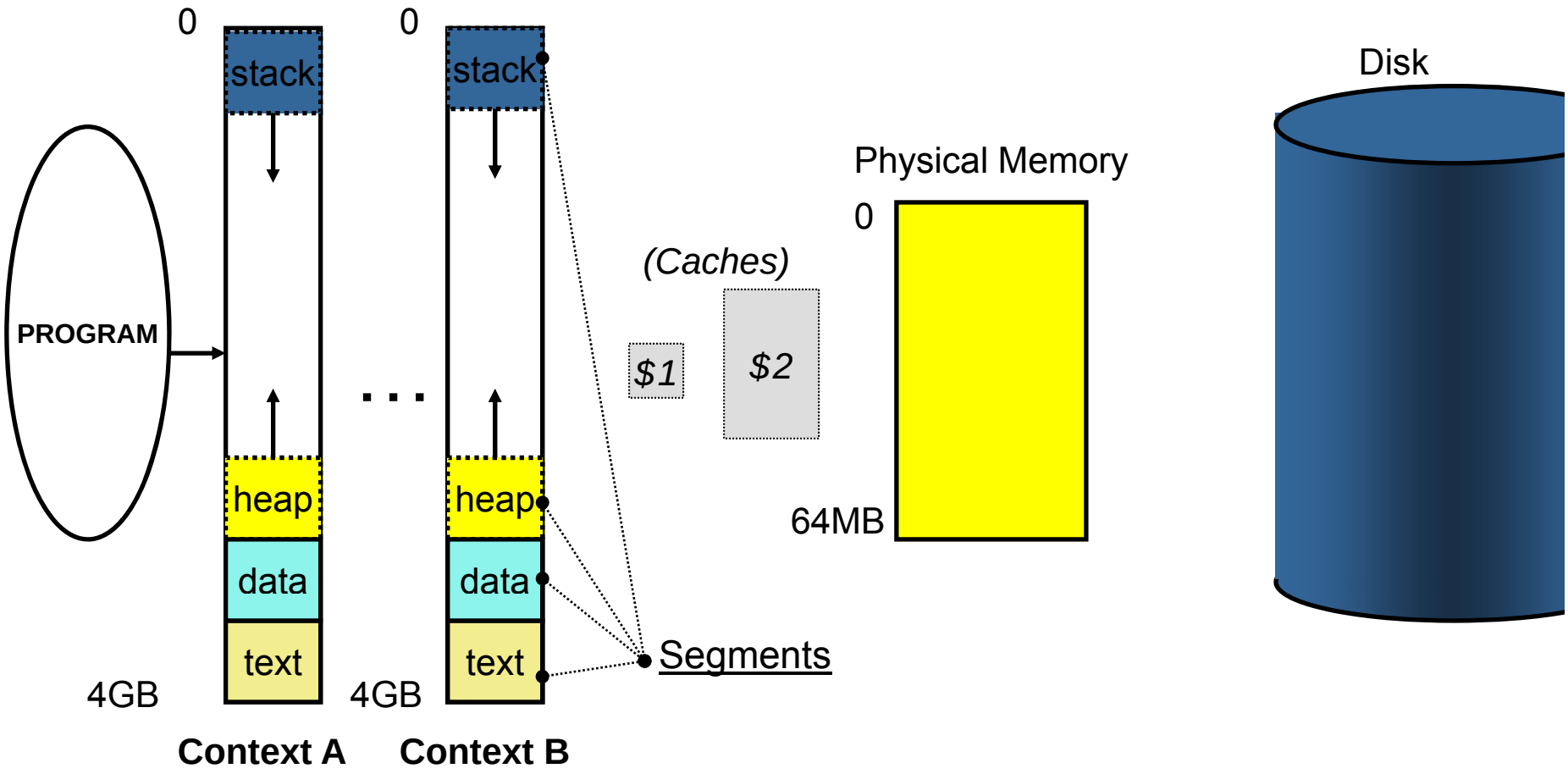
Virtual Memory System

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

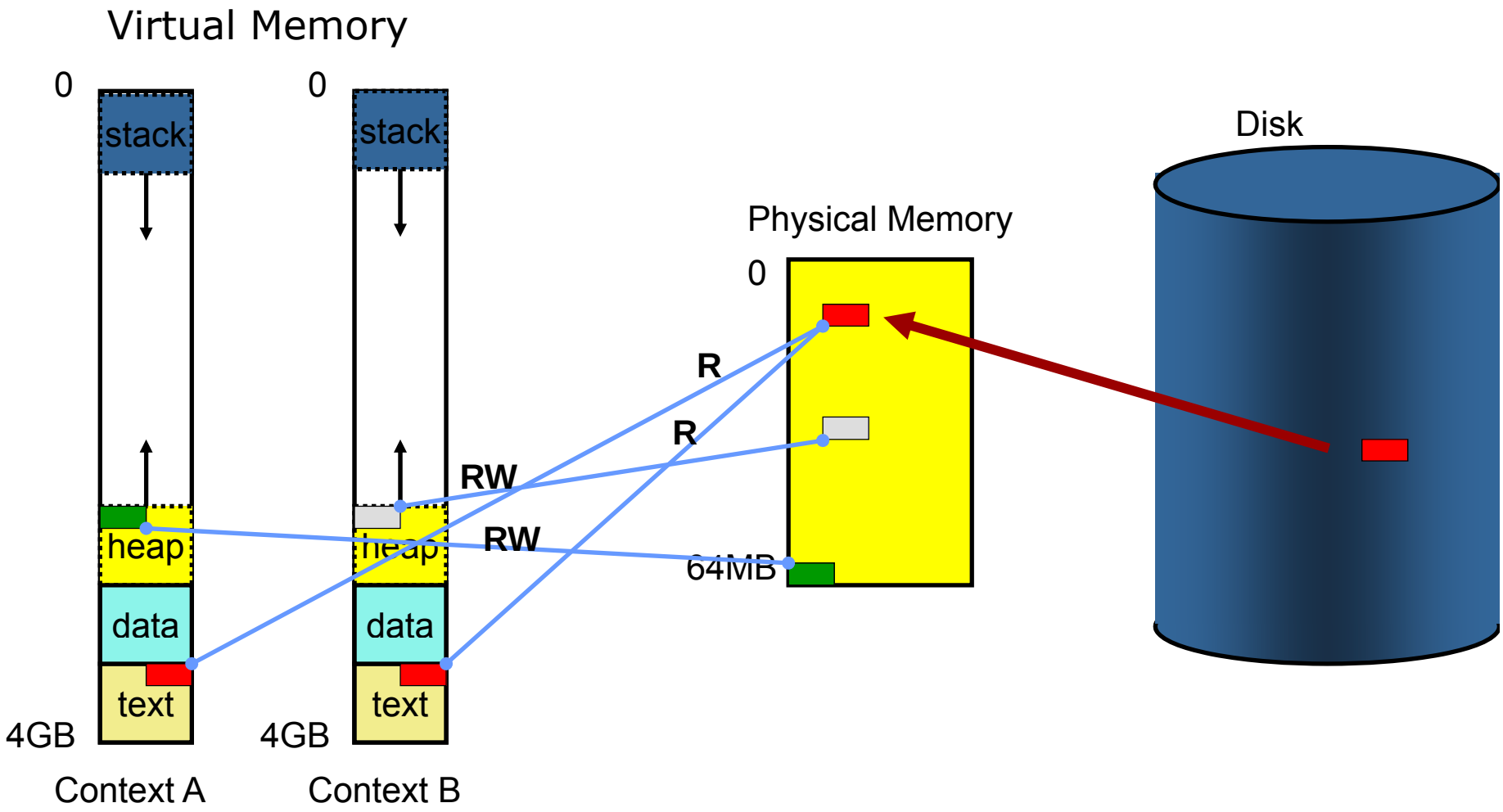
Physical Memory



Virtual and Physical Memory



Translation & Protection



Virtual memory — parameters

Compared to first-level cache parameters

- Replacement in cache handled by HW. Replacement in VM handled by SW
- VM hit latency very low (often zero cycles)
- VM miss latency huge (several kinds of misses)
- Allocation size is one ”page” 4kB and up)

Parameter	*First-level cache*	*Virtual memory*
Block (page) size	16-128 bytes	4K-64K bytes
Hit time	1-2 clock cycles	40-100 clock cycles
Miss penalty	8-100 clock cycles	700K-6000K clock cycles
(Access time)	(6-60 clock cycles)	(500K-4000K clock cycles)
(Transfer time)	(2-40 clock cycles)	(200K-2000K clock cycles)
Miss rate	0.5%-10%	0.00001%-0.001%
Data memory size	16 Kbyte - 1 Mbyte	16 Mbyte - 8 Gbyte

VM: Block placement

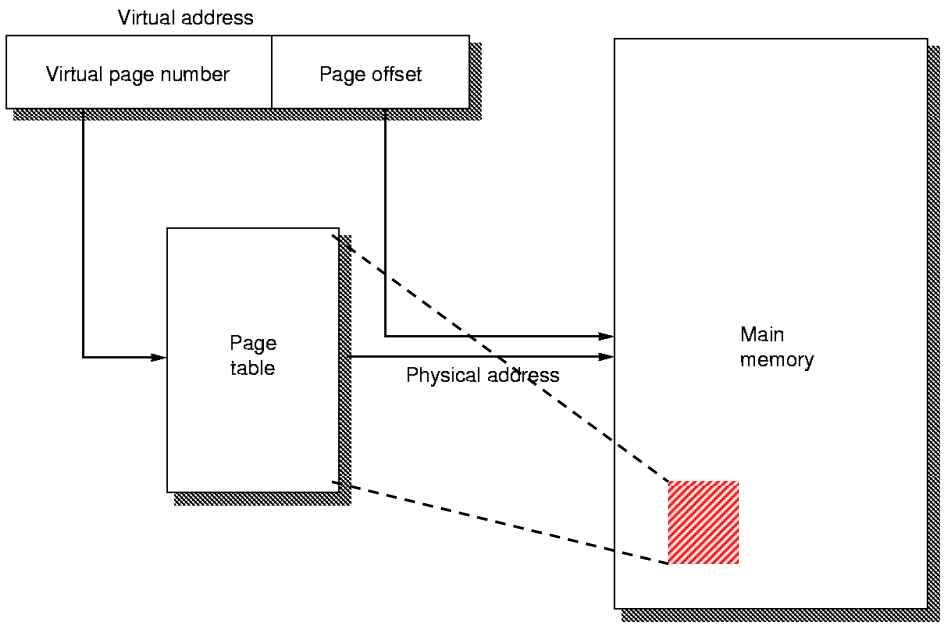
Where can a block (page) be placed in main memory?

What is the organization of the VM?

- The high miss penalty makes SW solutions to implement a ***fully associative address mapping*** feasible at page faults
- A page from disk may occupy any pageframe in PA
- Some restriction can be helpful (page coloring)

VM: Block identification

Use a page table stored in main



- Suppose 8 Kbyte pages, 48 bit virtual address
- Page table occupies

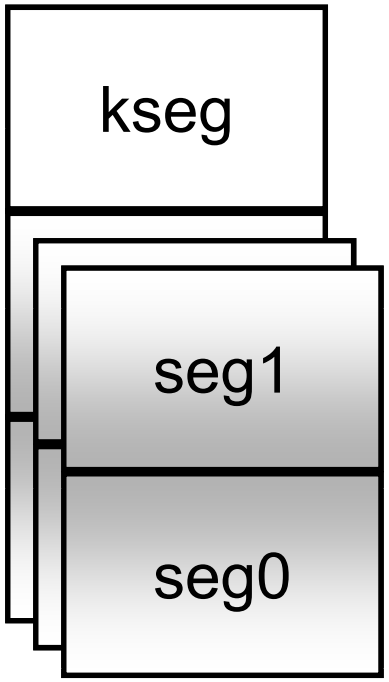
  $2^{48}/2^{13}$* 4B = 2^{37} = 128GB!!!

- Solutions:
  - ***Only one entry per physical page is needed***
  - Multi-level page table (dynamic)
  - Inverted page table (~hashing)

Address translation

- Multi-level table: The *Alpha 21064 (*

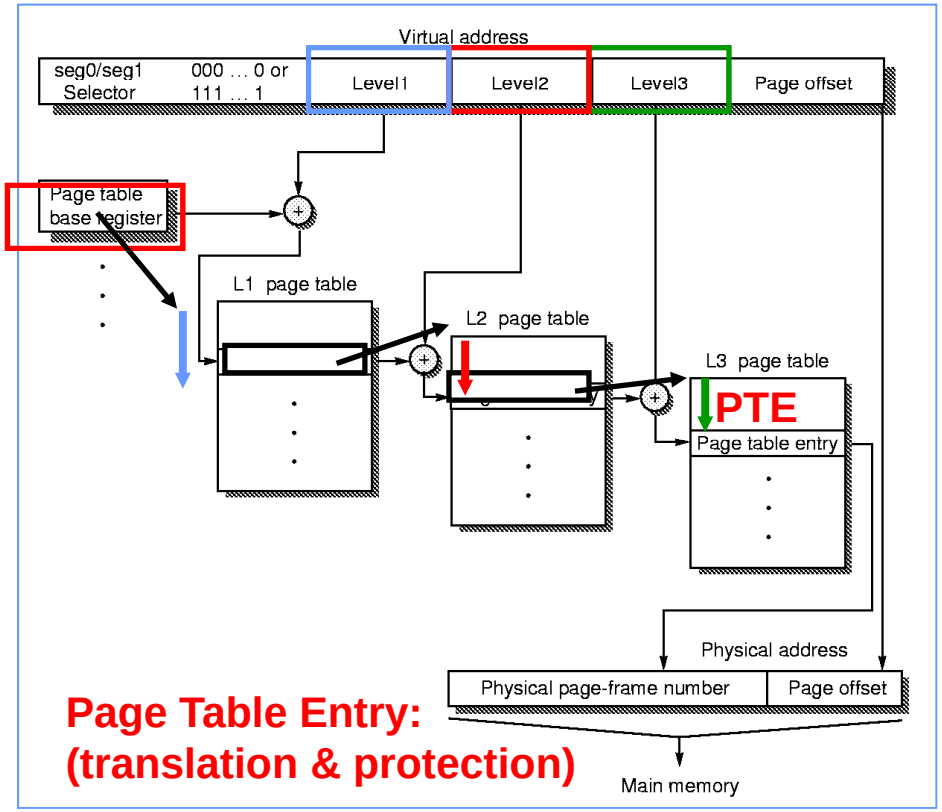
Segment is selected by bit 62 & 63 in addr.



Kernel segment
Used by OS. Does not use virtual memory.

User segment 1
Used for stack.

User segment 0
Used for instr. & static data & heap



Protection mechanisms

The address translation mechanism can be used to provide memory protection:

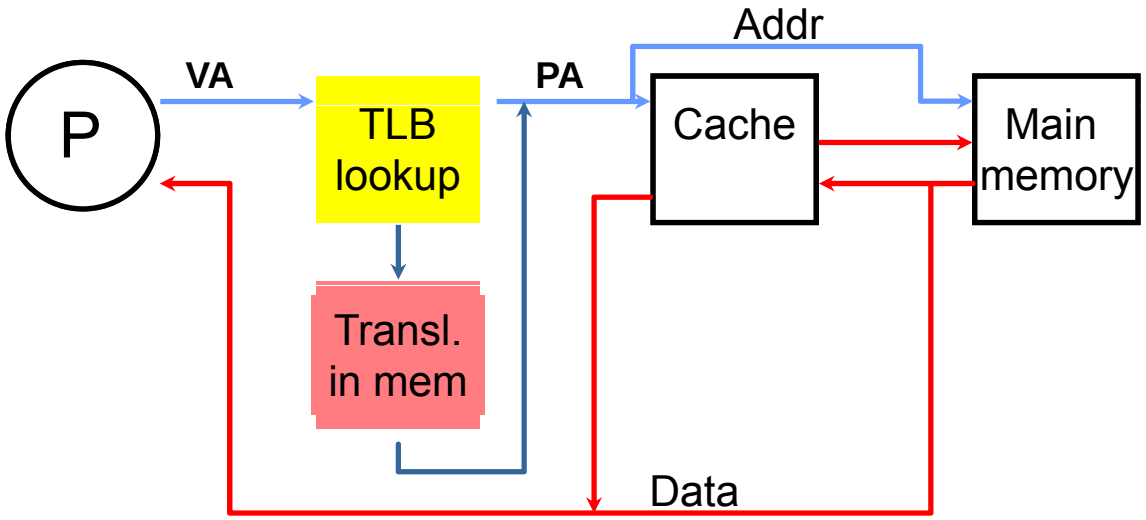
- Use ***protection attribute bits*** for each page
- Stored **in the page table entry** (PTE) (and TLB…)
- Each physical page gets its own **per process protection**
- **Violations** detected during the address translation **cause exceptions** (i.e., SW trap)
- ***Supervisor/user modes*** necessary to prevent user processes from changing e.g. PTEs

Fast address translation

How can we avoid three extra memory references for each original memory reference?

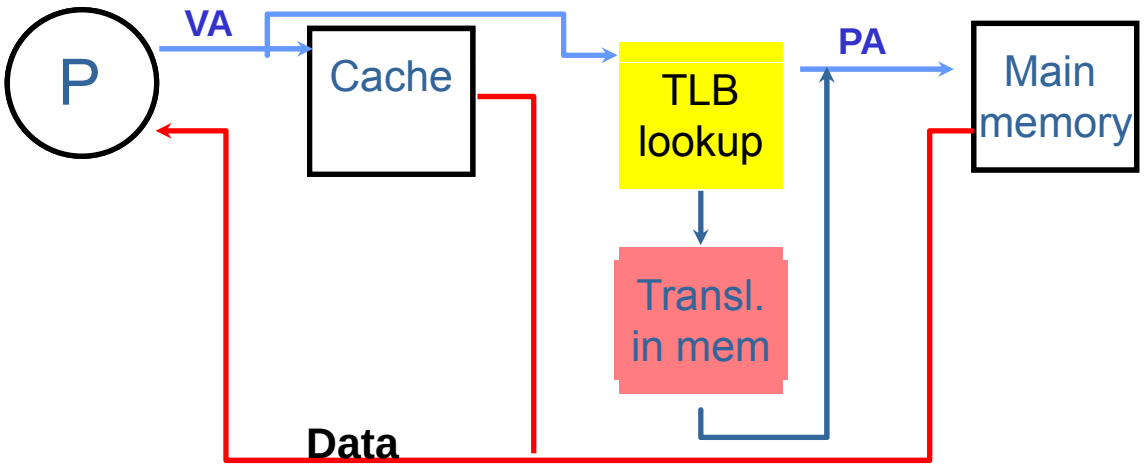
- Store the most commonly used address translations in a cache—***Translation Look-aside Buffer*** (TLB)

==> The caches rears their ugly faces again!



Do we need a fast TLB?

- Why do a TLB lookup for every L1 access?
- Why not cache virtual addresses instead?
  - Move the TLB on the other side of the cache
  - It is only needed for finding stuff in Memory anyhow
  - The TLB can be made larger and slower – or can it?

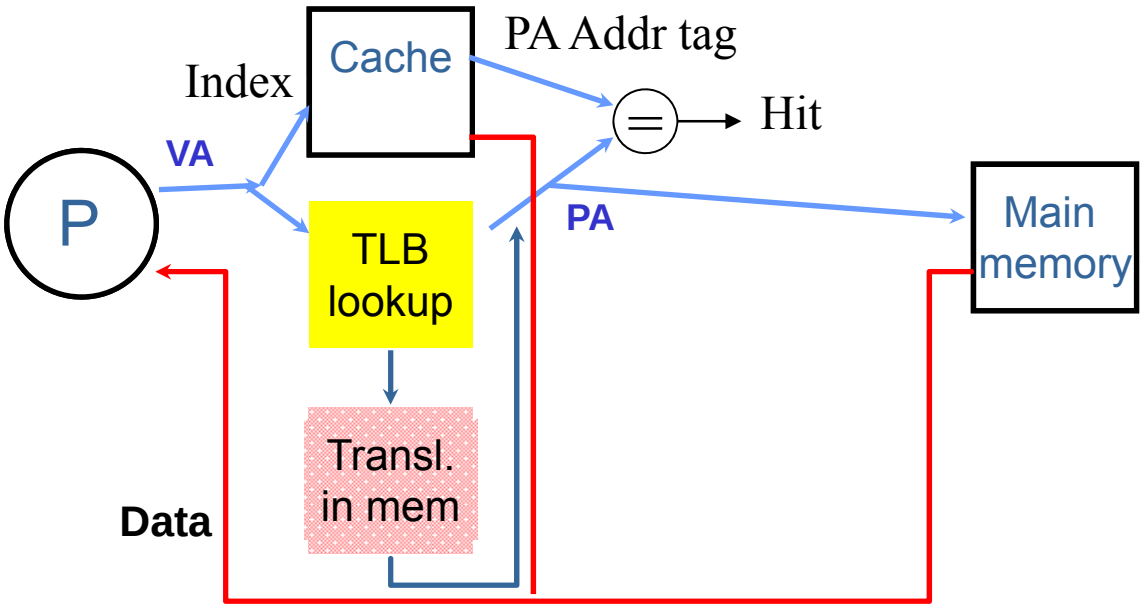


Aliasing Problem

The same physical page may be accessed using different virtual addresses

- A virtual cache will cause confusion -- a write by one process may not be observed
- Flushing the cache on each process switch is slow (and may only help partly)
- =>VIPT (VirtuallyIndexedPhysicallyTagged) is the answer
  - Direct-mapped cache no larger than a page
  - No more sets than there are cache lines on a page + logic
  - Page coloring can be used to guarantee correspondence between more PA and VA bits (e.g., Sun Microsystems)

Virtually Indexed Physically Tagged =VIPT

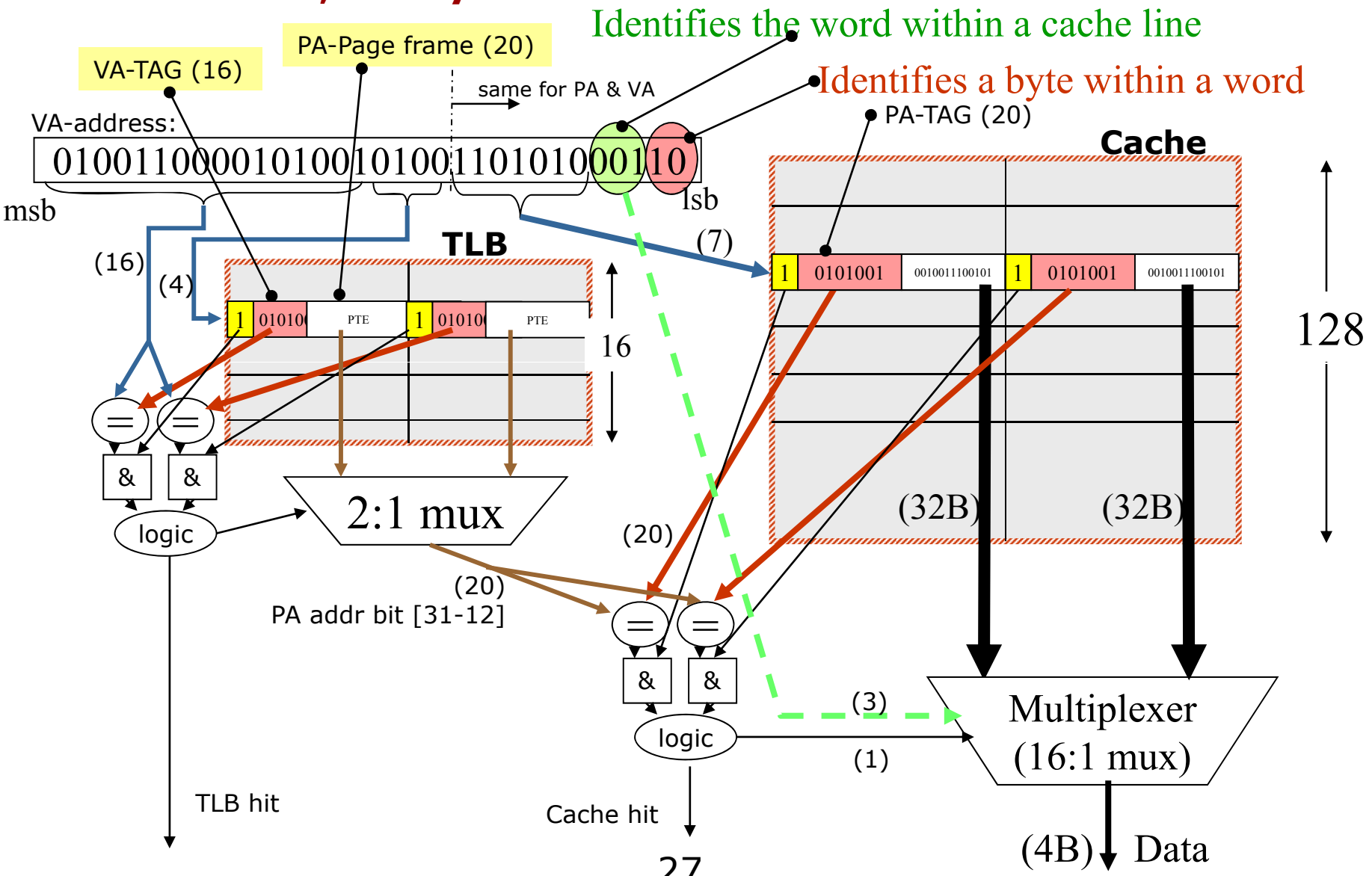


Have to guarantee that all aliases have the same index

- L1_cache_size < (page-size * associativity)
- Page coloring can help further

Putting it all together: VIPT

Cache: 8kB, 2-way, CL=32B, word=4B, page =4kB
TLB: 32 entries, 2-way



What is the capacity of the TLB

Typical TLB size = 0.5 - 2kB

Each translation entry 4 - 8B ==> 32 - 500 entries

Typical page size = 4kB - 16kB

TLB-reach = 0.1MB - 8MB

FIX:

- ***Multiple page sizes, e.g., 8kB and 8 MB***
- ***TSB -- A direct-mapped translation in memory as a "second-level TLB"***

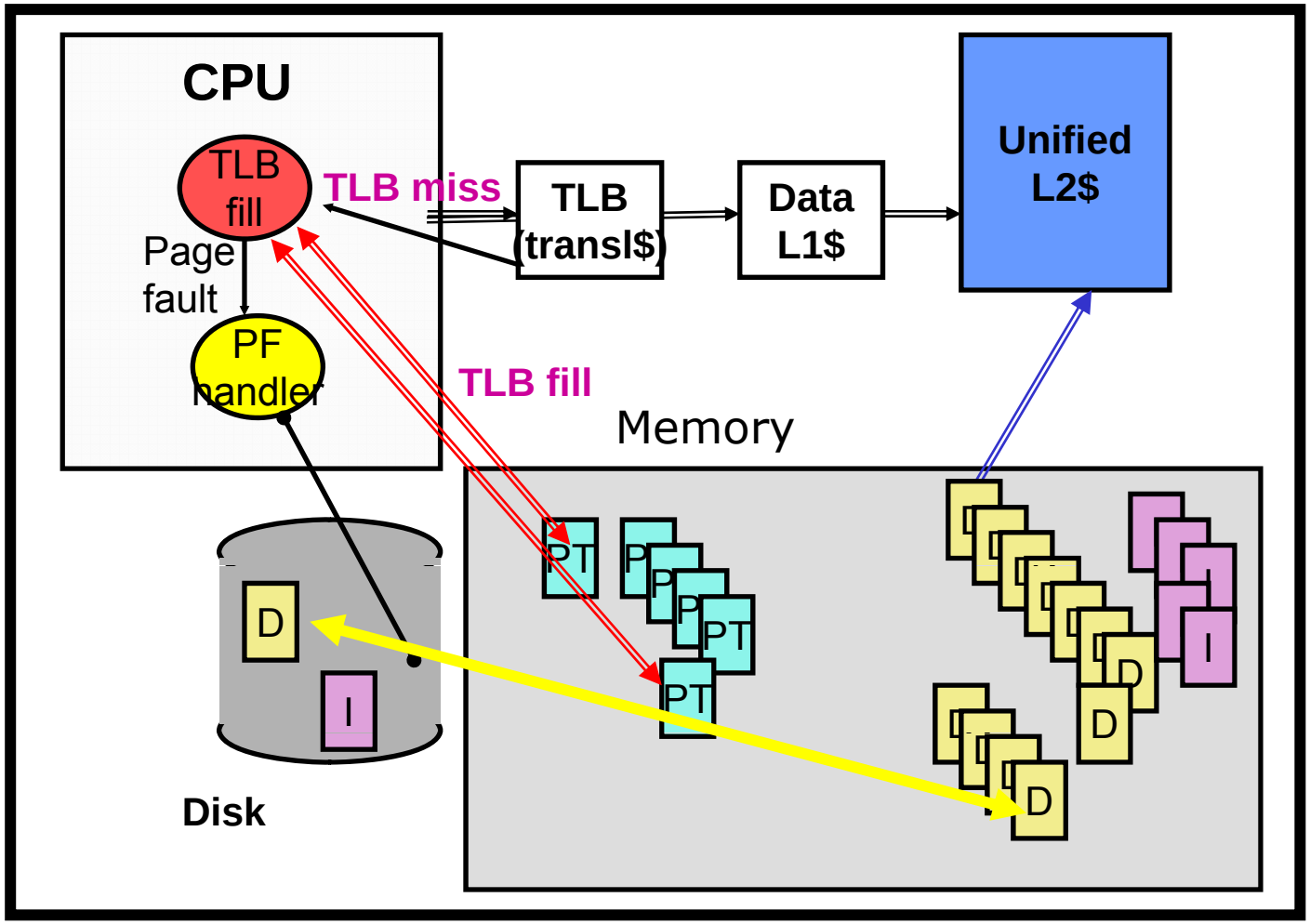
VM: Page replacement

Most important: ***minimize number of page faults***

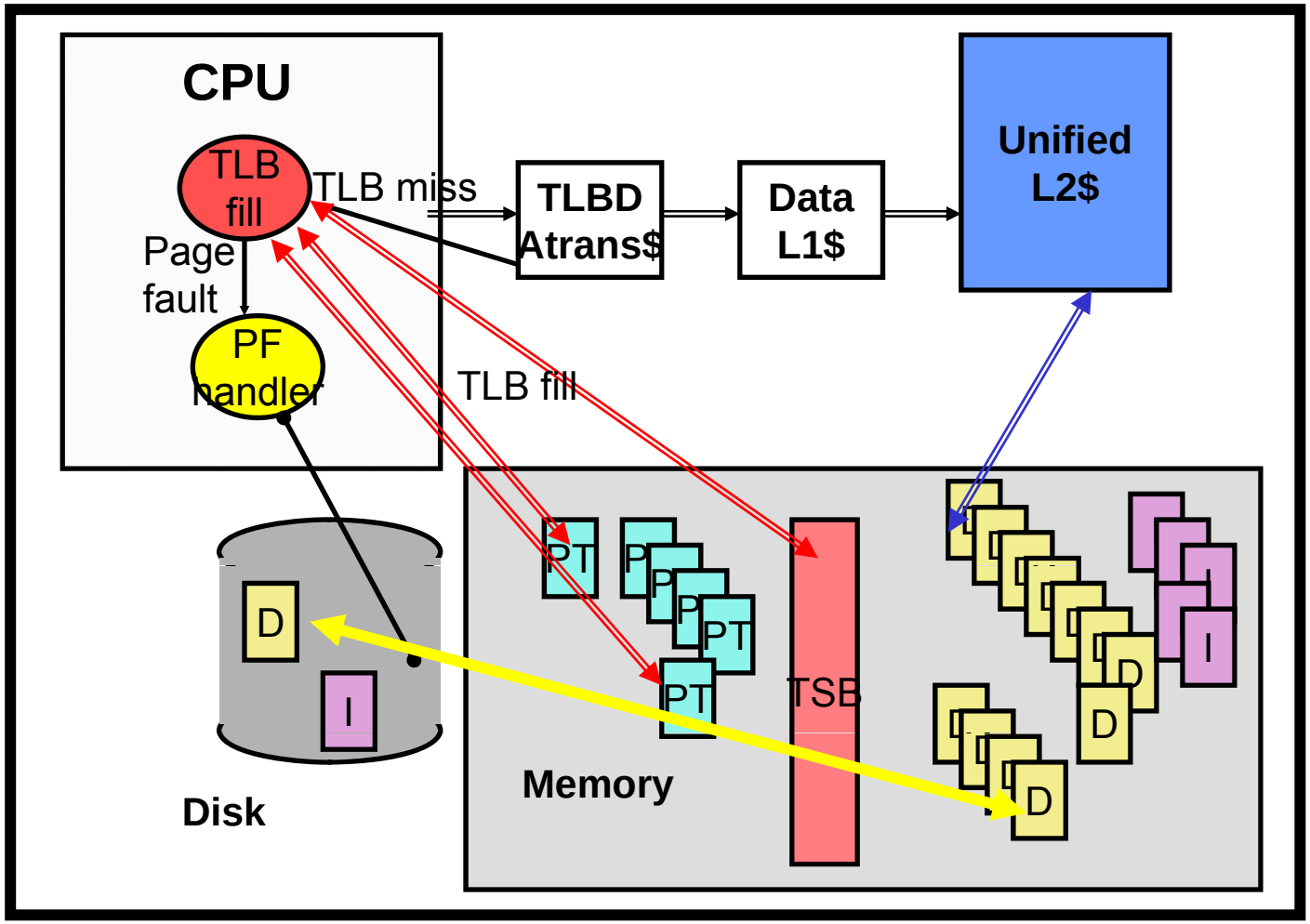
Page replacement strategies:

- FIFO—First-In-First-Out
- LRU—Least Recently Used
- Approximation to LRU
  - Each page has a ***reference bit*** that is set on a reference
  - The OS periodically resets the reference bits
  - When a page is replaced, a page with a reference bit that is not set is chosen

So far…


CPU
TLB fill
TLB miss
Page fault
PF handler
TLB (transl$)
Data L1$
Unified L2$
TLB fill
Memory
PT
P
PT
PT
D
I
Disk

Adding TSB (software TLB cache)



VM: Write strategy

Write back or Write through?

- ***Write back***!
- Write through is impossible to use:
  - Too long access time to disk
  - The write buffer would need to be *prohibitively* large
  - The I/O system would need an extremely high bandwidth

UPPSALA UNIVERSITET

VM dictionary

Virtual Memory System	The “cache” languge
Virtual address	~Cache address
Physical address	~Cache location
Page	~Huge cache block
Page fault	~Extremely painfull $miss
Page-fault handler	~The software filling the $
Page-out	Write-back if dirty

Caches Everywhere…

- D cache
- I cache
- L2 cache
- L3 cache
- ITLB
- DTLB
- TSB
- Virtual memory system
- Branch predictors
- Directory cache
- …

Exploring the Memory of a Computer System

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

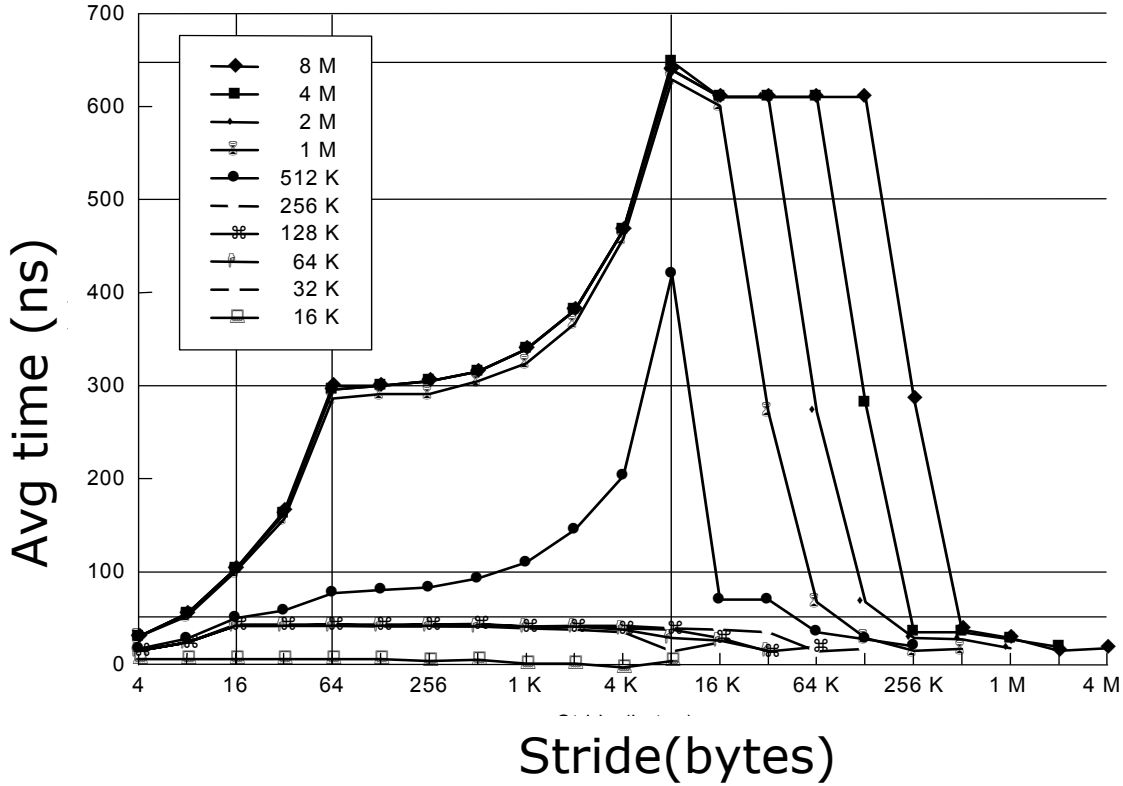
Micro Benchmark Signature

```
for (times = 0; times < Max; times++) /* many times*/

  for (i=0; i < ArraySize; i = i + Stride)
    dummy = A[i]; /* touch an item in the array */
```

Measuring the averge access time to memory, while varying ArraySize and Stride, will allow us to reverse-engineer the memory system.
(need to turn off HW prefetching…)

Micro Benchmark Signature

```
for (times = 0; times < Max; times++) /* many times*/

  for (i=0; i < ArraySize; i = i + Stride)
    dummy = A[i]; /* touch an item in the array */
```



Stepping through the array

```
for (times = 0; times < Max; times++) /* many times*/

  for (i=0; i < ArraySize; i = i + Stride)
    dummy = A[i]; /* touch an item in the array */
```

0 Array Size = 16, Stride=4

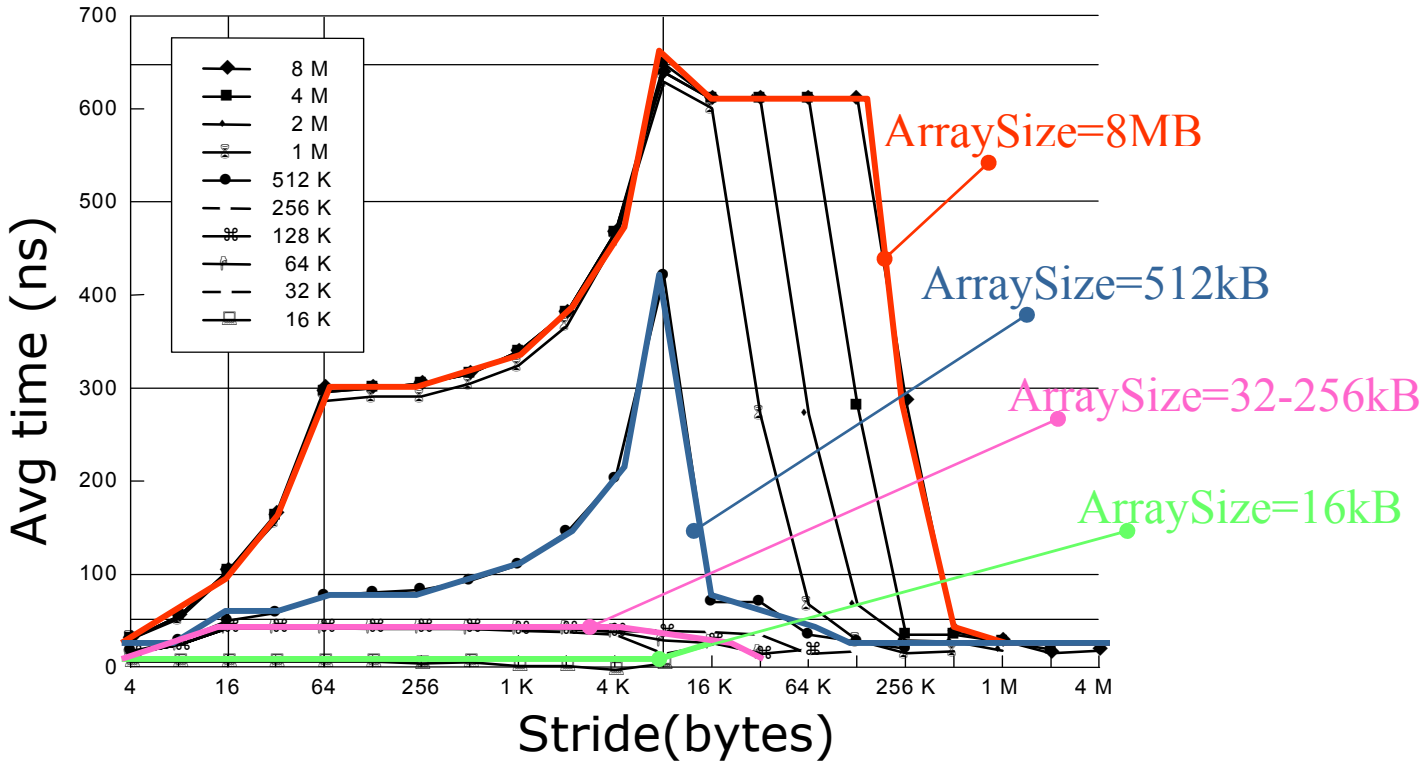
0 Array Size = 16, Stride=8…

0 Array Size = 32, Stride=4…

0 Array Size = 32, Stride=8…

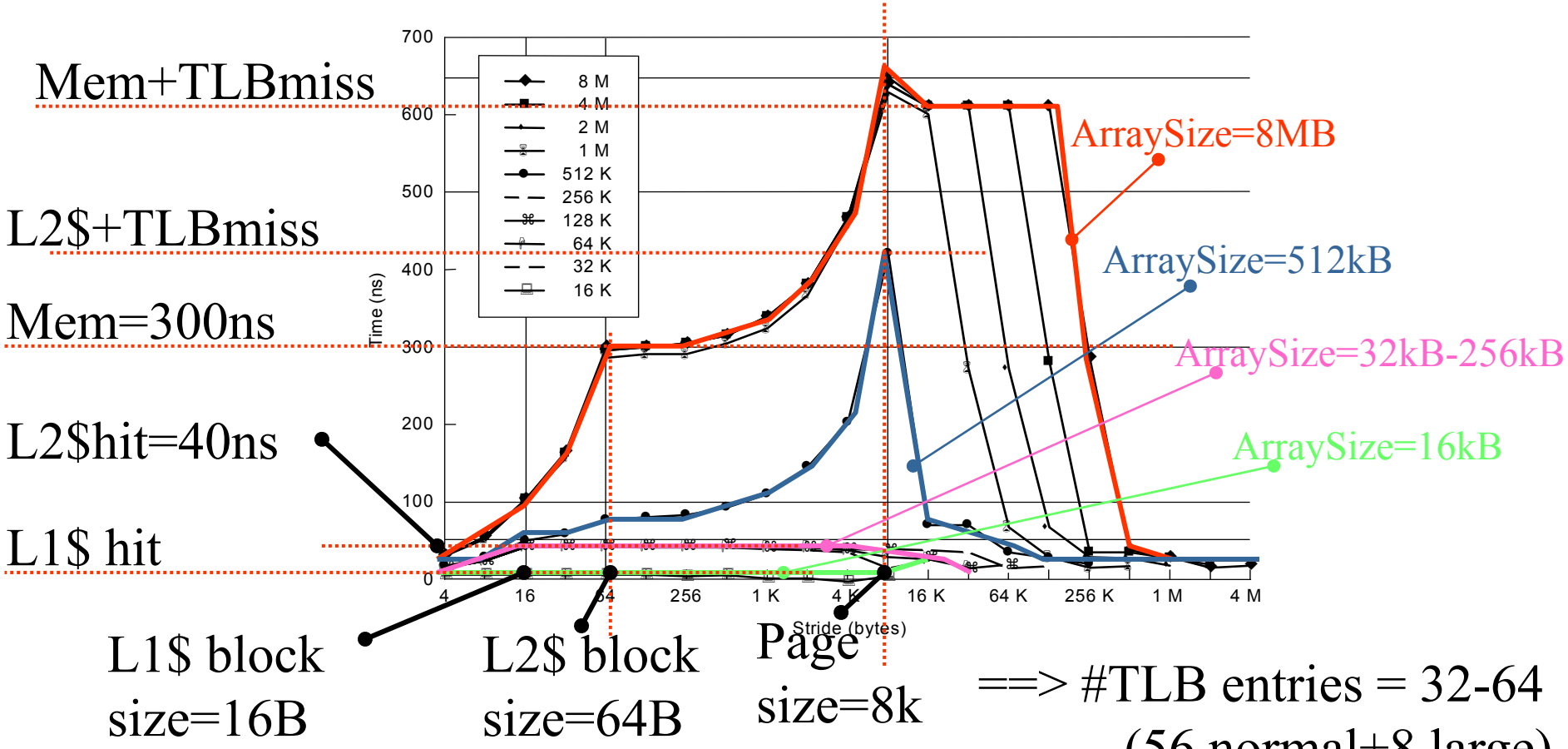
Micro Benchmark Signature

```
for (times = 0; times < Max; time++) /* many times*/

  for (i=0; i < ArraySize; i = i + Stride)
    dummy = A[i]; /* touch an item in the array */
```



Micro Benchmark Signature

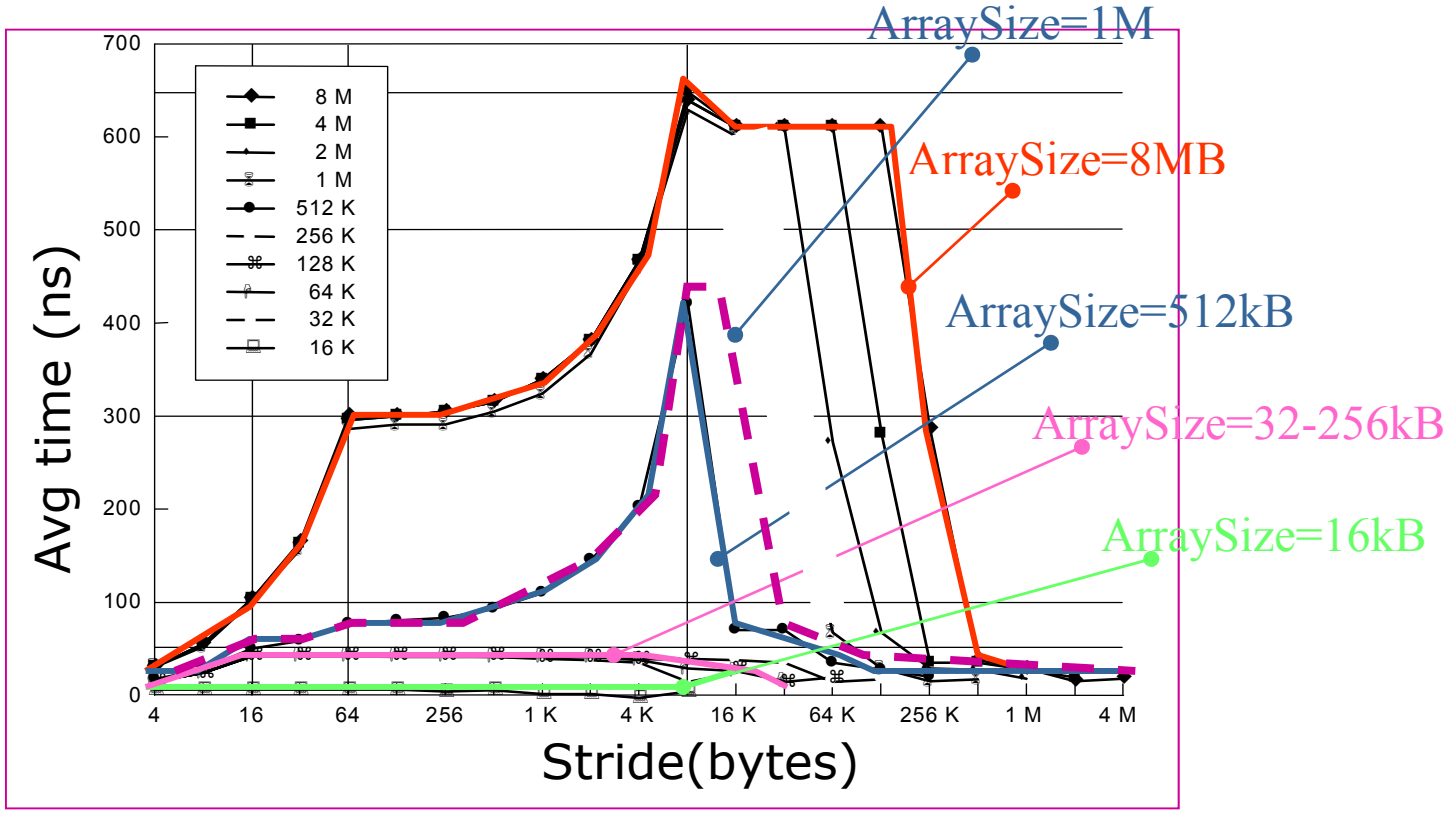
```
for (times = 0; times < Max; time++) /* many times*/

  for (i=0; i < ArraySize; i = i + Stride)
    dummy = A[i]; /* touch an item in the array */
```



==> #TLB entries = 32-64 (56 normal+8 large)

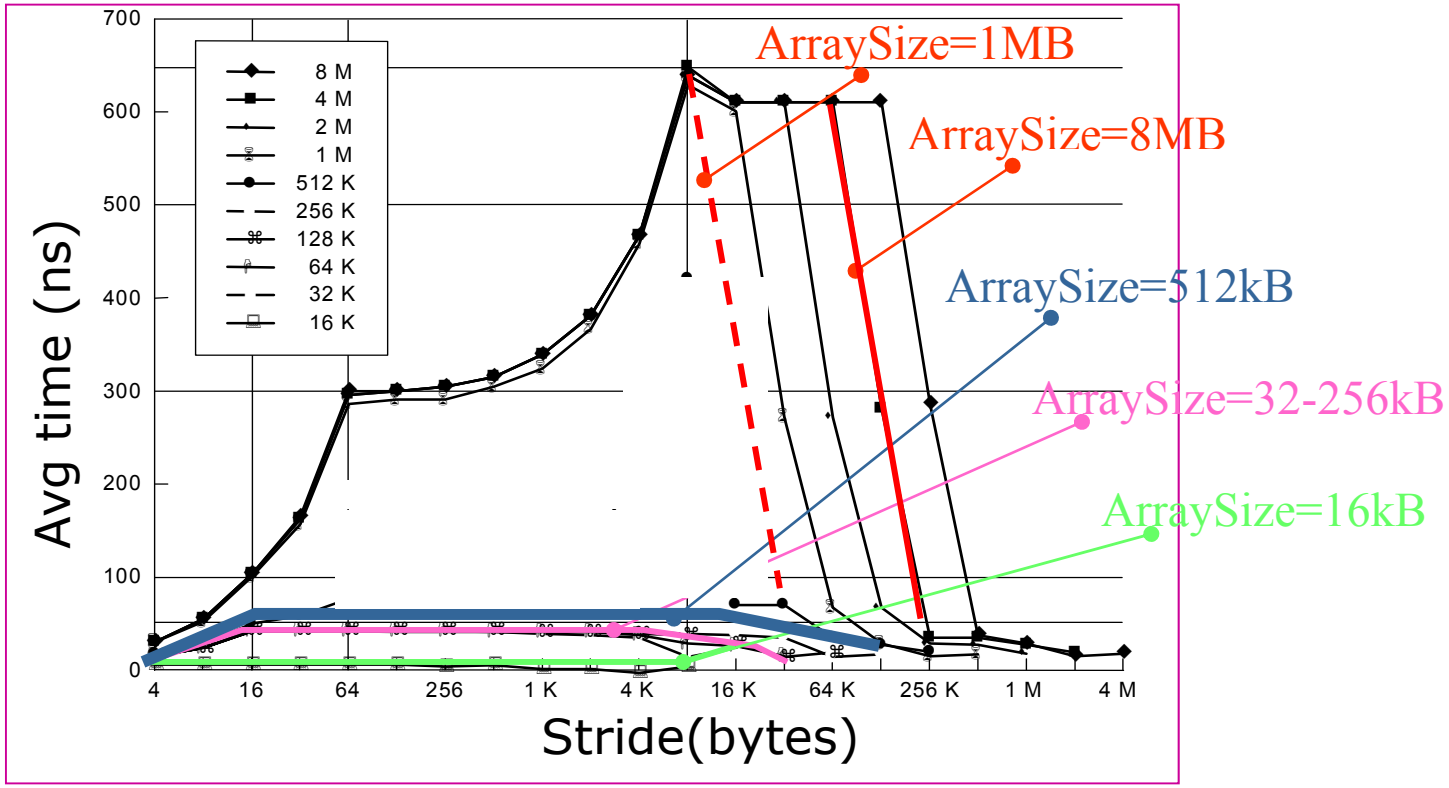
Twice as large L2 cache ???

```
for (times = 0; times < Max; time++) /* many times*/

  for (i=0; i < ArraySize; i = i + Stride)
    dummy = A[i]; /* touch an item in the array */
```



Twice as large TLB…

```
for (times = 0; times < Max; time++) /* many times*/

  for (i=0; i < ArraySize; i = i + Stride)
    dummy = A[i]; /* touch an item in the array */
```



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: ~50x
- Bandwidth difference L1$ and mem: ~20x
- Execute from L1$ instead from mem ==> 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
- …

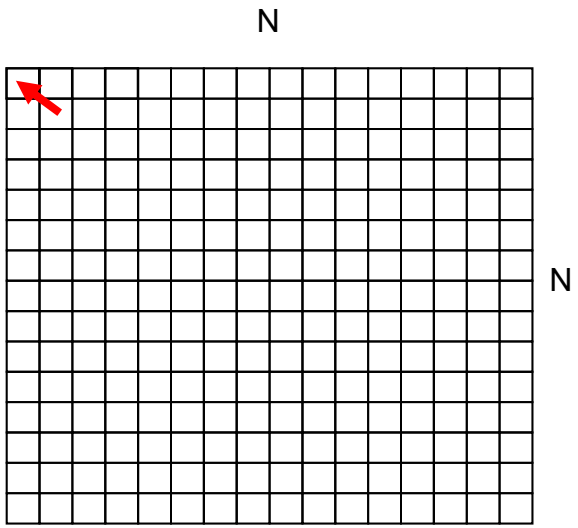
Final cache lingo slide

- **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

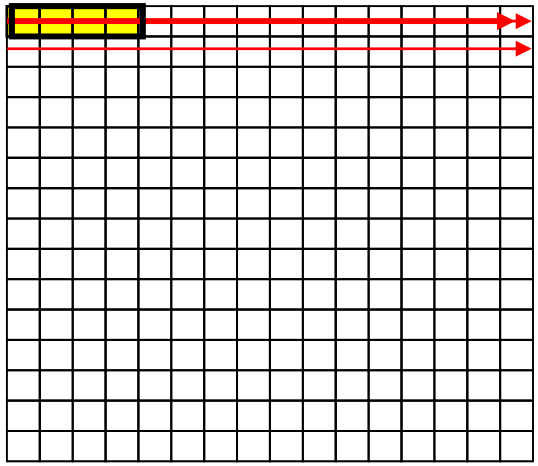
What can go Wrong? A Simple Example…

Perform a diagonal copy 10 times

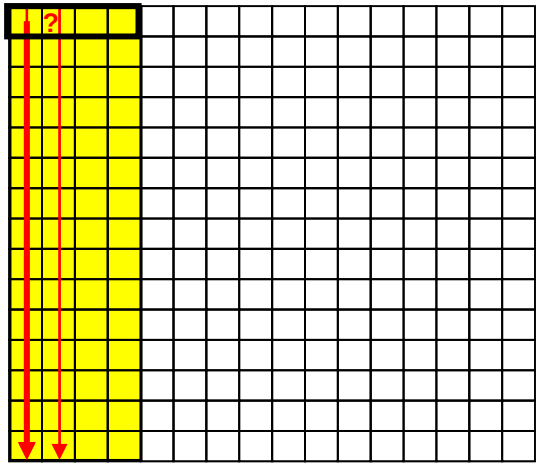


Example: Loop order

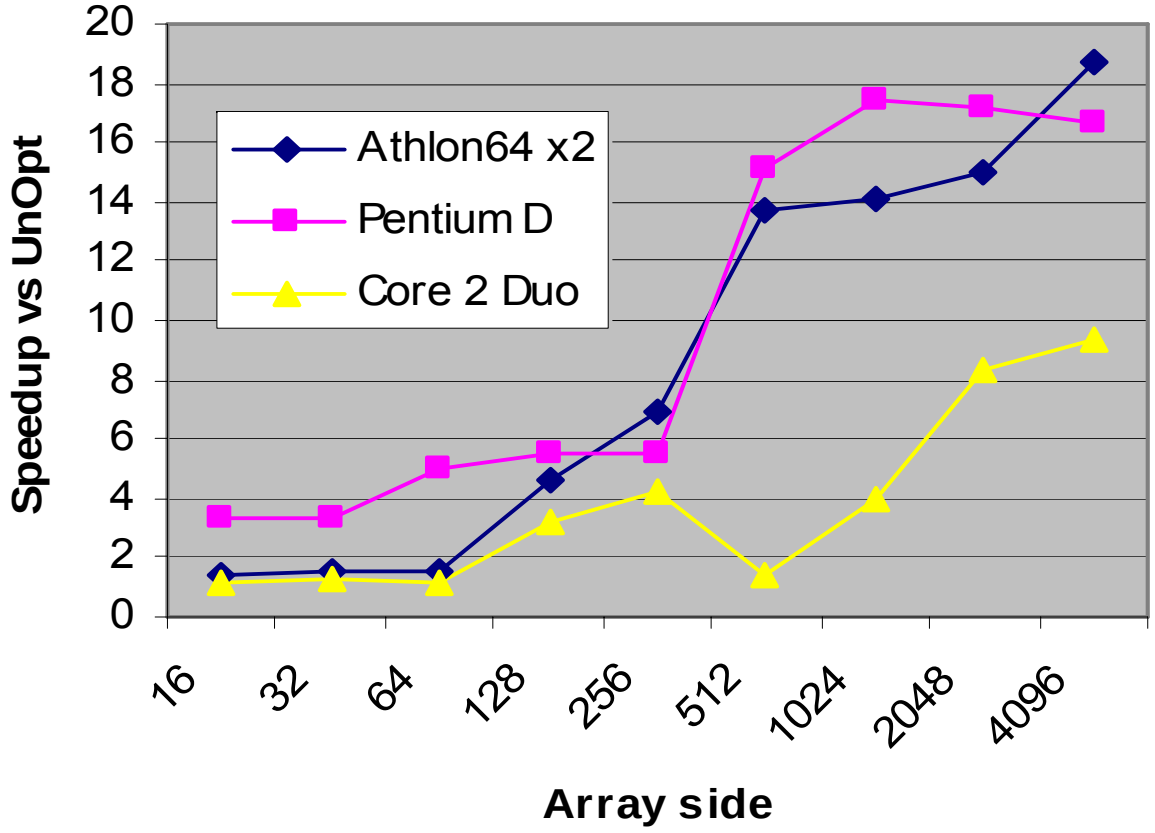
```
//Optimized Example A
for (i=1; i<N; i++) {
  for (j=1; j<N; j++) {
    A[i][j]= A[i-1][j-1];
  }
}
```



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



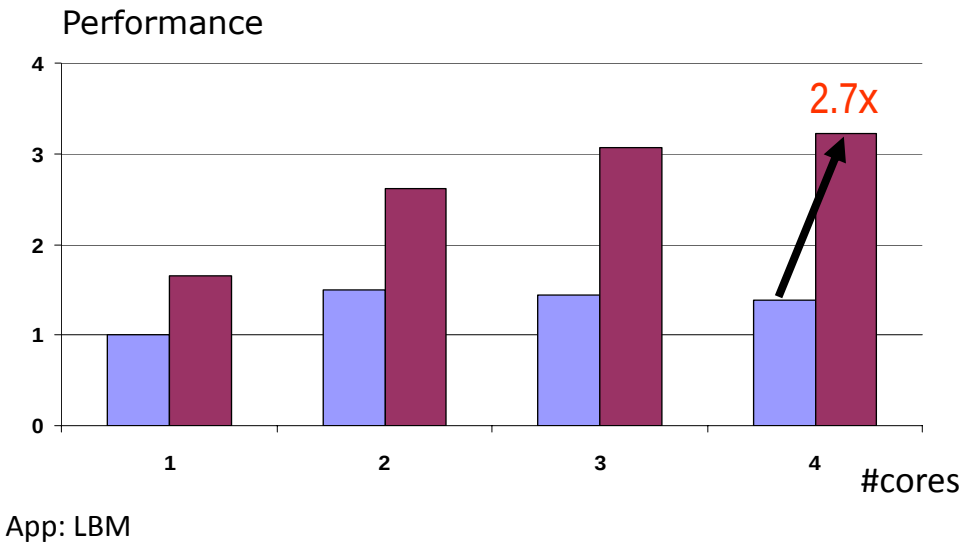
Performance Difference: Loop order



Demo Time!

ThreadSpotter

Example 1: The Same Application Optimized



Demo Time!

LBM:
Original code

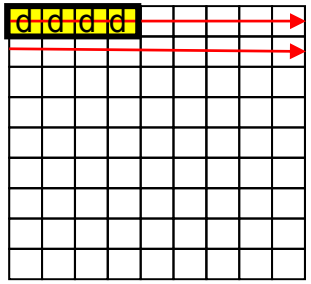
Optimization can be rewarding, but costly…

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

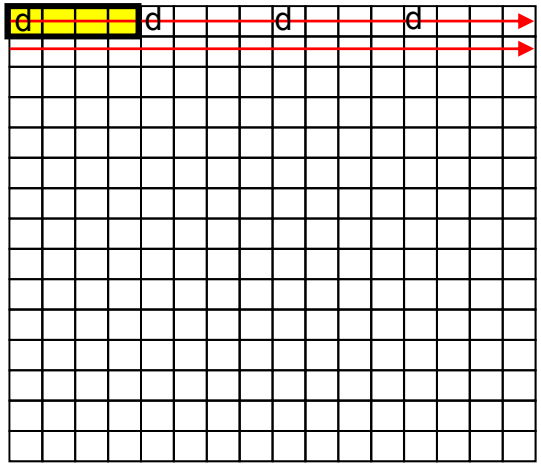
➔This fix required one line of code to change

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];
  }
}
```

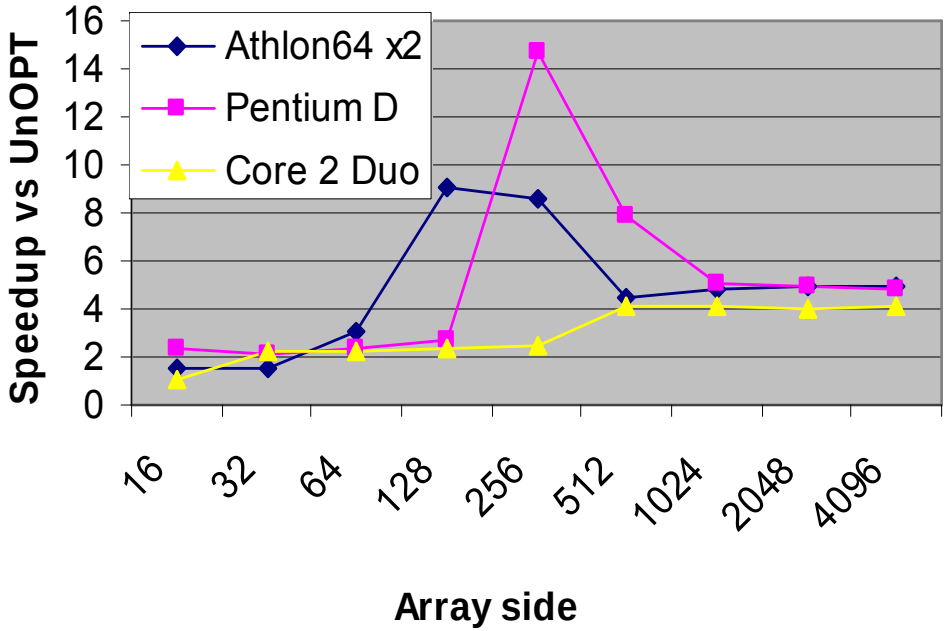



```
//Unoptimized Example A
for (i=1; i<N; i++) {
  for (j=1; j<N; j++) {
    A[i][j].d = A[i-1][j-1].d;
  }
}
```

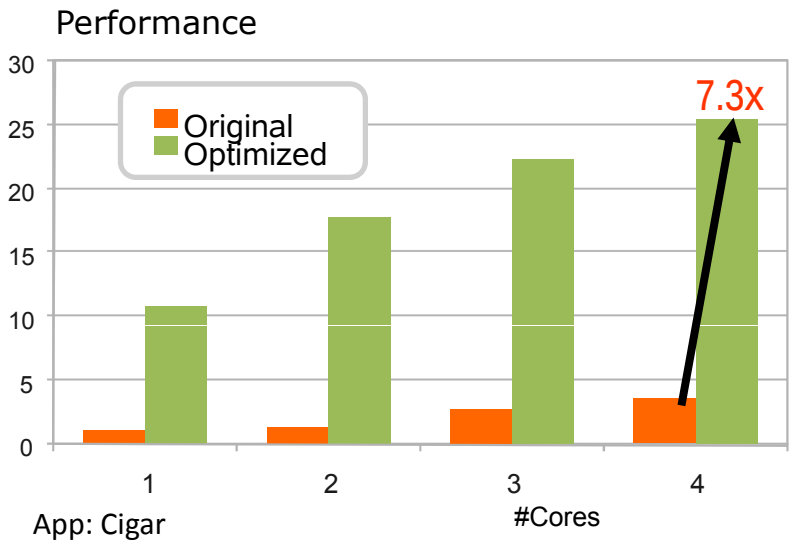



```
struct vec_type
{
        char a;
        char b;
        char c;
        char d;
};
```

Performance Difference: Sparse Data



Example 2: The Same Application Optimized



Looks like a perfect scalable application!

Are we done?

➔ Duplicate one data structure

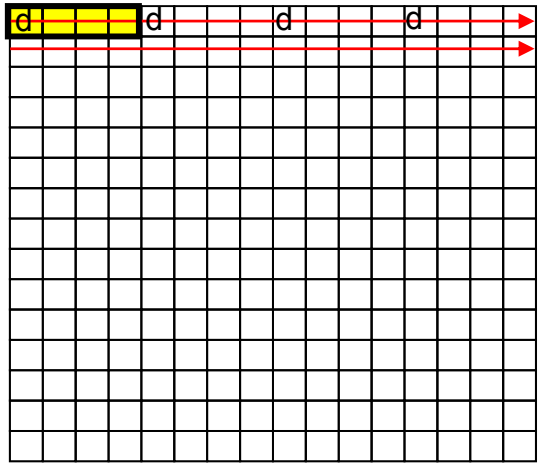
Demo Time!

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;
}
```



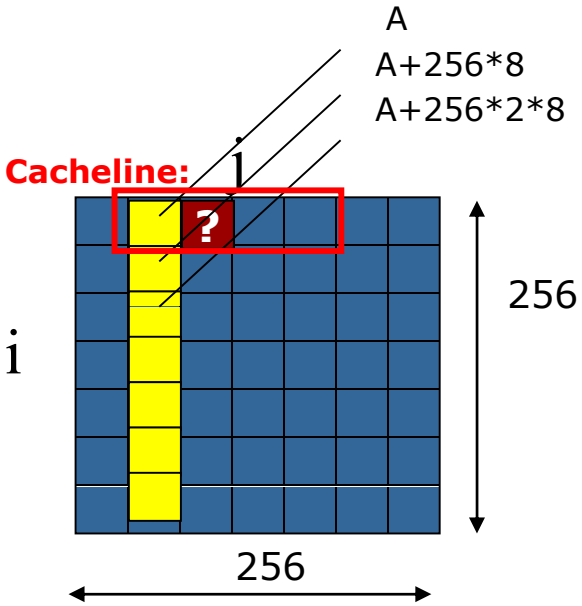

```
struct sparse_rec
{
        // size 80B
        char a;
        double f1;
        char b;
        double f2;
        char c;
        double f3;
        char d;
        double f4;
        char e;
        double f5;
};
```

```
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;
};
```

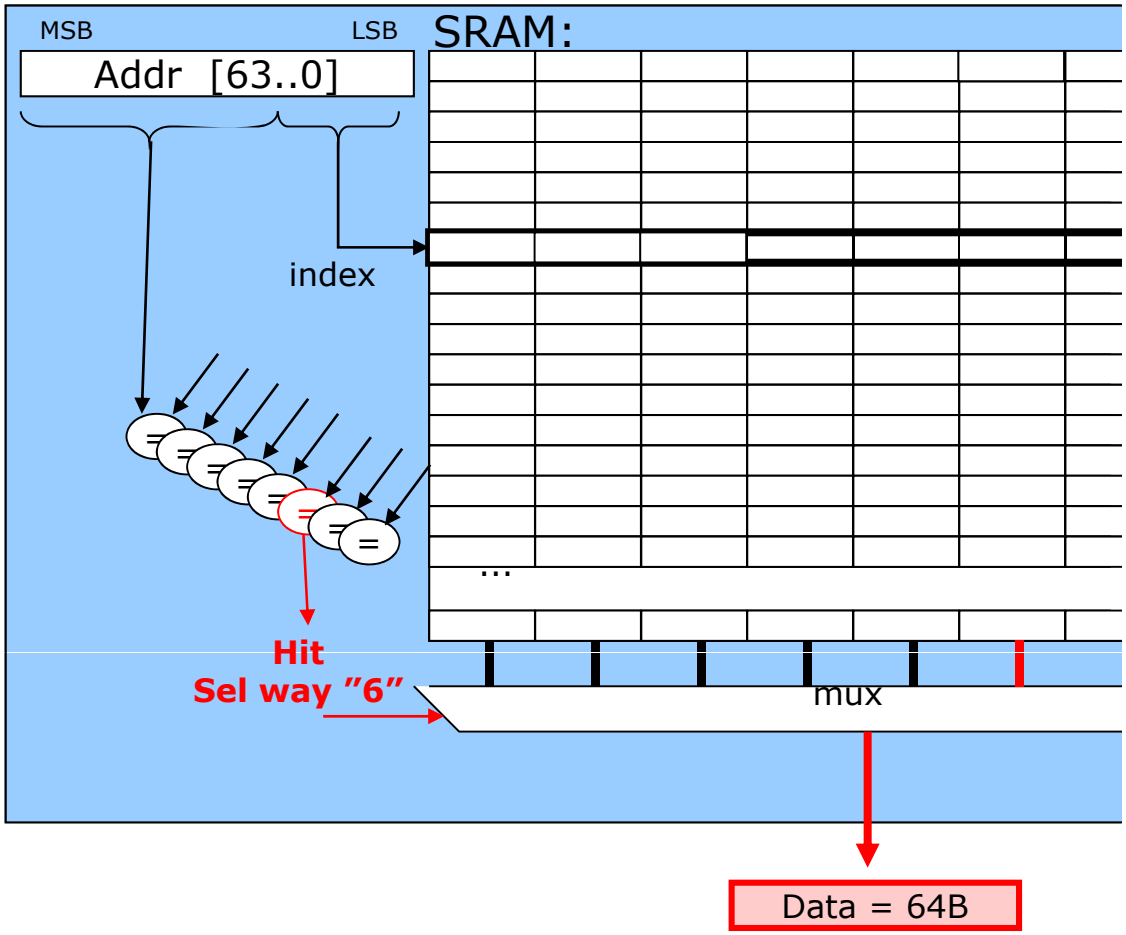
Loop Merging

```
/* 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;
```

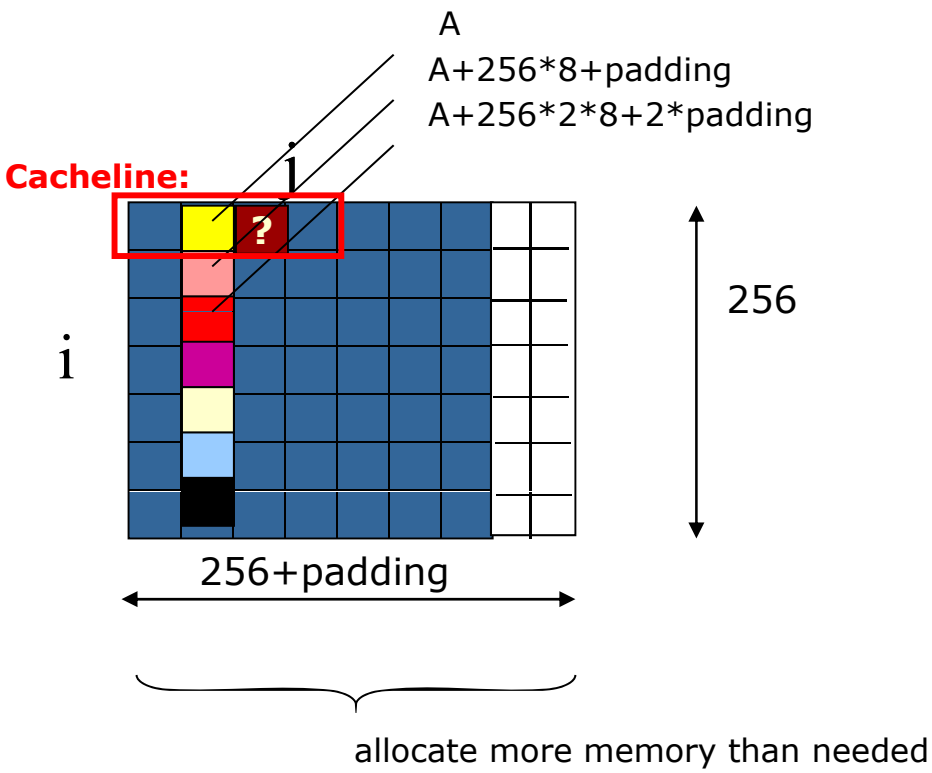
Padding of data structures



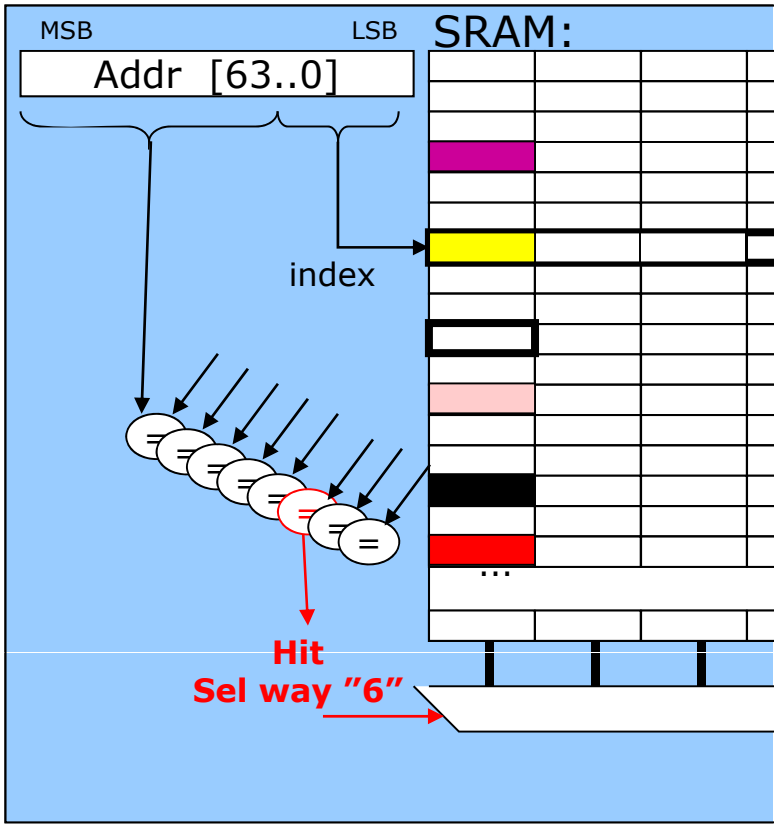
Generic Cache:



Padding of data structures

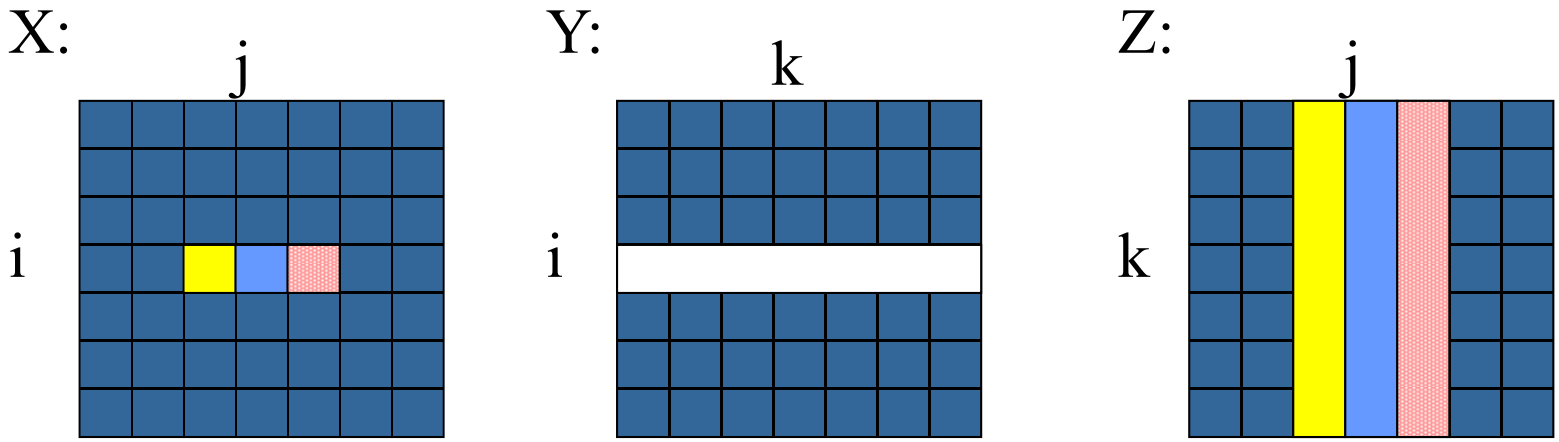


Generic Cache:



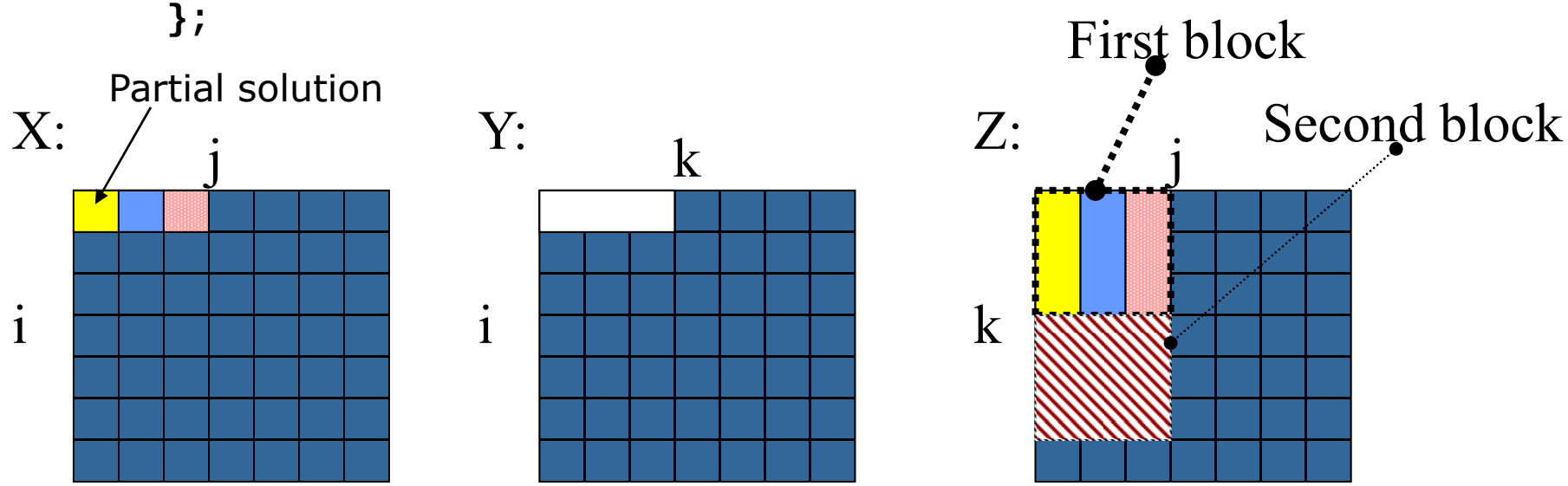
Blocking

```
/* 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;
        };
```



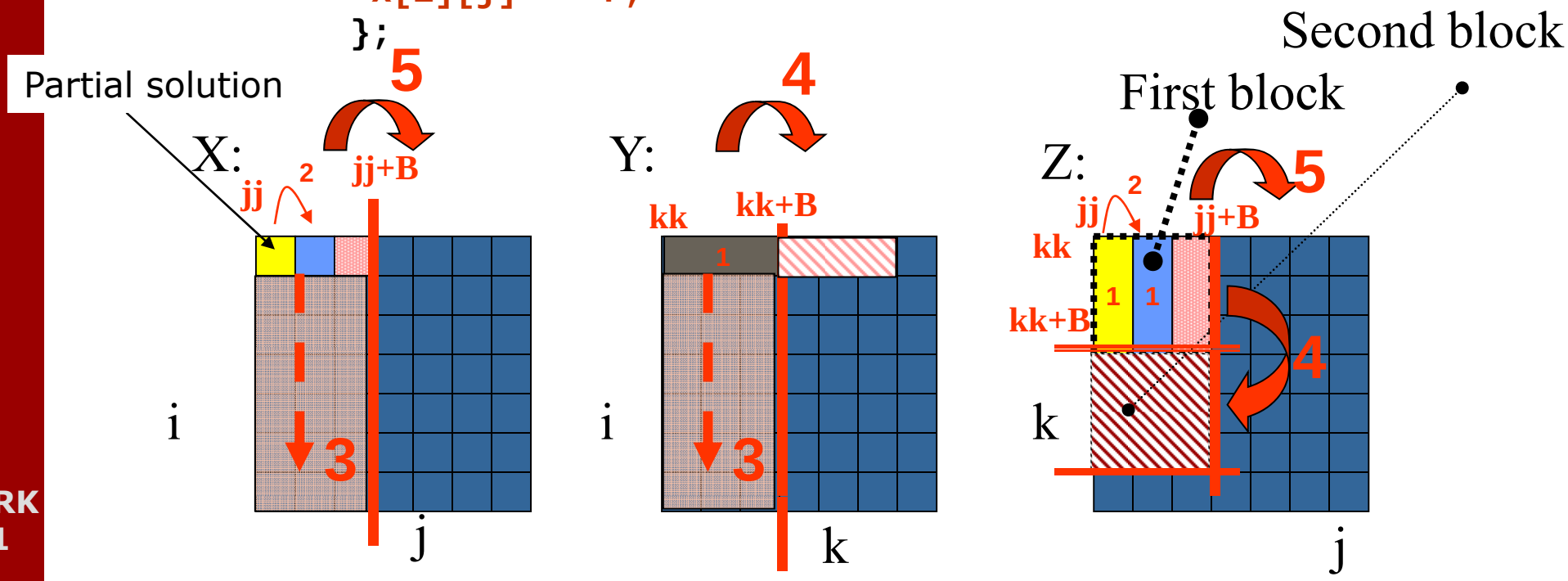
Blocking

```
/* 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;
        };
```



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];
```

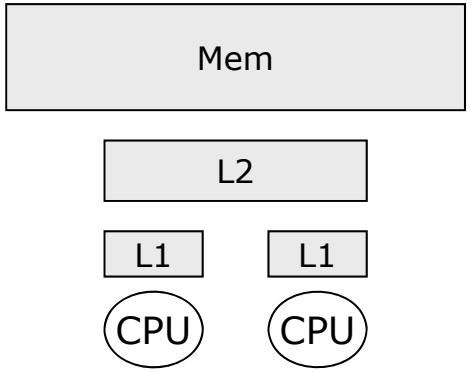
(Typically, the HW prefetcher will successfully prefetch sequential streams)

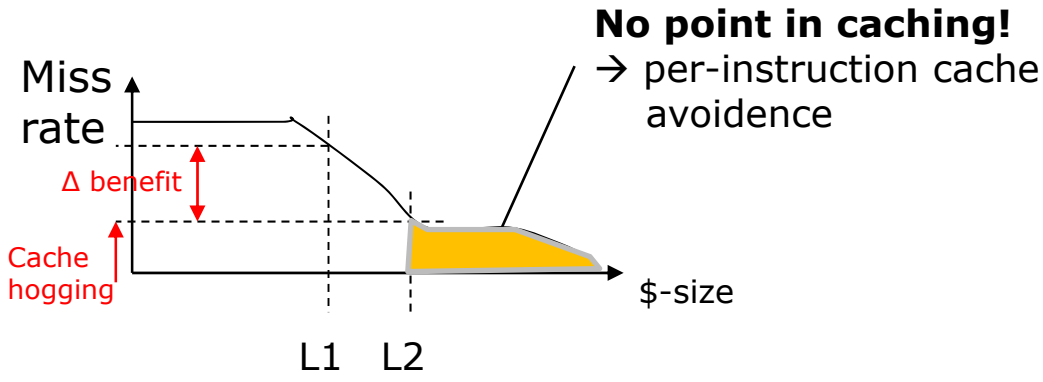
Cache Waste

```
/* 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*/
}
```

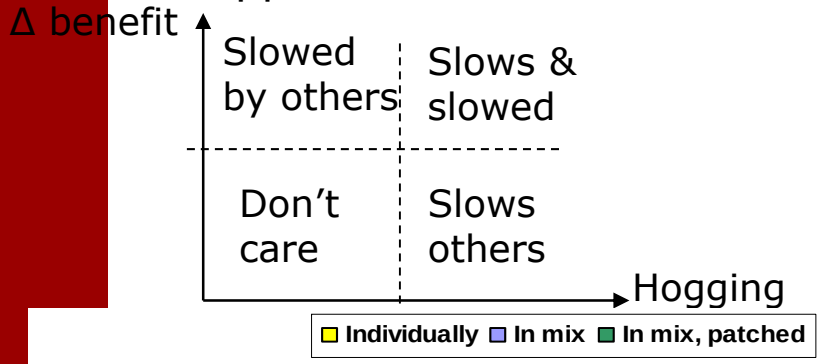
➔ Also important for single-threaded applications if they are co-scheduled and share cache with other applications.

Categorize and avoiding cache waste

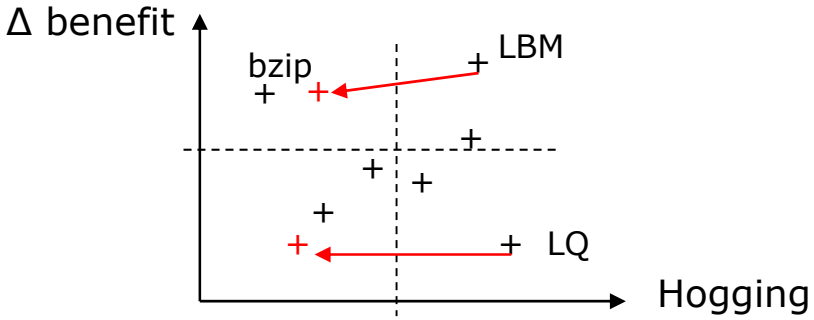


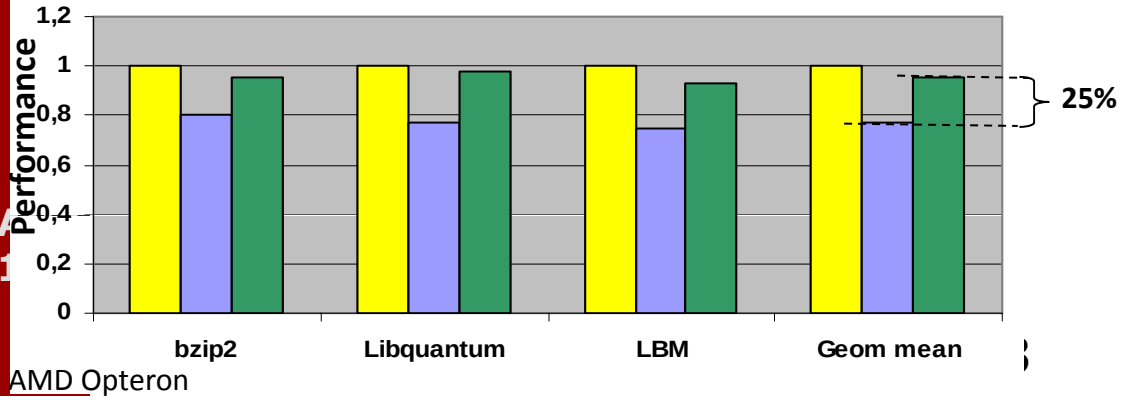


Application classification



Automatic "taming" of the hoggers

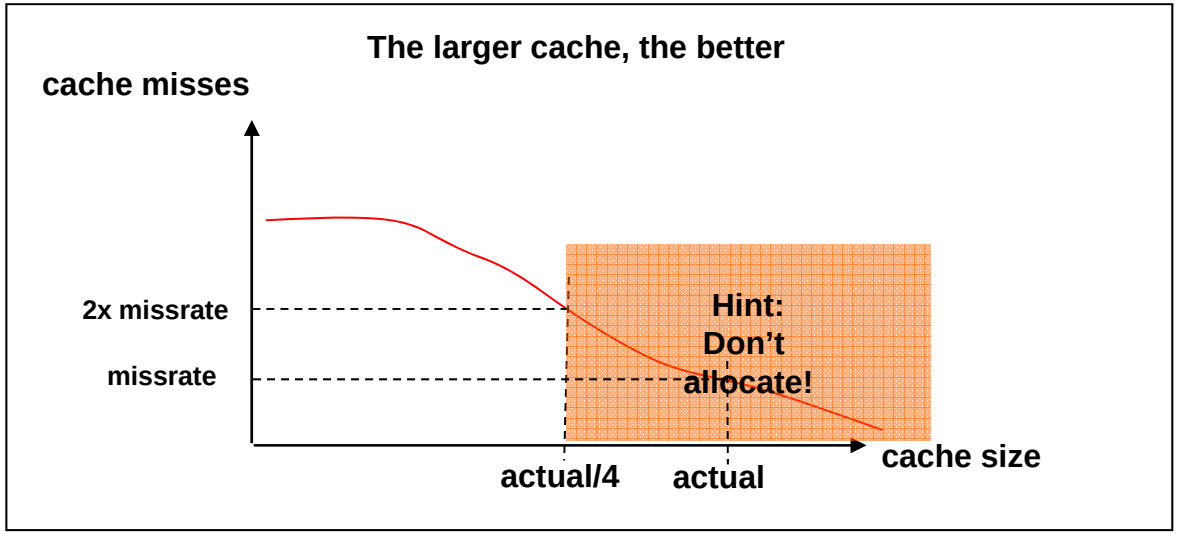


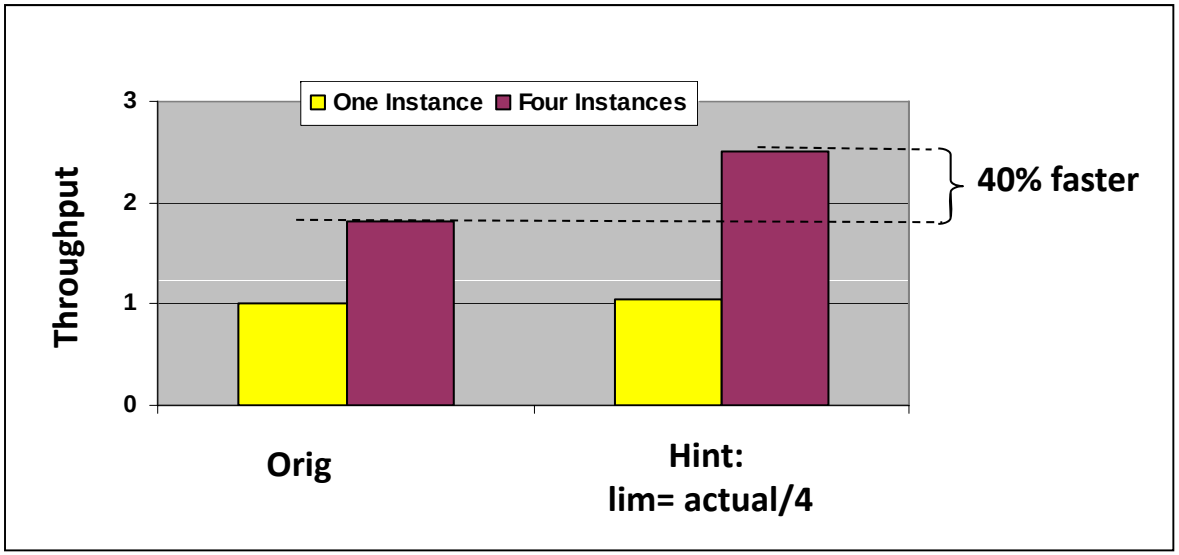


AMD Opteron

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)





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)

Commercial Break: ThreadSpotter

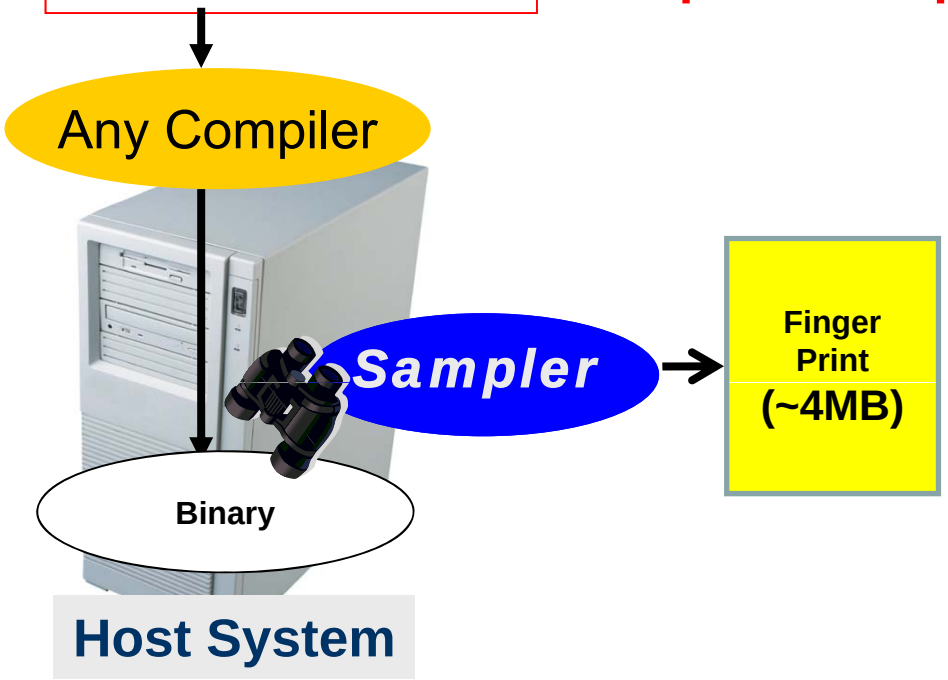
Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se

ThreadSpotter™

Source:
C, C++, Fortran, OpenMP…

```
/* Unoptimized  Array Multiplication: x = y * z    N = 1024 */
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;
      }
/* Unoptimized  Array Multiplication: x = y * z    N = 1024 */
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;
      }
```

Mission:
Find the SlowSpots™
Asses their importance
Enable for non-experts to fix them
Improve the productivity of performance experts



Source:
C, C++, For

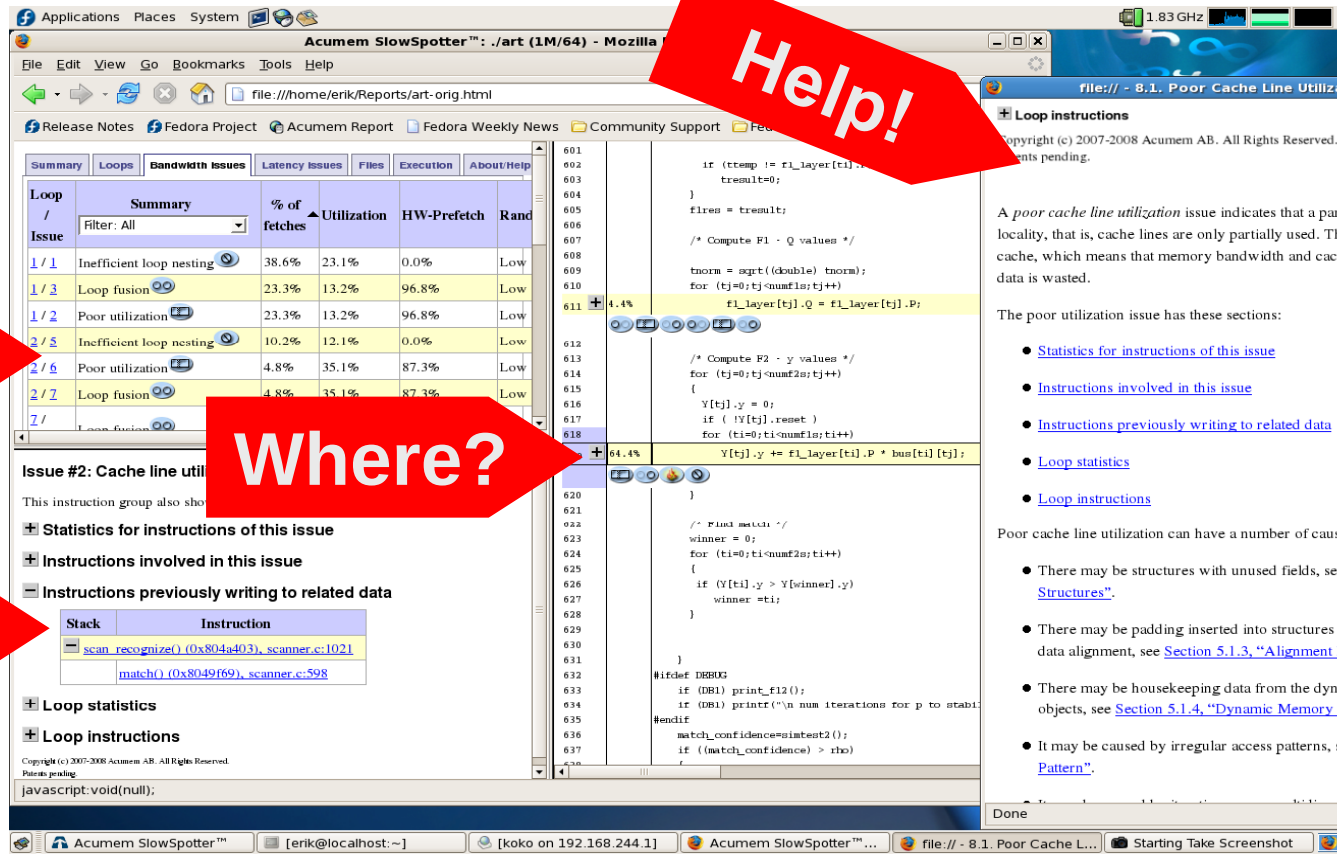
```
/* Unoptimized  Array Multiplicat
for (i = 0; i < N; i = i + 1)
  for (j = 0; j < N; j = j + 1)
    {r = 0;
    for (k = 0; k < N; k = k + 1)
      r =  r + y[i][k] * z[k][j];
    x[i][j] = r;
    }
/* Unoptimized  Array Multiplication: x = y * z    N = 1024 */
for (i = 0; i < N; i = i + 1)
  for (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;
    }
```

What?

Where?

How?

Help!



Any Compiler

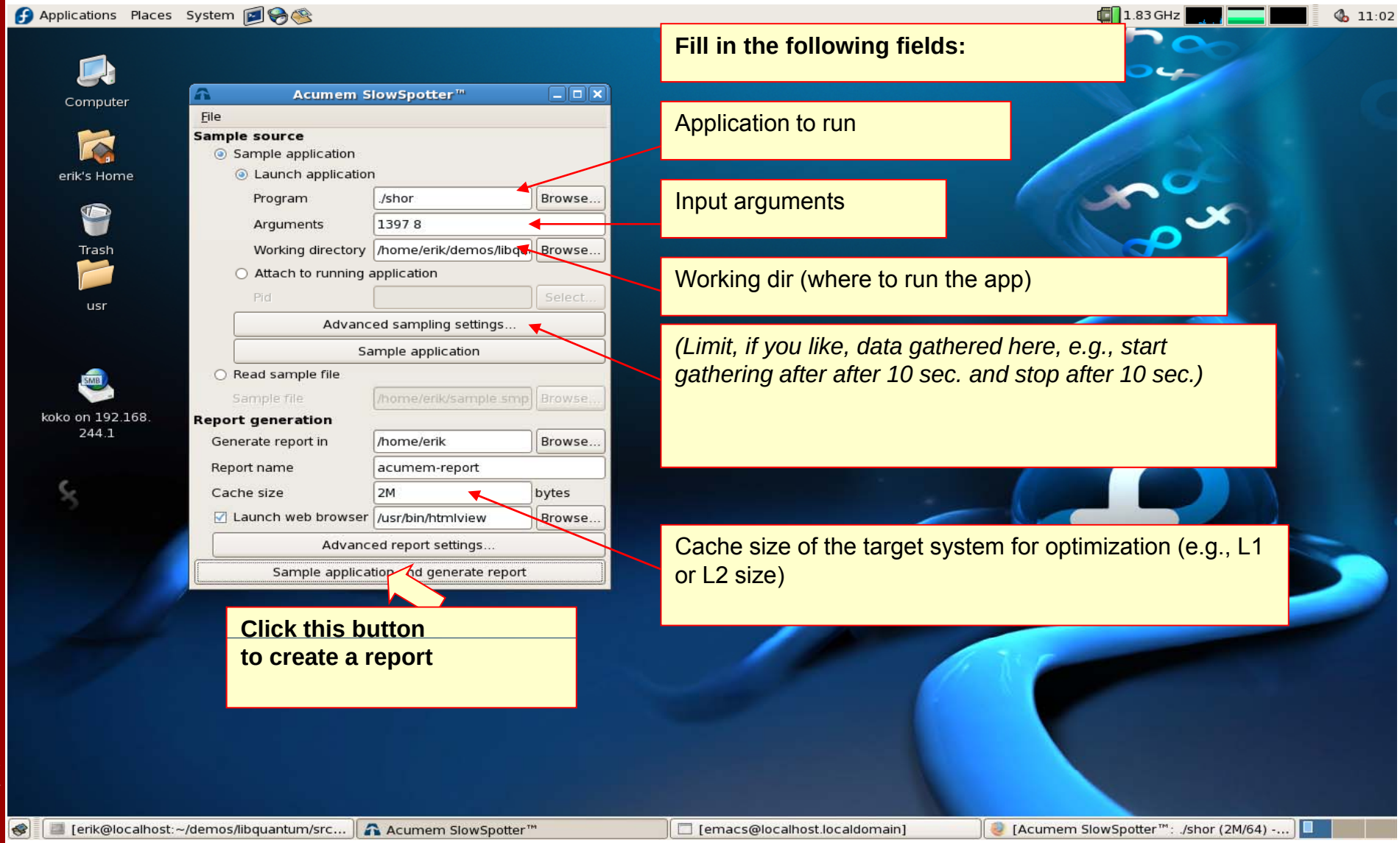
Binary

Sampler → Finger Print (~4MB) → Analysis → Advice

Target System Parameters

Host System

A One-Click Report Generation



Fill in the following fields:

Application to run

Input arguments

Working dir (where to run the app)

(Limit, if you like, data gathered here, e.g., start gathering after after 10 sec. and stop after 10 sec.)

Cache size of the target system for optimization (e.g., L1 or L2 size)

Click this button to create a report

Acumem VPE: ./art (64k/64) - Mozilla Firefox

Arkiv Redigera Visa Historik Bokmärken Verktyg Hjälp

file:///C:/Documents%20and%20Settings/Erik/Mina%20dokument/Demo/071112'

Summary | Loops | Bandwidth Issues | Latency Issues | Files | Execution | About/Help

Select a file in the file table, or follow a source code link from ... loop description.

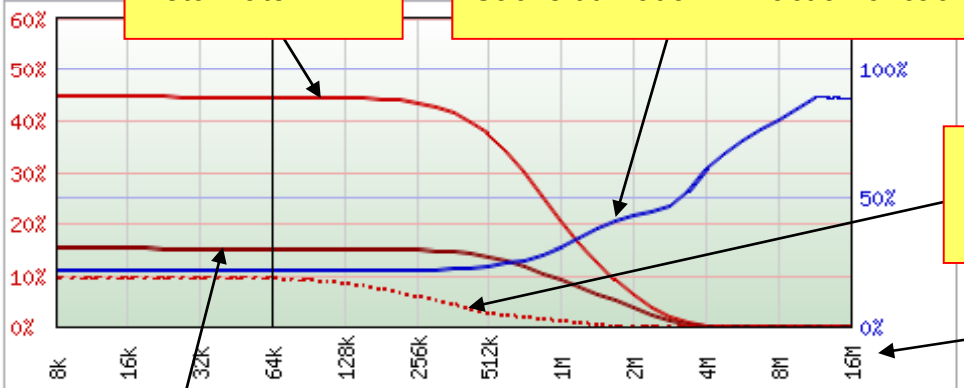
Fetch rate

Cache utilization ≈ Fraction of cache data utilized

Predicted fetch rate
(if utilization → 100%)

Cache size

Miss rate



Legend: -- Fetch rate .. Util. corr. fetch rate -- Miss rate -- Utilization

Miss rate	15.1%
Fetch rate	44.4%
Misses	5.45e+07
Fetches	1.60e+08
Cache line utilization	21.9%
Cache size	64k
Line size	64
Replacement policy	random

Klar

Start 3 U... Auto... Skick... Micr... Acu... SV 18:36

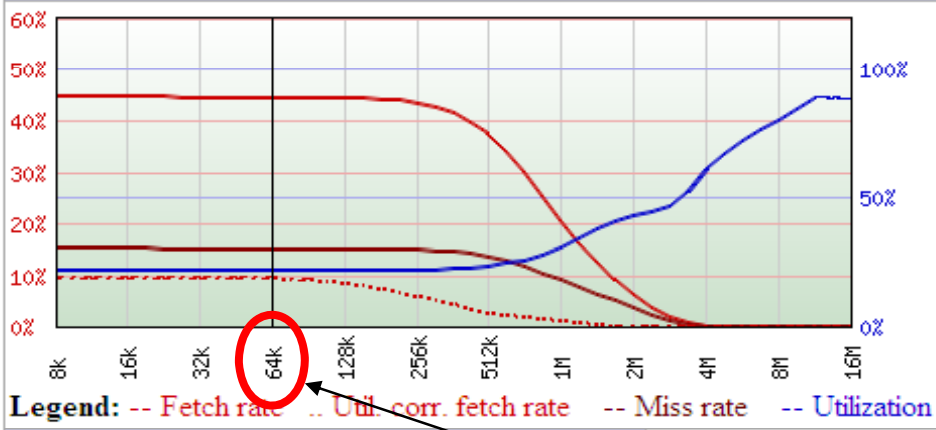
Acumem VPE: ./art (64k/64) - Mozilla Firefox

Arkiv Redigera Visa Historik Bokmärken Verktyg Hjälp

file:///C:/Documents%20and%20Settings/Erik/Mina%20dokument/Demo/071112'

Summary | Loops | Bandwidth Issues | Latency Issues | Files | Execution | About/Help

Miss rates and utilization



Miss rate	15.1%
Fetch rate	44.4%
Misses	5.45e+07
Fetches	1.60e+08
Cache line utilization	21.9%
Cache size	64k
Line size	64
Replacement policy	random

Cache size to optimize for

Select a file in the file table, or follow a source code link from an issue or a loop description.

Klar

Start 3 U... Auto... Skick... Micr... Acu... SV Google 18:36

Loop Focus Tab

Acumem VPE: ./art (64k/64) - Mozilla Firefox

Arkiv Redigera Visa Historik Bokmärken Verktyg

file:///C:/Document

Summary | Loops | Bandwidth Issues | Latency Issues | Files | Execution | About/Hel

Loop	% of misses	% of fetches	Utilization	Issues
1	85.8%	62.3%	17.7%	
2	9.5%	7.7%	23.8%	
4	0.0%	6.1%	34.1%	
3	0.0%	4.4%	23.7%	
6	0.0%	4.2%	36.8%	
7	0.0%	4.2%	25.1%	
5	0.0%	4.0%	26.1%	
8	4.2%	3.2%	18.4%	
9	0.0%	1.1%	23.5%	

Cache line utilization
Inefficient loop nesting
Random access pattern

List of bad loops

Loop 1

- Loop statistics
- Loop instructions
- Instruction groups in this loop, summary of issues
- Instruction group 1
- Instruction group 2
- Instruction group 3

Explaining what to do

```
600          tnorm += f1_layer[ti].P * f1_layer[ti].P;
601
602          if (ttemp != f1_layer[ti].P)
603             tresult=0;
604       }
605       f1res = tresult;
606
607       /* Compute F1 - Q values */
608
609       tnorm = sqrt((double) tnorm);
610       for (tj=0;tj<numf1s;tj++)
611  4.4%      f1_layer[tj].Q = f1_layer[tj].P;
612
613       /* Compute F2 - y values */
614       for (tj=0;tj<numf2s;tj++
615       {
616         Y[tj].y = 0;
617         if ( !Y[tj].reset )
618         for (ti=0;ti<numf1s;ti++)
619  65.4%      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
620       }
621
622       /* Find match */
623       winner = 0;
624       for (ti=0;ti<numf2s;ti++)
625       {
626        if (Y[ti].y > Y[winner].y)
627           winner =ti;
628       }
629
630
631    }
632 #ifdef DEBUG
633    if (DB1) print_f12();
634    if (DB1) printf("\n num iterations for p to stabilize = %i \n",j);
635 #endif
636    match_confidence=simtest2();
637    if ((match_confidence) > rho)
638    {
```

Spotting the crime

Klar

Start | 3 Ut... | Auto-... | Skick... | Micro... | Acum... | 18:47

Bandwidth Focus Tab

Acumem VPE: ./art (64k/64) - Mozilla Firefox

Summary | Loops | Bandwidth Issues | Latency Issues | Files | Execution | About/Help

Loop / Issue	Summary (Filter: All)	% of fetches	Utilization	HW-Prefetch	Randomness
1 / 3	Poor utilization	29.4%	12.4%	100.0%	Low
1 / 4	Loop fusion	29.4%	12.4%	97.6%	Low
1 / 1	Inefficient loop nesting	29.2%	12.6%	0.0%	Low
3 / 9	Loop fusion	4.4%	11.8%	97.3%	Low
3 / 8	Poor utilization	4.4%	23.7%	100.0%	Low
4 / 13	Loop fusion	4.2%	12.7%	96.7%	Low
4 / 12	Loop fusion	4.2%	12.7%	96.7%	Low
7 / 18	Poor utilization	4.2%	25.1%	100.0%	Low
4 / 10	Poor utilization				

List of Bandwidth SlowSpots

Issue #1: Inefficient loop nesting

This instruction group also show symptoms of: Cache line utilization, Hot-spot.

- Statistics for instructions of this issue
- Instructions involved in this issue
- Loop statistics
- Loop instructions

Explaining what to do

```
600            tnorm += f1_layer[ti].P * f1_layer[ti].P;
601
602            if (ttemp != f1_layer[ti].P)
603               tresult=0;
604         }
605         f1res = tresult;
606
607         /* Compute F1 - Q values */
608
609         tnorm = sqrt((double) tnorm);
610         for (tj=0;tj<numf1s;tj++)
611   4.4%        f1_layer[tj].Q = f1_layer[tj].P;
612
613         /* Compute F2 - y values */
614         for (tj=0;tj<numf2s;tj++)
615         {
616           Y[tj].y = 0;
617           if ( !Y[tj].reset
618           for (ti=0;ti<numf1s;ti++)
619   65.4%       Y[tj].y += f1_layer[ti].P * bus[ti][tj];
620         }
621
622         /* Find match */
623         winner = 0;
624         for (ti=0;ti<numf2s;ti++)
625         {
626          if (Y[ti].y > Y[winner].y)
627             winner =ti;
628         }
629
630
631      }
632   #ifdef DEBUG
633      if (DB1) print_f12();
634      if (DB1) printf("\n num iterations for p to stabilize = %i \n",
635   #endif
636      match_confidence=simtest2();
637      if ((match_confidence) > rho)
638      {
```

Spotting the crime

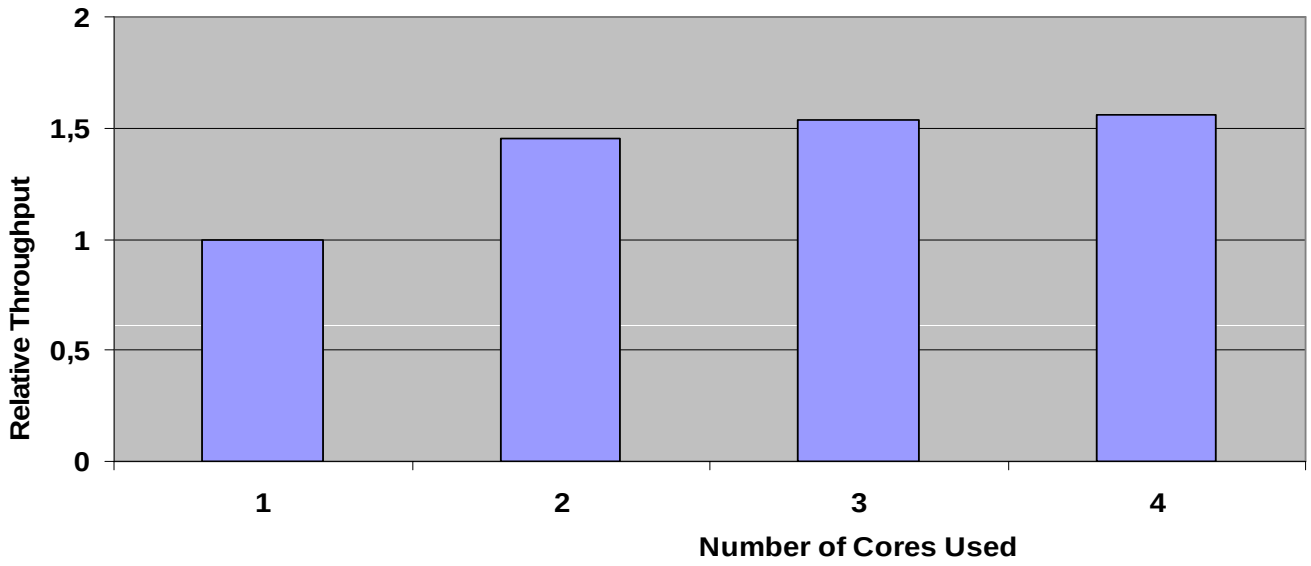
Klar

Start | Inkorgen f... | 2 Utforsk... | 071215 Ac... | Acumem ... | 03:15

Resource Sharing Example

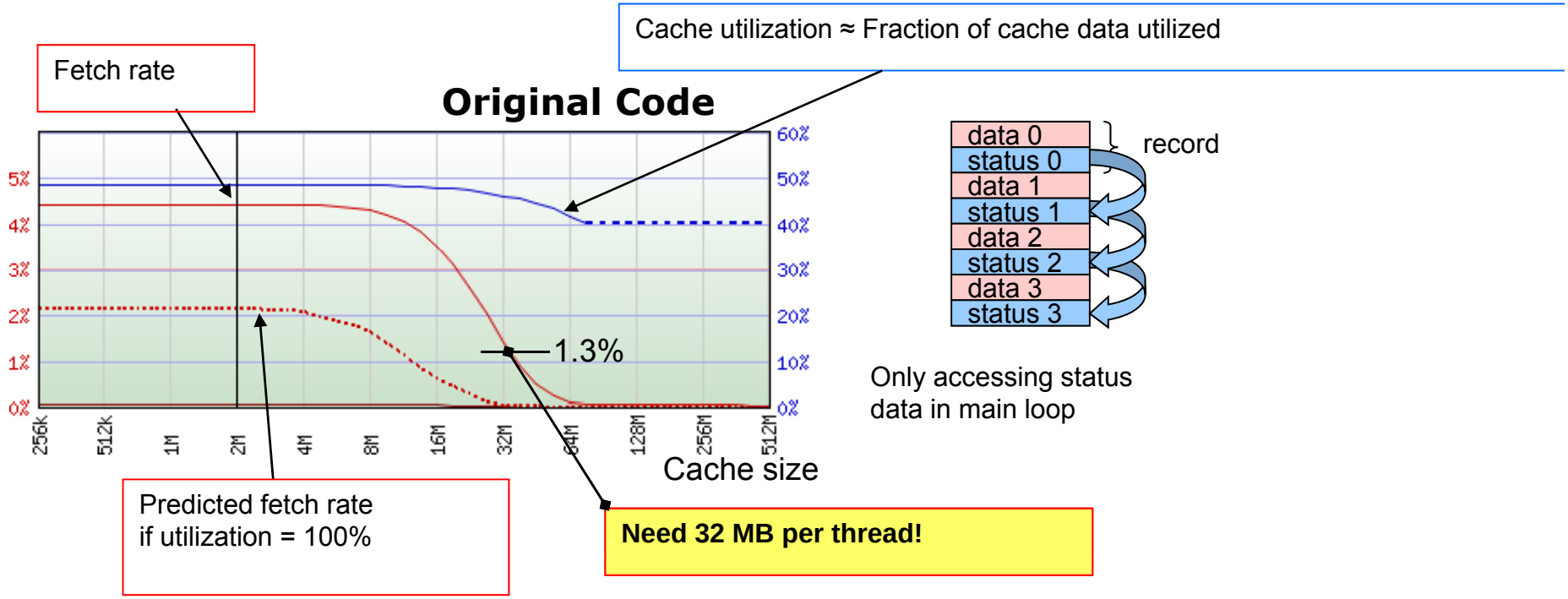
Libquantum
A quantum computer simulation
Widely used in research (download from: http://www.libquantum.de/)
4000+ lines of C, fairly complex code.
Runs an experiment in ~30 min

Throughput improvement:



Utilization Analysis

Libquantum



SlowSpotter's First Advice: Improve Utilization

➔Change one data structure

- Involves ~20 lines of code
- Takes a non-expert 30 min

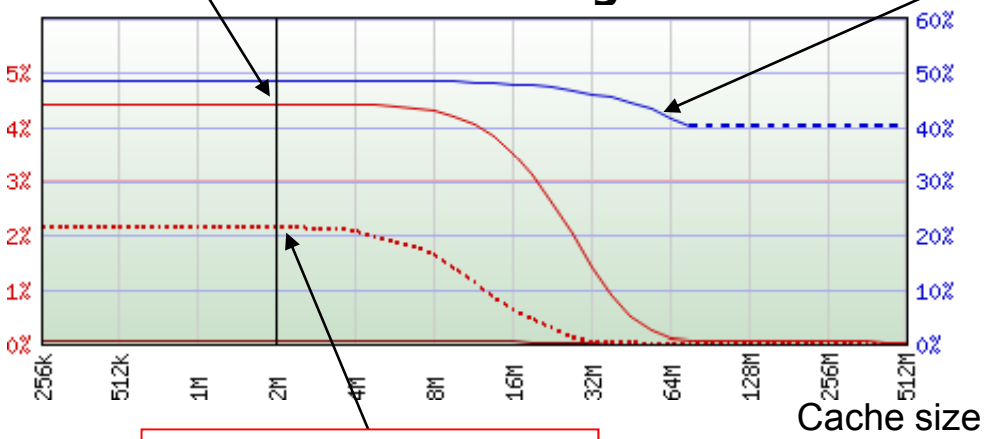
Utilization Analysis

Libquantum

Cache utilization ≈ Fraction of cache data utilized

Fetch rate

Original Code



Predicted fetch rate
if utilization = 100%

Utilization Optimization

```
for (i=0; i++; i<MAX) {
    ... = huge_data[i].status + ...
 }

for (i=0; i++; i<MAX) {
    ... = huge_data_status[i] + ...
 }
```

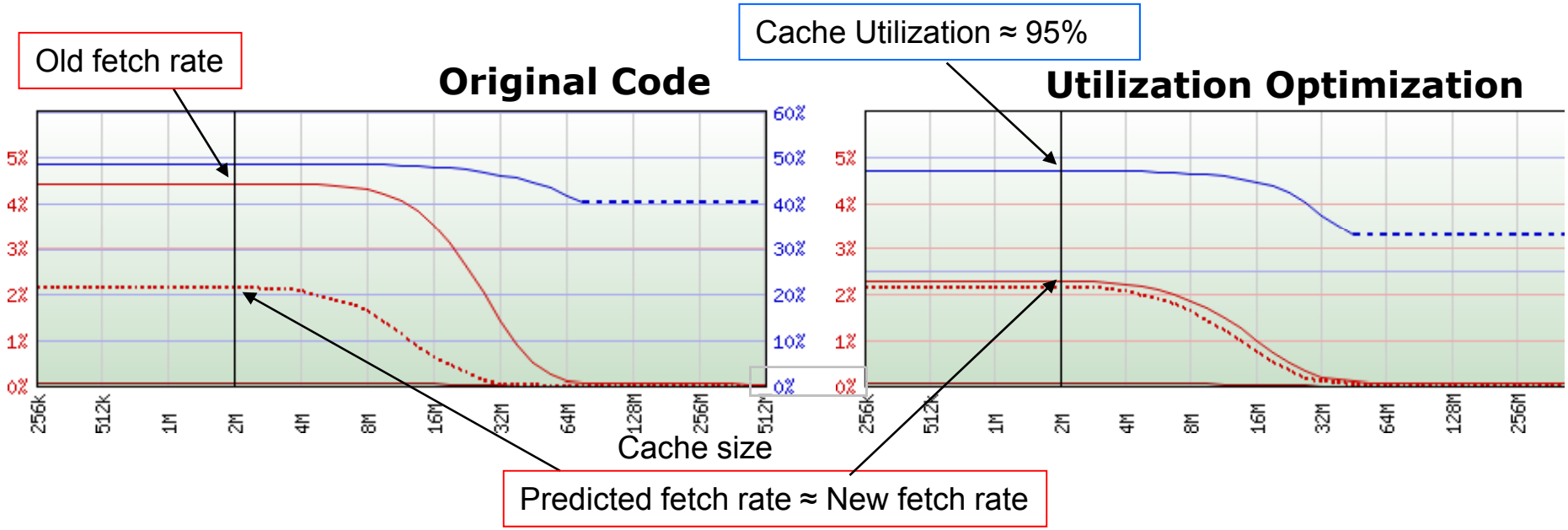
SlowSpotter's First Advice: Improve Utilization

➔Change one data structure

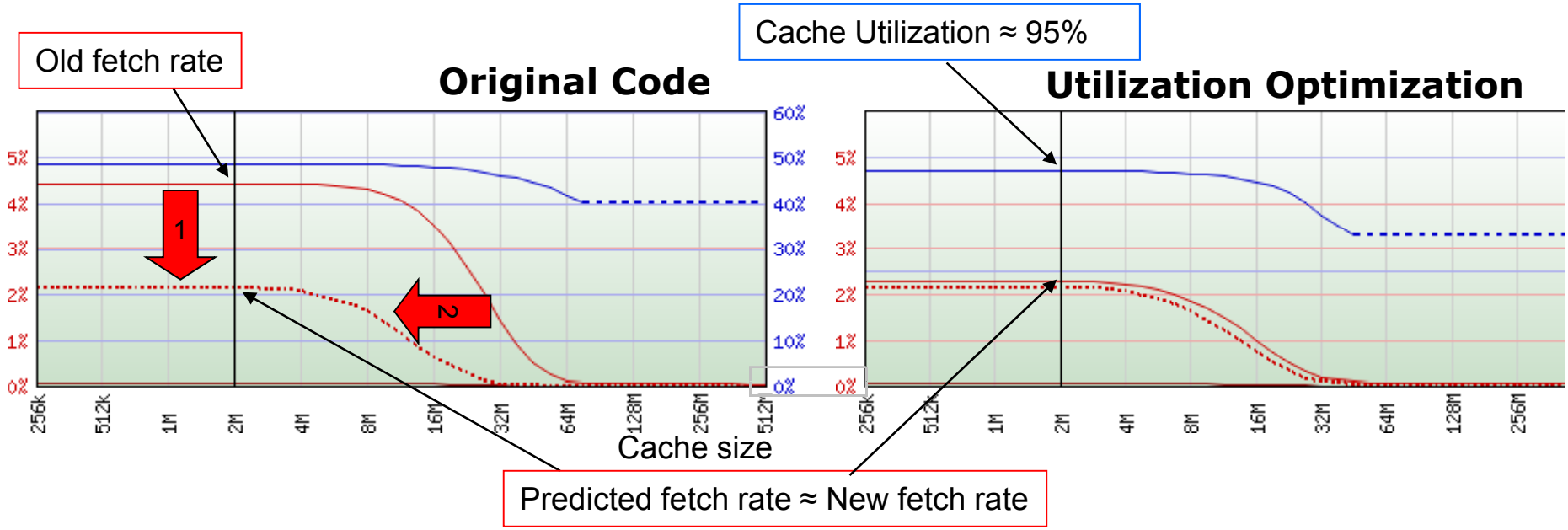
- Involves ~20 lines of code
- Takes a non-expert 30 min

After Utilization Optimization

Libquantum



Utilization Optimization

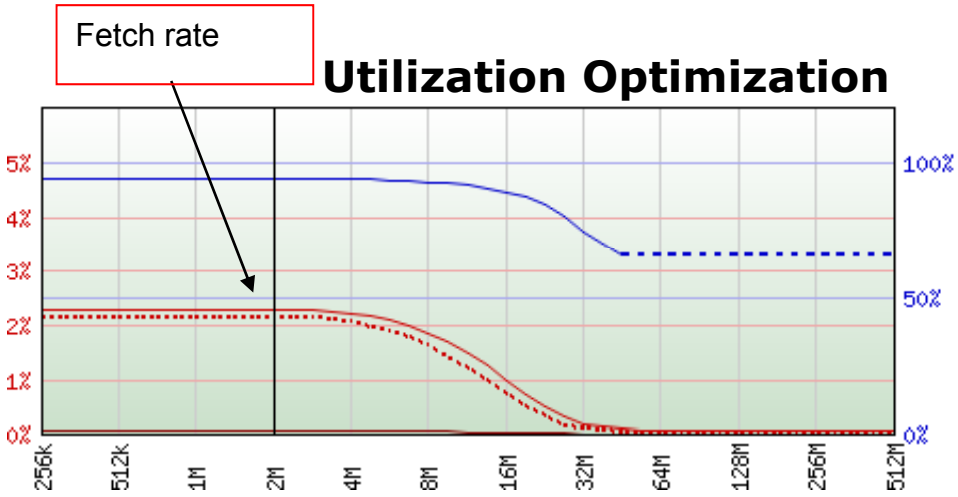


Two positive effects from better utilization

1. Each fetch brings in more useful data → lower fetch rate
2. The same amount of useful data can fit in a smaller cache → shift left

Reuse Analysis

Libquantum



Utilization + Fusion Optimization

```
...
toffoli(huge_data, ...)
cnot(huge_data, ...
...

...
fused_toffoli_cnot(huge_data,...)
...
```

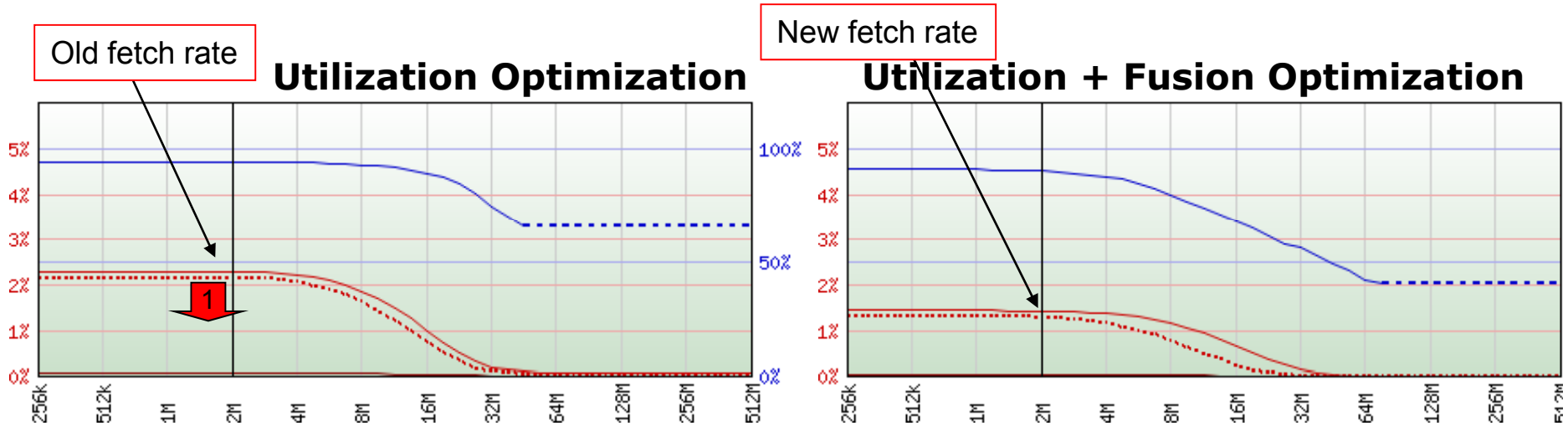
Second-Fifth SlowSpotter Advice: Improve reuse of data

➔Fuse functions traversing the same data

- Here: four fused functions created
- Takes a non-expert < 2h

Effect: Reuse Optimization

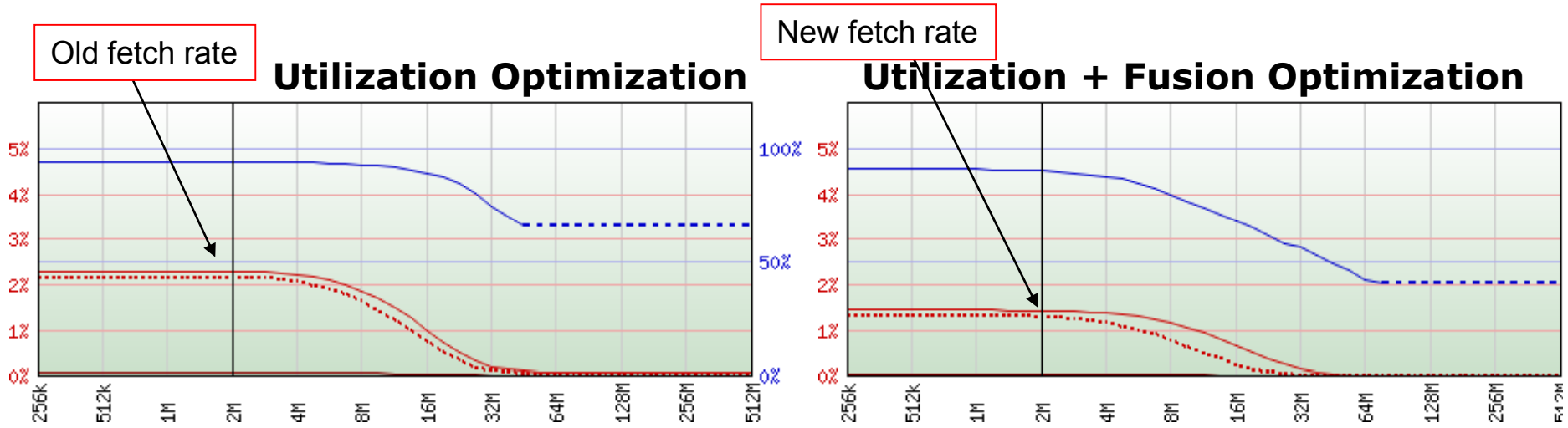
SPEC CPU2006-462.libquantum



- The miss in the second loop goes away
- Still need the same amount of cache to fit "all data"

Utilization + Reuse Optimization

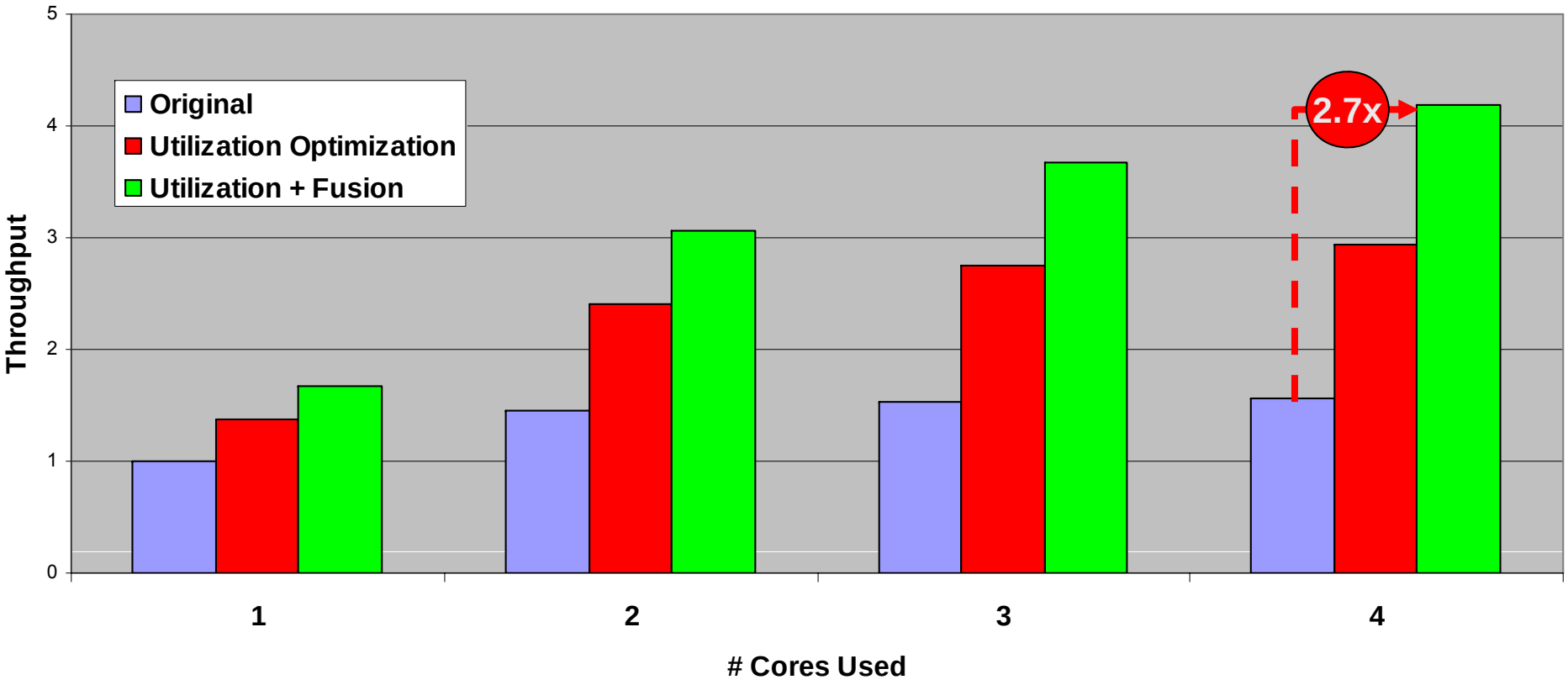
Libquantum



- Fetch rate down to 1.3% for 2MB
- Same as a 32 MB cache originally

Summary

Libquantum



UPPSALA UNIVERSITET

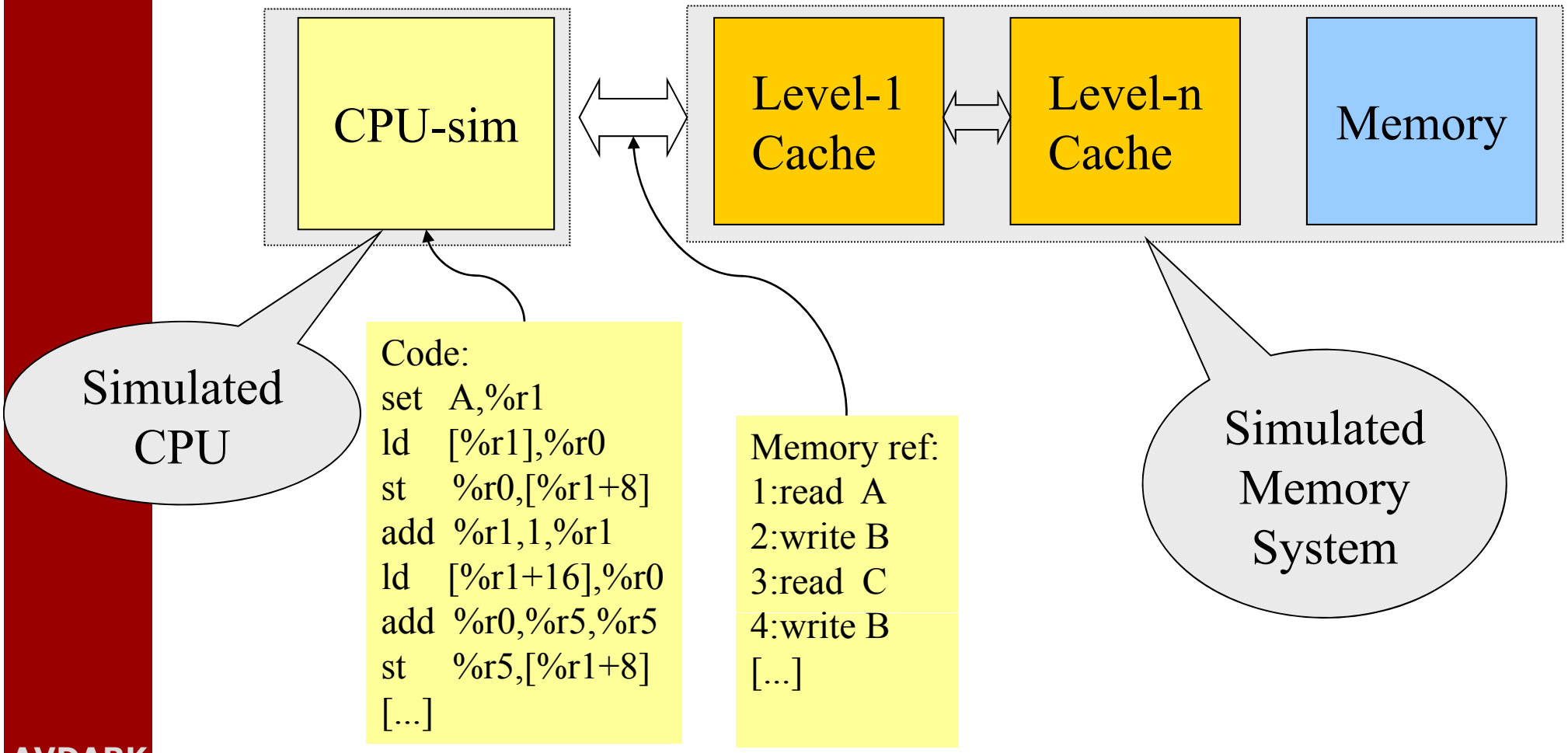
UPMARC Uppsala Programming for Multicore Architectures Research Center

- 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

Underneath the ThreadSpotter Hood

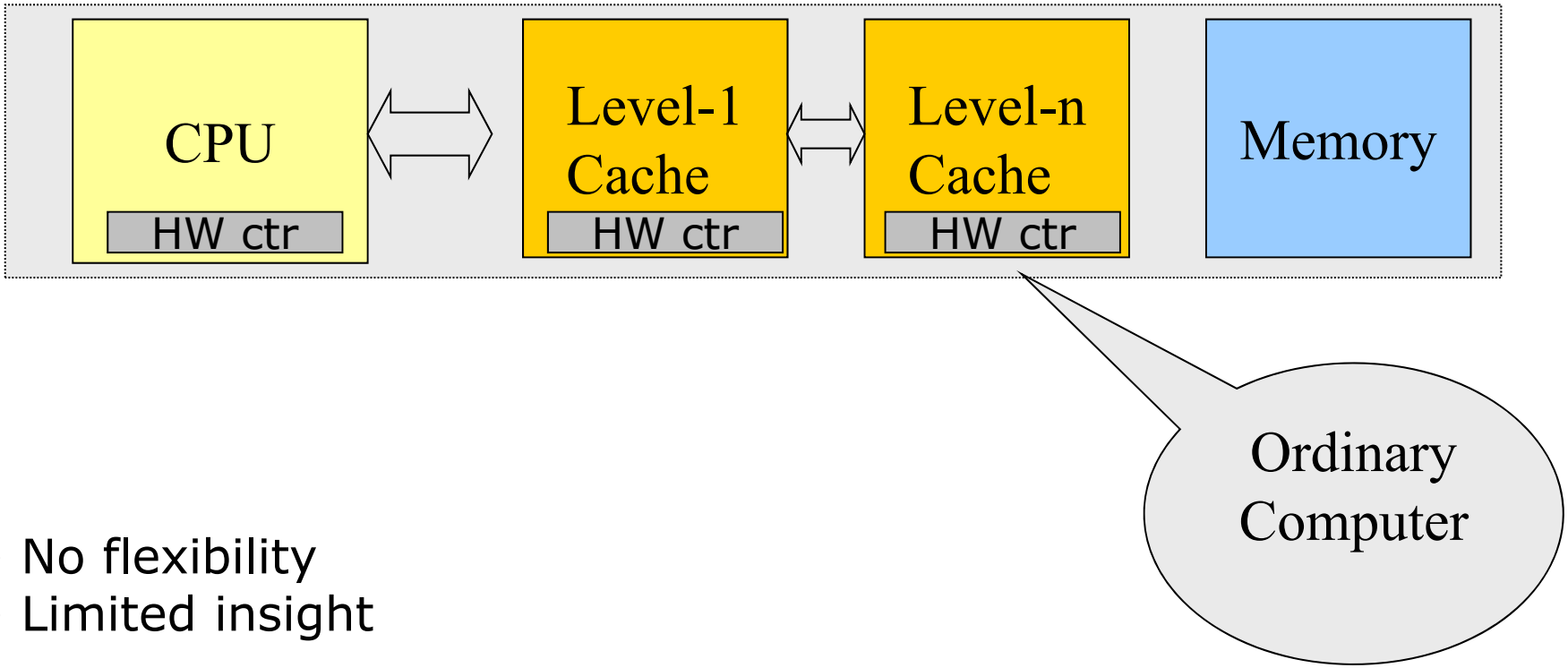
Great but Slow Insight: Simulation

Slowdown: ≈ 10 - 1000x



Limited Insight: Hardware Counters

Slowdown: ≈ 0%



- No flexibility
- Limited insight

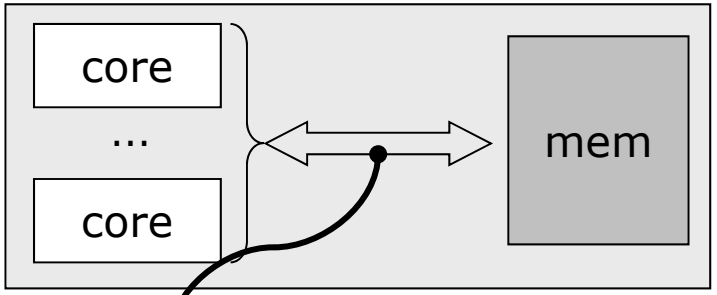
Insight: *"Instruction X misses Y% of the time in the cache"*
Architecturally dependent (!!)

StatCache: Insight and Efficiency

Slowdown 10% (for long-running applications)

Online Sampling

Host Computer



Randomly select accesses to monitor

Address Stream

1:read A
2:read B
3:read C
4:write C
5:read B
6:read D
7:read A
8:read E
9:read B

Reuse Distance=5

Reuse Distance=3

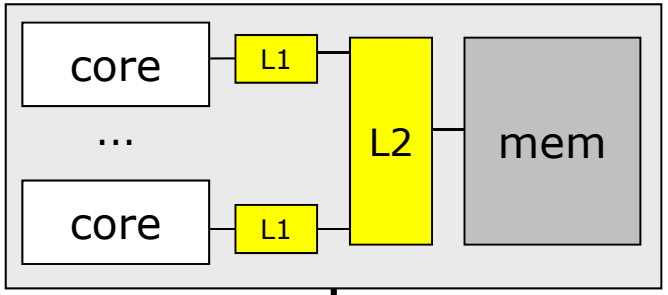
Sparse Sampler

Application Fingerprint

5, 3,...

Offline "Insight Technology"

Target Architecture

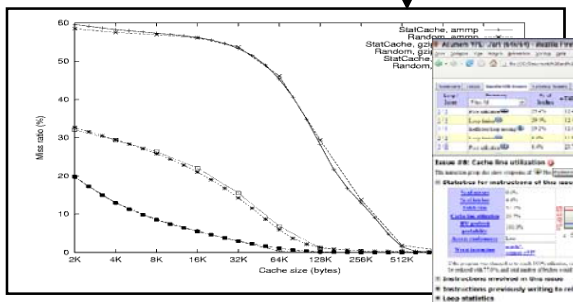


Architectural Parameters

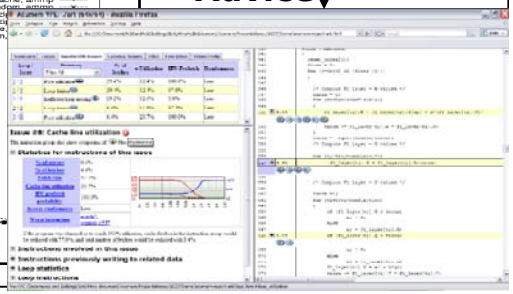
Probabilistic Cache Model

Acumem

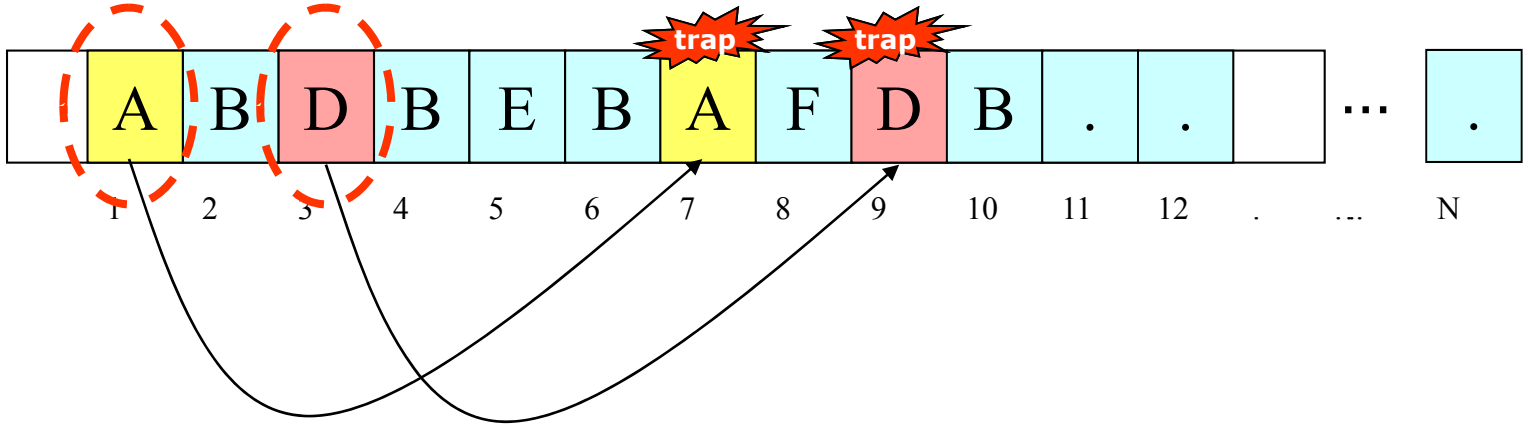
Modeled behavior



Advice



UART: Efficient sparse sampling



1. Use HW counter overflow to randomly select accesses to sample (e.g. ~on avergage every 1.000.000th access)

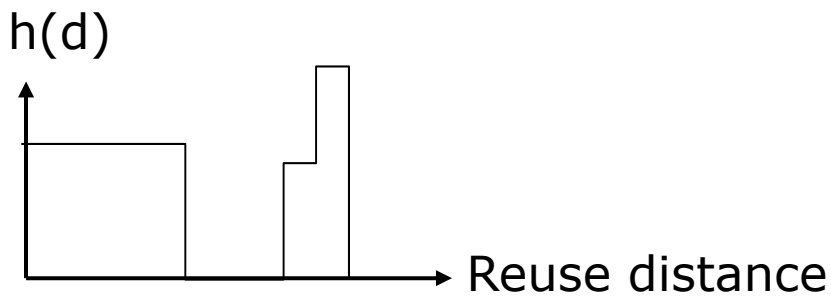
2. Set a watchpoint for the data cacheline they touch

3. Use HW counters to count #memory accesses until watchpoint trap

➔Sampling Overhead ~17% (10% at Acumem for long-running apps)

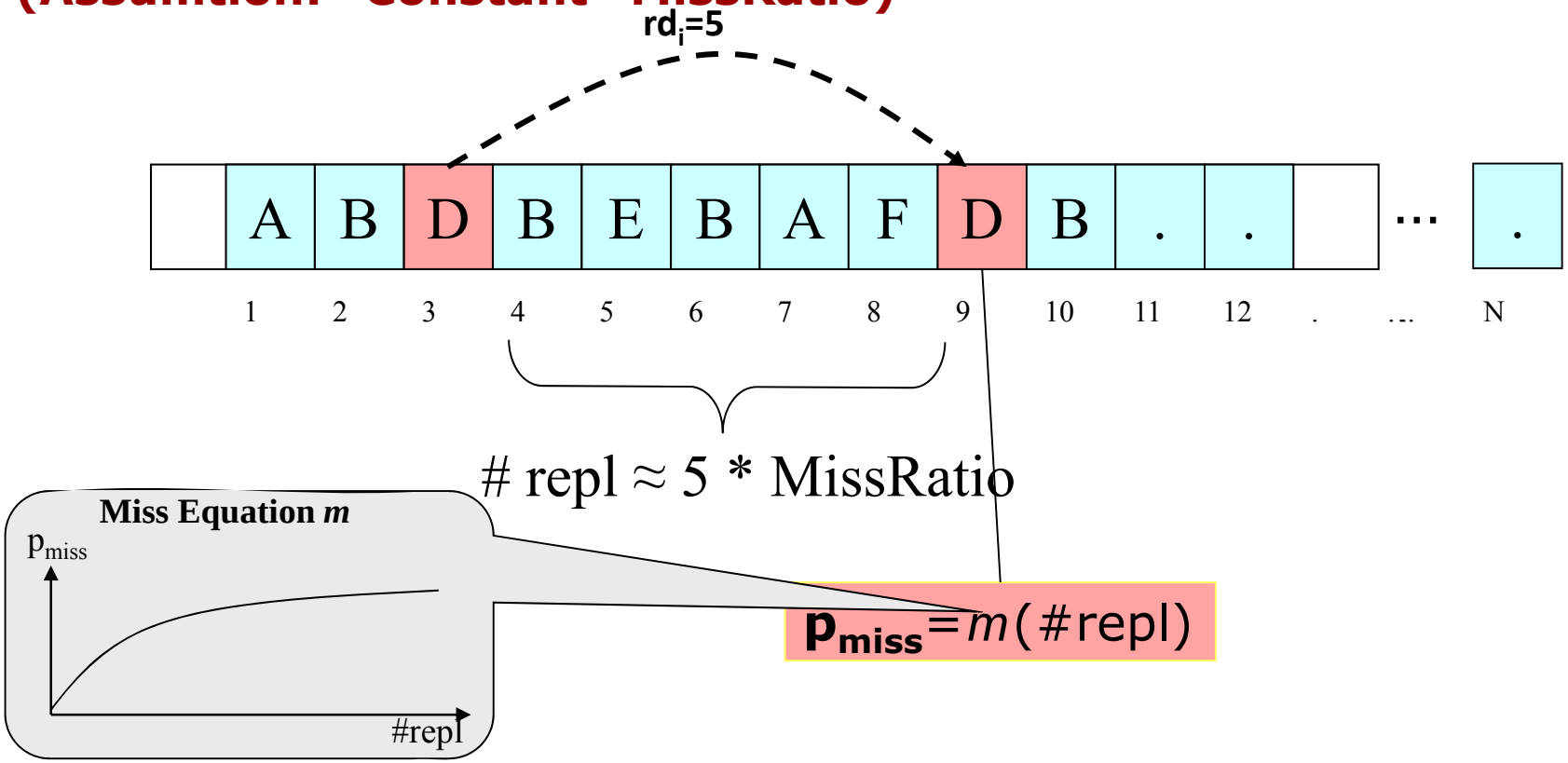
(Modeling with math < 100ms)

Fingerprint
≈ Sparse reuse distance histogram

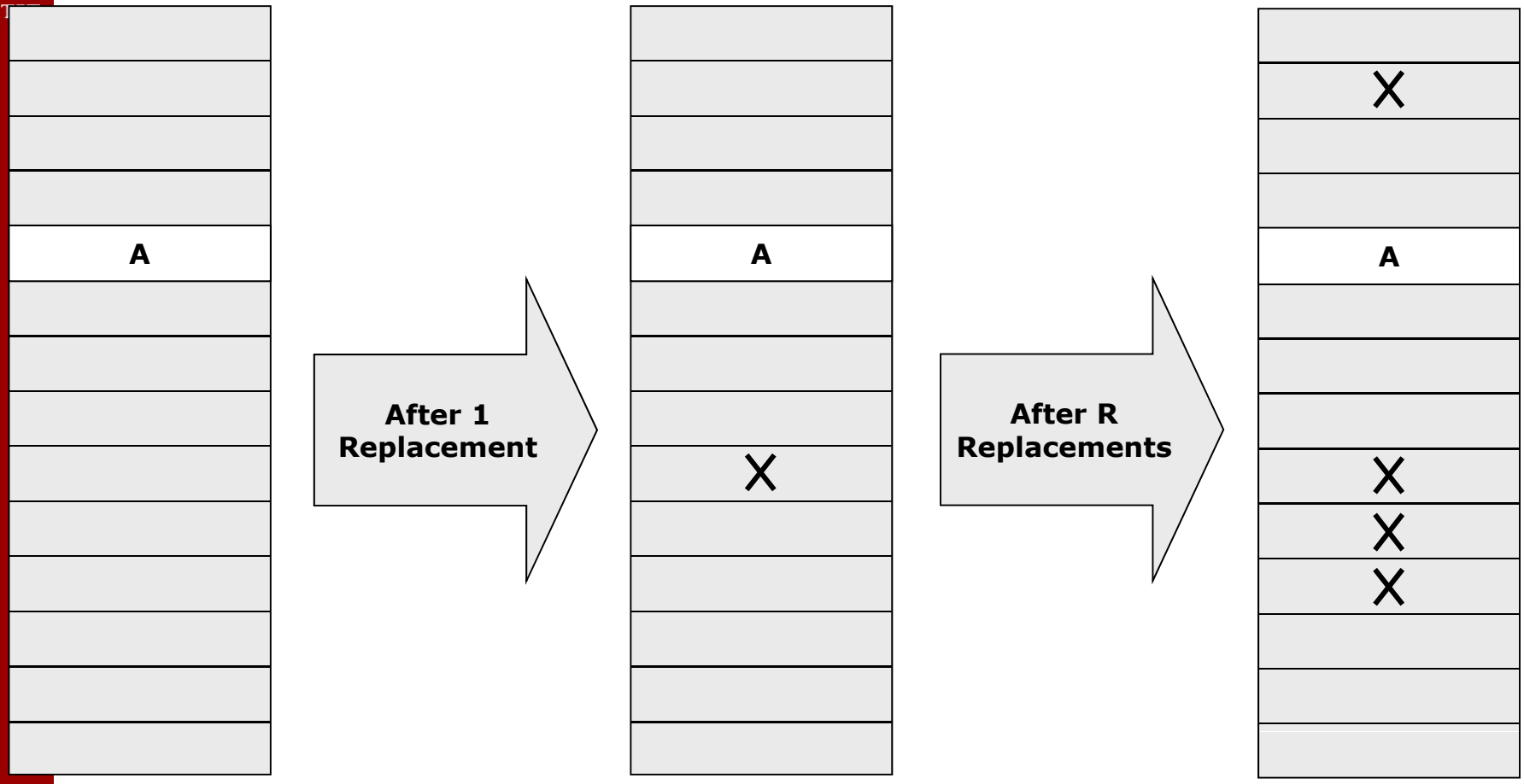


Modeling random caches with math

(Assumtion: "Constant" MissRatio)



Assuming a fully associative cache



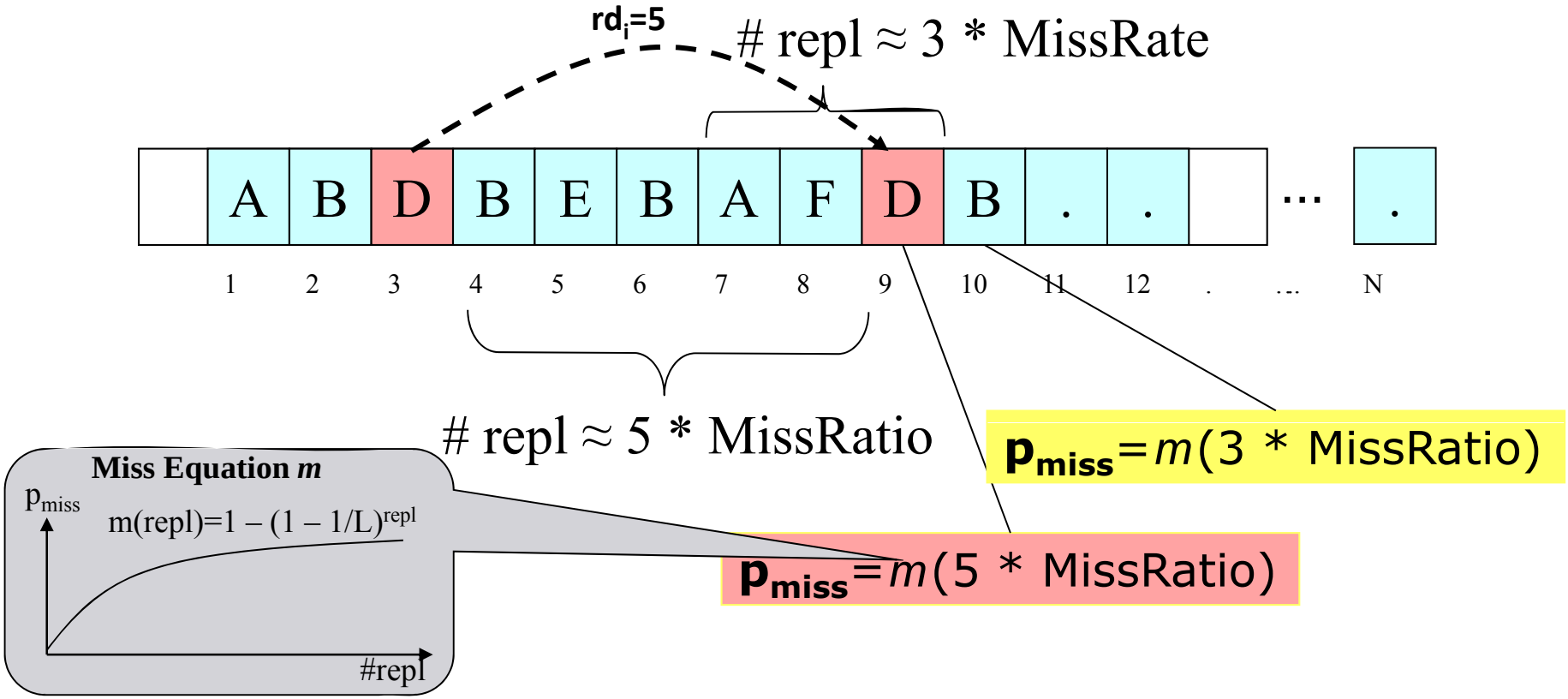
The cacheline A is in a cache with L cachelines

$(1 - 1/L)$ chance that A survives

$(1 - 1/L)^R$ chance that A survives

Modeling random caches with math

(Assumtion: "Constant" MissRatio)

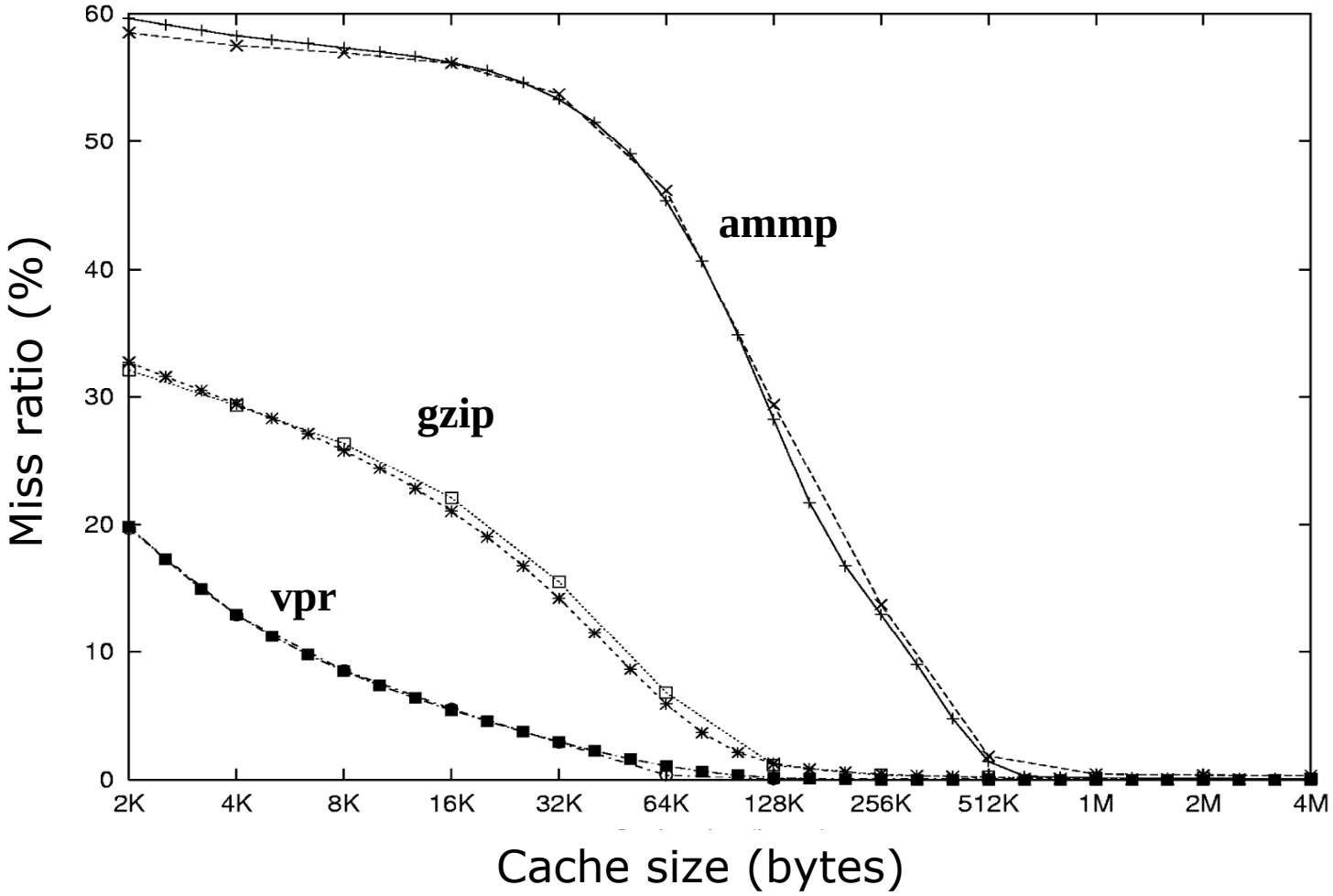


n samples: $\text{MissRatio} * n = \sum_{i=0}^{n} m(rd(i) * \text{MissRatio})$

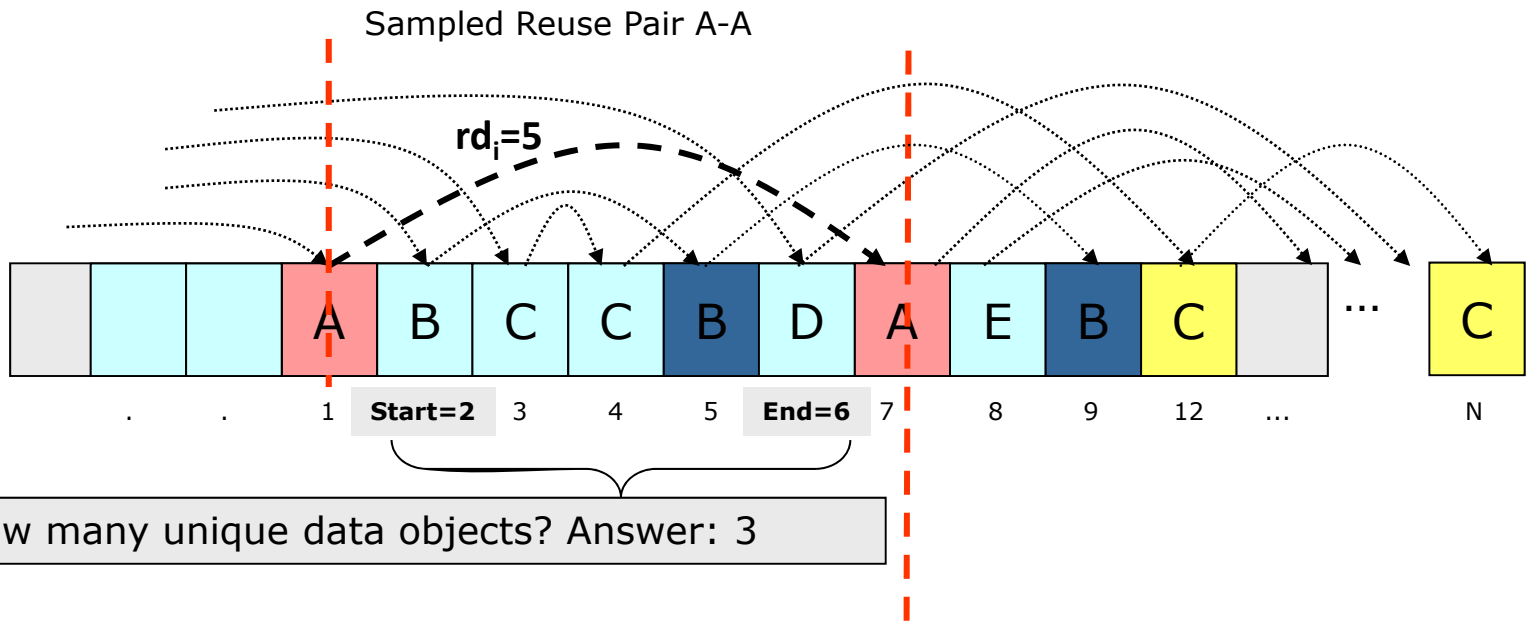
Can be solved in a "fraction of a second" for different L

Accuracy: Simulation vs. ”math” (Random replacement)

Comparing simulation (w/ slowdown 100x) and math (”fractions of a second”)



Modeling LRU Caches: Stack distance…



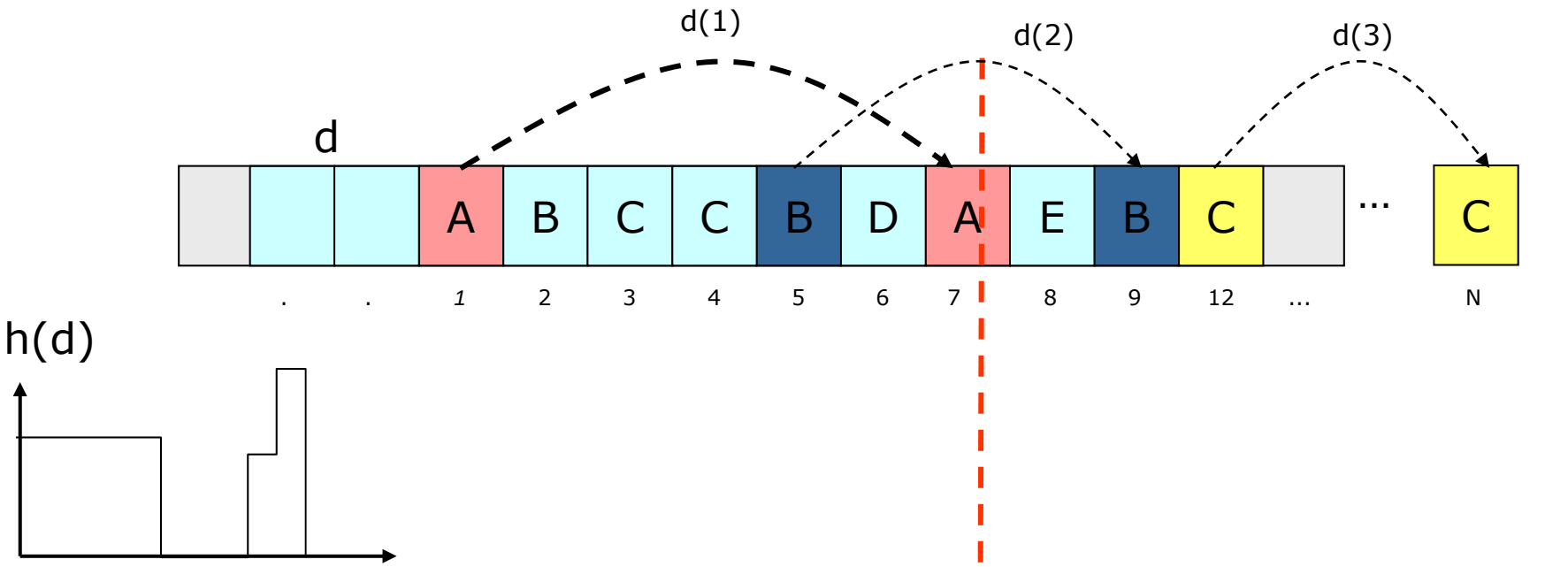
Stack Distance: How many unique data objects? Answer: 3

If we know all reuses: How many of the reuses 2-6 go beyond *End*? Answer: 3

$$\text{Stack\_distance} = \sum_{k=\text{Start}}^{\text{End}} [d(i) > (\text{End} - k + 2)]$$

<u>Foreach sample:</u> if (Stack_distance > L) miss++ else hit++

But we only know a few reuse distances…

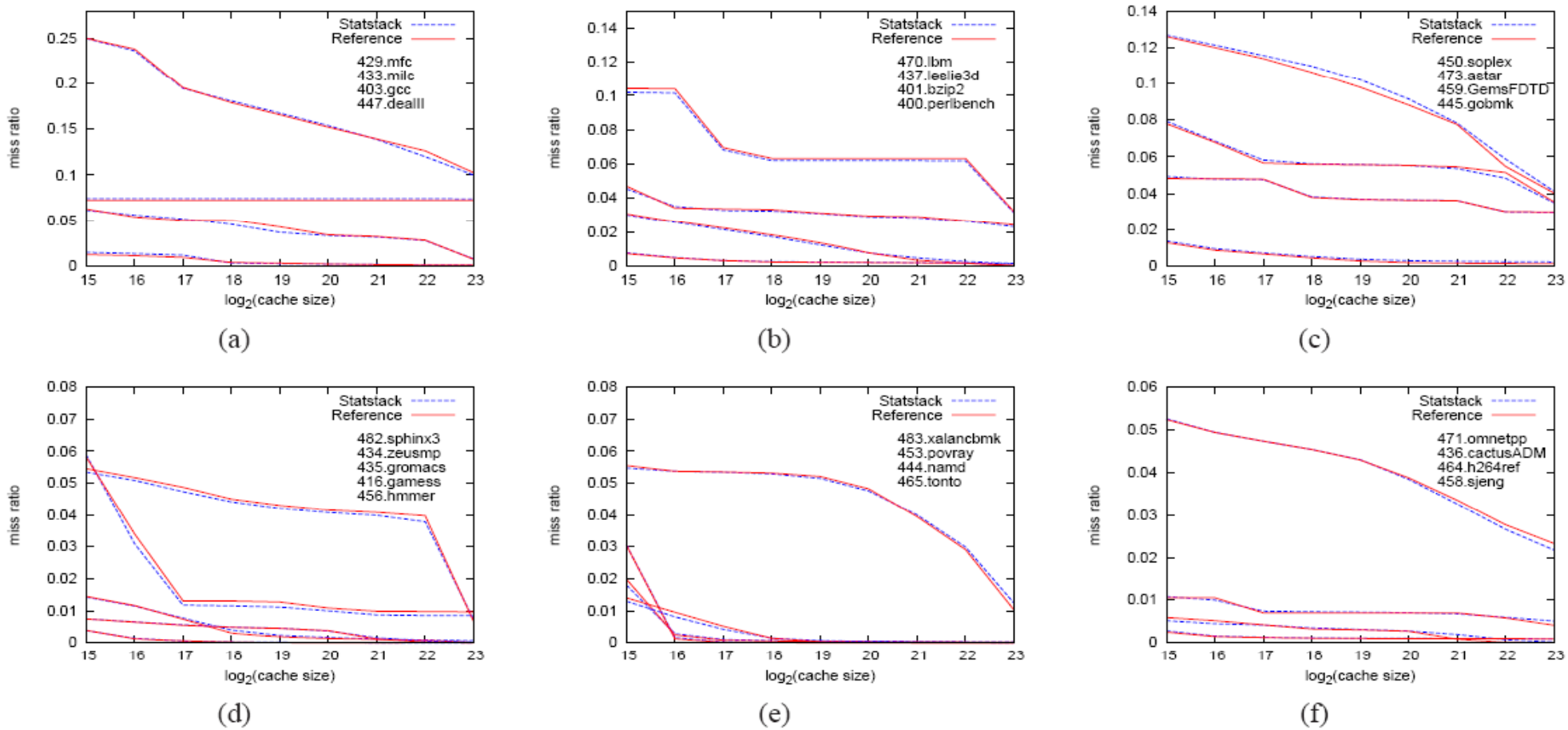


Estimate: How many of the reuses 2-6 go beyond *End*? Answer: Est_SD

Assume that the distribution **(aka histogram)** of sampled reuses is representative for all accesses in that "time window"

$$\text{Est\_SD} = \sum_{k=\text{Start}}^{\text{End}} \mathbf{p}[d(i) > (\text{End} - k)]$$

All SPEC 2006



Architecturally independent!

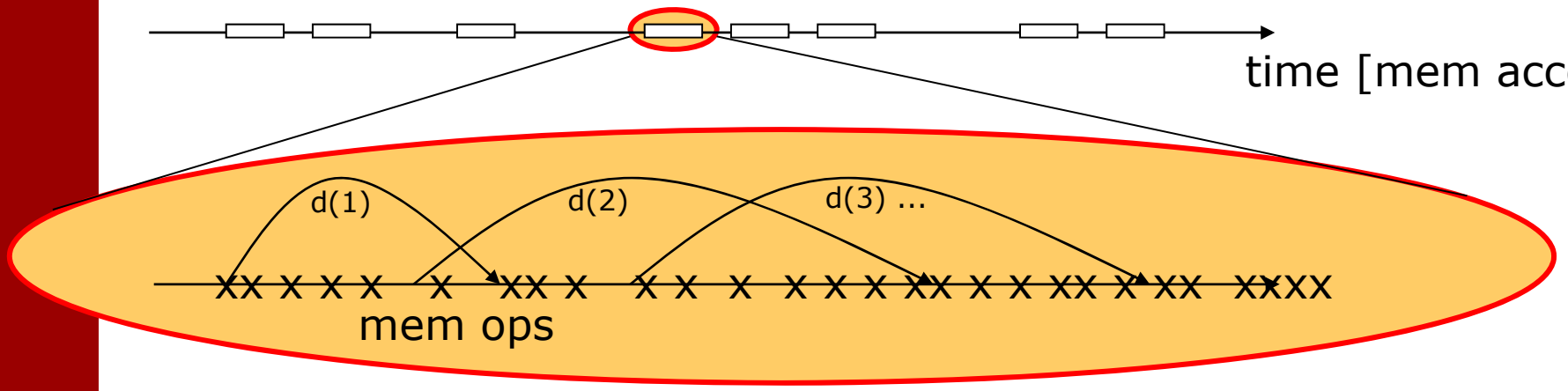
The fingerprint does not depend on the caches of the host architecture

Solve the equation for different targer architecture:

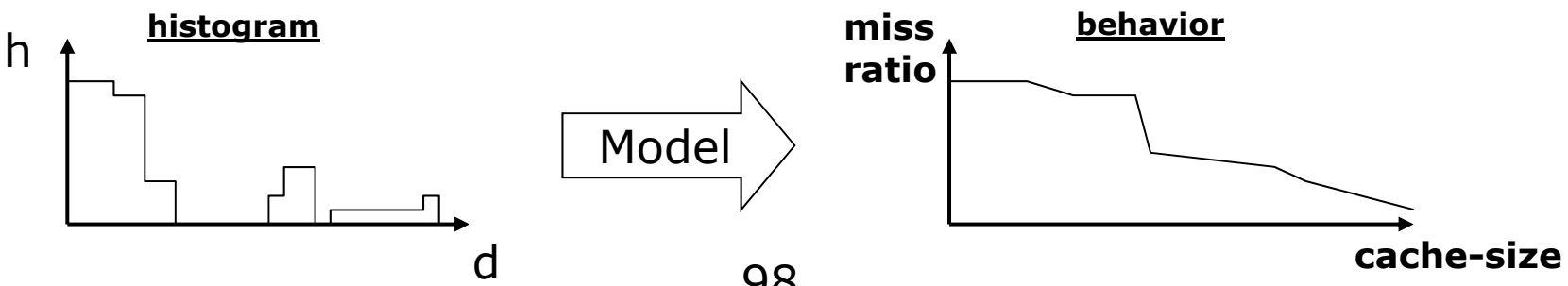
- Cache sizes
- Cacheline sizes
- Replacement algorithms {LRU, RND}
- Cache topology

In a nutshell

1. Sampling: Randomly select windows, and collect sparse reuse histograms from each window



2. Use histogram as input to model behavior of target arch.



Acumem Advice Heuristics

Sequential HW prefetching is modeled using math. Successful HW prefetches are only report in Bandwidth Issues.

Special heuristics have been designed to analyze the fingerprint to also tell the reason for cache misses