Motivation for Caching and Optimization of Cache Utilization

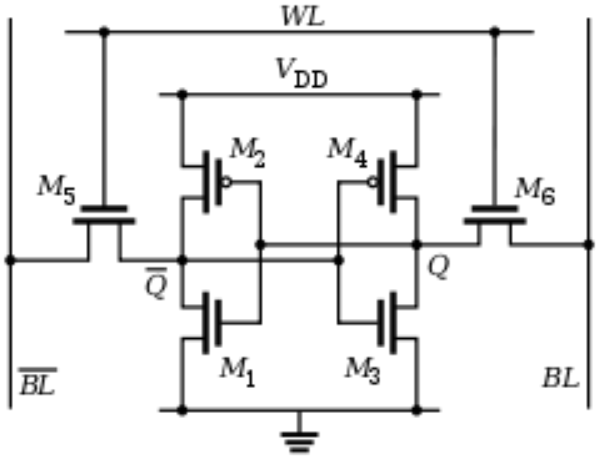
Agenda

- Memory Technologies
- Bandwidth Limitations
- Cache Organization
- Prefetching/Replacement
- More Complexity by Coherence
- Performance Optimization
- Examples
  - Partially used structure
  - Alignment problem
  - Inefficient loop nesting
  - False sharing
- Questions

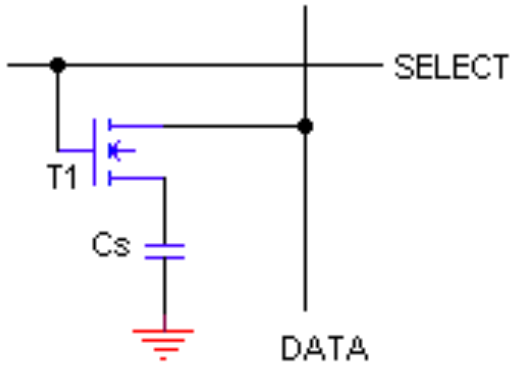
Memory Technologies

SRAM vs. DRAM

SRAM Cell



DRAM Cell



~140 F²

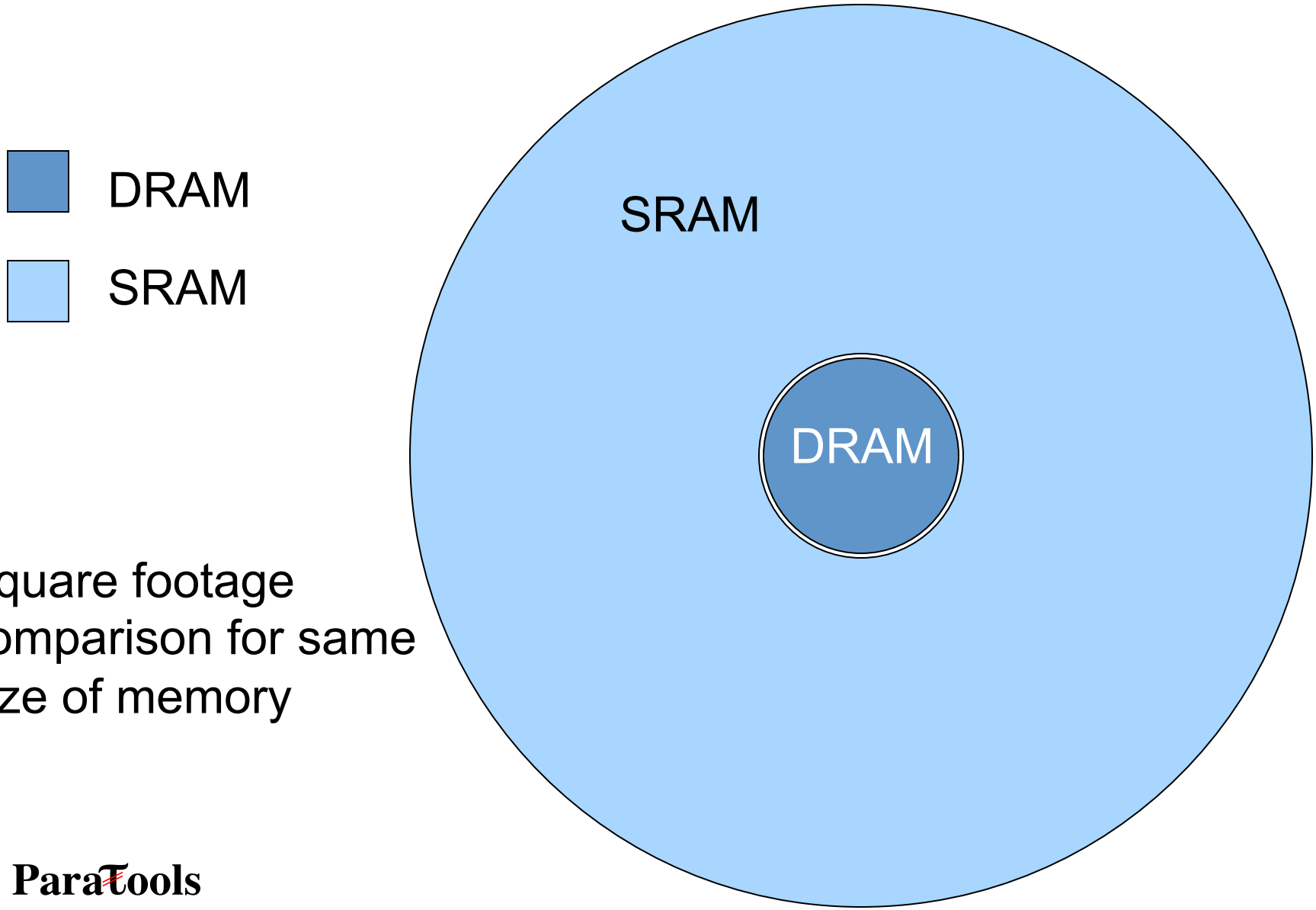
Access Time: < 0.1 ns

~ 6 – 10 F²

Access Time: < 7 ns

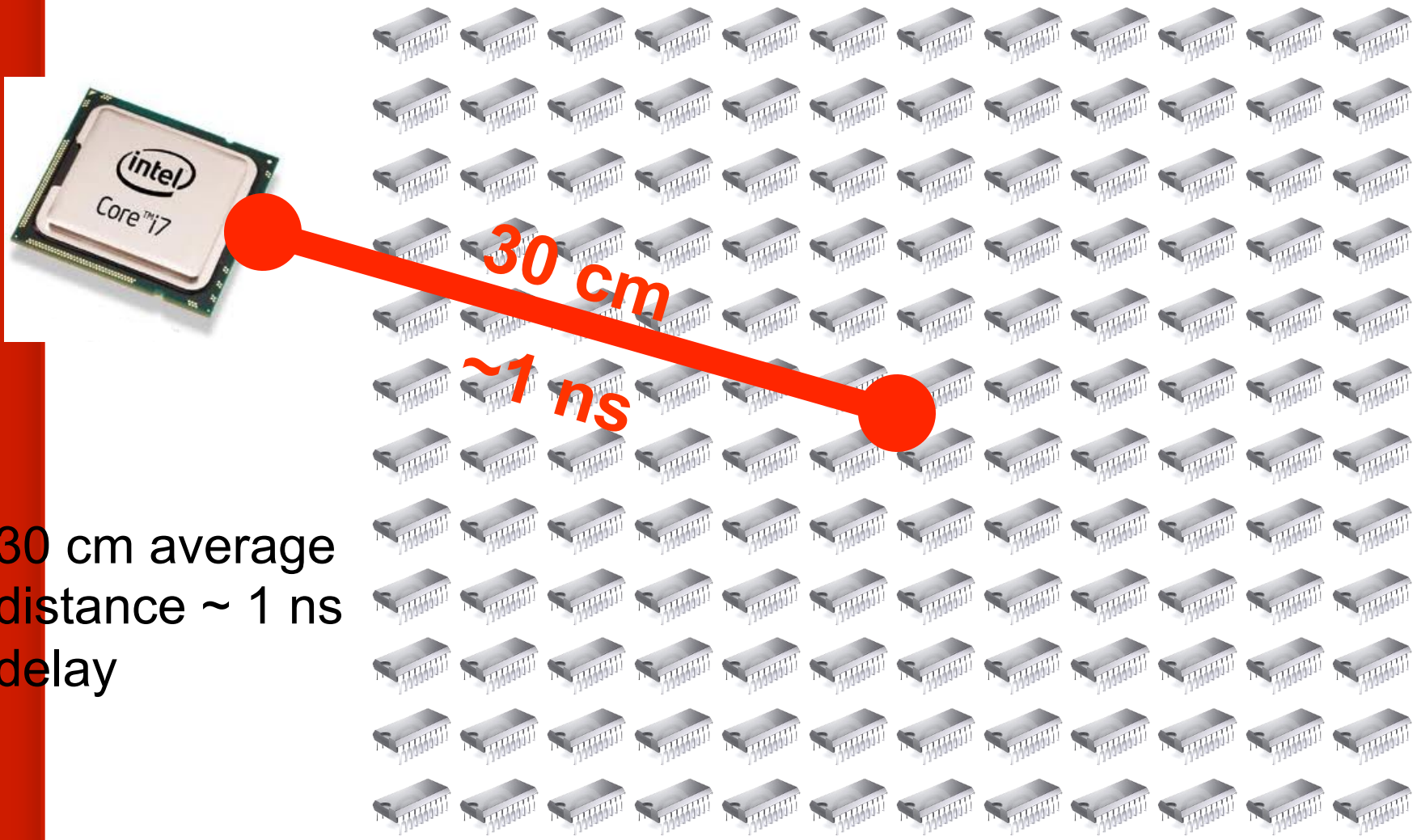
ParaTools

SRAM vs. DRAM



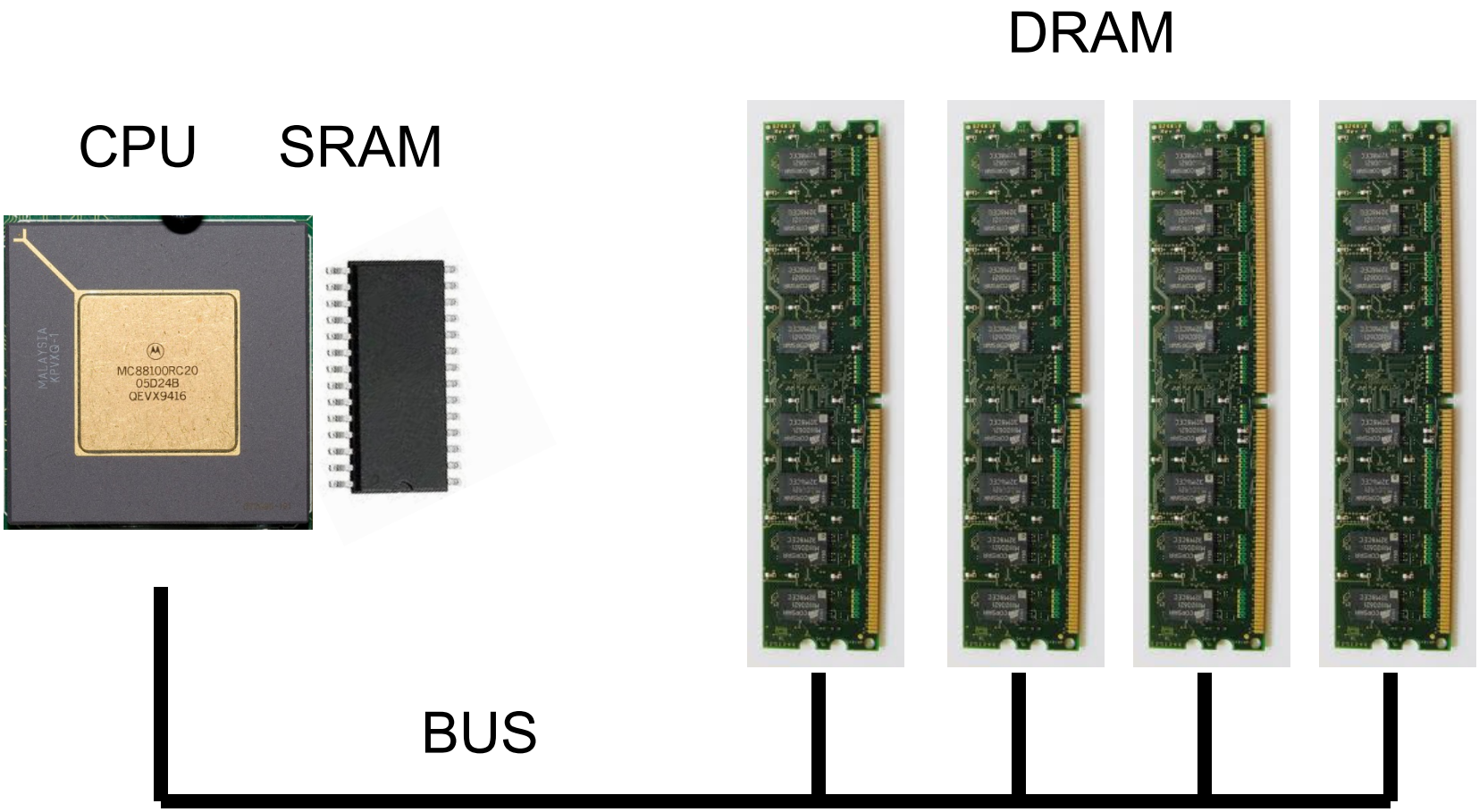
Square footage comparison for same size of memory

SRAM Technology for Main Memory?



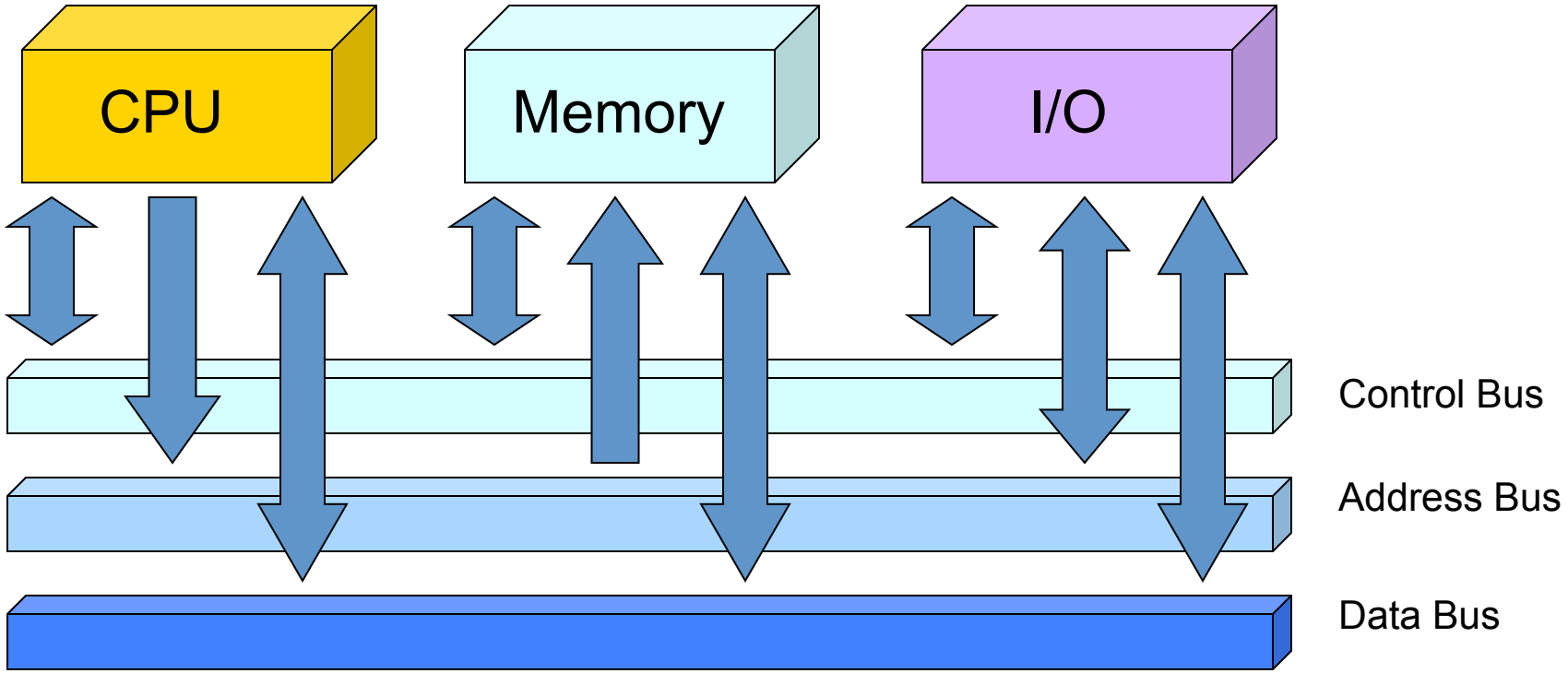
30 cm average distance ~ 1 ns delay

Use of SRAMs

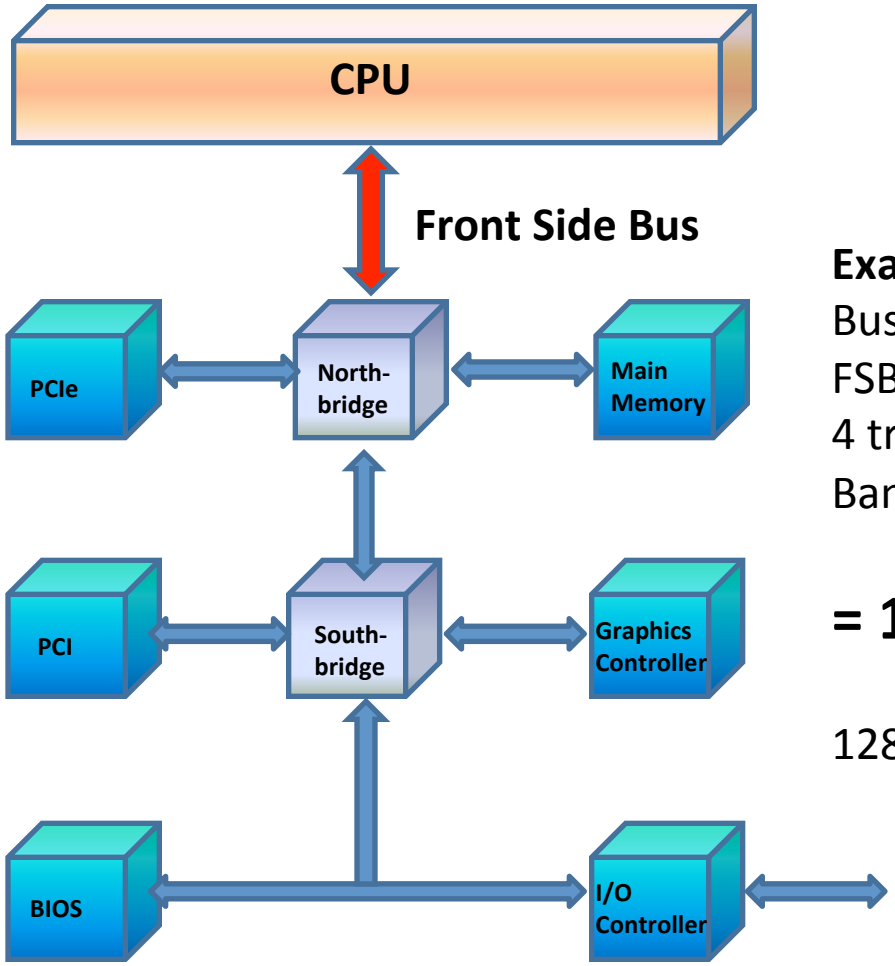


Bandwidth Limitations

Von-Neumann-Bottleneck



Bandwidth Front Side Bus (FSB)



Example:
Bus Width: 64 bit (8 byte)
FSB Clock: 400 MHz
4 transfers/cycle
Bandwidth: 8 byte x 400 MHz x 4 transfers/cycle

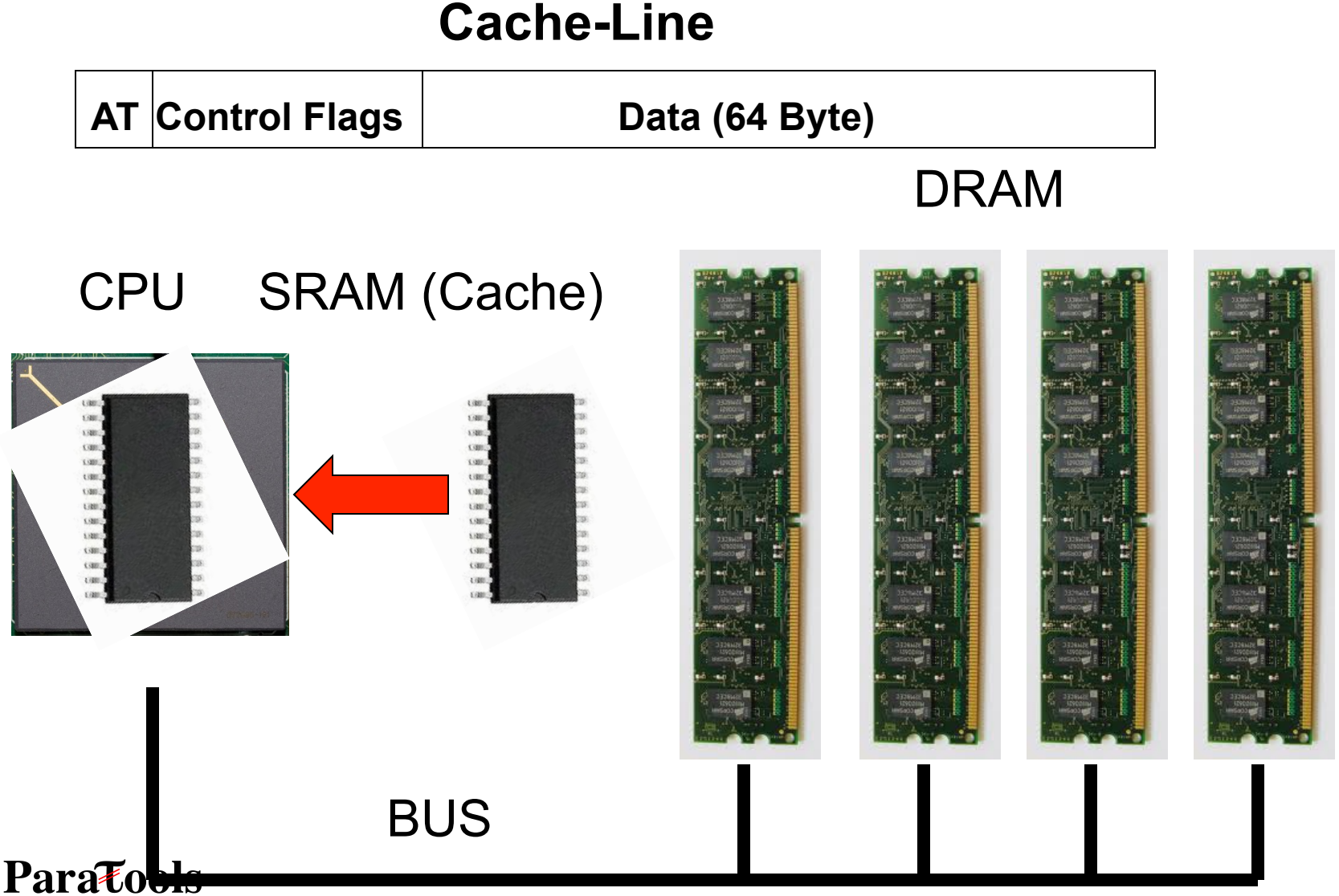
= 12800 MB/s

12800 MB/s / 8 cores = 1600 MB/s

Caching

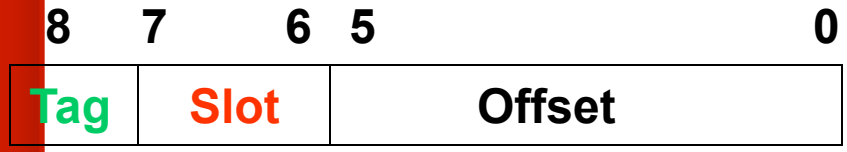
Cache-Line

AT	Control Flags	Data (64 Byte)



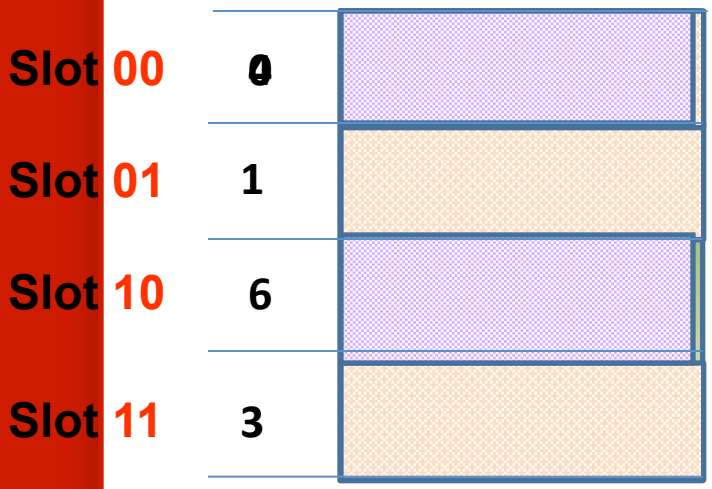
Cache Organization

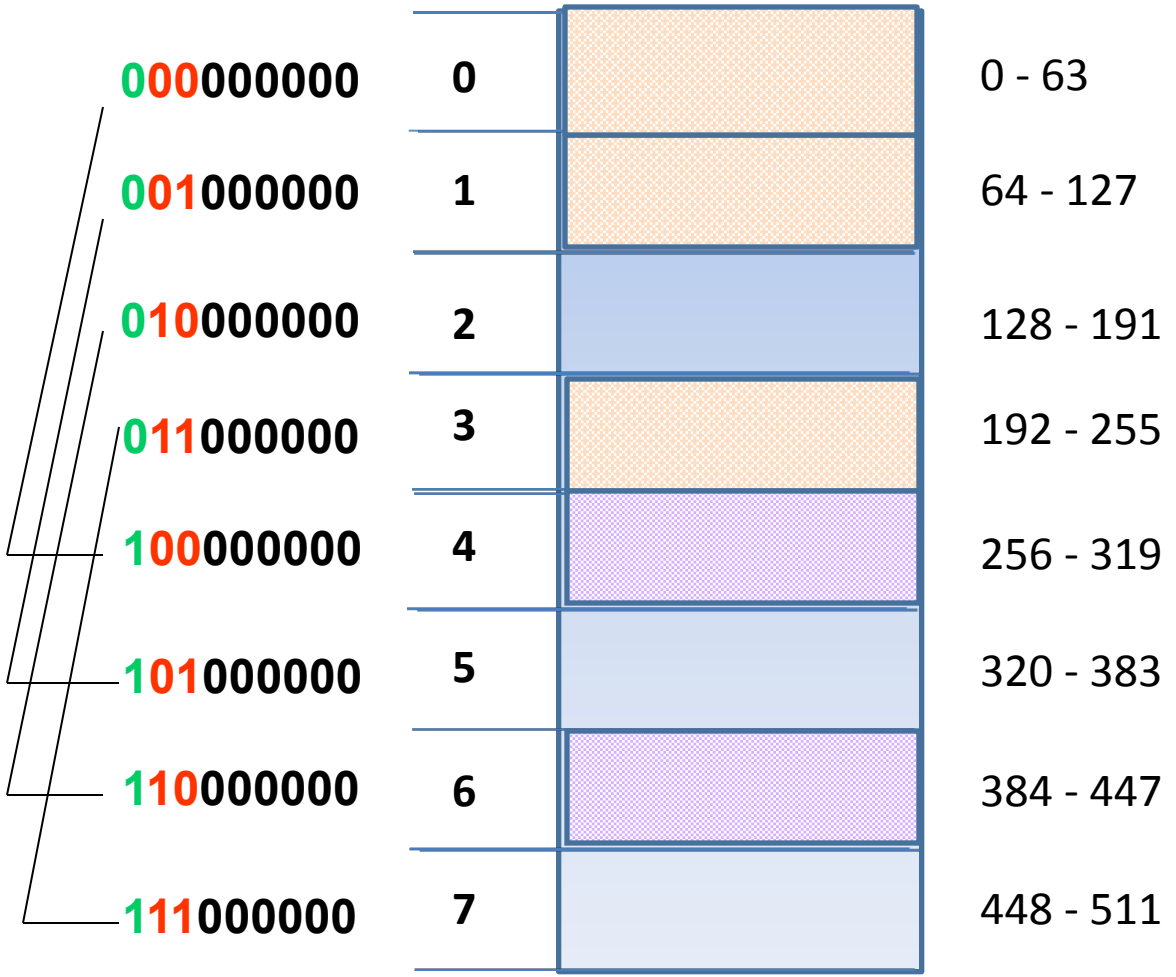
Direct Mapped Cache (DM)



Main Memory 512 byte

Cache 256 byte
(4 Cache-Lines 64 byte)



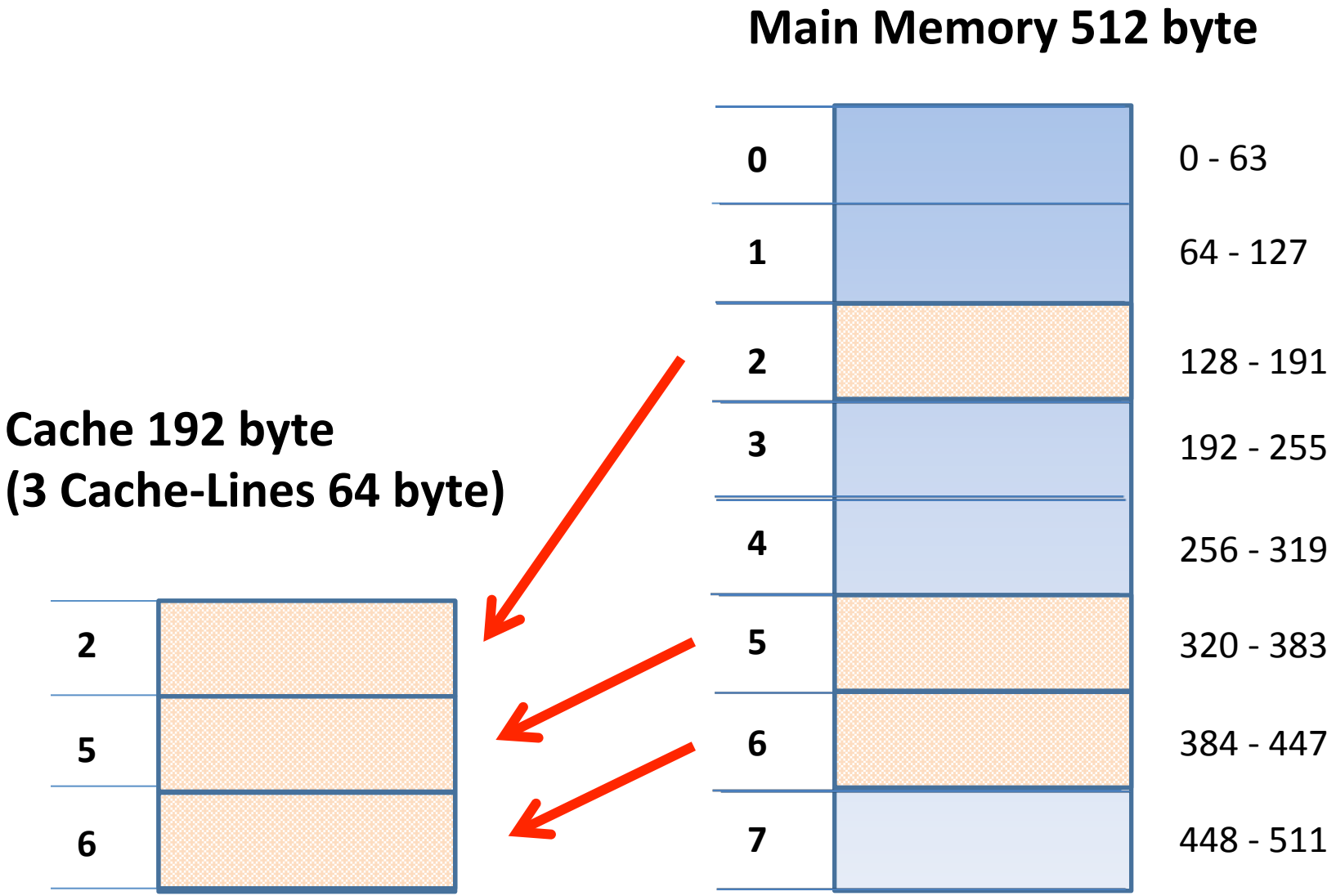


ParaTools

Analogy Direct Mapped Cache (DM)

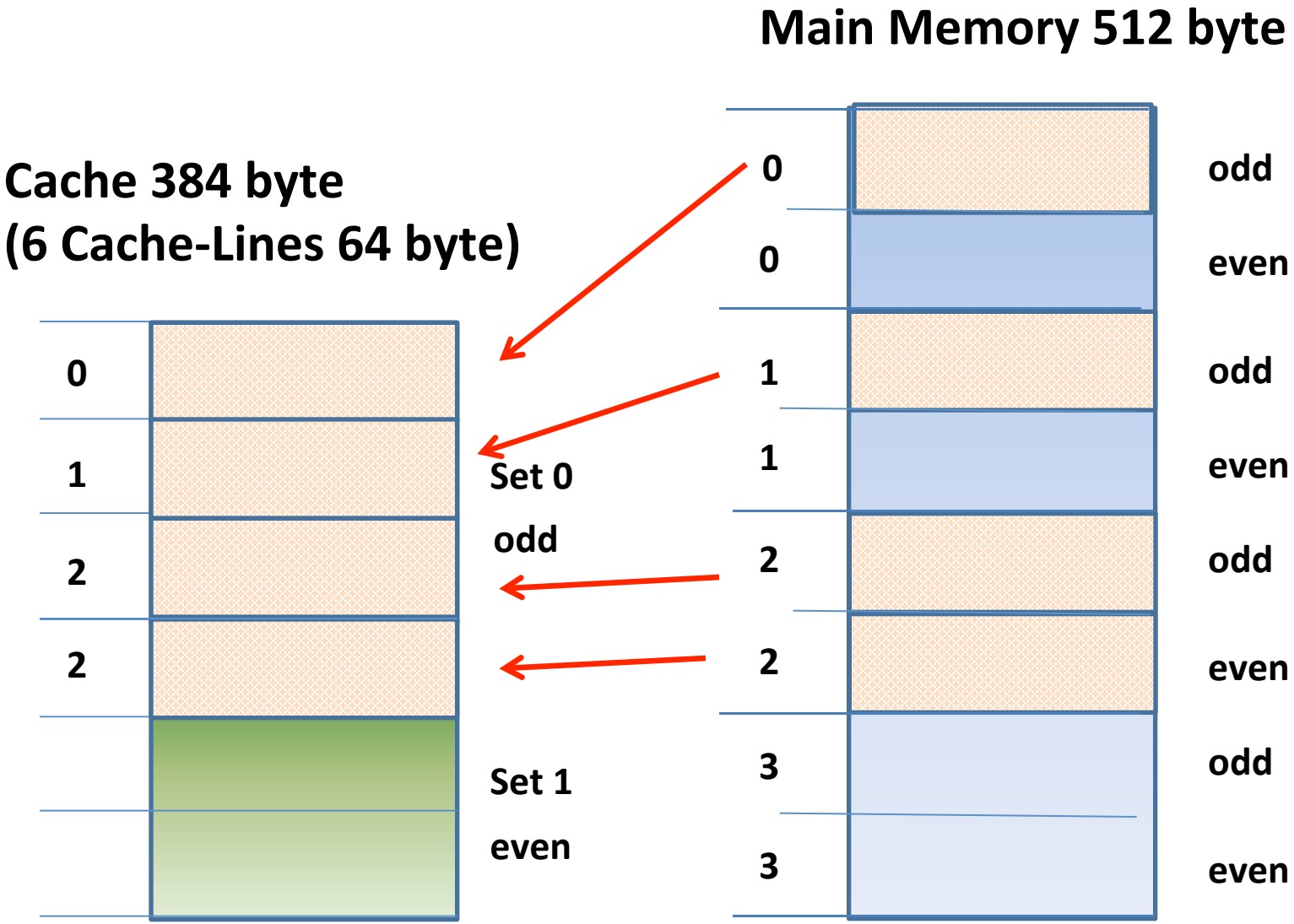
P

Fully Associative Cache (FA)



Analogy Fully Associative Cache (FA)

Set Associative Cache (SA)



ParaTools

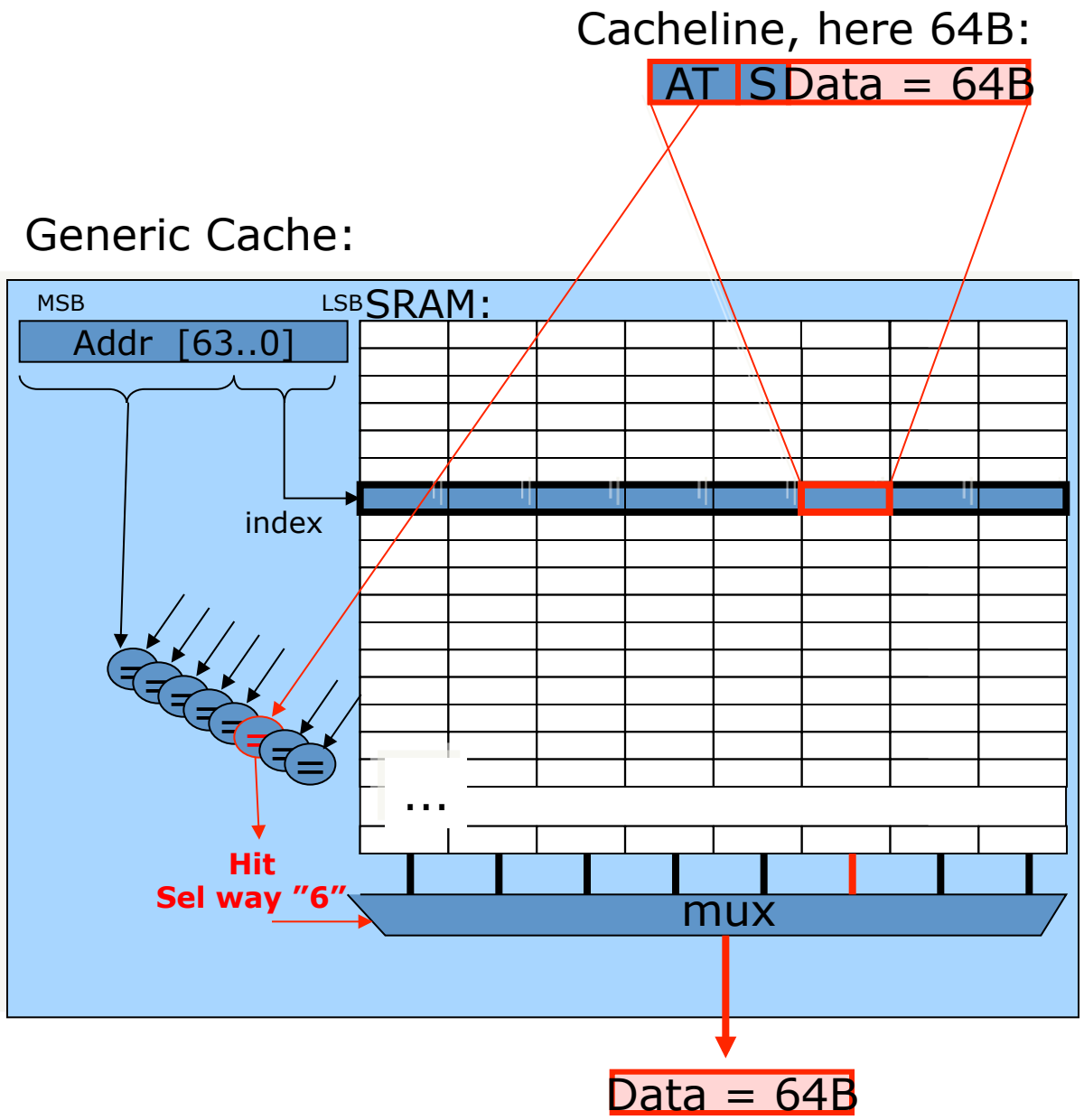
Analogy Set Associative Cache (SA)

Cache implementation

L3 € 24MB

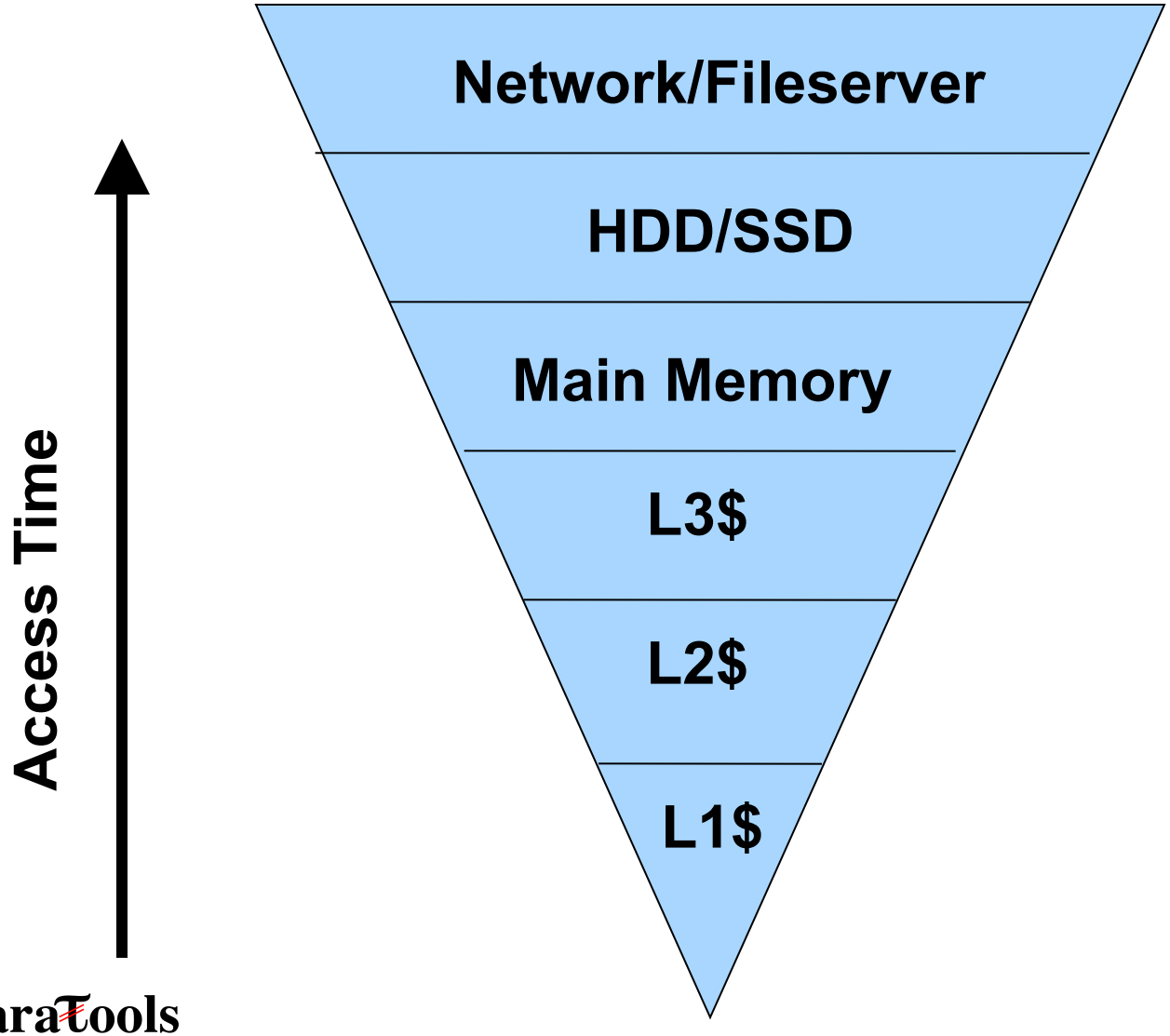
L2 $ 256kB

D$ 64kB

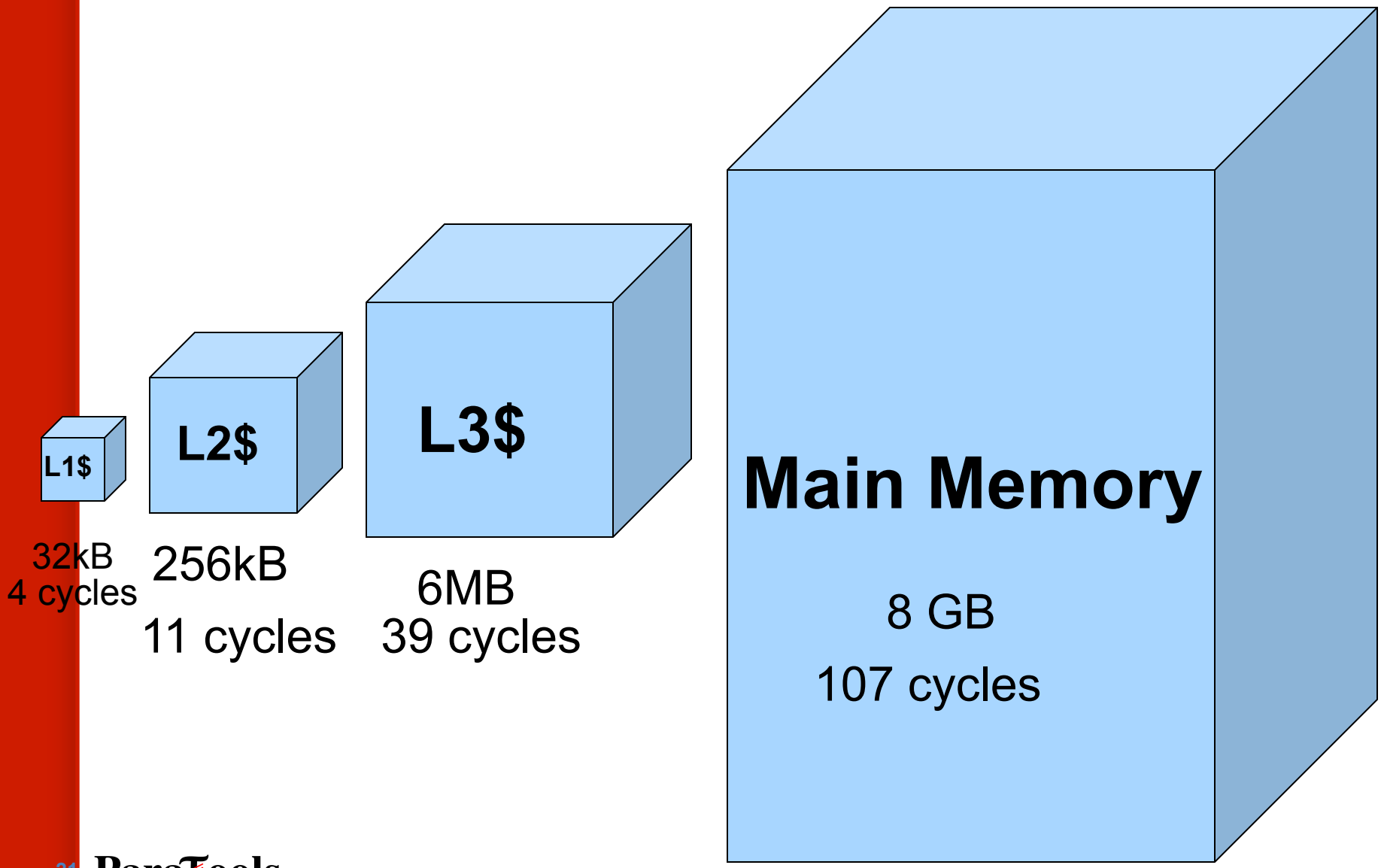


ParaTools

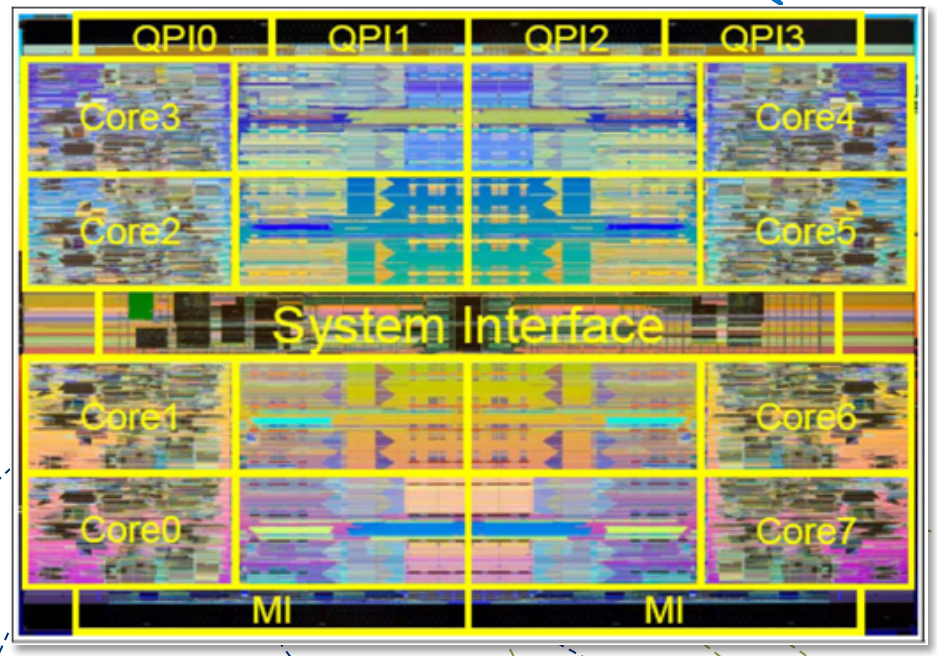
Access Time vs. Size

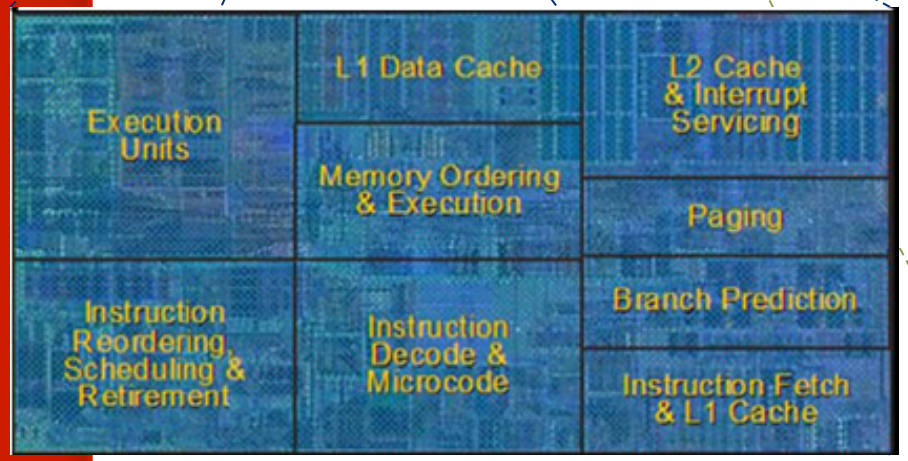


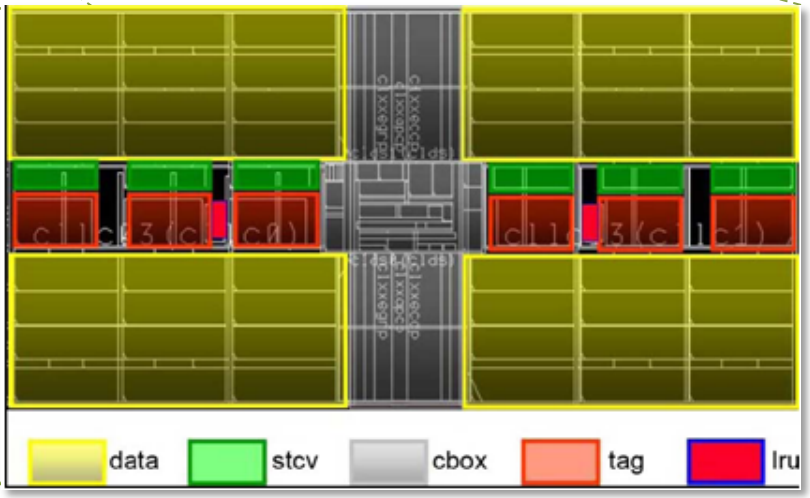
Cache Level Latency



How is the silicon used (i7-Ex)?

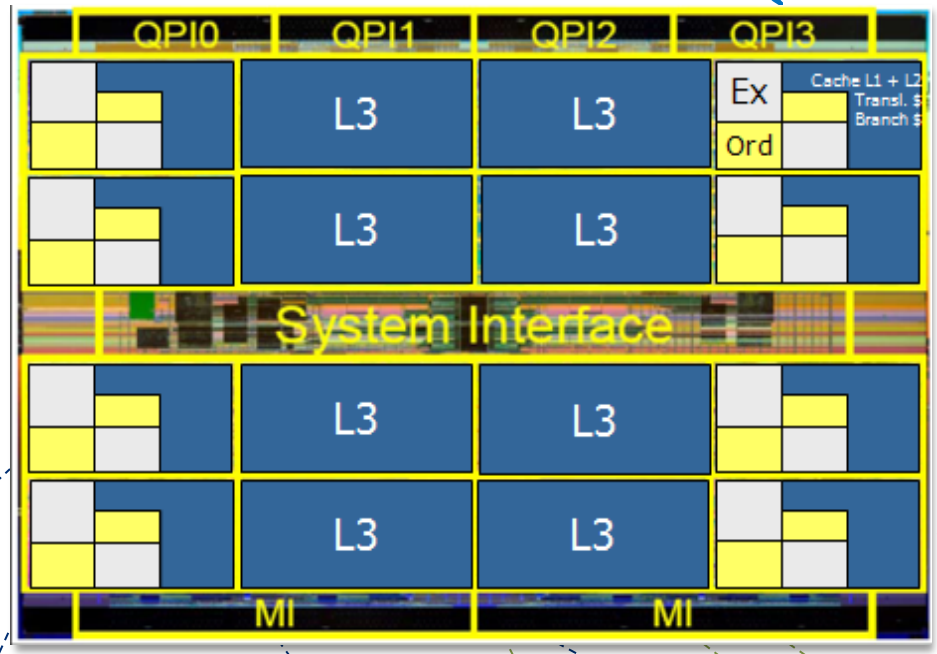


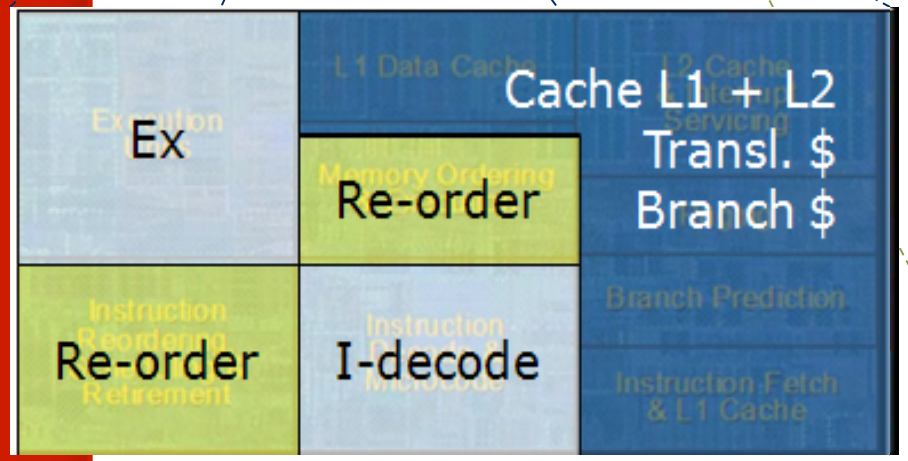


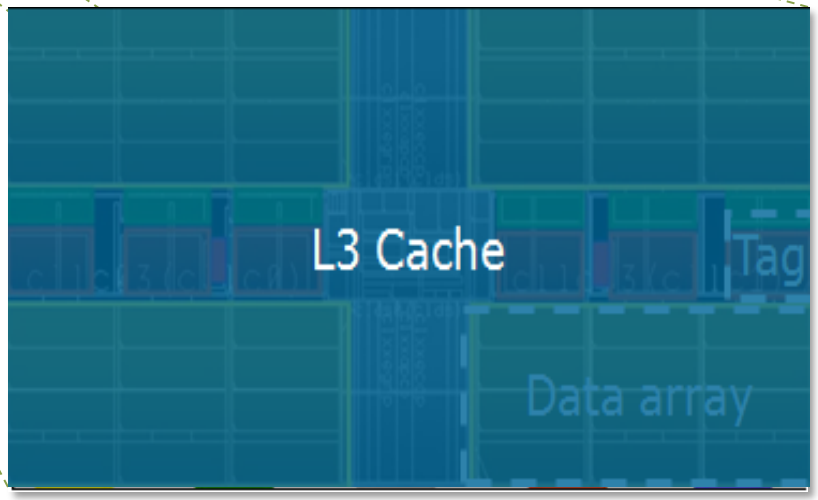


ParaTools

How is the silicon used (i7-Ex)?







ParaTools

Prefetching/Replacement

HW Prefetching

The prefetcher anticipates application's next memory access and prefetches the data to the cache

Sequential prefetching: Sequential streams [to a page]. Some number of prefetch streams supported. Often only for L2 and L3.

PC-based prefetching: Detects strides from the same PC. Often also for L1.

Adjacent prefetching: On a miss, also bring in the "next" cache line. Often only for L2 and L3.

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)

Replacement Policies

- **LRU** (Least Recently Used):
Evicts the cache line that was least recently used

- **Random** replacement:
A random cache line is selected for eviction

- **FIFO** (First In First Out):
The oldest entry will be removed

- **LFU** (Least Frequently Used):
The least requested cache line will be evicted

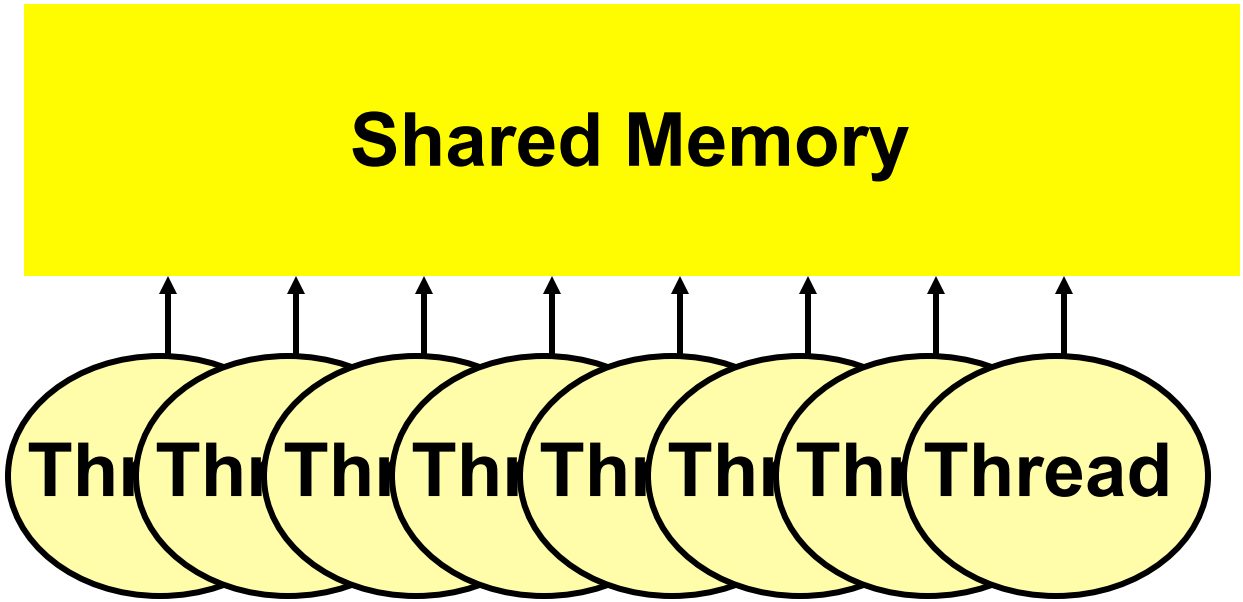
Analogy Office

Replace

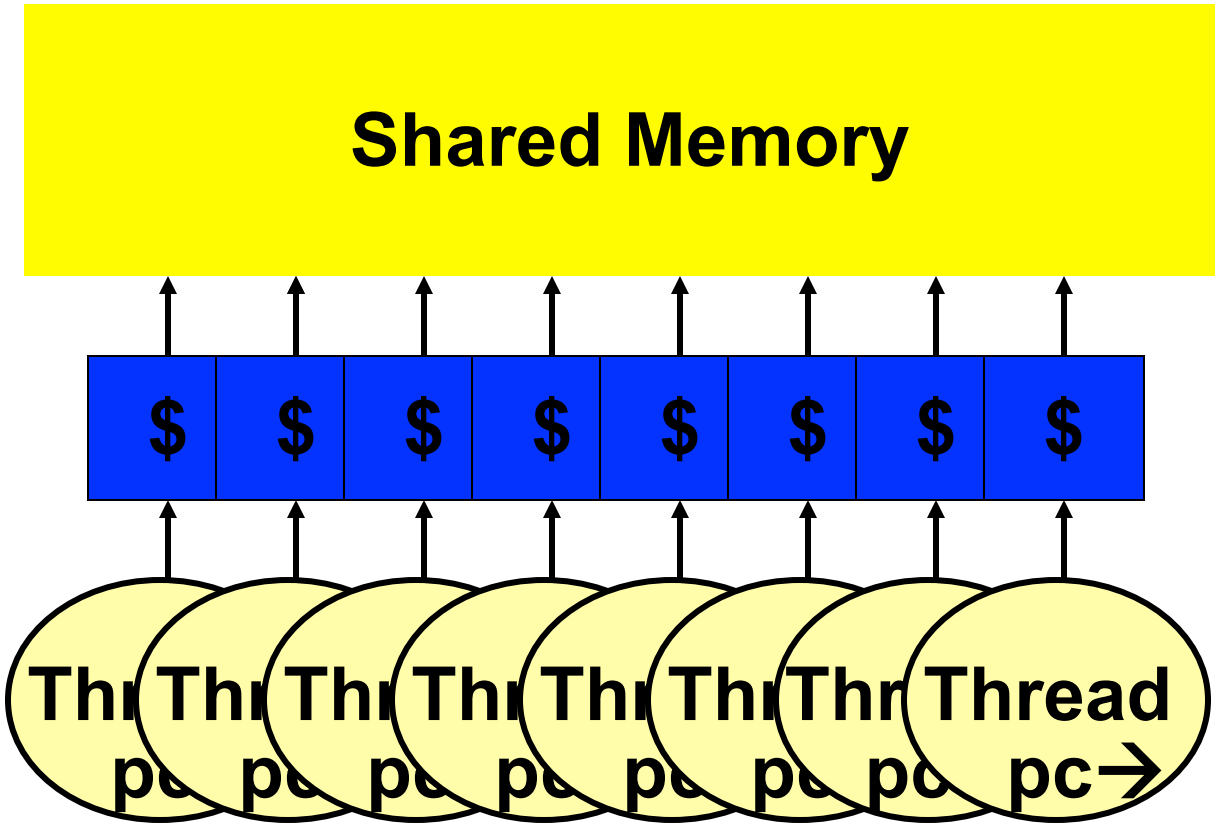

Archive
Access Time > 2 min

More Complexity by Coherence

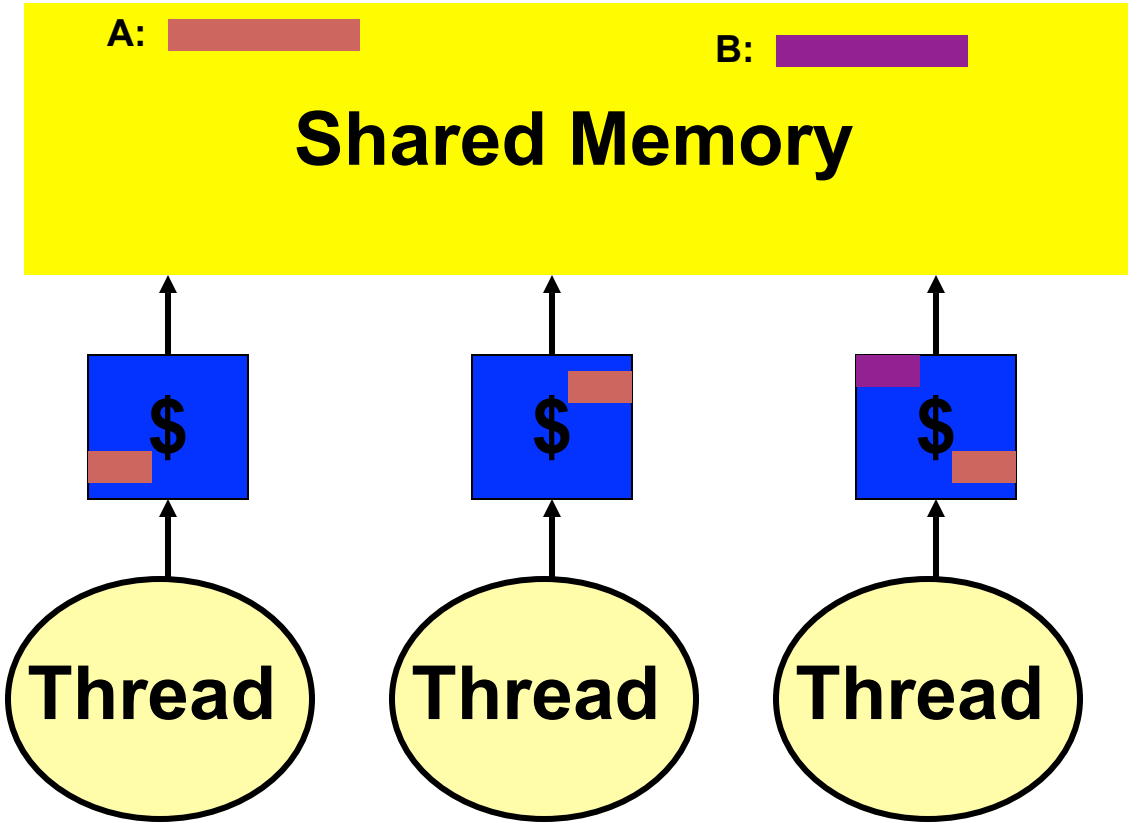
Programming Model:



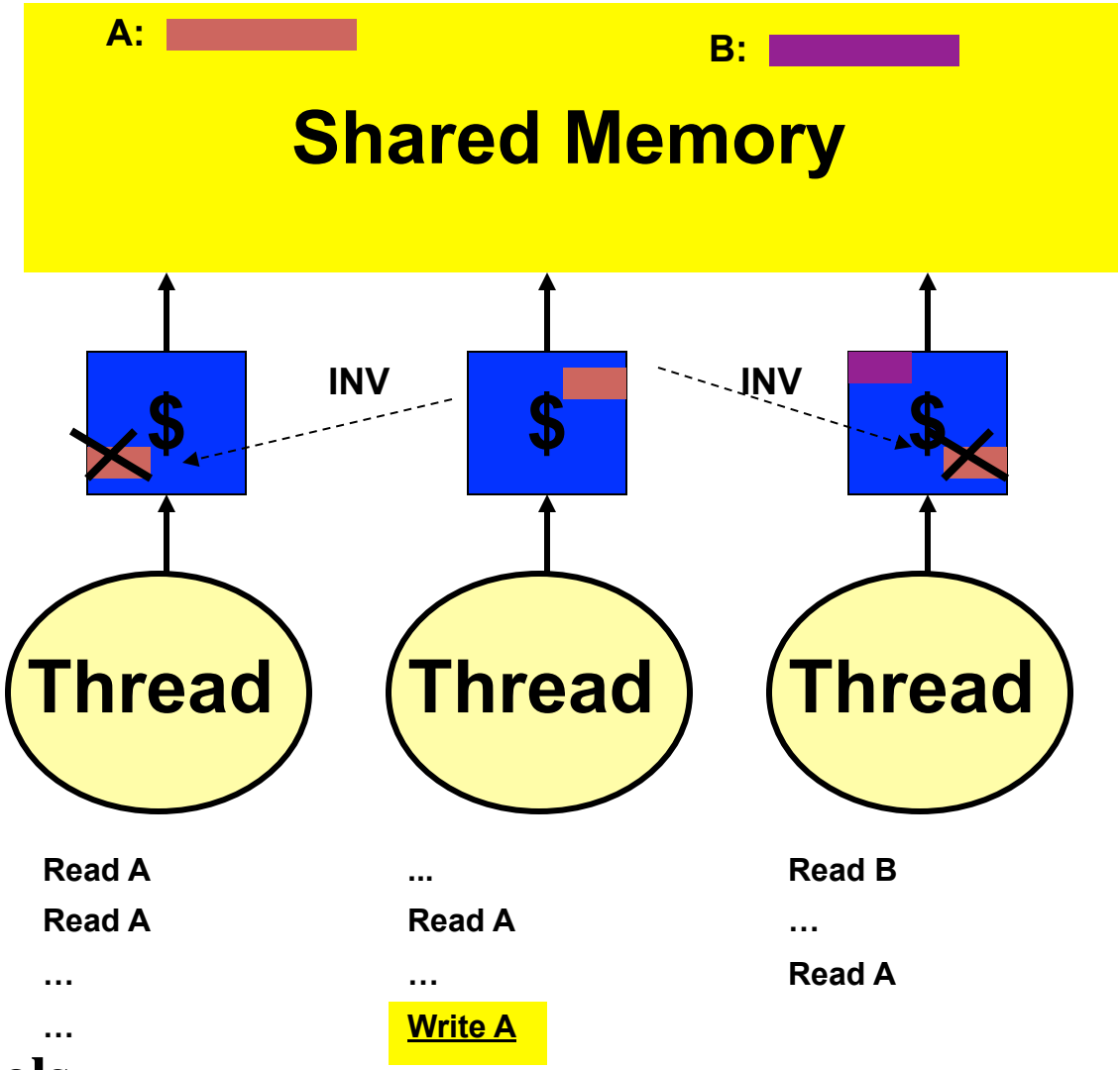
Adding Caches: More Concurrency



Caches:
Automatic Replication of Data

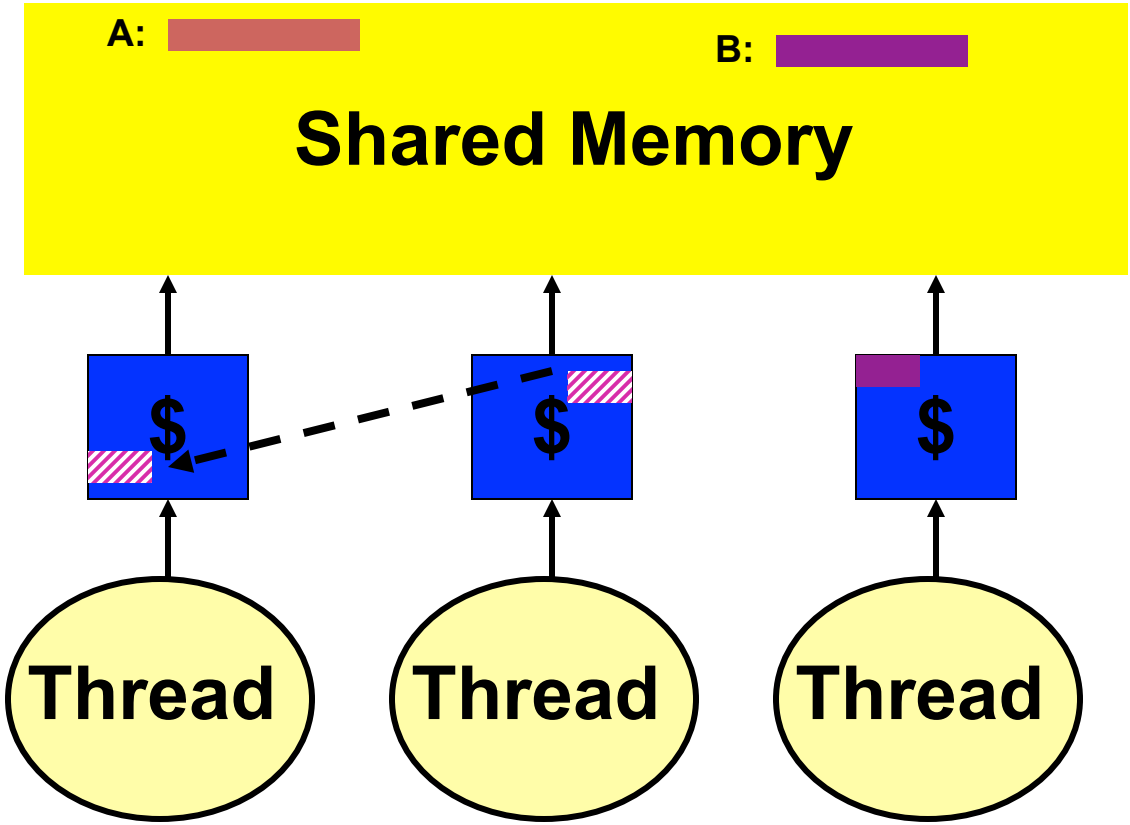


The Cache Coherent Memory System

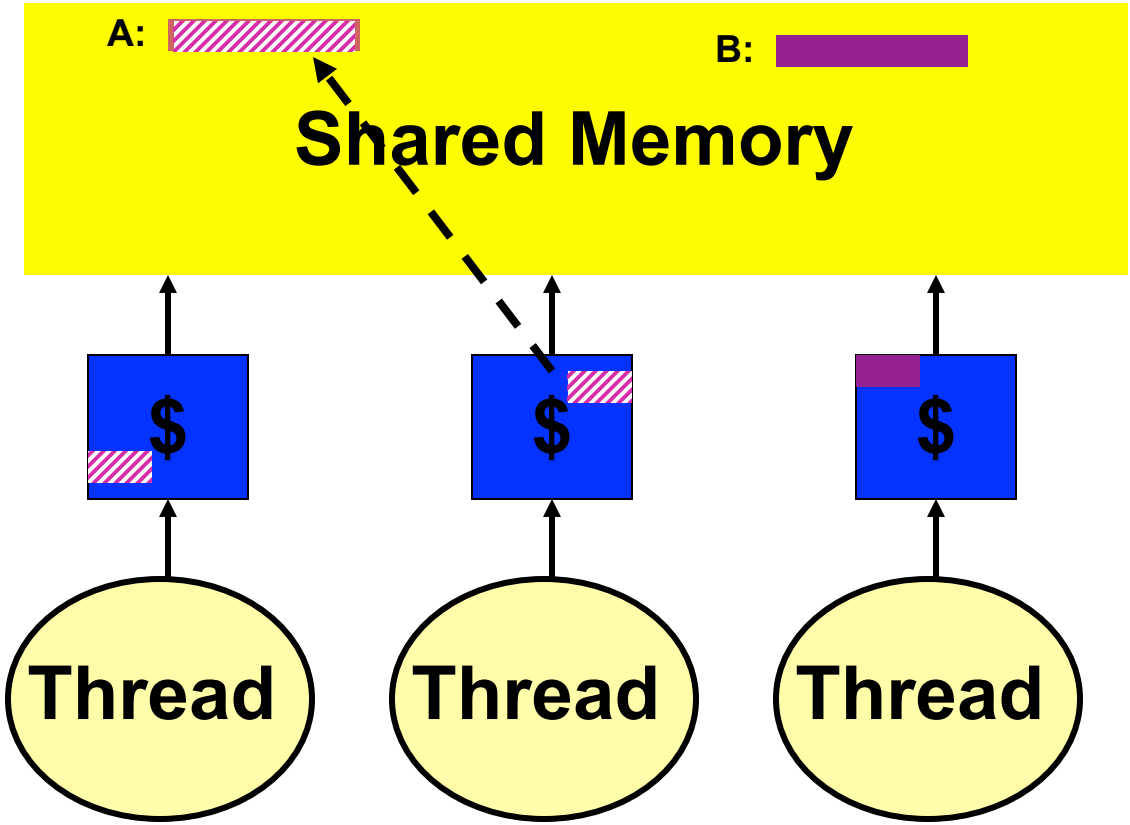


ParaTools

The Cache Coherent 2



Writeback


A:
B:
Shared Memory
$
$
$
Thread
Thread
Thread
Read A
Read A
…
…
Read A
…
Read A
…
Write A …
A gets replaced
Read B
…
Read A

Cache Line States in the MESI Protocol

M – Modified:
The data in the cache line is modified and is guaranteed to only reside in this cache. The copy in main memory is not up to date, so when the cache line leaves the modified state the data must be written back to main memory.

E – Exclusive:
The data in the cache line is unmodified, but is guaranteed to only reside in this cache.

S – Shared:
The data in the cache line is unmodified, and there may also be copies of it in other caches.

I – Invalid:
The cache line does not contain valid data.

Performance Optimization

Optimizing for Cache Performance

Keep the active footprint small

Try to let the processor fetch as much data as possible from cache and not from main memory (assist the prefetcher, don't confuse it!)

Try to ensure that every fetched cache line will consist of 100% data which actually will be used

Let the thread do the job which has the required data already stored in its private cache

...

What is the potential gain of optimizing cache usage?

Latency difference L1$ and mem: ~25-50x

Bandwidth difference L1$ and mem: ~20x

Repeated TLB misses adds a factor ~2-3x

Execute from L1$ instead from mem
==> 50-150x improvement

At least a factor 2-4x is within reach

Cache Lingo

Miss ratio: **What is the likelihood that a memory access will miss in a cache?**

Fetch ratio: **What is the likelihood that a memory access will cause a fetch from main memory [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 "ParaTools ThreadSpotter-ish" language**

Examples

Partially Used Structure

```
struct DATA
{
    int a;
    int b;
    int c;
    int d;
};

DATA * pMyData;
pMyData = new DATA[10*1024*1024];

for (long i=0; i<10*1024*1024; i++)
{
    pMyData[i].a = pMyData[i].b;
}
```

Partially Used Structure

Explanation

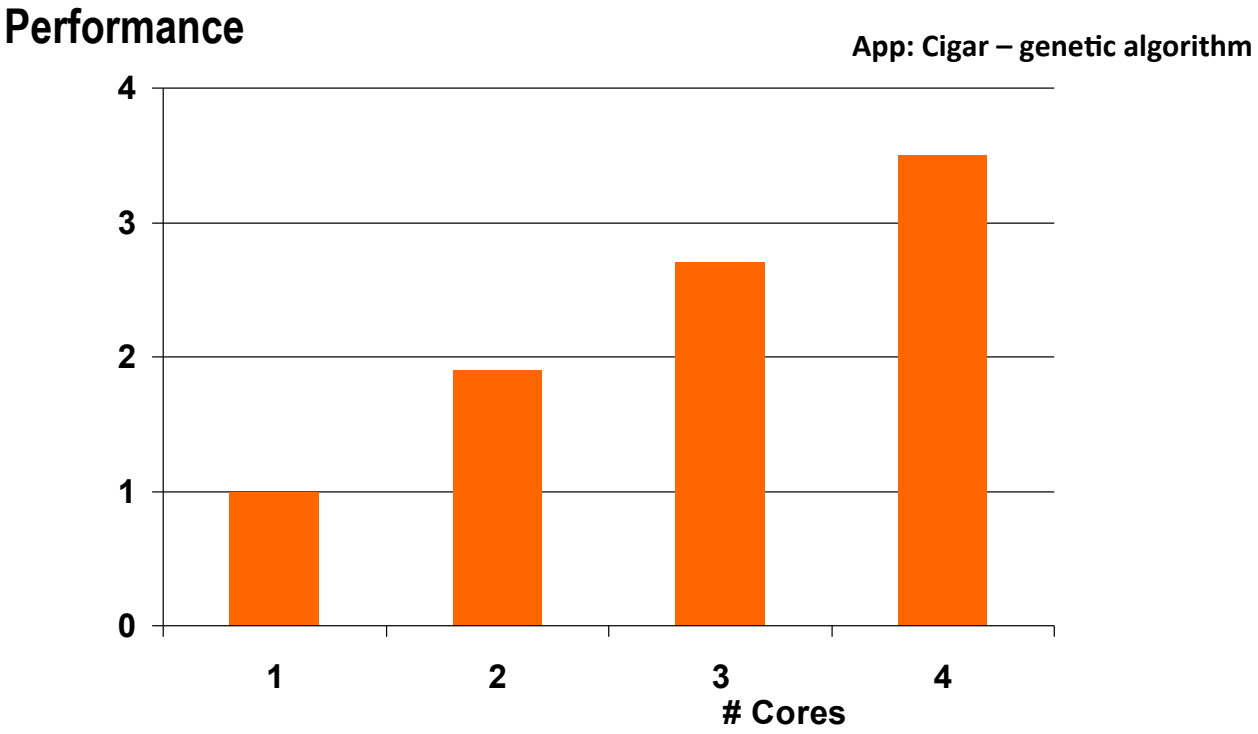
Partially Used Structure

Fully Used Structure

Partially Used Structure - Fixed

```
struct DATA
{
    int a;
    int b;
};



DATA * pMyData;
pMyData = new DATA[10*1024*1024];

for (long i=0; i<10*1024*1024; i++)
{
    pMyData[i].a = pMyData[i].b;
}
```

Example: A scalable parallel application?

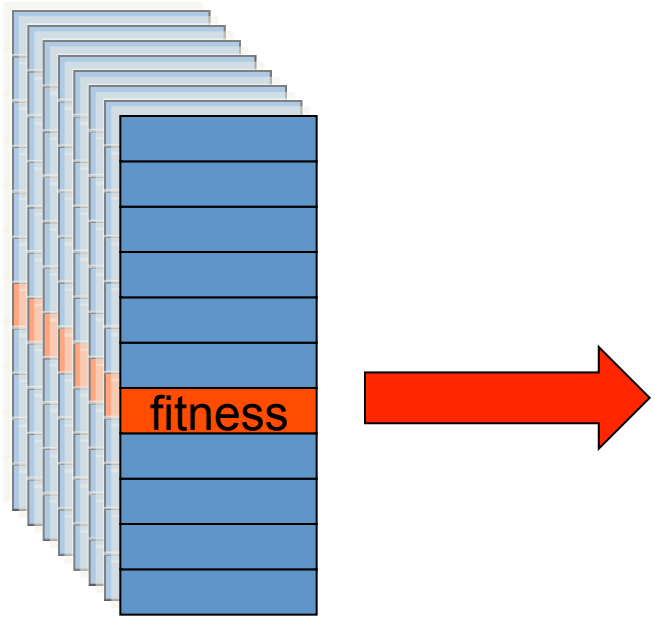


Looks like a perfect scalable application!

Are we done?

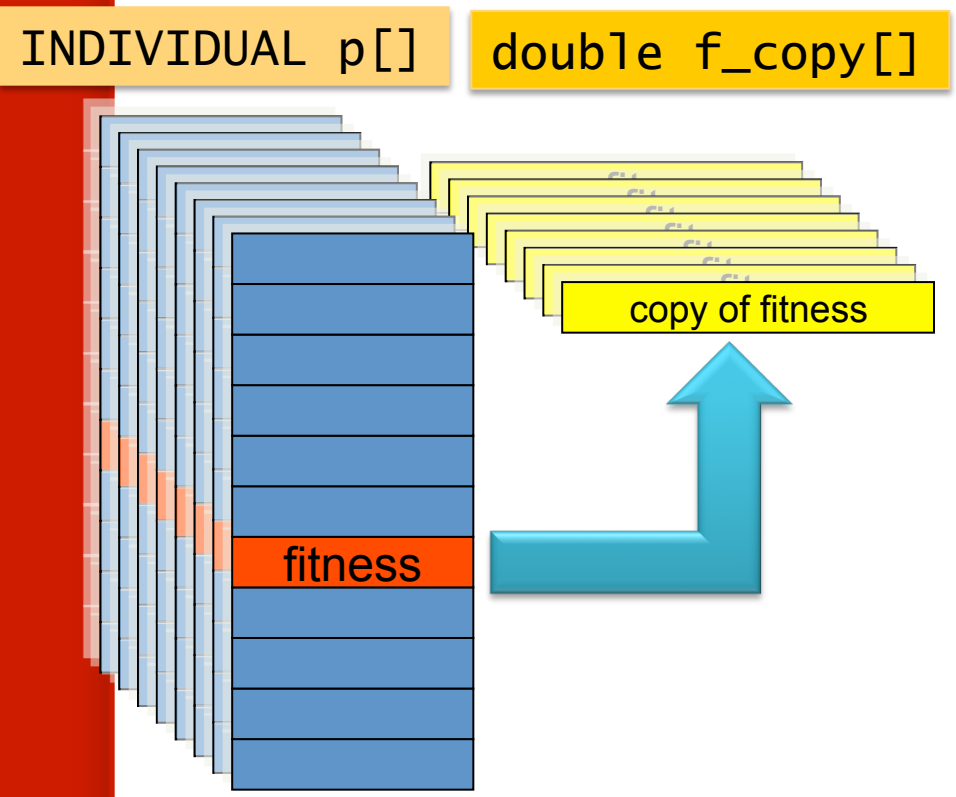
Cigar: The worst slow spot

INDIVIDUAL p[]



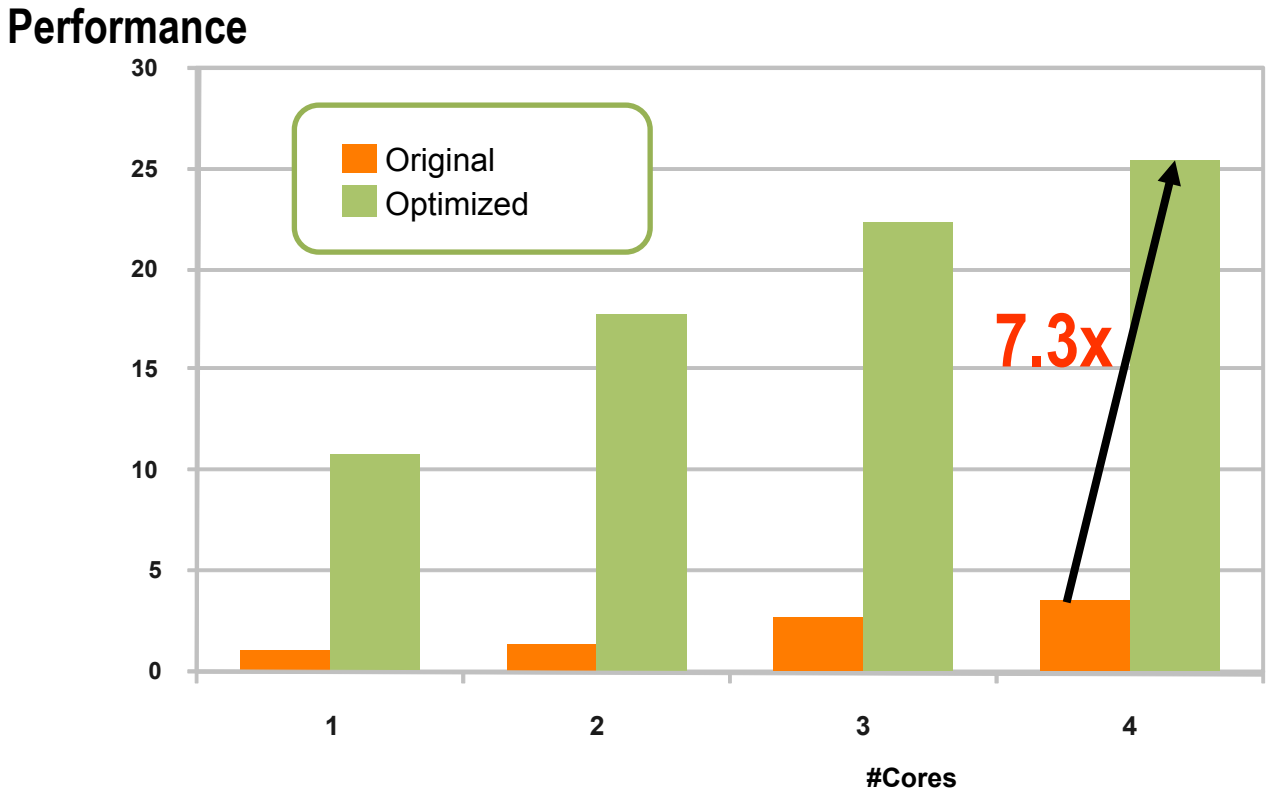

```
for (current = 0;
    current < howmany-1;
    current++)
{
 max = current;
 /* Find Next best */
 for (i = current+1;
      i < size; i++)
  if((p+rank[i])->fitness>
   (p+rank[max])->fitness)
      max = i;

 SwapInt(&rank[current],
        &rank[max]);
}
```

Cigar: Code change. Duplicate data




```
for (current = 0;
    current < howmany-1;
    current++)
{
 max = current;
 /* Find Next best */
 for (i = current+1;
      i < size; i++)
  if (f_copy[rank[i]] >
      f_copy[rank[max]])
      max = i;

 SwapInt(&rank[current],
        &rank[max]);
}
```

The same application optimized



- ParaTools ThreadSpotter advice: Change one data structure

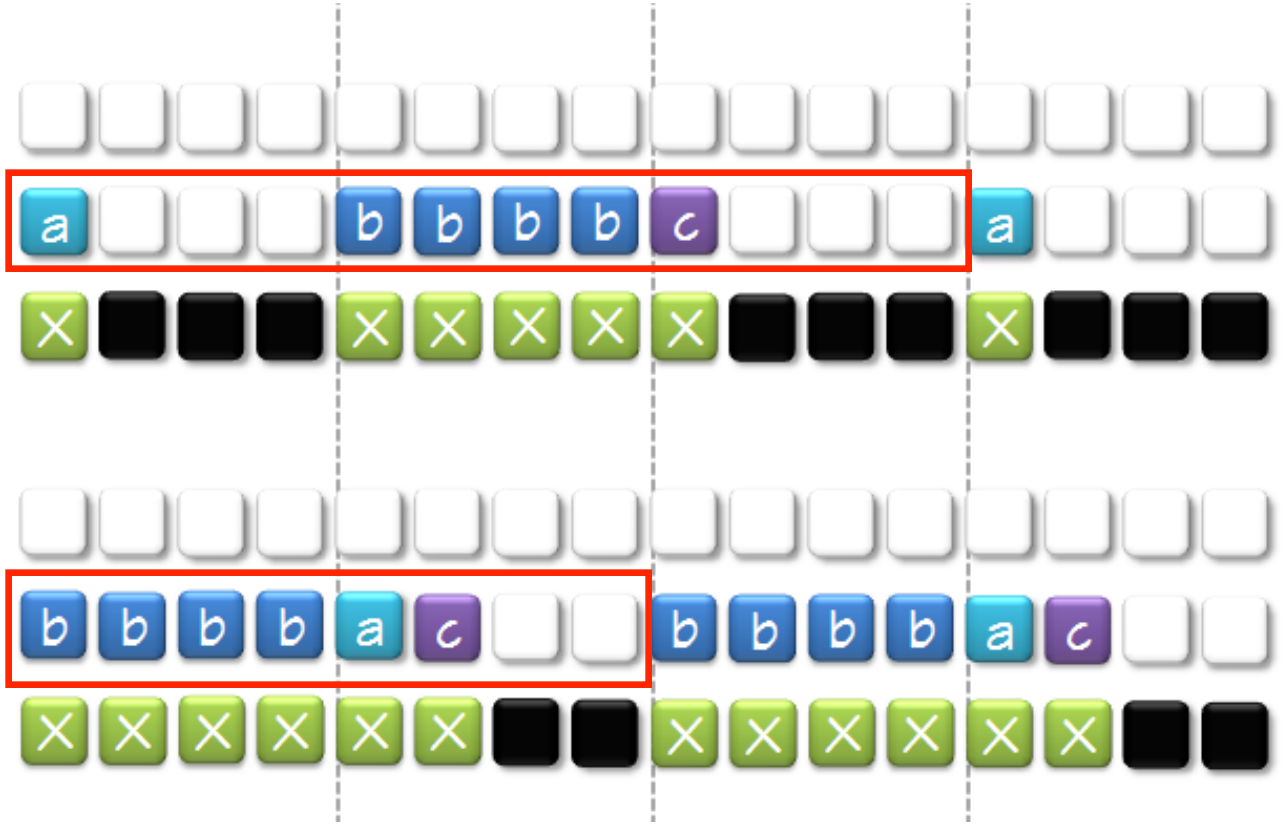
Alignment Problem

```
struct DATA
{
    char a;
    int b;
    char c;
};

DATA * pMyData;
pMyData = new DATA[36*1024*1024];

for (long i=0; i<36*1024*1024; i++)
{
    pMyData[i].a++;
}
```

Alignment Problem

Explanation

Inefficient Alignment



Fixed

Alignment Problem - Fixed

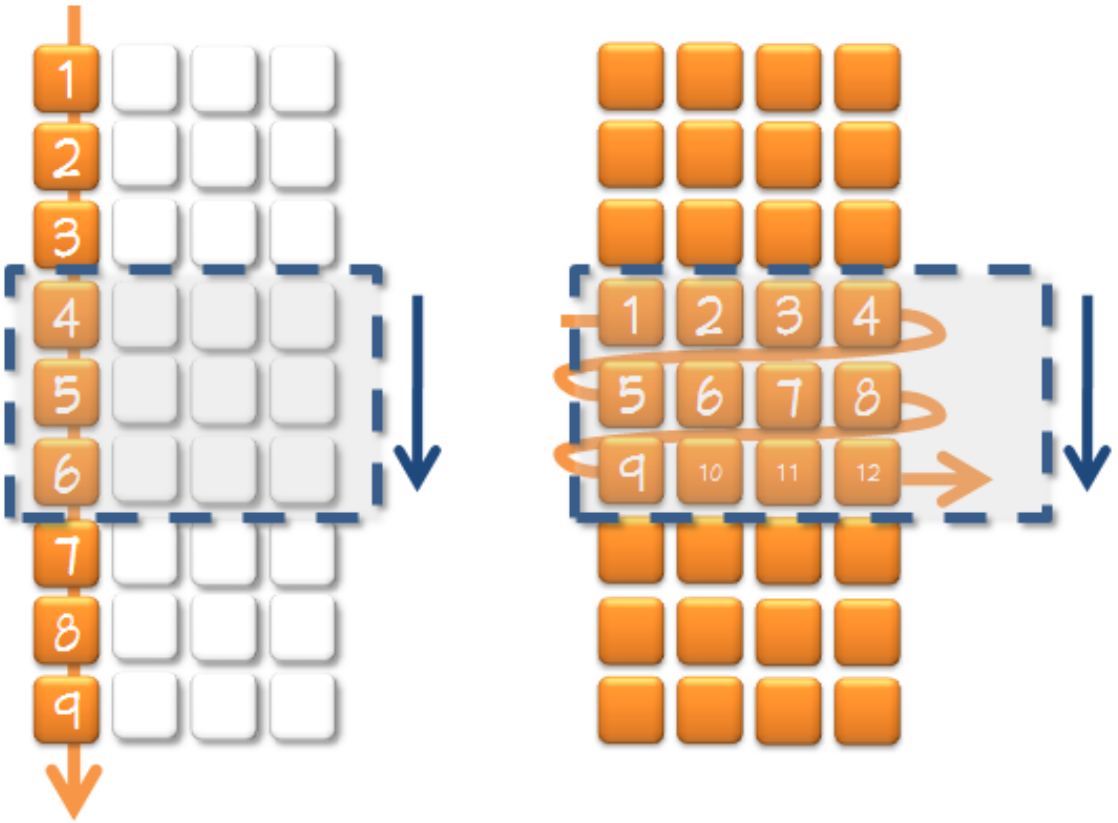
```
struct DATA
{
      int b;
       char a;
      char c;
};

DATA * pMyData;
pMyData = new DATA[36*1024*1024];

for (long i=0; i<36*1024*1024; i++)
{
   pMyData[i].a++;
}
```

Inefficient Loop Nesting

```
#define SIZE (36000*32*32)
#define ROWSIZE 16
#define NBROWS (SIZE/ROWSIZE)

char * p;
p = new char[SIZE];

long nbRows = NBROWS;
long sRowSize = ROWSIZE;

for (long x=0; x<sRowSize; x++)
for (long y=0; y<nbRows; y++)
{
   p[x+y*sRowSize]++;
}
```

Inefficient Loop Nesting

Explanation



ParaTools

Inefficient Loop Nesting - Fixed

```
#define SIZE (36000*32*32)
#define ROWSIZE 16
#define NBROWS (SIZE/ROWSIZE)

char * p;
p = new char[SIZE];

long nbRows = NBROWS;
long sRowSize = ROWSIZE;

for (long y=0; y<nbRows; y++)
for (long x=0; x<sRowSize; x++)
{
   p[x+y*sRowSize]++;
}
```

False Sharing

```
int sum1;
int sum2;

void thread1(int v[], int v_count) {
sum1 = 0;
for (int i = 0; i < v_count; i++)
sum1 += v[i];
}

void thread2(int v[], int v_count) {
sum2 = 0;
for (int i = 0; i < v_count; i++)
sum2 += v[i];
}
```

False Sharing (Fixed)

```
int __attribute__((aligned(64))) sum1;
int __attribute__((aligned(64))) sum2;

void thread1(int v[], int v_count) {
sum1 = 0;
for (int i = 0; i < v_count; i++)
  sum1 += v[i];
}

void thread2(int v[], int v_count) {
sum2 = 0;
for (int i = 0; i < v_count; i++)
  sum2 += v[i];
}
```

Questions ?

Thank you