清華大學
Tsinghua University

第1次实验：存储层次分析及程序优化

清華大學
Tsinghua University

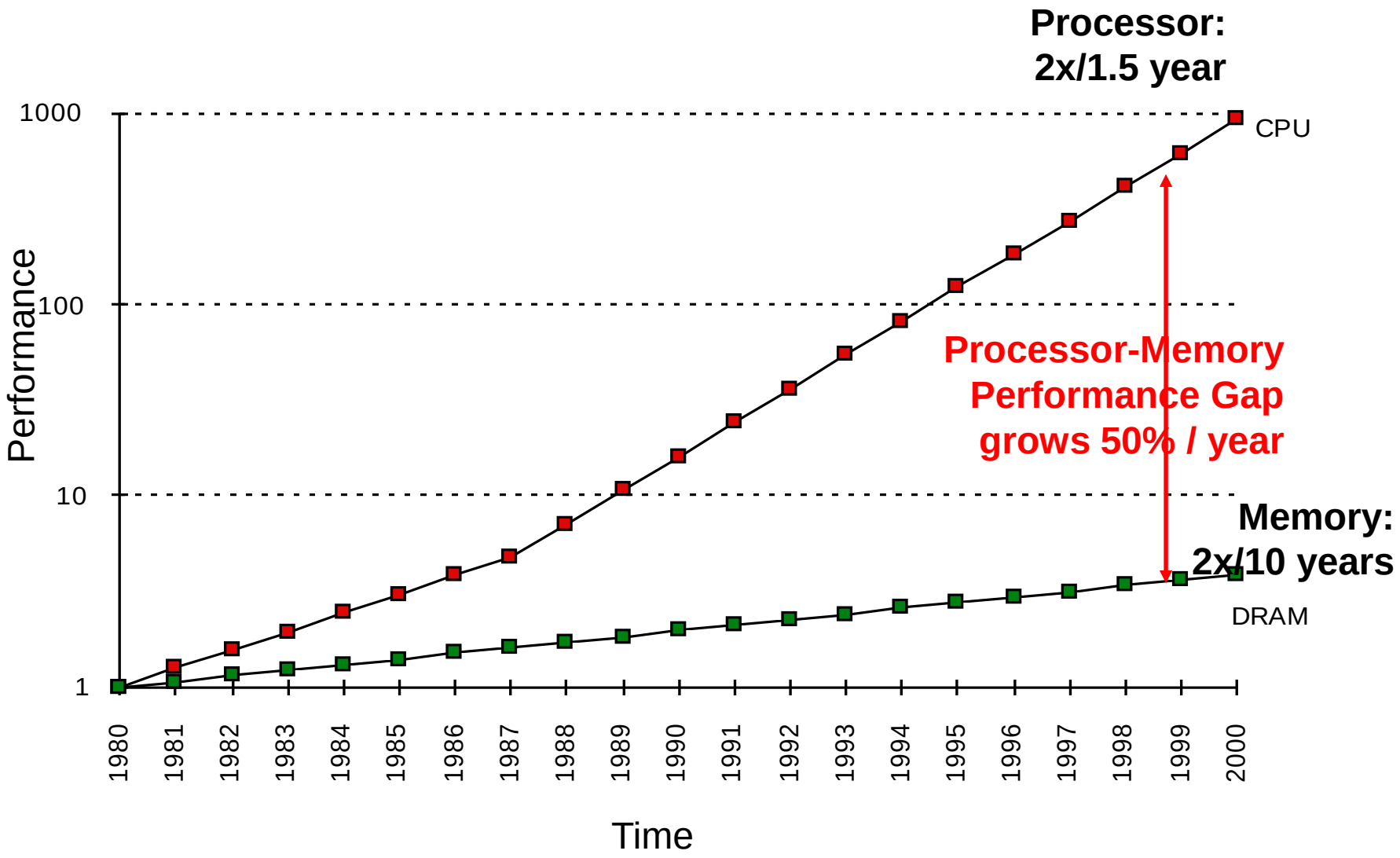
实验内容

- 测量Cache参数
  - Size
  - Line size
  - Ways of associativity
  - Write strategy
  - Replacement policy
- 程序优化
  - 根据Cache结构对矩阵乘程序进行优化

实验要求

- 请阅读文档：实验要求

Processor-DRAM Latency Gap



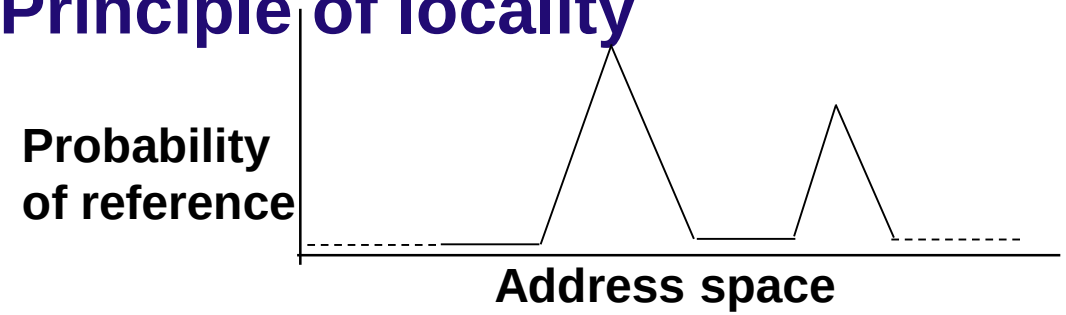
Cache Size

- CMP中常见的Cache组织
  - Private L1 D Cache
  - Private L1 I Cache
  - Private L2 Cache
  - Shared L3 Cache (increase with #core)
- 实验要求测量
  - L1 D Cache (32KB?)
  - L2 Cache (256KB?)
  - L1 I and L3 Cache (optional)

产品名称	发行日期	内核数
Intel® Xeon® Processor E7-8850 v2 (24M Cache, 2.30 GHz)	Q1'14	12
Intel® Xeon® Processor E7-8857 v2 (30M Cache, 3.00 GHz)	Q1'14	12
Intel® Xeon® Processor E7-8870 v2 (30M Cache, 2.30 GHz)	Q1'14	15
Intel® Xeon® Processor E7-8880 v2 (37.5M Cache, 2.50 GHz)	Q1'14	15
Intel® Xeon® Processor E7-8880L v2 (37.5M Cache, 2.20 GHz)	Q1'14	15
Intel® Xeon® Processor E7-8890 v2 (37.5M Cache, 2.80 GHz)	Q1'14	15
Intel® Xeon® Processor E7-8891 v2 (37.5M Cache, 3.20 GHz)	Q1'14	10
Intel® Xeon® Processor E7-8893 v2 (37.5M Cache, 3.40 GHz)	Q1'14	6
Intel® Xeon® Processor E7-4809 v2 (12M Cache, 1.90 GHz)	Q1'14	6
Intel® Xeon® Processor E7-4820 v2 (16M Cache, 2.00 GHz)	Q1'14	8
Intel® Xeon® Processor E7-4830 v2 (20M Cache, 2.20 GHz)	Q1'14	10
Intel® Xeon® Processor E7-4850 v2 (24M Cache, 2.30 GHz)	Q1'14	12
Intel® Xeon® Processor E7-4860 v2 (30M Cache, 2.60 GHz)	Q1'14	12

Why hierarchy works?

- **Principle of locality**



Rule of thumb:
Programs spend
90% of their
execution time in
only 10% of code

- **Temporal locality: recently accessed items are likely to be accessed in the near future**
  ⇒ Keep them close to the processor
- **Spatial locality: items whose addresses are near one another tend to be referenced close together in time**
  ⇒ Move blocks consisted of contiguous words to the upper level

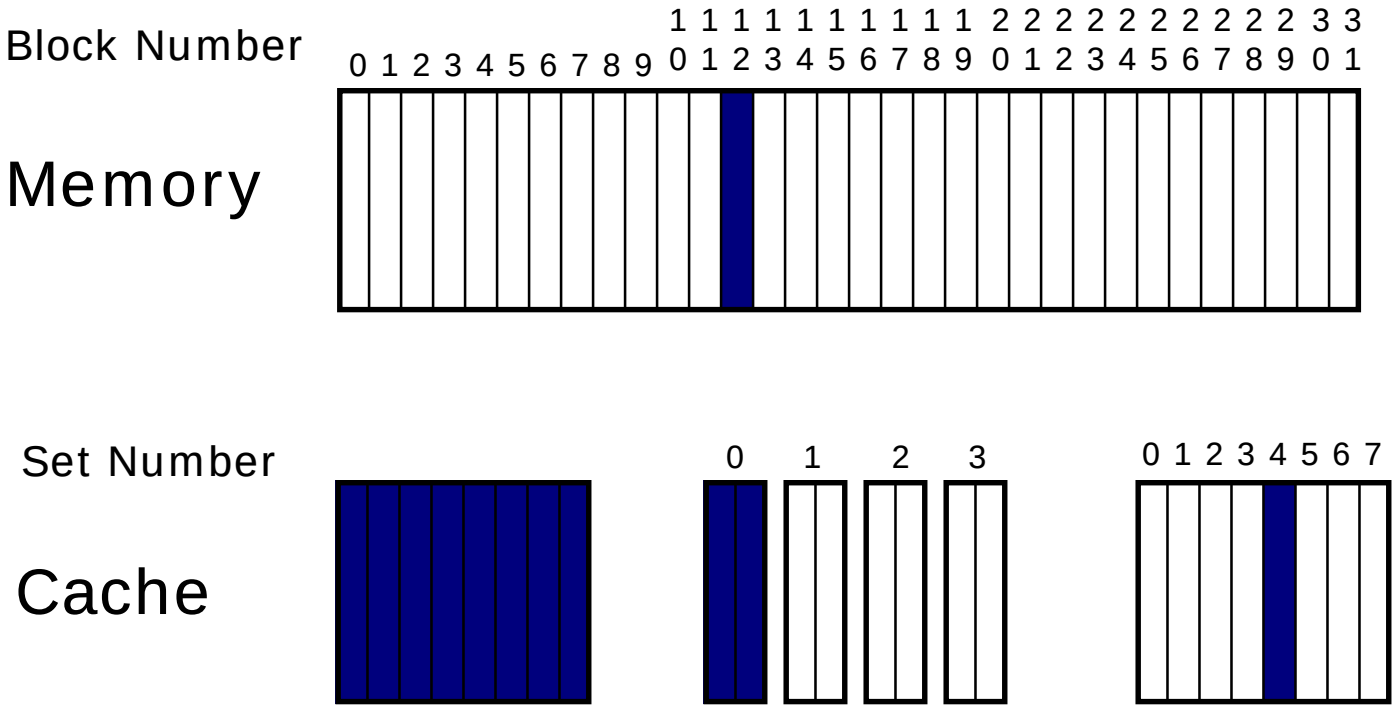
Cache Line

- To exploit spatial locality
- Dose cache line size remains same between different cache levels?
- The same cache line size simplifies design.
  - Intel Core i7/i5/i3 uses 64B line size in each level.
  - Can you verify this?
- But it does not have to…
  - Intel Pentium 4 has 64B L1 cache lines and 128B L2 cache lines.
    - Slightly improve the hit rate
    - Require a longer latency to refill

Four Questions for Memory Heir.

- **Q#1: Where can a block be placed in the upper level?**
  ⇒ Block placement
  - direct-mapped, fully associative, set-associative
- **Q#2: How is a block found if it is in the upper level?**
  ⇒ Block identification
- **Q#3: Which block should be replaced on a miss?**
  ⇒ Block replacement
  - Random, LRU (Least Recently Used)
- **Q#4: What happens on a write? ⇒ Write strategy**
  - Write-through vs. write-back
  - Write allocate vs. No-write allocate

Placement Policy



block 12
can be placed

Ways of Associativity

- **#block in one set**
- **Distinguish placement policy through ways of associativity.**
  - **Set-associative**
    - 1 < #set < #block of cache
    - 1 < ways of associativity < #block of cache
  - **Direct-mapped**
    - #set = #block of cache
    - ways of associativity = 1
  - **Fully associative**
    - only one set
    - ways of associativity = #block of cache

Replacement Policy

- LRU (Least Recently Used)
- PLRU (Pseudo LRU)
  - Tree-PLRU
  - Bit-PLRU
- MRU (Most Recently Used)
- LFU (Least Frequently Used)
- Adaptive/Dynamic replacement policy
  - LRU vs. LFU
  - DIP (LRU vs. BIP)
  - DRRIP (SRRIP vs. BRRIP)

清華大學
Tsinghua University

Write Strategy

- Write through vs. write back
  - What happens on a write-hit
  - Design a method to test this is required!

- Write allocate vs. no-write allocate
  - What happens on a write-miss
  - Can you test this?

如何测量Cache参数?

- 观察测试程序的平均内存访问时间AMAT
  - 测试程序需实现特定的access pattern
  - 需对其它cache参数进行假设
    - 假设的完整性将影响实验和分析的准确性与复杂性

- For example: 如何测量cache size?
  - The behavior of LRU replacement policies with cyclic access patterns is useful for measuring cache sizes and latencies.
  - Using a cyclic pattern results in sharp changes in latency between cache levels.

Access Patterns

- Recency-friendly (e.g. stack access pattern)
  - Near-immediate re-reference interval
  - Benefit from LRU replacement

$$(a_1, a_2, \ldots, a_{k-1}, a_k, a_k, a_{k-1}, \ldots, a_2, a_1)^N$$

- Thrashing (e.g. cyclic access pattern)
  - Distant re-reference interval
  - LRU receives zero cache hits
  - Benefit from MRU replacement

$$(a_1, a_2, \ldots, a_k)^N$$

Access Patterns (Cont.)

- **Streaming access pattern**
  - **Infinite re-reference interval**
  - **No cache hits under any replacement policy**

$$(a_1, a_2, a_3, a_4, \ldots a_k)$$

- **Mixed access pattern**
  - **An access pattern with near-immediate or distant re-reference interval followed by a reference to a sequence with a probability.**
  - **Benefit from LFU**

$$[(a_1 , \ldots , a_k , a_k , \ldots , a_1)^A \; P (a_1 , a_2 , \ldots , a_k , a_{k+1} \ldots , a_m)]^N$$

$$[(a_1 , \ldots , a_k)^A \; P (b_1 , b_2 , \ldots , b_m)]^N$$

"scan"

Access Patterns (Cont.)

- 选择哪种access pattern进行实验?

- 取决于实验设计及假设
  - 当假设替换策略为LRU时，可以利用cyclic access pattern测量cache大小.
    - Cyclic access pattern可以使LRU替换策略每次访问都出现miss.
  - 利用LRU和MRU替换策略在不同access pattern上的不同表现，测量cache替换策略.

- Access pattern的实现?

Cyclic Access Pattern

- **Simplest implementation**

```
unsigned char array[SIZE] = {0};
unsigned char touch;
for (int i = 0; i < ITER; i++) {
    for (int j = 0; j < SIZE; j += STRIDE) {
        touch = array[j];
    }
}
```

Tsinghua University

Cyclic Access Pattern (Cont.)

- **Data-dependent access**
  - **Measure latency**
  - **Avoid parallel memory accesses**
    - New address depends on previous loaded data
- **Read rather than write**
  - **Avoid troubles related to write strategy**
    - Buffered, data-dependent …
  - **Of course, you have to write to memory when you test write strategy.**

Tsinghua University

Cyclic Access Pattern (Cont.)

- **Hardware prefetching**
  - **Processor loads data into cache before the data is accessed. (Prediction)**
  - **Access pattern needs randomness.**

- **Software prefetching**
  - **Prefetch instructions**
  - **Turn off prefetch related compiler options such as compiler optimization.**

Cyclic Access Pattern (Cont.)

- **Initialize array**

```
void * array[SIZE] = {NULL}
for (int i = 0; i < SIZE; i++) {
    array[i] = &array[i];
}
for (int i = SIZE - 1; i >= 0; i--) {
    if (i < STRIDE) continue;
    int j = rand()%(i/STRIDE) * STRIDE + i%STRIDE;
    void * tmp = array[i];
    array[i] = array[j];
    array[j] = tmp;
}
register void * p = &array[rand()%SIZE];
```

清華大學
Tsinghua University

Cyclic Access Pattern (Cont.)

- **Memory access**

```
for (int i = 0; i < ITRS; i++) {
    p = *(void**)p
    p = *(void**)p
    …
}
```

清華大學
Tsinghua University

Cyclic Access Pattern (Cont.)

- **Reduce the impact of TLB misses**
  - **We want to measure time handling cache misses, not time handling TLB misses.**
  - **Linux hugepages (2MB pages)**

Cyclic Access Pattern (Cont.)

- Mount hugetlbfs
  - 编译内核时配置CONFIG_HUGETLB_PAGE和CONFIG_HUGETLBFS选项
  - mount none /mnt/huge -t hugetlbfs

- Array allocation

```
int fd = open("mnt/huge/temp", O_CREAT | O_RDWR, 0755);
void ** array = (void**)mmap(0, SIZE*sizeof(void*), \
                PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

Access Pattern实现总结

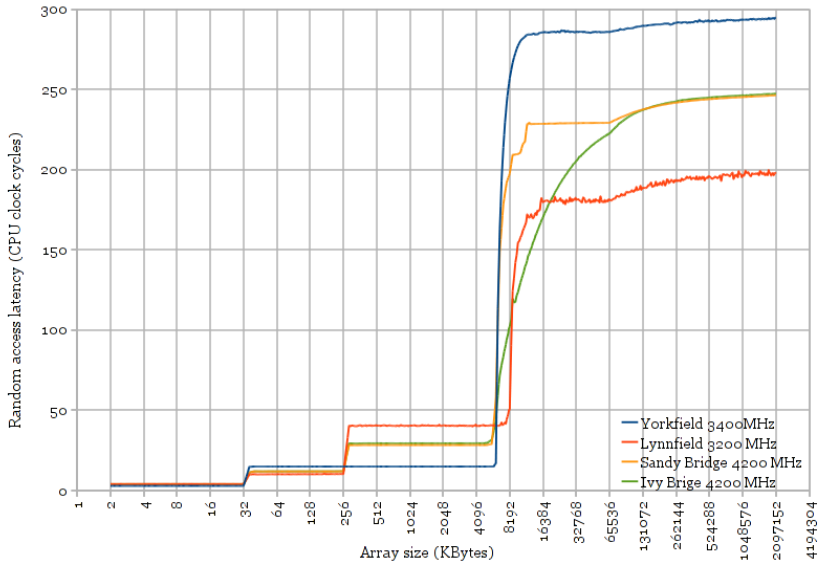
- 实现方法及相关细节
  - Data-dependent read
  - Prefetching
  - Hugepages
- 不建议使用复杂的库(如STL)
  - 数据结构复杂，难以控制一次操作访问内存的次数.
- 保证不同次实验的结果有可比性
  - 比较一次访存的平均延迟
  - 比较总延迟，保证每次实验的访存次数相同
- 各种access pattern的实现均需注意以上问题

计时功能实现

- 每次实验保证足够的访存次数
  - 单次访存延迟太小，不能精确测量.
- 计时函数选择
  - clock函数
    - ANSI标准
    - 不计算子进程消耗的时间
  - times函数
    - 计算子进程消耗的时间
    - 区分用户代码和内核代码消耗的时间
  - clock_gettime、gettimeofday函数
    - 精度更高
- 利用shell的time命令检验

测量Cache Size

- Access pattern
  - Cyclic
- Replacement policy
  - LRU
- STRIDE
  - Cache line size
- AMAT
  - 在SIZE大小超过cache大小时产生跳变



测量Cache Size (Cont.)

- 不断从内存读取连续数组中的数据，观察平均读取速度.
- 当数组大小超过L1 D Cache Size后，会出现cache读缺失，平均读取速度会有一个突然的增加.
- 同理，当数组大小超过L2 Cache Size后，平均读取速度也会有一个突然的增加.

测量Cache Size (Cont.)

- 测量cache size，主要是利用L1 Cache、L2 Cache和主存之间的速度差异。如果一个数组能够完全存放在L1 D Cache中，那么不停地循环访问数组。第一遍循环时，数组从主存搬移到L1 D Cache中。以后每次循环直接从L1 Cache中访问数据，速度很快。
- 如果数组超过了L1 D Cache的大小，就不能整个存放在L1 Cache中。当以cache line size为步长，间隔地循环访问数组时，可以使得每次访问都是缺失的，必须访问L2 Cache。L2 Cache的速度低于L1 Cache，从而使得访问数据所需的时间跳跃式地增大。

测量Cache Size (Cont.)

- 当STRIDE没有刚好取cache line size时
  - If STRIDE < cache line size
    - 跳变的高度变低
    - STRIDE越接近cache line size，跳变的高度越接近最大值.
  - If STRIDE > cache line size
    - 发生跳变的位置向右偏移
    - 跳变的高度不变

测量Cache Size (Cont.)

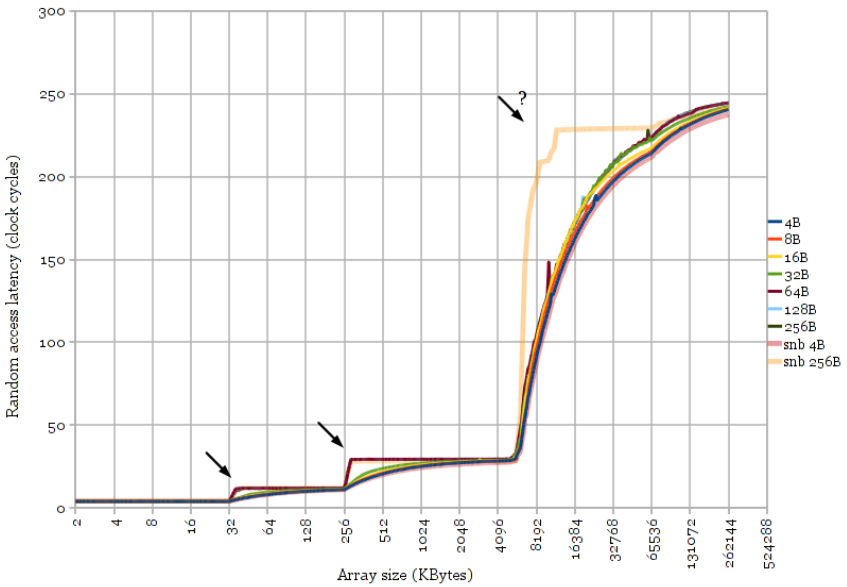
- 考虑placement policy对测量结果的影响
  - Fully associative
    - 测量结果与之前的估计一致
  - Direct-mapped
    - 当#block in array - #block in cache < #block in cache时，不能保证每次访存都缺失.
    - 导致跳变的过程变缓
  - Set-associative
    - 当#block in array - #block in cache < #set时，不能保证每次访存都缺失.
    - 导致跳变的过程变缓

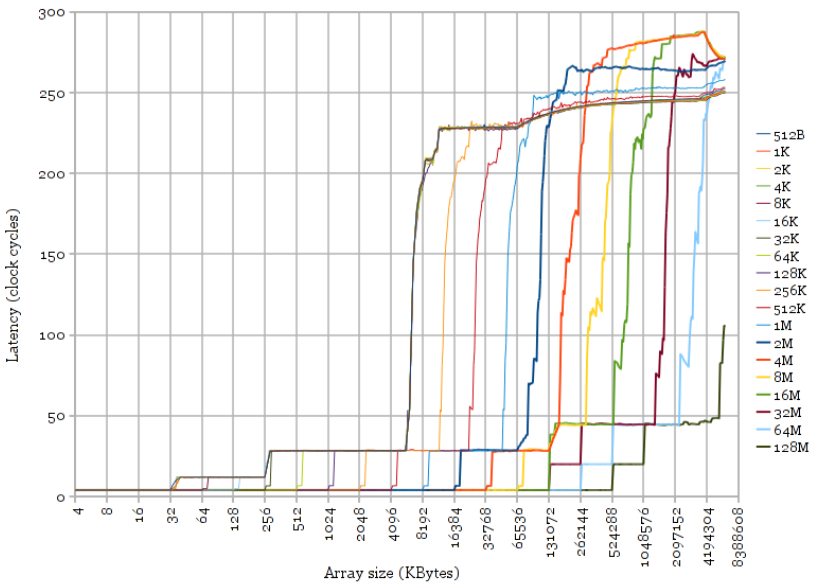
测量Cache Size (Cont.)

- 考虑replacement policy对测量结果的影响
  - 有些替换策略使得跳变的过程变缓(如PLRU)
    - 为什么是使跳变的过程变缓，而不是跳变的高度变低?
  - 有些替换策略使得跳变不再发生(如MRU)
    - 当数组大小超过cache大小时，延迟不再产生跳变，而是改变增长的斜率.

测量Cache Size

- 当STRIDE大小发生改变时，测量结果的变化.





测量Cache Line Size

- 在测量cache size时，我们已经讨论了STRIDE不等于cache line size对所测得延迟的影响.
- 能否以此为依据测量cache line size?
  - 使数组大小超过L1 Cache Size
  - 逐步增加STRIDE
  - 当STRIDE = cache line size时，延迟升至最大.
- 测量所得的访问延迟曲线应当是怎样的?
  - 访问延迟由较低渐变到最大值
- 其它cache参数是否会对测量结果造成影响?

测量Cache Line Size (Cont.)

- 当访问数组中的元素时，如果是连续访问，因为cache line的第一个字节缺失后，会将整个cache line移入cache，因此后续访问的命中率会很高。
- 如果访问是间断的，对数组间隔顺序访问，命中率就会降低，平均访问延迟增大。当间隔达到一定的大小，即超过cache line size，将造成每次都缺失的最坏情况，平均访问延迟达到最大。

测量Ways of Associativity

- 在测量cache size时，我们已经讨论了placement policy对测量结果的影响.
- 能否以此为依据测量ways of associativity?
  - 当#block in array - #block in cache < #set时，不能保证每次访存都缺失.
  - 当#block in array - #block in cache > #set时，每次访存都会缺失.
  - 由此估计set的个数，进而结合cache size计算ways of associativity.
- 准确性不容易保障
- 其它方法?
  - 构造新的access pattern

测量Ways of Associativity (Cont.)

- 使用一个2倍cache size大小的数组.
- 将数组分为2^n块，只访问其中的奇数块.
- 逐渐增大n的取值，当某一次访问时间变慢时，2^{n-2}就是ways of associativity的值.

测量Replacement Policy

- 需针对不同的replacement policy进行有针对性的区分
- For example: LRU vs. MRU
  - LRU对stack access pattern的命中率较高，对cyclic access pattern的命中率很低.
  - MRU对stack access pattern和cyclic access pattern的命中率都较高.
- 参考资料
  - Intel Ivy Bridge Cache Replacement Policy
  - http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/

测量Write Strategy

- Write through vs. write back
  - Write through在write-hit和write-miss时的访问延迟相似.
  - Write back在write-hit时的访问延迟明显小于write-miss.
  - Write Buffer对测量结果是否有影响?
- 能否测量write allocate vs. no-write allocate?
  - Write allocate写完成后的读访问延迟低
  - No-write allocate写完成后的读访问延迟高

Cache参数测量总结

- 各参数的参考测量方法
- 注重思考测量方法的原理
- 注重分析测量方法的误差
- 不仅仅局限于编写代码
- 扩展
  - 各级cache的cache line size是否相同？
  - 各级cache的替换策略是否相同？
  - 新的测量方法？
  - 使用新的access pattern?
  - 发掘其它有意义的cache参数?
  - …

矩阵乘优化

- 能否应用前面测量所得的cache参数信息?

- For example: 分块相乘
  - 利用已知的cache size信息对矩阵进行分块，使得cache可以容纳下分块后的矩阵.

- 优化方法
  - 尽量使用与cache有关的优化
  - 尽量不要使用并行化
  - 尽量不要使用算法层次的优化

Question:

```
int sum_array_rows(int a[M][N])
{
  int i, j, sum = 0;

  for (i = 0; i < M; i++)
     for (j = 0; j < N; j++)
        sum += a[i][j];
  return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
  int i, j, sum = 0;

  for (j = 0; j < N; j++)
     for (i = 0; i < M; i++)
        sum += a[i][j];
  return sum;
}
```

Which is faster?

Matrix Multiplication

```
 for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

Which is faster?

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

5th Miss Rate Reduction Technique: Compiler Optimizations

- Reduction comes from software
- McFarling [1989] reduced caches misses by 75% (8KB, DM, 4 byte blocks) in software
- Instructions
  - Reorder procedures in memory so as to reduce conflict misses
  - Profiling to look at conflicts(using tools they developed)
- Data
  - Merging Arrays: improve spatial locality by single array of compound elements vs. 2 arrays
  - Loop Interchange: change nesting of loops to access data in order stored in memory
  - Loop Fusion: Combine 2 independent loops that have same looping and some variables overlap
  - Blocking: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows