UNDERSTANDING PROCESSOR CACHE EFFECTS WITH VALGRIND & VTUNE

Chester Rebeiro

Embedded Lab

IIT Kharagpur

Is Time Proportional to Iterations?

- SIZE = 64MBytes;
- unsigned int A[SIZE];

- *Iterations A:*

  ```
  for(i=0; i<SIZE; i+=1) A[i] *= 3;
  ```

- *Iterations B:*

  ```
  for(i=0; i<SIZE; i+=16) A[i] *= 3;
  ```

- Is **Time(A) / Time(B) = 16 ?**

Is Time Proportional to Iterations?

- Not Really !
  - **We get Time(A)/Time(B) = 3 !**
  - Straight forward pencil-and-paper analysis will not suffice
  - A deeper understanding is needed
  - For this we use profiling tools

Tools for Profiling Software

- Static Program Modification
  - Automatic insertion of code to record performance attributes at run time.
  - Example : QPT (Quick program profiling and tracing) for MIPS and SPARC systems, Gprof, ATOM
- Hardware Counters
  - Requires support from processor for hardware performance monitoring
  - **VTune (commercial – Intel)**, oprofile, perfmon
- Simulators
  - For simulation of the platform behavior
  - **Valgrind** (x86 Simulation), **Simplescalar**

Valgrind

- **Opensource : http://valgrind.org**
- Valgrind is an instrumentation framework for building dynamic analysis tools.
- There are tools for
  - **Memory checking :** to detect memory management problems such as no uninitilized data, leaky, overlapped memcpy's etc.
  - **Cachegrind :** is a cache profiler
  - **Callgrind :** Extends cachegrind and in addition provides information about *callgraphs*.
  - **Massif :** is a heap profiler
  - **Helgrind :** is useful in multi-threaded programs.

Cachegrind

- Pinpoints the sources of cache misses in the code.
- Can simulate L1, L2, and D1 cache memories
- On Modern processors :
  - L1 cache miss costs around 10 clock cycles
  - L2 cache miss can cost as much as 200 clock cycles.

Iteration Example Revisited with Cachegrind

- SIZE = 64MBytes;
- unsigned int A[SIZE];

- *Iterations A:*

  for(i=0; i<SIZE; **i+=1**) A[i] *= 3;
- *Iterations B:*

  for(i=0; i<SIZE; **i+=16**) A[i] *= 3;

- Is the ratio of **Time(A) / Time(B) = 16 ?**

Running Cachegrind

```
./valgrind --tool=cachegrind --cachegrind-out-file=sorts/cg1.out  sorts/loops
```

Tool

Output file name

Executable

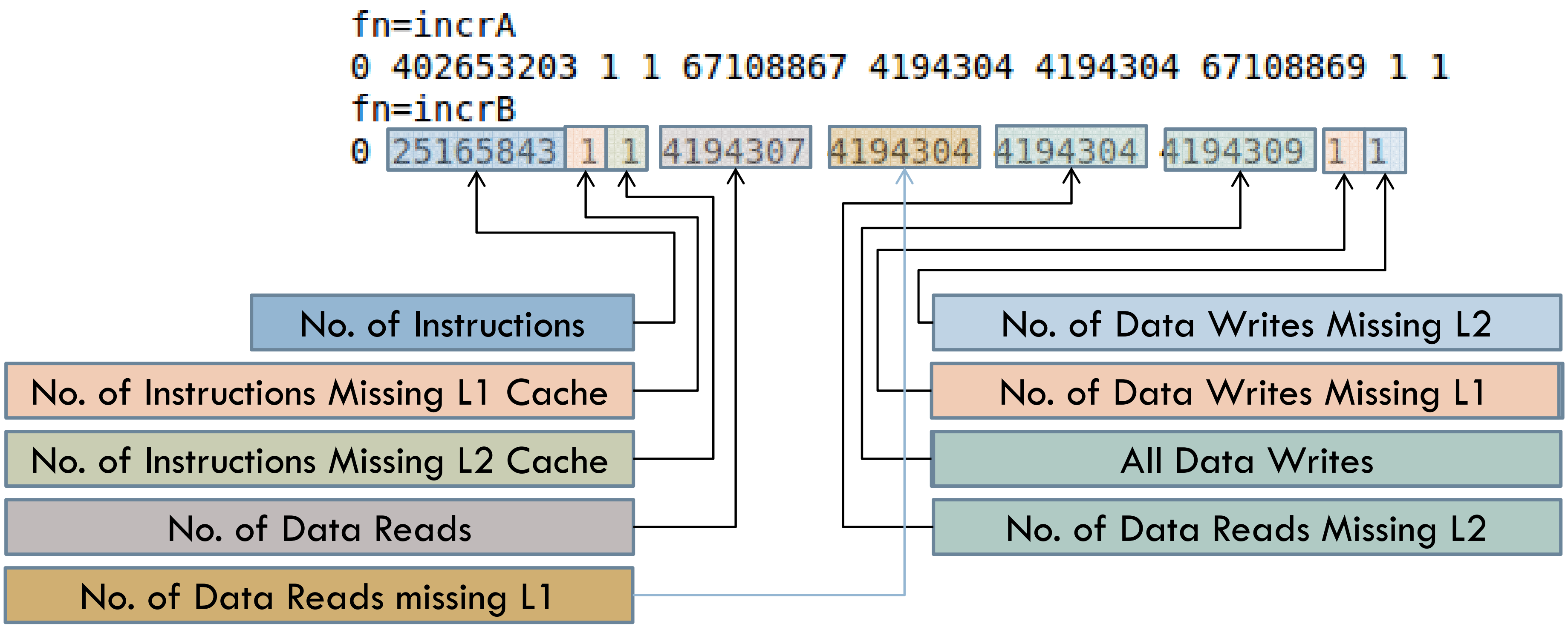
Console Output :

No. of instructions

No. of misses in I1

```
==24431==
==24431== I   refs:      427,911,025
==24431== I1  misses:            693
==24431== LLi misses:            689
==24431== I1  miss rate:        0.00%
==24431== LLi miss rate:        0.00%
==24431==
==24431== D   refs:      142,642,258  (71,328,739 rd   + 71,313,519 wr)
==24431== D1  misses:      8,389,711  ( 8,389,444 rd   +        267 wr)
==24431== LLd misses:      8,389,639  ( 8,389,378 rd   +        261 wr)
==24431== D1  miss rate:         5.8% (      11.7%     +        0.0%  )
==24431== LLd miss rate:         5.8% (      11.7%     +        0.0%  )
==24431==
==24431== LL refs:         8,390,404  ( 8,390,137 rd   +        267 wr)
==24431== LL misses:       8,390,328  ( 8,390,067 rd   +        261 wr)
==24431== LL miss rate:          1.4% (        1.6%    +        0.0%  )
```

Output of Cachegrind (cg1.out)



cg_annotate

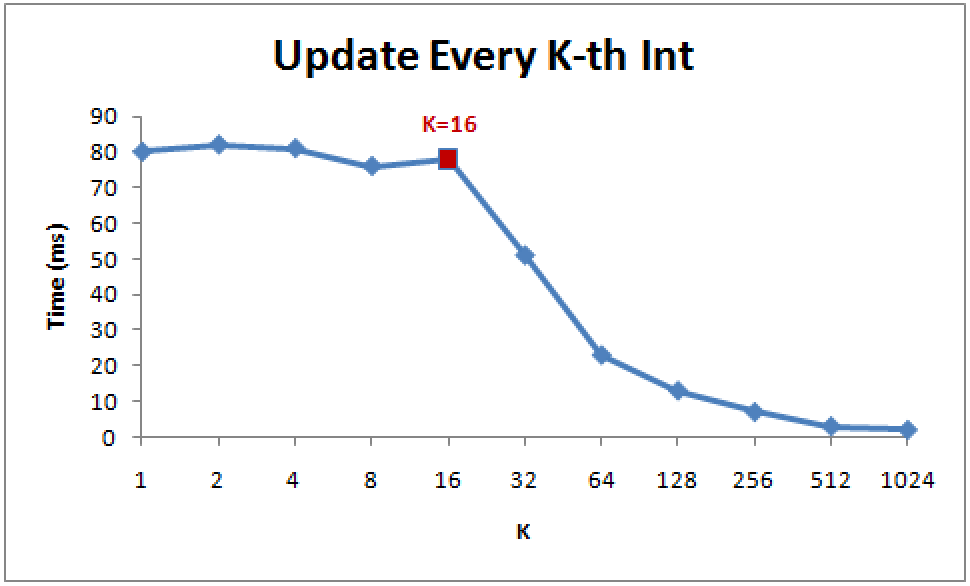
```
[chester@anubis bin]$ cg_annotate sorts/cg1.out
--------------------------------------------------------------------------------
I1 cache:         32768 B, 64 B, 4-way associative
D1 cache:         32768 B, 64 B, 8-way associative
LL cache:         8388608 B, 64 B, 16-way associative
Command:          sorts/loops
Data file:        sorts/cg1.out
Events recorded:  Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Events shown:     Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Event sort order: Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Thresholds:       99 0 0 0 0 0 0 0 0
Include dirs:
User annotated:
Auto-annotation:  off

--------------------------------------------------------------------------------
         Ir I1mr ILmr         Dr      D1mr      DLmr         Dw D1mw DLmw
--------------------------------------------------------------------------------
427,911,025  693  689 71,328,739 8,389,444 8,389,378 71,313,519  267  261  PROGRAM TOTALS

--------------------------------------------------------------------------------
         Ir I1mr ILmr         Dr      D1mr      DLmr         Dw D1mw DLmw  file:function
--------------------------------------------------------------------------------
402,653,203    1    1 67,108,867 4,194,304 4,194,304 67,108,869    1    1  ???:incrA
 25,165,843    1    1  4,194,307 4,194,304 4,194,304  4,194,309    1    1  ???:incrB
```

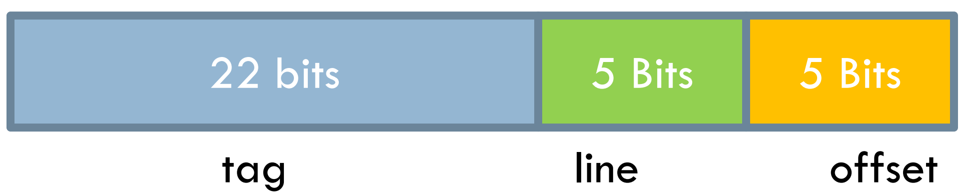
Effects of Cache Line

- Unsigned int takes 4 bytes
- Data cache line is of 64 bytes
- So every 16th byte falls in a new cache line and results in a cache miss



Direct Mapped Cache

- Consider a Direct Mapped Cache with
  - 1024 Bytes
  - 32 byte cache line
- Number of Cache Lines = 1024/32 = 32
- Assume memory address is of 32 bits



- For ex: Address = 0x12345678
  - Offset : $(11000)_2$
  - Line : $(10011)_2$

Direct Mapped Cache

```c
#define SIZE        (8 * 1024 * 1024)


unsigned int A[SIZE][8];
```

```c
unsigned int incrA()
{
        int i;
        int rep = 1 << 20;
        unsigned int x;

        for(i=0; i<rep; i++){
                x ^= A[0][0];
                x ^= A[31][0];
        }

        return x;
}
```

```c
unsigned int incrA()
{
        int i;
        int rep = 1 << 20;
        unsigned int x;

        for(i=0; i<rep; i++){
                x ^= A[0][0];
                x ^= A[32][0];
        }

        return x;
}
```

Cache Grind Results for Direct Mapped

```
==25666== I   refs:      8,474,937
==25666== I1  misses:          591
==25666== LLi misses:          586
==25666== I1  miss rate:      0.00%
==25666== LLi miss rate:      0.00%
==25666==
==25666== D   refs:      7,373,749  (7,364,175 rd   + 9,574 wr)
==25666== D1  misses:        8,977  (    7,017 rd   + 1,960 wr)
==25666== LLd misses:          819  (      611 rd   +   208 wr)
==25666== D1  miss rate:       0.1% (      0.0%     +  20.4%  )
==25666== LLd miss rate:       0.0% (      0.0%     +   2.1%  )
```

A[31][0]

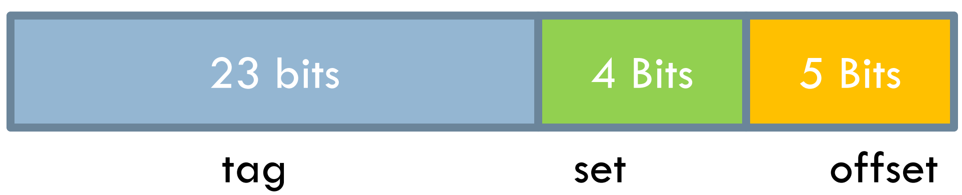
Thrashing in Cache Memories

A[32][0]

```
==25673== I   refs:      8,474,937
==25673== I1  misses:          591
==25673== LLi misses:          586
==25673== I1  miss rate:      0.00%
==25673== LLi miss rate:      0.00%
==25673==
==25673== D   refs:      7,373,749  (7,364,175 rd   + 9,574 wr)
==25673== D1  misses:    2,106,127  (2,104,167 rd   + 1,960 wr)
==25673== LLd misses:          819  (      611 rd   +   208 wr)
==25673== D1  miss rate:      28.5% (     28.5%     +  20.4%  )
==25673== LLd miss rate:       0.0% (      0.0%     +   2.1%  )
```

Set Associative Cache

- Consider a Direct Mapped Cache with
  - 1024 Bytes, 32 byte cache line
  - 2 way set-associative
- Number of Cache Lines = 1024/32 = 32 (5 bits)
- Number of sets = 32/2 = 16 (4 bits)
- Assume memory address is of 32 bits



- For ex: Address = 0x12345678
  - Offset : $(11000)_2$
  - Set: $(0011)_2$

2-way Cache Prevents Thrashing

```
==25673== I   refs:      8,474,937
==25673== I1  misses:          591
==25673== LLi misses:          586
==25673== I1  miss rate:      0.00%
==25673== LLi miss rate:      0.00%
==25673==
==25673== D   refs:      7,373,749  (7,364,175 rd   + 9,574 wr)
==25673== D1  misses:    2,106,127  (2,104,167 rd   + 1,960 wr)
==25673== LLd misses:          819  (      611 rd   +   208 wr)
==25673== D1  miss rate:      28.5% (     28.5%     +  20.4%  )
==25673== LLd miss rate:       0.0% (      0.0%     +   2.1%  )
```

Direct Mapped

```
==7415== I   refs:      8,474,973
==7415== I1  misses:          591
==7415== LLi misses:          586
==7415== I1  miss rate:      0.00%
==7415== LLi miss rate:      0.00%
==7415==
==7415== D   refs:      7,373,760  (7,364,183 rd   + 9,577 wr)
==7415== D1  misses:        8,406  (    6,625 rd   + 1,781 wr)
==7415== LLd misses:          819  (      614 rd   +   205 wr)
==7415== D1  miss rate:       0.1% (      0.0%     +  18.5%  )
==7415== LLd miss rate:       0.0% (      0.0%     +   2.1%  )
```

2-way set associative

Traversal for Large Matrices

```
void traverse(unsigned long long m[][N])
{
	int i,j;

	for(i=0; i<N; ++i){
		for(j=0; j<N; ++j){
			m[i][j] = <something>
		}
	}
}
```

- ROW MAJOR
- Miss Rate/Iteration: 8/B

```
void traverse(unsigned long long m[][N])
{
	int i,j;

	for(i=0; i<N; ++i){
		for(j=0; j<N; ++j){
			m[j][i] = <something>
		}
	}
}
```

- COLUMN MAJOR
- Miss Rate/Iteration: 1

Matrix Multiplication Example

- We need to multiply C = A*B

```
#define N 128

unsigned long long A[N][N];
unsigned long long B[N][N];
unsigned long long C[N][N];

void mul_ijk()
{
        int i, j, k;
        register sum;

        for (i=0; i<N; ++i){
                for(j=0; j<N; ++j){
                        sum = 0;
                        for(k=0; k<N; ++k){
                                sum += (A[i][k] * B[k][j]);
                        }
                        C[i][j] = sum;
                }
        }
}
```

Matrix A is accessed in Row Major
Matrix B is accessed in Column Major

Analysis of Matrix Multiplication

```
D   refs:        280,270,794  (276,327,182 rd   + 3,943,612 wr)
D1  misses:      169,747,024  (169,261,262 rd   +   485,762 wr)
LLd misses:           99,144  (        630 rd   +    98,514 wr)
D1  miss rate:         60.5% (        61.2%     +      12.3%  )
LLd miss rate:          0.0% (         0.0%     +       2.4%  )
```

- Huge miss rate because B is accessed in column major fashion.
- So, each access to B results in a cache miss.
- A solution, is to find B transpose, then only row major traversal is required.

Matrix Multiplication (Naïve Transpose)

```
void mul_ijk()
{
        int i, j, k;
        register sum;

        transpose1(B);
        for (i=0; i<N; ++i){
                for(j=0; j<N; ++j){
                        sum = 0;
                        for(k=0; k<N; ++k){
                                sum += (A[i][k] * B[j][k]);
                        }
                        C[i][j] = sum;
                }
        }
}
```

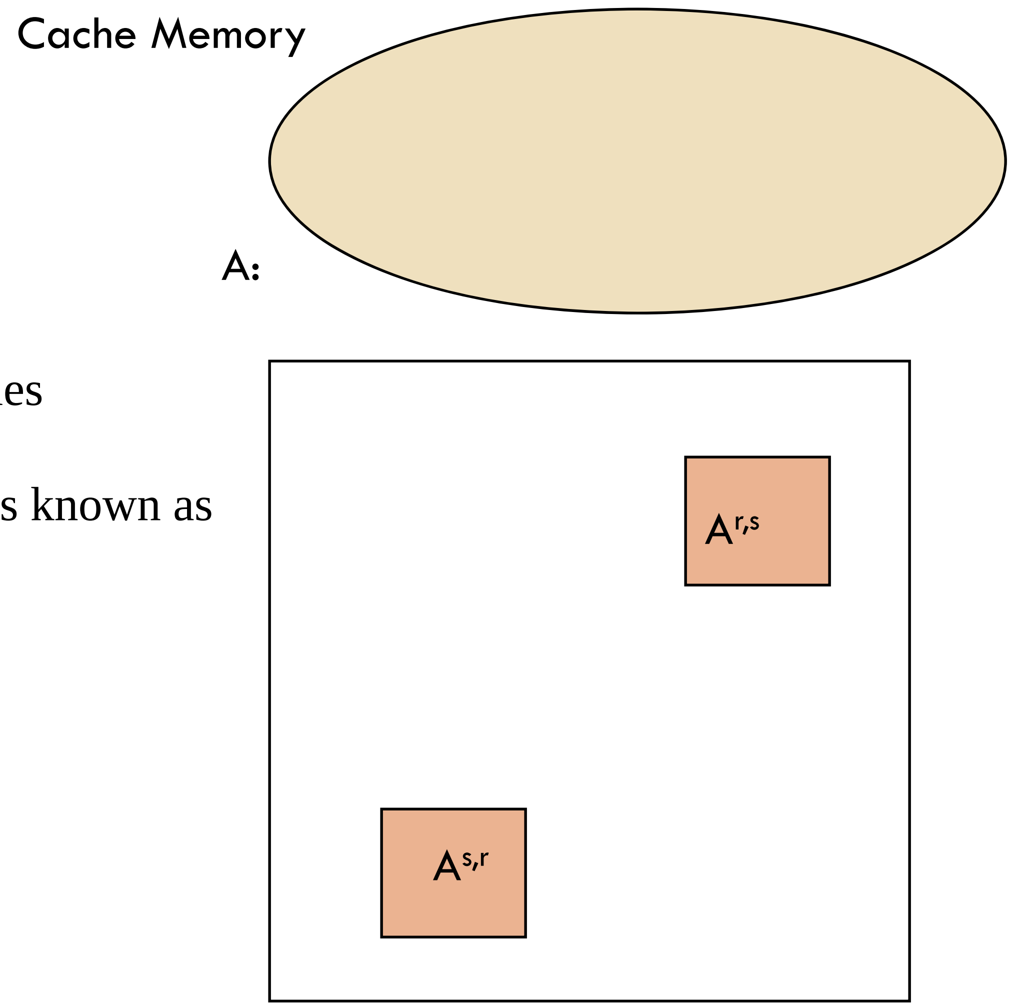
```
void transpose1(unsigned long long B[][N])
{
        int i, j;
        unsigned long long tmp;

        for (i=0; i < N; ++i){
                for(j=i+1; j<N; ++j){
                        tmp = B[i][j];
                        B[i][j] = B[j][i];
                        B[j][i] = tmp;
                }
        }
}
```

```
==8744== D   refs:      5,002,950  (4,729,484 rd   + 273,466 wr)
==8744== D1  misses:    2,287,416  (2,254,114 rd   +  33,302 wr)
==8744== LLd misses:        6,981  (      631 rd   +   6,350 wr)
==8744== D1  miss rate:      45.7% (     47.6%     +    12.1%  )
==8744== LLd miss rate:       0.1% (      0.0%     +     2.3%  )
```

Reduction in number of misses by a factor of almost 98%

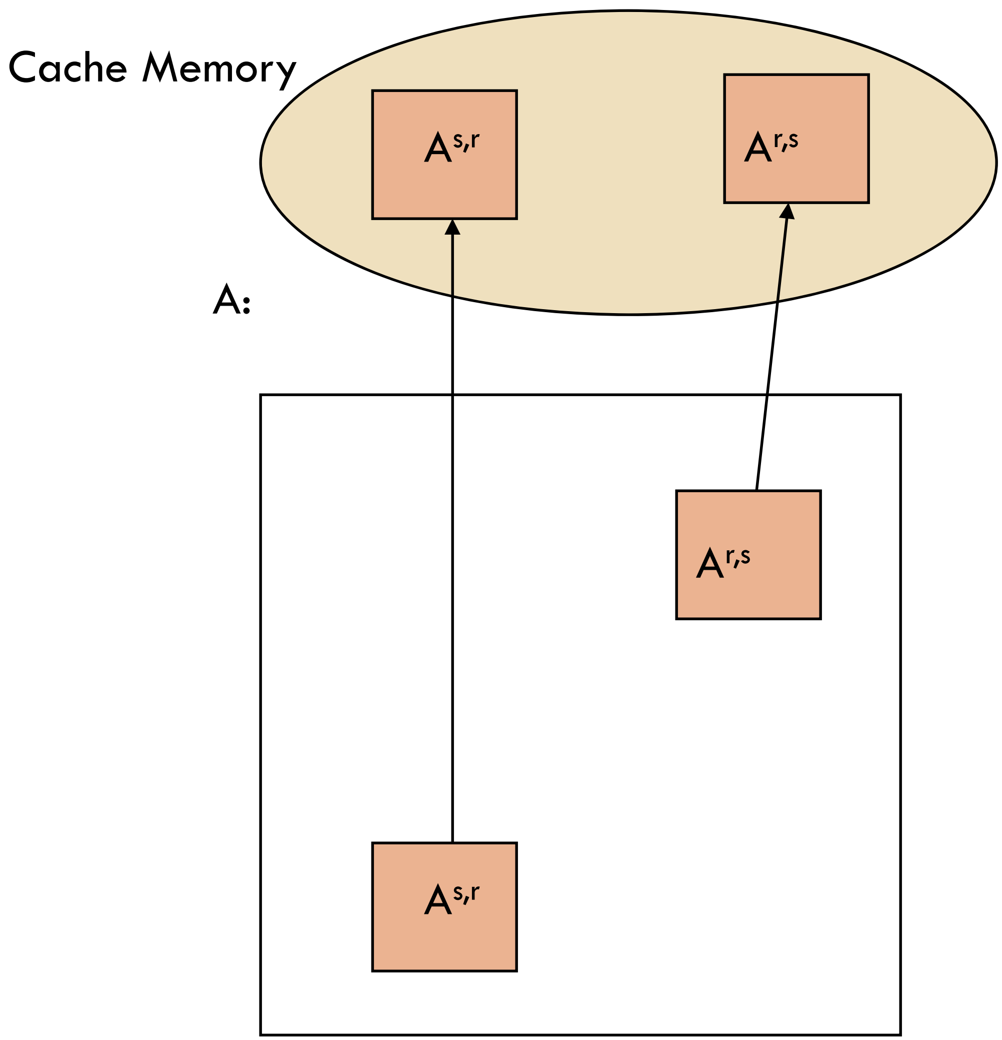
A Better Transpose



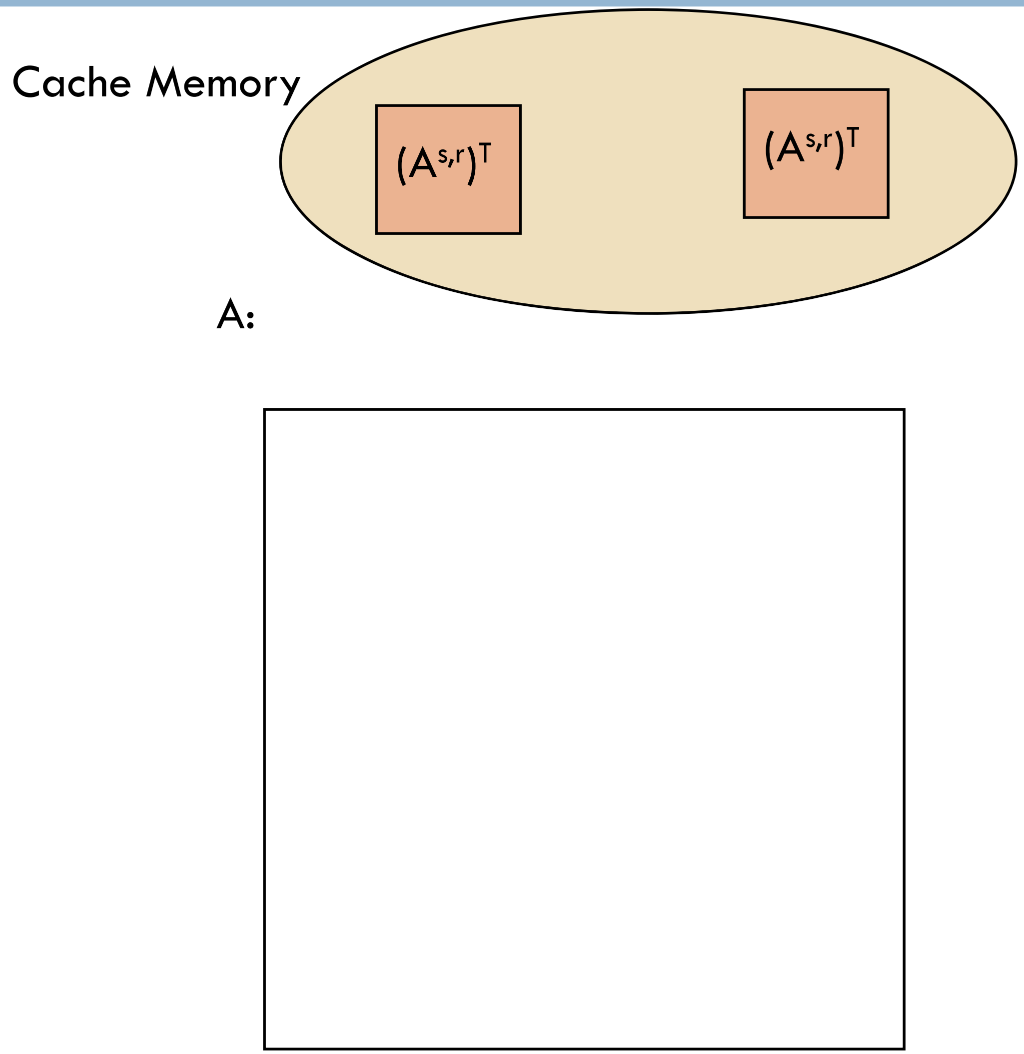
Partition the Matrix into Tiles

Tile - Each sub-matrix $A^{r,s}$ is known as tile.

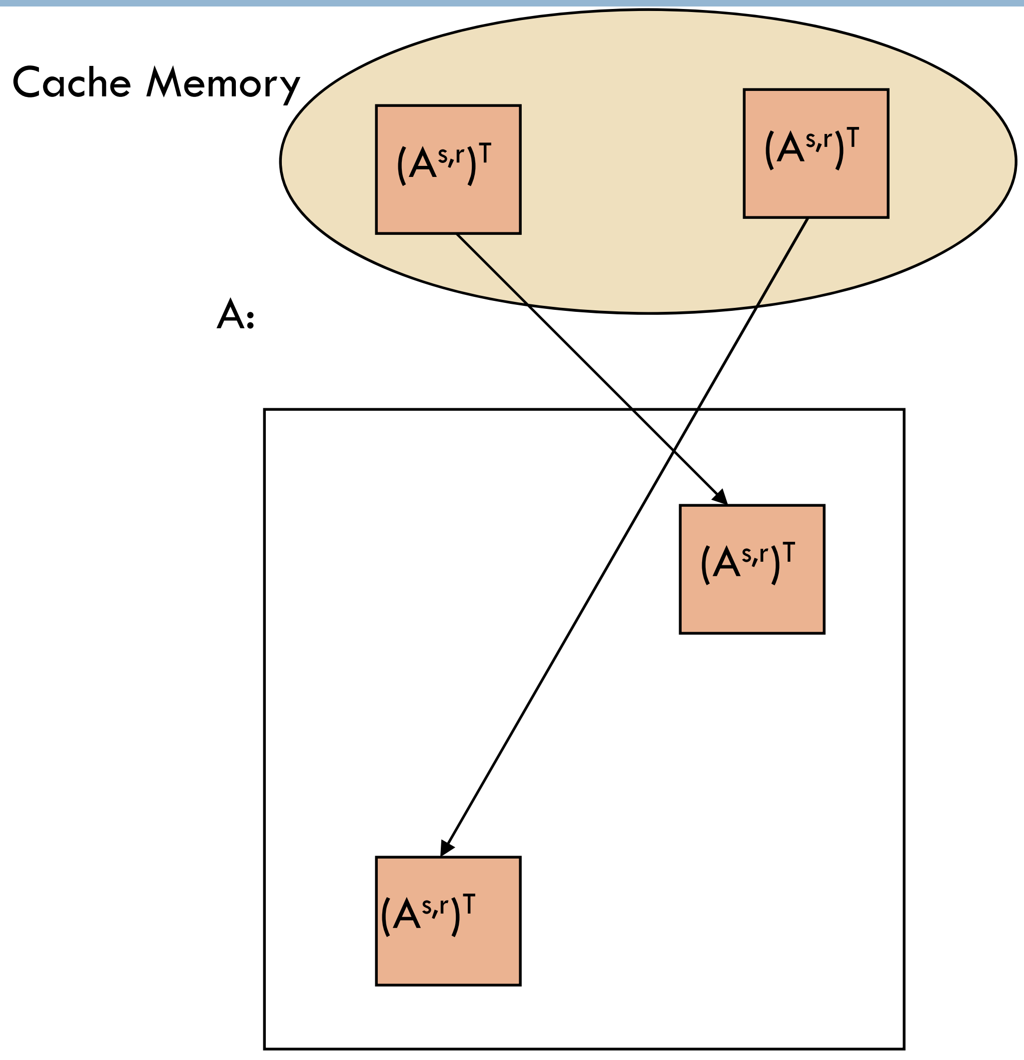
A Better Transpose (load)



A Better Transpose (transpose)



A Better Transpose (transfer)



Cache Oblivious Algorithms

- An algorithm designed to take advantage of a CPU cache without explicit knowledge the cache parameters.

- New branch of algorithm design.

- Optimal Cache-oblivious algorithms are known for the
  - Cooley-Tukey FFT algorithm
  - Matrix Multiplication
  - Sorting
  - Matrix Transposition

Summary for Cachegrind

- Easy to use tool to analyze cache memory behavior for various configurations
- Slow, around 20x to 100x slower than normal.
- What you *simulate* is not what you may *get !*
- What is needed is a way to analyze software at run-time

Related vs Unrelated Memory Accesses

Related Data Accesses

```
res ^= T1[t1];
res ^= T2[res & 0x3F];
res ^= T3[res & 0x3F];
res ^= T4[res & 0x3F];
res ^= T5[res & 0x3F];
res ^= T6[res & 0x3F];
res ^= T7[res & 0x3F];
res ^= T8[res & 0x3F];
res ^= T9[res & 0x3F];
res ^= T10[res & 0x3F];
res ^= T11[res & 0x3F];
res ^= T12[res & 0x3F];
res ^= T13[res & 0x3F];
res ^= T14[res & 0x3F];
res ^= T15[res & 0x3F];
res ^= T16[res & 0x3F];
```

Unrelated Data Accesses

```
res ^= T1[t1 & 0x3F];
res ^= T2[t2 &  0x3F];
res ^= T3[t3 & 0x3F];
res ^= T4[t4 & 0x3F];
res ^= T5[t5 & 0x3F];
res ^= T6[t6 & 0x3F];
res ^= T7[t7 & 0x3F];
res ^= T8[t8 & 0x3F];
res ^= T9[t9 & 0x3F];
res ^= T10[t10 & 0x3F];
res ^= T11[t11 & 0x3F];
res ^= T12[t12 & 0x3F];
res ^= T13[t13 & 0x3F];
res ^= T14[t14 & 0x3F];
res ^= T15[t15 & 0x3F];
res ^= T16[t16 & 0x3F];
```

Time(Related Data Access) =
Five x Time(Unrelated Data Accesses)

Vtune

- Vtune is an tool for real-time performance analysis of software.
- Unlike Valgrind has less overhead.
- Uses MSRs : Model Specific Performance-Monitoring Counters
  - Model Specific because MSRs for one processor may not be compatible with another
- There are two banks of registers :
  - IA32_PERFEVTSELx : Performance event select MSRs
  - IA32_PMCx : Performance monitoring event counters

References

- Valgrind website : http://**valgrind**.org/
- Intel, Vtune : http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/
- Igor Ostrovosky, Gallary of Cache Effects : http://igoro.com/archive/gallery-of-processor-cache-effects/
- Siddhartha Chatterjee and Sandeep Sen , Cache Friendly Matrix Transposition

Thank You