Understanding Application Memory Performance
Ulrich Drepper
Consulting Engineer, Red Hat
Why Singling Out Memory?

- Speed of Computer Main Memory does not keep up

<table>
<thead>
<tr>
<th></th>
<th>Clock</th>
<th>Memory Access</th>
<th>Effective Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>early 80s</td>
<td>1MHz</td>
<td>1 cycle</td>
<td>1 MHz</td>
</tr>
<tr>
<td>today</td>
<td>4GHz</td>
<td>250 cycles</td>
<td>16 Mhz</td>
</tr>
</tbody>
</table>

- Memory cannot get much faster, latency-wise
  Energy = Capacity \cdot Voltage^2 \cdot Frequency

- Increased competition for memory connection due to many-core processors
Why Is Memory Performance Optimization Hard?

- Memory technology not well understood
- There are so many places where memory is accessed
- Effects not local
  - Entire program should be understood for best results
  - Other processes can have effects, too
- Hardware Complications
  - Multi-core
  - NUMA
Important Factors

- Cache Line Utilization
- Memory Page Utilization
- TLB Branch Utilization
- Avoid just-in-time reading:
  - Help hardware prefetching
  - Use explicit software prefetching
- Parallelism
  - Concurrent cache-line use
  - Frequent cache-line transfer
- Non-local access
An Example: Matrix Multiplication

for (size_t i = 0; i < X; ++i)
  for (size_t j = 0; j < Z; ++j)
    for (size_t k = 0; k < Y; ++k)
      res[i][j] += mul1[i][k] * mul2[k][j];

Both matrixes have size 2048x2048
• 8,589,934,592 multiplications and additions
• 3GHz Intel Core2
• Runtime: 678 sec!
• 12,669,520 FLOPS
Measure!

Oprofile: statistical profiling

- Use hardware performance counters (10 sec each)

<table>
<thead>
<tr>
<th>Event</th>
<th>Count</th>
<th>Event</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_CLK_UNHALTED</td>
<td>5,302,632,000</td>
<td>L1D_REPL</td>
<td>183,174,500</td>
</tr>
<tr>
<td>INST RETIRED</td>
<td>435,096,000</td>
<td>L2_LINES_IN.ANY</td>
<td>240,435,000</td>
</tr>
<tr>
<td>RESOURCE_STALLS</td>
<td>1,886,790,000</td>
<td>L2_LINES_IN.DEMAND</td>
<td>126,758,500</td>
</tr>
<tr>
<td>IFU_MEM_STALL</td>
<td>262,414,500</td>
<td>PAGE_WALKS</td>
<td>154,154,000</td>
</tr>
<tr>
<td>ITLB_MISS_RETIRED</td>
<td>28,500</td>
<td>DTLB_MISSSES.ANY</td>
<td>118,965,000</td>
</tr>
<tr>
<td>L1I_MISSES</td>
<td>253,500</td>
<td>DTLB_MISSSES.MISS_LD</td>
<td>131,460,500</td>
</tr>
<tr>
<td>L2_IFETCH</td>
<td>20,000</td>
<td>DTLB_MISSSES.MISS_ST</td>
<td>24,500</td>
</tr>
<tr>
<td>L1D_CACHE_LD</td>
<td>139,074,500</td>
<td>L1D_CACHE_LD</td>
<td>177,222,500</td>
</tr>
<tr>
<td>STORE_BLOCK</td>
<td>141,500</td>
<td>L1D_CACHE_ST</td>
<td>122,000</td>
</tr>
</tbody>
</table>

What does each number mean?
Relativity

- Absolute numbers hard to interpret
- Create ratios (appendix B, Intel Optimization Manual)
- Ratios are independent of length of sampling
- No universal levels for ratios:
  - Memory-intensive code has more cache misses
  - Arithmetic-intensive code with have less, but more dependencies
Important Ratios

- Clocks per Instruction Retired
  \[ \frac{CPU_{\_}CLK_{\_}HALTED}{INST_{\_}RETIRED} \]
  In multi-scalar processors, optimum > 1

- Instruction Fetch Stall
  \[ \frac{CYCLES_{\_}L1I_{\_}MEM_{\_}STALLED}{CPU_{\_}CLK_{\_}HALTED} \]
  Any stall bad. Code should be predictable

- Virtual Table Use
  \[ \frac{BR_{\_}IND_{\_}CALL_{\_}EXEC}{INST_{\_}RETIRED} \]
  Possible reason for instruction fetch stalls: indirect calls
Important Ratios

- **Load Rate:**
  - \( \text{L1D\_CACHE\_LD.MESI/CPU\_CLK\_UNHALTED} \)
  - Large number of loads means load/store buffers full all the time

- **Store Order Block**
  - \( \text{STORE\_BLOCK.ORDER/CPU\_CLK\_UNHALTED} \)
  - Ratio of cycles in which instructions are held up because of write ordering due to cache misses

- **L1 Data Cache Miss Rate**
  - \( \text{L1D\_REPL/INST\_RETIRED} \)
  - How many instructions cause L1 cache misses
Important Ratios

• L2 Cache Miss Rate
  \[ \text{L2_LINES_IN/INST RETIRED} \]
  Instructions which cause L2 misses

• TLB Miss Penalty
  \[ \text{PAGE_WALKS/CPU_CLK_UNHALTED} \]
  Cycles spent waiting for page table walks

• DTLB Miss Rate
  \[ \text{DTLB_MISSES/INST RETIRED} \]
  Instructions which cause DTLB misses
## Ratios for the Example

- Some of the memory-related ratios:

<table>
<thead>
<tr>
<th>Ratio</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_CLK_UNHALTED/INST_RETIRED</td>
<td>12.19</td>
</tr>
<tr>
<td>RESOURCE_STALLS.RS_FULL/CPU_CLK_UNHALTED</td>
<td>77.36%</td>
</tr>
<tr>
<td>IFU_MEM_STALL/CPU_CLK_UNHALTED</td>
<td>8.84%</td>
</tr>
<tr>
<td>L1D_CACHE_LD/CPU_CLK_UNHALTED</td>
<td>0.03</td>
</tr>
<tr>
<td>L1D_REPL/INST_RETIRED</td>
<td>15.30%</td>
</tr>
<tr>
<td>L2_LINES_IN.ANY/INST_RETIRED</td>
<td>19.20%</td>
</tr>
<tr>
<td>L2_LINES_IN.DEMAND/INST_RETIRED</td>
<td>9.60%</td>
</tr>
<tr>
<td>PAGE_WALKS/CPU_CLK_UNHALTED</td>
<td>4.12%</td>
</tr>
<tr>
<td>DTLB_MISSES.MISS_LD/INST_RETIRED</td>
<td>9.90%</td>
</tr>
<tr>
<td>DTLB_MISSES.MISS_ST/INST_RETIRED</td>
<td>0.00%</td>
</tr>
<tr>
<td>L1D_CACHE_LD/INST_RETIRED</td>
<td>14.31%</td>
</tr>
<tr>
<td>L1D_CACHE_ST/INST_RETIRED</td>
<td>0.01%</td>
</tr>
</tbody>
</table>
Slightly Revised: Matrix Multiplication

```c
for (size_t k = 0; k < Y; ++k)
    for (size_t j = 0; j < Z; ++j)
        for (size_t i = 0; i < X; ++i)
            res[i][j] += mul1[i][k] * mul2[k][j];
```

• Now: 38 sec, 94% faster!
## Visible Improvement

<table>
<thead>
<tr>
<th>Metric</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_CLK_UNHALTED/INST RETIRED</td>
<td>1.4</td>
</tr>
<tr>
<td>RESOURCE_STALLS_RS_FULL/CPU_CLK_UNHALTED</td>
<td>8.87%</td>
</tr>
<tr>
<td>IFU_MEMSTALL/CPU_CLK_UNHALTED</td>
<td>8.97%</td>
</tr>
<tr>
<td>L1D_CACHE_LD/CPU_CLK_UNHALTED</td>
<td>0.26</td>
</tr>
<tr>
<td>L1D_CACHE_ST/INST RETIRED</td>
<td>0.71%</td>
</tr>
<tr>
<td>PAGE_WALKS/CPU_CLK_UNHALTED</td>
<td>0.53%</td>
</tr>
<tr>
<td>DTLB_MISSES.MISS_LD/INST RETIRED</td>
<td>0.03%</td>
</tr>
<tr>
<td>L1D_CACHE_LD/INST RETIRED</td>
<td>6.46%</td>
</tr>
<tr>
<td>L1D_CACHE_ST/INST RETIRED</td>
<td>0.71%</td>
</tr>
<tr>
<td>PAGE_WALKS/CPU_CLK_UNHALTED</td>
<td>0.53%</td>
</tr>
<tr>
<td>DTLB_MISSES.MISS_ST/INST RETIRED</td>
<td>0.02%</td>
</tr>
<tr>
<td>L2_LINES_IN.ANY/INST RETIRED</td>
<td>1.32%</td>
</tr>
<tr>
<td>L2_LINES_IN.DEMAND/INST RETIRED</td>
<td>0.08%</td>
</tr>
<tr>
<td>IFU_MEMSTALL/CPU_CLK_UNHALTED</td>
<td>8.97%</td>
</tr>
<tr>
<td>L1D_CACHE_LD/CPU_CLK_UNHALTED</td>
<td>0.26</td>
</tr>
<tr>
<td>L1D_CACHE_ST/INST RETIRED</td>
<td>0.71%</td>
</tr>
<tr>
<td>PAGE_WALKS/CPU_CLK_UNHALTED</td>
<td>0.53%</td>
</tr>
<tr>
<td>DTLB_MISSES.MISS_LD/INST RETIRED</td>
<td>0.03%</td>
</tr>
<tr>
<td>L1D_CACHE_LD/INST RETIRED</td>
<td>6.46%</td>
</tr>
<tr>
<td>L1D_CACHE_ST/INST RETIRED</td>
<td>0.71%</td>
</tr>
</tbody>
</table>

**Improvement**

- 88.56%
- 88.54%
- 1.47%
- 814.29%
- 87.19%
- 99.75%
- 99.17%
- 87.19%
- 99.75%
- 2200.00%
- 54.88%
- 7010.00%
Use Huge Pages

- mount hugetlbfs at /mnt/huge
- Use mmap with file descriptor for file under /mnt/huge

<table>
<thead>
<tr>
<th>Metric</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_CLK_UNHALTED/INST_RETIRED</td>
<td>1.32</td>
</tr>
<tr>
<td>DTLB_MISSES.MISS_LD/INST_RETIRED</td>
<td>0.02%</td>
</tr>
<tr>
<td>DTLB_MISSES.MISS_ST/INST_RETIRED</td>
<td>0.02%</td>
</tr>
</tbody>
</table>

5.73% 8.00% 4.35%
Tiling

Fill in entire cache lines before they are evicted:

```c
#define SM (64 / sizeof (double))
for (i = 0; i < X; i += SM)
    for (j = 0; j < Z; j += SM)
        for (k = 0; k < Y; k += SM)
            for (i2 = 0, rres = &RES(i, j), rmul1 = &MUL1(i, k); i2 < SM;
                 ++i2, rres += Y, rmul1 += X)
            for (k2 = 0, rmul2 = &MUL2(k, j); k2 < SM; ++k2, rmul2 += Z)
                for (j2 = 0; j2 < SM; ++j2)
                    rres[j2] += rmul1[k2] * rmul2[j2];
```
Tiling can help significantly

<table>
<thead>
<tr>
<th>Metric</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_CLK_UNHALTED/INST RETIRED</td>
<td>1.29</td>
</tr>
<tr>
<td>RESOURCESTALLS.RS_FULL/CPU_CLK_UNHALTED</td>
<td>8.87%</td>
</tr>
<tr>
<td>IFU_MEM_STALL/CPU_CLK_UNHALTED</td>
<td>8.34%</td>
</tr>
<tr>
<td>L1D_CACHE_LD/CPU_CLK_UNHALTED</td>
<td>0.23</td>
</tr>
<tr>
<td>L1D_REPL/INST RETIRED</td>
<td>1.32%</td>
</tr>
<tr>
<td>L2_LINES_IN.ANY/INST RETIRED</td>
<td>0.90%</td>
</tr>
<tr>
<td>L1D_CACHE_LD/INST RETIRED</td>
<td>4.58%</td>
</tr>
<tr>
<td>L1D_CACHE_ST/INST RETIRED</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
Where is Time Spent?

- It's simple if looking at the code is sufficient
  \[ \text{res}[i][j] += \text{mul1}[i][k] \times \text{mul2}[k][j]; \]
- Use oprofile and observe location of events
- Select all interesting counters with opcontrol
- opannotate –source
  - Show all counters next to each line
- Opannotate –assembly
  - Show next to assembler instructions
  - Not precise since PEBS is not supported!
### Annotated Listing

<table>
<thead>
<tr>
<th>Function</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache Load</td>
<td>11058 2.5025 111 0.5398</td>
</tr>
<tr>
<td></td>
<td>2 4.5e-04 0 0</td>
</tr>
<tr>
<td>L2 Cache Load</td>
<td>84957 19.2266 1693 8.2324</td>
</tr>
<tr>
<td></td>
<td>11053 2.5014 1138 5.5337</td>
</tr>
<tr>
<td></td>
<td>6245 1.4133 340 1.6533</td>
</tr>
<tr>
<td></td>
<td>2397 0.5425 1454 7.0703</td>
</tr>
<tr>
<td></td>
<td>3568 0.8075 292 1.4199</td>
</tr>
<tr>
<td></td>
<td>30993 7.0140 2634 12.8082</td>
</tr>
</tbody>
</table>

```c
unsigned long int hash;
size_t idx;

hash_entry *table = (hash_entry *) htab->table;

hash = 1 + hval % htab->size;
idx = hash;

if (table[idx].used) {
    if (table[idx].used==hval && table[idx].keylen == keylen && memcmp (table[idx].key, key, keylen) == 0)
        return idx;
    hash = 1 + hval % (htab->size - 2);
    do {
        if (idx <= hash)
            idx = htab->size + idx - hash;
        else
            idx -= hash;
    } while (table[idx].used==hval&&table[idx].keylen==keylen
```
Problems of Parallelism

- False sharing of cache lines:
  - Unintentionally use same cache line in different threads
  - Happens with global variables
  - Should not happen that often with dynamic memory
  - Group variables and align them

- Common working set:
  - Multiple threads working on same data (good!)
  - Produced output placed in same memory location (bad!)
  - Use per-thread working area and consolidate in end

- Synchronization:
  - Highly contested cache lines for sync primitives
Ratios for Multi-Thread Problems

- Modified Data Sharing Ratio:
  \[
  \frac{\text{EXT\_SNOOP/INST\_RETIRED}}{}
  \]
  Instructions which cause modified cache line from other core to be retrieved

- Locked Operations Impact:
  \[
  \frac{(\text{L1D\_CACHE\_LOCK\_DURATION} + 20 \times \text{L1D\_CACHE\_LOCK})}{\text{CPU\_CLK\_UNHALTED}}
  \]
  How many cycles used for atomic operations. Should be near zero
Summary

- There are many layers to memory performance
- Each program has different characteristics
- Statistical profiling can
  - Give general overview
  - Pinpoint hotspots
- Often program logic has to be significantly rethought
Backup Slides
Common Memory Performance Problems

- Higher-level language semantics for strings
  - String implementation cannot guess what is needed
  - Programmer should provide estimates to C++ class
- In general: temporary values
C++ Temporaries

```c
vec operator+(vec a, vec b) {
    vec r;
    for (size_t c = 0; c < a.size; ++c)
        r.data[c] = a.data[c] + b.data[c];
    return r;
}
```

Good style, bad performance
C++ Temporaries Help in C++0x

```cpp
vec &&operator+(vec &&a, vec &&b) {
    for (size_t c = 0; c < a.size; ++c)
        a.data[c] += b.data[c];
    return a;
}
```

- && indicates rvalue reference, i.e., reference to temporary
- Can be reused for result instead of new allocation