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
  
  \[ \text{Energy} = \text{Capacity} \cdot \text{Voltage}^2 \cdot \text{Frequency} \]

- 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
Overview

Request

Not here

Nor here Physical Address

Send Request

Send Data

Read Command

Prepare Data

Use Data

Store Data

Check here

Determine DIMM(s)

And here

Memory Controller

Memory

Memory

Memory

Memory

Memory

Memory

Memory

Memory
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_MEMSTALL</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_MISSES.ANY</td>
<td>118,965,000</td>
</tr>
<tr>
<td>L1I_MISSES</td>
<td>253,500</td>
<td>DTLB_MISSES.MISS_LD</td>
<td>131,460,500</td>
</tr>
<tr>
<td>L2_IFETCH</td>
<td>20,000</td>
<td>DTLB_MISSES.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
  \[ \frac{\text{CPU_CLK_HALTED}}{\text{INST_RETIRED}} \]
  In multi-scalar processors, optimum > 1

- Instruction Fetch Stall
  \[ \frac{\text{CYCLES_L1I_MEM_STALLED}}{\text{CPU_CLK_HALTED}} \]
  Any stall bad. Code should be predictable

- Virtual Table Use
  \[ \frac{\text{BR_IND_CALL_EXEC}}{\text{INST_RETIRED}} \]
  Possible reason for instruction fetch stalls: indirect calls
Important Ratios

• Load Rate:
  \[ \frac{L1D\_CACHE\_LD\_MESI}{CPU\_CLK\_UNHALTED} \]
  Large number of loads means load/store buffers full all the time

• Store Order Block
  \[ \frac{STORE\_BLOCK\_ORDER}{CPU\_CLK\_UNHALTED} \]
  Ratio of cycles in which instructions are held up because of write ordering due to cache misses

• L1 Data Cache Miss Rate
  \[ \frac{L1D\_REPL}{INST\_RETIRED} \]
  How many instructions cause L1 cache misses
Important Ratios

- **L2 Cache Miss Rate**
  
  $\frac{L2_{-LINES\_IN}}{INST\_RETIRED}$
  
  Instructions which cause L2 misses

- **TLB Miss Penalty**

  $\frac{PAGE\_WALKS}{CPU\_CLK\_UNHALTED}$
  
  Cycles spent waiting for page table walks

- **DTLB Miss Rate**

  $\frac{DTLB\_MISSES}{INST\_RETIRED}$
  
  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_MEMSTALL / 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% 88.54%</td>
</tr>
<tr>
<td>IFU_MEM_STALL/CPU_CLK_UNHALTED</td>
<td>8.97% -1.47%</td>
</tr>
<tr>
<td>L1D_CACHE_LD/CPU_CLK_UNHALTED</td>
<td>0.26 -814.29%</td>
</tr>
<tr>
<td>L1D_REPL/INST RETIRED</td>
<td>15.39% -0.59%</td>
</tr>
<tr>
<td>L2_LINES_IN.ANY/INST RETIRED</td>
<td>1.32% 93.15%</td>
</tr>
<tr>
<td>L2_LINES_IN.DEMAND/INST RETIRED</td>
<td>0.08% 99.17%</td>
</tr>
<tr>
<td>PAGE_WALKS/CPU_CLK_UNHALTED</td>
<td>0.53% 87.19%</td>
</tr>
<tr>
<td>DTLB_MISSES.MISS_LD/INST RETIRED</td>
<td>0.03% 99.75%</td>
</tr>
<tr>
<td>DTLB_MISSES.MISS_ST/INST RETIRED</td>
<td>0.02% -2200.00%</td>
</tr>
<tr>
<td>L1D_CACHE_LD/INST RETIRED</td>
<td>6.46% 54.88%</td>
</tr>
<tr>
<td>L1D_CACHE_ST/INST RETIRED</td>
<td>0.71% -7010.00%</td>
</tr>
</tbody>
</table>
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>
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];
```
Improvement

<table>
<thead>
<tr>
<th>Metric</th>
<th>Improvement</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_CLK_UNHALTED/INST RETIRED</td>
<td>1.29</td>
<td>7.67%</td>
</tr>
<tr>
<td>RESOURCE_STALLS_RS_FULL/CPU_CLK_UNHALTED</td>
<td>8.87%</td>
<td>0.00%</td>
</tr>
<tr>
<td>IFU_MEM_STALL/CPU_CLK_UNHALTED</td>
<td>8.34%</td>
<td>7.02%</td>
</tr>
<tr>
<td>L1D_CACHE_LD/CPU_CLK_UNHALTED</td>
<td>0.23</td>
<td>11.72%</td>
</tr>
<tr>
<td>L1D_REPL/INST RETIRED</td>
<td>1.32%</td>
<td>91.44%</td>
</tr>
<tr>
<td>L2_LINES_IN.ANY/INST RETIRED</td>
<td>0.90%</td>
<td>31.84%</td>
</tr>
<tr>
<td>L1D_CACHE_LD/INST RETIRED</td>
<td>4.58%</td>
<td>29.03%</td>
</tr>
<tr>
<td>L1D_CACHE_ST/INST RETIRED</td>
<td>0.00%</td>
<td>99.44%</td>
</tr>
</tbody>
</table>

Tiling can help significantly.
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!
<table>
<thead>
<tr>
<th>Function</th>
<th>Total</th>
<th>L1 Cache Load</th>
<th>L2 Cache Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotated Listing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11058</td>
<td>2.5025</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5398</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4.5e-04</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>84957</td>
<td>19.2266</td>
<td>1693</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.2324</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11053</td>
<td>2.5014</td>
<td>1138</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.5337</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6245</td>
<td>1.4133</td>
<td>340</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.6533</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2397</td>
<td>0.5425</td>
<td>1454</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.0703</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3568</td>
<td>0.8075</td>
<td>292</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.4199</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30993</td>
<td>7.0140</td>
<td>2634</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.8082</td>
<td></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);
} else
```
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:
  
  \[
  \text{EXT\_SNOOP/INST\_RETIRED}
  \]
  
  Instructions which cause modified cache line from other core to be retrieved

• Locked Operations Impact:
  
  \[
  \frac{(L1D\_CACHE\_LOCK\_DURATION+20\times L1D\_CACHE\_LOCK)}{CPU\_CLK\_UNHALTED}
  \]
  
  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
Questions?

drepper@redhat.com | people.redhat.com/drepper