Memory Technology

Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se
Main memory characteristics

Performance of main memory (from 3rd Ed... faster today)

- **Access time**: time between address is latched and data is available (~50ns)
- **Cycle time**: time between requests (~100 ns)
- **Total access time**: from Id to REG valid (~150ns)

- Main memory is built from **DRAM**: Dynamic RAM
- 1 transistor/bit ==> more error prune and slow
- Refresh and precharge
- Cache memory is built from **SRAM**: Static RAM
  - about 4-6 transistors/bit
The address is multiplexed Row/Address Strobe (RAS/CAS)
“Thin” organizations (between x16 and x1) to decrease pin load
Refresh of memory cells decreases bandwidth
Bit-error rate creates a need for error-correction (ECC)
SRAM organization

- Address is typically not multiplexed
- Each cell consists of about 4-6 transistors
- Wider organization (x18 or x36), typically few chips
- Often parity protected (ECC becoming more common)
Error Detection and Correction

Error-correction and detection
- E.g., 64 bit data protected by 8 bits of ECC
  - Protects DRAM and high-availability SRAM applications
  - Double bit error detection (“crash and burn”)
  - Chip kill detection (all bits of one chip stuck at all-1 or all-0)
  - Single bit correction
  - Need “memory scrubbing” in order to get good coverage

Parity
- E.g., 8 bit data protected by 1 bit parity
  - Protects SRAM and data paths
  - Single-bit “crash and burn” detection
  - Not sufficient for large SRAMs today!!
Correcting the Error

- Correction on the fly by hardware
  - no performance-glitch
  - great for cycle-level redundancy
  - fixes the problem for now...

- Trap to software
  - correct the data value and write back to memory

- Memory scrubber
  - kernel process that periodically touches all of memory
Improving main memory performance

- Page-mode => faster access within a small distance
- Improves bandwidth per pin -- not time to critical word
- Single wide bank improves access time to the complete CL
- Multiple banks improves bandwidth
Newer kind of DRAM...

- SDRAM (5-1-1-1 @100 MHz)
  - Mem controller provides strobe for next seq. access
- DDR-DRAM (5-½-½-½)
  - Transfer data on both edges
- RAMBUS
  - Fast unidirectional circular bus
  - Split transaction addr/data
  - Each DRAM devices implements RAS/CAS/refresh... internally
- CPU and DRAM on the same chip?? (IMEM)...
Newer DRAMs ...
(Several DRAM arrays on a die)

<table>
<thead>
<tr>
<th>Name</th>
<th>Clock rate (MHz)</th>
<th>BW (GB/s per DIMM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDR-260</td>
<td>133</td>
<td>2,1</td>
</tr>
<tr>
<td>DDR-300</td>
<td>150</td>
<td>2,4</td>
</tr>
<tr>
<td>DDR2-533</td>
<td>266</td>
<td>4,3</td>
</tr>
<tr>
<td>DDR2-800</td>
<td>400</td>
<td>6,4</td>
</tr>
<tr>
<td>DDR3-1066</td>
<td>533</td>
<td>8,5</td>
</tr>
<tr>
<td>DDR3-1600</td>
<td>800</td>
<td>12,8</td>
</tr>
</tbody>
</table>
Modern DRAM (1)

From AnandTech:
Everything You Always Wanted to Know About SDRAM: But Were Afraid to Ask
http://www.anandtech.com/show/3851/everything-you-always-wanted-to-know-about-sdram-memory-but-were-afraid-to-ask
Figure 6. Page-hit timing (with precharge and subsequent bank access)

From AnandTech:
Everything You Always Wanted to Know About SDRAM: But Were Afraid to Ask
http://www.anandtech.com/show/3851/everything-you-always-wanted-to-know-about-sdram-memory-but-were-afraid-to-ask
Timing “page miss”

Figure 8. Page-miss timing

From AnandTech:
Everything You Always Wanted to Know About SDRAM: But Were Afraid to Ask
http://www.anandtech.com/show/3851/everything-you-always-wanted-to-know-about-sdram-memory-but-were-afraid-to-ask
The Endian Mess

Numbering the bytes

Big Endian

Store the value 0x5F

Store the string Hello

Little Endian
Virtual Memory System

Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se
Physical Memory

PROGRAM

Physical Memory
0
64MB

Disk
Virtual and Physical Memory
Translation & Protection

Virtual Memory

Disk

Physical Memory

4GB

Context A

4GB

Context B

4GB

Context A

4GB

Context B

Dept of Information Technology | www.it.uu.se © Erik Hagersten | http://user.it.uu.se/~eh
Virtual memory — parameters

Compared to first-level cache parameters

- Replacement in cache handled by HW. Replacement in VM handled by SW
- VM hit latency very low (often zero cycles)
- VM miss latency huge (several kinds of misses)
- Allocation size is one “page” 4kB and up

<table>
<thead>
<tr>
<th>Parameter</th>
<th>First-level cache</th>
<th>Virtual memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block (page) size</td>
<td>16-128 bytes</td>
<td>4K-64K bytes</td>
</tr>
<tr>
<td>Hit time</td>
<td>1-2 clock cycles</td>
<td>40-100 clock cycles</td>
</tr>
<tr>
<td>Miss penalty</td>
<td>8-100 clock cycles</td>
<td>700K-6000K clock cycles</td>
</tr>
<tr>
<td>(Access time)</td>
<td>(6-60 clock cycles)</td>
<td>(500K-4000K clock cycles)</td>
</tr>
<tr>
<td>(Transfer time)</td>
<td>(2-40 clock cycles)</td>
<td>(200K-2000K clock cycles)</td>
</tr>
<tr>
<td>Miss rate</td>
<td>0.5%-10%</td>
<td>0.00001%-0.001%</td>
</tr>
<tr>
<td>Data memory size</td>
<td>16 Kbyte - 1 Mbyte</td>
<td>16 Mbyte - 8 Gbyte</td>
</tr>
</tbody>
</table>
VM: Block placement

Where can a block (page) be placed in main memory?
What is the organization of the VM?

- The high miss penalty makes SW solutions to implement a **fully associative address mapping** feasible at page faults
- A page from disk may occupy any pageframe in PA
- Some restriction can be helpful (page coloring)
Use a page table stored in main memory:

- Suppose 8 Kbyte pages, 48 bit virtual address
- Page table occupies $2^{48}/2^{13} \times 4B = 2^{37} = 128GB!!!$

Solutions:

- Only one entry per physical page is needed
- Multi-level page table (dynamic)
- Inverted page table (~hashing)
Address translation

- Multi-level table: The Alpha 21064

Segment is selected by bit 62 & 63 in addr.

- **kseg**
  - **Kernel segment**
  - Used by OS.
  - Does not use virtual memory.

- **seg0**
  - **User segment 0**
  - Used for instr. & static data & heap

- **seg1**
  - **User segment 1**
  - Used for stack.

---

Page Table Entry:
(translation & protection)
Protection mechanisms

The address translation mechanism can be used to provide memory protection:

- Use **protection attribute bits** for each page
- Stored **in the page table entry** (PTE) (and TLB...)
- Each physical page gets its own **per process protection**
- **Violations** detected during the address translation **cause exceptions** (i.e., SW trap)
- **Supervisor/user modes** necessary to prevent user processes from changing e.g. PTEs
Fast address translation

How can we avoid three extra memory references for each original memory reference?

- Store the most commonly used address translations in a cache—*Translation Look-aside Buffer* (TLB)

===> *The caches rears their ugly faces again!*

![Diagram of address translation process]
Do we need a fast TLB?

- Why do a TLB lookup for every L1 access?
- Why not cache virtual addresses instead?
  - Move the TLB on the other side of the cache
  - It is only needed for finding stuff in Memory anyhow
  - The TLB can be made larger and slower – or can it?
### Aliasing Problem

The same physical page may be accessed using different virtual addresses:

- A virtual cache will cause confusion -- a write by one process may not be observed.
- Flushing the cache on each process switch is slow (and may only help partly).
- => VIPT (Virtually Indexed Physically Tagged) is the answer:
  - Direct-mapped cache no larger than a page.
  - No more sets than there are cache lines on a page + logic.
  - Page coloring can be used to guarantee correspondence between more PA and VA bits (e.g., Sun Microsystems).
Virtually Indexed Physically Tagged = VIPT

Have to guarantee that all aliases have the same index
- L1_cache_size < (page-size * associativity)
- Page coloring can help further
Putting it all together: VIPT

Cache: 8kB, 2-way, CL=32B, word=4B, page =4kB
TLB: 32 entries, 2-way

Identifies the word within a cache line

Identifies a byte within a word

VA-TAG (16)
PA-Page frame (20)

same for PA & VA

Identifies the word within a cache line

Identifies a byte within a word

TLB hit
Cache hit

Multiplexer (16:1 mux)
What is the capacity of the TLB

Typical TLB size = 0.5 - 2kB
Each translation entry 4 - 8B ==> 32 - 500 entries
Typical page size = 4kB - 16kB
\textbf{TLB-reach} = 0.1MB - 8MB

\textbf{FIX:}

- \textit{Multiple page sizes, e.g., 8kB and 8 MB}
- \textit{TSB -- A direct-mapped translation in memory as a “second-level TLB”}
Most important: minimize number of page faults

Page replacement strategies:

- FIFO—First-In-First-Out
- LRU—Least Recently Used
- Approximation to LRU
  - Each page has a reference bit that is set on a reference
  - The OS periodically resets the reference bits
  - When a page is replaced, a page with a reference bit that is not set is chosen
So far...

![Diagram of CPU, TLB, TLB fill, TLB miss, Data L1$, Unified L2$, PF handler, Page fault, TLB fill, TLB miss, Data L1$]
Adding TSB (software TLB cache)
VM: Write strategy

Write back or Write through?

- **Write back**!
- Write through is impossible to use:
  - Too long access time to disk
  - The write buffer would need to be *prohibitively* large
  - The I/O system would need an extremely high bandwidth
VM dictionary

Virtual Memory System
Virtual address
Physical address
Page
Page fault
Page-fault handler
Page-out

The “cache” language
~Cache address
~Cache location
~Huge cache block
~Extremely painfull $miss
~The software filling the $
Write-back if dirty
Caches Everywhere...

- D cache
- I cache
- L2 cache
- L3 cache
- ITLB
- DTLB
- TSB
- Virtual memory system
- Branch predictors
- Directory cache
- ...

Dept of Information Technology | www.it.uu.se
© Erik Hagersten | http://user.it.uu.se/~eh
Exploring the Memory of a Computer System

Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se
for (times = 0; times < Max; times++) /* many times*/

    for (i=0; i < ArraySize; i = i + Stride)
        dummy = A[i]; /* touch an item in the array */

Measuring the average access time to memory, while varying ArraySize and Stride, will allow us to reverse-engineer the memory system.
(need to turn off HW prefetching...)
for (times = 0; times < Max; times++) /* many times*/

for (i=0; i < ArraySize; i = i + Stride)
    dummy = A[i]; /* touch an item in the array */
Stepping through the array

```
for (times = 0; times < Max; times++) /* many times*/
    for (i=0; i < ArraySize; i = i + Stride)
        dummy = A[i]; /* touch an item in the array */
```

- Array Size = 16, Stride=4
- Array Size = 16, Stride=8...
- Array Size = 32, Stride=4...
- Array Size = 32, Stride=8...
for (times = 0; times < Max; time++) /* many times*/

for (i=0; i < ArraySize; i = i + Stride)
    dummy = A[i]; /* touch an item in the array */
for (times = 0; times < Max; time++) /* many times*/
for (i=0; i < ArraySize; i = i + Stride)
dummy = A[i]; /* touch an item in the array */
Twice as large L2 cache ???

```c
for (times = 0; times < Max; time++) /* many times*/

for (i=0; i < ArraySize; i = i + Stride)
dummy = A[i]; /* touch an item in the array */
```

![Graph showing the relationship between ArraySize and Stride with average time in nanoseconds as a function of Stride (bytes). The graph includes multiple lines representing different ArraySize values: 8 MB, 512 kB, 32-256 kB, and 16 kB, with corresponding legends indicating the ArraySize values.]
Twice as large TLB...

```c
for (times = 0; times < Max; time++) /* many times*/
    for (i=0; i < ArraySize; i = i + Stride)
        dummy = A[i]; /* touch an item in the array */
```
Optimizing for Multicores

Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se
Optimizing for the memory system: What is the potential gain?

- Latency difference $L1$ and mem: $\sim 50x$
- Bandwidth difference $L1$ and mem: $\sim 20x$
- Execute from $L1$ instead from mem $\Rightarrow$ 50-150x improvement
- At least a factor 2-4x is within reach
Optimizing for cache performance

- Keep the active footprint small
- Use the entire cache line once it has been brought into the cache
- Fetch a cache line prior to its usage
- Let the CPU that already has the data in its cache do the job
- ...
Final cache lingo slide

- **Miss ratio**: What is the likelihood that a memory access will miss in a cache?
- **Miss rate**: D:o per time unit, e.g. per-second, per-1000-instructions
- **Fetch ratio/rate**: What is the likelihood that a memory access will cause a fetch to the cache [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 Acumem-ish language
What can go Wrong?
A Simple Example...

Perform a diagonal copy 10 times
Example: Loop order

//Optimized Example A
for (i=1; i<N; i++) {
    for (j=1; j<N; j++) {
        A[i][j] = A[i-1][j-1];
    }
}

//Unoptimized Example A
for (j=1; j<N; j++) {
    for (i=1; i<N; i++) {
        A[i][j] = A[i-1][j-1];
    }
}
Performance Difference: Loop order

- Athlon64 x2
- Pentium D
- Core 2 Duo

Array side

Speedup vs UnOpt

Demo Time!
ThreadSpotter
Example 1: The Same Application Optimized

Optimization can be rewarding, but costly...

- Require expert knowledge about MC and architecture
- Weeks of wading through performance data

► This fix required one line of code to change
Example: Sparse data usage

//Optimized Example A
for (i=1; i<N; i++) {
    for (j=1; j<N; j++) {
        A_d[i][j] = A_d[i-1][j-1];
    }
}

//Unoptimized Example A
for (i=1; i<N; i++) {
    for (j=1; j<N; j++) {
        A[i][j].d = A[i-1][j-1].d;
    }
}
Performance Difference: Sparse Data

The graph shows the speedup vs UnOPT for different processors (Athlon64 x2, Pentium D, and Core 2 Duo) with varying array side sizes. The x-axis represents the array side size, and the y-axis represents the speedup. The graph indicates a performance difference in handling sparse data for different processors.
Example 2: The Same Application Optimized

Looks like a perfect scalable application!
Are we done?
⇒ Duplicate one data structure
Example: Sparse data allocation

```c
sparse_rec sparse [HUGE];
for (int j = 0; j < HUGE; j++)
{
    sparse[j].a = 'a'; sparse[j].b = 'b'; sparse[j].c = 'c'; sparse[j].d = 'd'; sparse[j].e = 'e';
    sparse[j].f1 = 1.0; sparse[j].f2 = 1.0; sparse[j].f3 = 1.0; sparse[j].f4 = 1.0; sparse[j].f5 = 1.0;
}
```

```
struct sparse_rec
{
    // size 80B
    char a;
    double f1;
    char b;
    double f2;
    char c;
    double f3;
    char d;
    double f4;
    char e;
    double f5;
};

struct dense_rec
{
    // size 48B
    double f1;
    double f2;
    double f3;
    double f4;
    double f5;
    char a;
    char b;
    char c;
    char d;
    char e;
};
```
Loop Merging

/* Unoptimized */
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        a[i][j] = 2 * b[i][j];
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        c[i][j] = K * b[i][j] + d[i][j]/2

/* Optimized */
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        a[i][j] = 2 * b[i][j];
        c[i][j] = K * b[i][j] + d[i][j]/2;
Padding of data structures

Generic Cache:

SRAM:

Data = 64B
Padding of data structures

Generic Cache:

Cacheline:

allocate more memory than needed
/* Unoptimized ARRAY: x = y * z */
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        {r = 0;
         for (k = 0; k < N; k = k + 1)
             r = r + y[i][k] * z[k][j];
         x[i][j] = r;
        };

X:  j
   i

Y:  k
   i

Z:  j
   k
/* Optimized ARRAY: X = Y * Z */

for (jj = 0; jj < N; jj = jj + B)
for (kk = 0; kk < N; kk = kk + B)
for (i = 0; i < N; i = i + 1)
  for (j = jj; j < min(jj+B,N); j = j + 1)
    {r = 0;
     for (k = kk; k < min(kk+B,N); k = k + 1)
       r = r + y[i][k] * z[k][j];
     x[i][j] += r;}

/* Optimized ARRAY: X = Y * Z */
for (jj = 0; jj < N; jj = jj + B)
  for (kk = 0; kk < N; kk = kk + B)
    for (i = 0; i < N; i = i + 1)
      for (j = jj; j < min(jj+B,N); j = j + 1)
        {r = 0;
         for (k = kk; k < min(kk+B,N); k = k + 1)
           r = r + y[i][k] * z[k][j];
         x[i][j] += r;
        }
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)
Cache Waste

/* Unoptimized */
for (s = 0; s < ITERATIONS; s++) {
    for (j = 0; j < HUGE; j++)
        x[j] = x[j+1]; /* will hog the cache but not benefit*/
    for (i = 0; i < SMALLER_THAN_CACHE; i++)
        y[i] = y[i+1]; /* will be evicted between usages */
}
/* Optimized */
for (s = 0; s < ITERATIONS; s++) {
    for (j = 0; j < Huge; j++) {
        PREFETCH_NT x[j+1] /* will be installed in L1, but not L3 (AMD) */
        x[j] = x[j+1];
        for (i = 0; i < SMALLER_THAN_CACHE; i++)
            y[i] = y[i+1]; /* will always hit in the cache*/
    }

⇒ Also important for single-threaded applications if they are co-scheduled and share cache with other applications.
Categorize and avoiding cache waste

Application classification

<table>
<thead>
<tr>
<th>Slowed by others</th>
<th>Slows &amp; slowed</th>
<th>Don’t care</th>
<th>Slows others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slows others</td>
<td>L1</td>
<td>L2</td>
<td>Mem</td>
</tr>
<tr>
<td></td>
<td>CPU</td>
<td>CPU</td>
<td></td>
</tr>
</tbody>
</table>

Miss rate

\[ \Delta \text{benefit} \]

Cache hogging

\[ \Delta \text{size} \]

Automatic "taming" of the hoggers

\[ \Delta \text{benefit} \]

Application classification

Slowed by others

Don’t care

Slows others

Hogging

Individually

In mix

In mix, patched

Hogging

Performance

bzip2

Libquantum

LBM

Geom mean

25%

Example: Hints to avoid cache pollution (non-temporal prefetches)

- The larger cache, the better
- Cache misses
- 2x missrate
- missrate
- Hint: Don’t allocate!
- Actual
- Actual/4
- Cache size

- One Instance
- Four Instances
- Throughput
- 40% faster
- Orig
- Hint: lim= actual/4
Some performance tools

Free licenses
- Oprofile
- GNU: gprof
- AMD: code analyst
- Google performance tools
- Virtual Inst: High Productivity Supercomputing (http://www.vi-hps.org/tools/)

Not free
- Intel: Vtune and many more
- ThreadSpotter (of course😊)
- HP: Multicore toolkit (some free, some not)
Commerical Break: ThreadSpotter

Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se
ThreadSpotter™

Source:
C, C++, Fortran, OpenMP…

Mission:
Find the SlowSpots™
Asses their importance
Enable for non-experts to fix them
Improve the productivity of performance experts
Source: C, C++, Fortran...

What?

Where?

How?

Any Compiler

Finger Print (~4MB)

Analysis

Advice

Host System

Sampler

Target System Parameters

Mission:
Find the SlowSpots™
Asses their importance

/* Unoptimized Array Multiplication: x = y * z    N = 1024 */
for (i = 0; i < N; i = i + 1)
  for (j = 0; j < N; j = j + 1)
  {
      r = 0;
      for (k = 0; k < N; k = k + 1)
          r = r + y[i][k] * z[k][j];

      x[i][j] = r;
  } /* Uti i d A M lti li ti * N 1024 */

How?

Any Compiler

Finger Print (~4MB)

Analysis

Advice

Host System

Sampler

Target System Parameters

A poor cache line utilization issue indicates that a process locality, that is, cache lines are only partially used. If the cache, which means that memory bandwidth and cache data is wasted.

The poor utilization issue has these sections:

- Statistics for instructions of this issue
- Instructions previously written to related data
- Loop statistics
- Loop instructions

Poor cache line utilization can have a number of causes:

- There may be structures with unused fields, see Structures.
- There may be padding inserted into structures data alignment, see Section 5.1.3, "Alignment.
- There may be housekeeping data from the dynamic memory, see Section 5.1.1, "dynamic Memory.
- It may be caused by irregular access patterns, as shown.

© Erik Hagersten | http://user.it.uu.se/~eh
A One-Click Report Generation

Fill in the following fields:

- Application to run
- Input arguments
- Working dir (where to run the app)

(Limit, if you like, data gathered here, e.g., start gathering after after 10 sec. and stop after 10 sec.)

Cache size of the target system for optimization (e.g., L1 or L2 size)

Click this button to create a report
Fetch rate

Cache utilization ≈ Fraction of cache data utilized

Predicted fetch rate (if utilization → 100%)

Cache size
Select a file in the file table, or follow a source code link from an issue or a loop description.

Cache size to optimize for: 64k
Loop Focus Tab

Spotting the crime

List of bad loops

Explaining what to do
### Bandwidth Focus Tab

#### Spotting the crime

#### List of Bandwidth SlowSpots

<table>
<thead>
<tr>
<th>Loop</th>
<th>Issue</th>
<th>Summary</th>
<th>% of fetches</th>
<th>Utilization</th>
<th>HW-Prefetch</th>
<th>Randomness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>Poor utilization</td>
<td>29.4%</td>
<td>17.4%</td>
<td>100.0%</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>1/4</td>
<td>Loop fusion</td>
<td>29.4%</td>
<td>24.4%</td>
<td>97.0%</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>1/1</td>
<td>Inefficient loop nesting</td>
<td>29.2%</td>
<td>126.8%</td>
<td>0.0%</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>5/9</td>
<td>Loop fusion</td>
<td>4.2%</td>
<td>11.8%</td>
<td>97.1%</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>3/12</td>
<td>Poor utilization</td>
<td>4.2%</td>
<td>9.7%</td>
<td>96.6%</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>4/12</td>
<td>Loop fusion</td>
<td>4.2%</td>
<td>12.7%</td>
<td>97.6%</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>7/18</td>
<td>Poor utilization</td>
<td>4.2%</td>
<td>25.1%</td>
<td>100.0%</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>4/10</td>
<td>Poor utilization</td>
<td>4.2%</td>
<td>97.4%</td>
<td>96.7%</td>
<td>Low</td>
<td></td>
</tr>
</tbody>
</table>

#### Explaining what to do

**Issue #1: Inefficient loop nesting**

This instruction group also shows symptoms of: Cache line utilization, Hot-spot.  

**Statistics for instructions of this issue**

**Instructions involved in this issue**

**Loop statistics**

**Loop instructions**

---

*Acumem VPE: /art (64k/64) - Mozilla Firefox*
Resource Sharing Example

**Libquantum**
A quantum computer simulation
Widely used in research (download from: [http://www.libquantum.de/](http://www.libquantum.de/))
4000+ lines of C, fairly complex code.
Runs an experiment in ~30 min

Throughput improvement:
Utilization Analysis

Libquantum

Fetch rate

Original Code

Cache utilization \approx \text{Fraction of cache data utilized}

Predicted fetch rate if utilization = 100%

Need 32 MB per thread!

SlowSpotter’s First Advice: Improve Utilization

- Change one data structure
  - Involves \sim 20 lines of code
  - Takes a non-expert 30 min
Utilization Analysis

Libquantum

Fetch rate

Cache utilization ≈ Fraction of cache data utilized

Original Code

Utilization Optimization

for (i=0; i++; i<MAX) {
    ... = huge_data[i].status + ...
}

for (i=0; i++; i<MAX) {
    ... = huge_data_status[i] + ...
}

SlowSpotter’s First Advice: Improve Utilization

- Change one data structure
  - Involves ~20 lines of code
  - Takes a non-expert 30 min
After Utilization Optimization

Libquantum

Original Code

Utilization Optimization

Old fetch rate

Cache Utilization ≈ 95%

Predicted fetch rate ≈ New fetch rate
Utilization Optimization

Two positive effects from better utilization
1. Each fetch brings in more useful data → lower fetch rate
2. The same amount of useful data can fit in a smaller cache → shift left
Reuse Analysis

Libquantum

Utilization Optimization

Fetch rate

Utilization + Fusion Optimization

... toffoli(huge_data, ...)
cnot(huge_data, ...)
...
fused_toffoli_cnot(huge_data, ...)
...

Second-Fifth SlowSpotter Advice: Improve reuse of data

→ Fuse functions traversing the same data
  - Here: four fused functions created
  - Takes a non-expert < 2h
The miss in the second loop goes away.

Still need the same amount of cache to fit “all data”
Utilization + Reuse Optimization

Libquantum

Old fetch rate

Utilization Optimization

New fetch rate

Utilization + Fusion Optimization

- Fetch rate down to 1.3% for 2MB
- Same as a 32 MB cache originally
Summary

Libquantum

Utilization Optimization
Utilization + Fusion

Throughput

# Cores Used

1 2 3 4

2.7x
Uppsala Programming for Multicore Architecture Center

62 MSEK grant / 10 years [$9M/10y]
+ related additional grants at UU = 130MSEK

Research areas:

- Performance modeling
- New parallel algorithms
- Scheduling of threads and resources
- Testing & verification
- Language technology
- MC in wireless and sensors
Underneath the ThreadSpotter Hood
Great but Slow Insight: Simulation

Slowdown: ≈ 10 - 1000x

Code:
set A,%r1
ld [%r1],%r0
st %r0,[%r1+8]
add %r1,1,%r1
ld [%r1+16],%r0
add %r0,%r5,%r5
st %r5,[%r1+8]
[...]

Memory ref:
1:read A
2:write B
3:read C
4:write B
[...]

Simulated CPU

Simulated Memory System

CPU-sim
Level-1 Cache
Level-n Cache
Memory
Limited Insight: Hardware Counters

Slowdown: $\approx 0\%$

- No flexibility
- Limited insight

Insight: "Instruction X misses Y\% of the time in the cache"  
Architecturally dependent (!!)
StatCache: Insight and Efficiency
Slowdown 10% (for long-running applications)

Online Sampling

Host Computer

- core
- ...mem
- core

Address Stream

1: read A
2: read B
3: read C
4: write C
5: read B
6: read D
7: read A
8: read E
9: read B

Randomly select accesses to monitor

Sparse Sampler

Application Fingerprint

5, 3, ...

Offline “Insight Technology”

Target Architecture

- core
- L1
- L2
- mem

Architectural Parameters

Probabilistic Cache Model

Modeled behavior

Acumem

Reuse Distance=5

Advice

Reuse Distance=3

Dept of Information Technology | www.it.uu.se

© Erik Hagersten | http://user.it.uu.se/~eh
UART: Efficient sparse sampling

1. Use HW counter overflow to randomly select accesses to sample (e.g. ~on average every 1,000,000th access)

2. Set a watchpoint for the data cacheline they touch

3. Use HW counters to count #memory accesses until watchpoint trap

→ Sampling Overhead ~17% (10% at Acumem for long-running apps)

(Modeling with math < 100ms)
Fingerprint
≈ Sparse reuse distance histogram
Modeling random caches with math

(Assumption: “Constant” MissRatio)

\[ \text{rd}_i = 5 \]

\[ \# \text{repl} \approx 5 \times \text{MissRatio} \]

Miss Equation:

\[ p_{\text{miss}} = m(\#\text{repl}) \]
Assuming a fully associative cache

The cacheline A

(1 – 1/L) chance that A survives

(1 – 1/L)^R chance that A survives
Modeling random caches with math
(Assumption: "Constant" MissRatio)

Miss Equation $m$

$p_{\text{miss}} = m(5 \times \text{MissRatio})$

$p_{\text{miss}} = m(3 \times \text{MissRatio})$

$n$ samples: $\text{MissRatio} \times n = \sum_{i=0}^{n} m(\text{rd}(i) \times \text{MissRatio})$

Can be solved in a "fraction of a second" for different $L$
Accuracy: Simulation vs. “math” (Random replacement)

Comparing simulation (w/ slowdown 100x) and math (“fractions of a second”)
Modeling LRU Caches: Stack distance...

Stack Distance: How many unique data objects? Answer: 3

If we know all reuses: How many of the reuses 2-6 go beyond End? Answer: 3

Stack_distance = \[ \sum_{k=Start}^{End} [d(i) > (End - k + 2)] \]

Foreach sample: if (Stack_distance > L) miss++ else hit++
But we only know a few reuse distances...

Assume that the distribution (aka histogram) of sampled reuses is representative for all accesses in that "time window"

\[
\text{Est}_\text{SD} = \sum_{k=\text{Start}}^{\text{End}} p[d(i) > (\text{End} - k)]
\]
Architecturally independent!

The fingerprint does not depend on the caches of the host architecture

Solve the equation for different target architecture:

- Cache sizes
- Cacheline sizes
- Replacement algorithms \{LRU, RND\}
- Cache topology
In a nutshell

1. Sampling: Randomly select windows, and collect sparse reuse histograms from each window

2. Use histogram as input to model behavior of target arch.
Sequential HW prefetching is modeled using math. Successful HW prefetches are only reported in Bandwidth Issues.

Special heuristics have been designed to analyze the fingerprint to also tell the reason for cache misses.