Reading Response

• Due Friday, December 10, 6am

SHARK: Architectural support for autonomic protection against stealth by rootkit exploits

http://portal.acm.org/citation.cfm?id=1521783

• New lab on Monday (quick & easy)

• Wednesday Lab: Reserve for lecture or review
Memory Hierarchy

Registers

Cache

Main memory

Magnetic disk

Tape

Optical disk
Multi-Level Caches ($’s circa 2007)

Options: separate data and instruction caches, or a unified cache

- Larger, slower, cheaper

<table>
<thead>
<tr>
<th></th>
<th>Regs</th>
<th>L1 d-cache</th>
<th>L1 i-cache</th>
<th>Unified L2 Cache</th>
<th>Memory</th>
<th>disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>size:</td>
<td>200 B</td>
<td>8-64 KB</td>
<td>32 B</td>
<td>1-4MB SRAM</td>
<td>128 MB DRAM</td>
<td>30 GB</td>
</tr>
<tr>
<td>speed:</td>
<td>3 ns</td>
<td>3 ns</td>
<td>6 ns</td>
<td>60 ns</td>
<td>8 ms</td>
<td></td>
</tr>
<tr>
<td>$/Mbyte:</td>
<td>$100/MB</td>
<td>$1.50/MB</td>
<td>$0.05/MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>line size:</td>
<td>8 B</td>
<td>32 B</td>
<td>32 B</td>
<td>8 KB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Intel Pentium Cache Hierarchy

Processor Chip

L1 Data
1 cycle latency
16 KB
4-way assoc
Write-through
32B lines

L1 Instruction
16 KB, 4-way
32B lines

Regs.

L2 Unified
128KB--2 MB
4-way assoc
Write-back
Write allocate
32B lines

Main Memory
Locality

• **Principle of Locality** states within a given period of time, programs tend to reference a relatively confined area of memory repeatedly.

• This Principle is a measurable phenomena and produces a **working set**.

• The working set is the set of memory locations referenced over a fixed period of time, or window, extending from the current time into the past.
Spatial Locality, or the fact that when a given address has been referenced, it is likely that addresses near it will be referenced within a short period of time.

Temporal Locality, or the fact that once a particular memory item has been referenced, it is likely that it will be referenced again within a short period of time.
Consider a “block” in a cache

• **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future

• **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time
2-Level Hierarchy

- A two level hierarchy suggests that memory can be structured in two layers, or two interacting layers.

  - **Primary level** -- small and fast
  
  - **Secondary level** -- large and slow

- This relationship indicates a trade-off in memory technologies between processing time expense and hardware purchasing expense.
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10)  /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23)  /* ... up to 8 MB */
#define MAXSTRIDE 16        /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

int data[MAXELEMS];    /* The array we'll be traversing */

int main()
{
    int size;       /* Working set size (in bytes) */
    int stride;     /*Stride (in array elements) */
    double Mhz;     /* Clock frequency */

    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    Mhz = mhz(0);      /* Estimate the clock frequency */
    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride, Mhz));
        printf("\n");
    }
    exit(0);
}
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz) {
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride);        /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0); /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
The Memory Mountain

Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified L2 cache

read throughput (MB/s)

Slopes of Spatial Locality

Ridges of Temporal Locality

stride (words)

working set size (bytes)
Hits vs misses

\[ t_a = h t_p + (1 - h) t_s \]

- \( t_a \): actual access time
- \( t_p \): primary memory access time
- \( t_s \): secondary memory access time
- \( h \): hit ratio, or hits per memory reference
Hits vs misses

\[ t_a = h t_p + (1 - h) t_s \]

Assume:
- \( t_p \) -- primary memory access time is 10 ns.
- \( t_s \) -- secondary memory access time is 100 ns.
**Hits vs misses**

\[ t_a = h t_p + (1 - h) t_s \]

<table>
<thead>
<tr>
<th>h</th>
<th>ht_p</th>
<th>(1-h)t_s</th>
<th>t_a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>0.2</td>
<td>2</td>
<td>80</td>
<td>82</td>
</tr>
<tr>
<td>0.4</td>
<td>4</td>
<td>60</td>
<td>64</td>
</tr>
<tr>
<td>0.5</td>
<td>5</td>
<td>50</td>
<td>55</td>
</tr>
<tr>
<td>0.6</td>
<td>6</td>
<td>40</td>
<td>46</td>
</tr>
<tr>
<td>0.8</td>
<td>8</td>
<td>20</td>
<td>28</td>
</tr>
<tr>
<td>1.0</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
Matrix Multiplication Example

**Major Cache Effects to Consider**
- Total cache size
  - Exploit temporal locality and keep the working set small (e.g., by using blocking)
- Block size
  - Exploit spatial locality

**Description:**
- Multiply N x N matrices
- O(N^3) total operations
- Accesses
  - N reads per source element
  - N values summed per destination
    - but may be able to hold in register

```c
/* ijk */
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;
  }
}
```

Variable `sum` held in register
Miss Rate Analysis for Matrix Multiply

**Assume:**
- Cache Line (Block) size = 32 bytes (fits 4 64-bit words)
- Matrix dimension (N) is very large
  - Approximate 1/N as 0.0
- Cache is not even big enough to hold multiple rows

**Analysis Method:**
- Look at access pattern of inner loop

![Diagram showing access pattern of inner loop](image)
Layout of C Arrays in Memory (review)

- **C arrays allocated in row-major order**
  - each row in contiguous memory locations

- **Stepping through columns in one row:**
  - for (i = 0; i < N; i++)
    - sum += a[0][i];
  - accesses successive elements
  - if block size (B) > 4 bytes, exploit spatial locality
    - compulsory miss rate = 4 bytes / B

- **Stepping through rows in one column:**
  - for (i = 0; i < n; i++)
    - sum += a[i][0];
  - accesses distant elements
  - no spatial locality!
    - compulsory miss rate = 1 (i.e. 100%)
Matrix Multiplication (ijk)

```c
/* ijk */
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;
    }
}
```

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (jik)

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

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (kij)

```c
/* kij */
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];
    }
}
```

Misses per Inner Loop Iteration:

- A
- B
- C

Approx speedup relative to 1.25?

Cache = 10ms and memory = 100ms

\[ t_a = h t_p + (1 - h) t_s \]

n is BIG
Matrix Multiplication (kij)

```c
/* kij */
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];
    }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (ikj)

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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misses</td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (jki)

```c
/* jki */
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;
    }
}
```

**Inner loop:**
- 

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (kji)

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

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
### Summary of Matrix Multiplication

**ijk (& jik):**
- 2 loads, 0 stores
- misses/iter = 1.25

```c
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;
    }
}
```

**kij (& ikj):**
- 2 loads, 1 store
- misses/iter = 0.5

```c
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];
    }
}
```

**jki (& kji):**
- 2 loads, 1 store
- misses/iter = 2.0

```c
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;
    }
}
```
Pentium Matrix Multiply Performance

- Miss rates are helpful but not perfect predictors.