12
Tuning Software for Speed
18-548/15-548 Memory System Architecture
Philip Koopman
October 14, 1998

Assignments

By next class read about Main Memory Architecture:
• Read: Cragon 5.1-5.1.5
  • Supplemental Reading:
    – Hennessy & Patterson 5.6
    – IBM App. Note: Understanding DRAM

Homework 7 due October 21

Lab 4 due October 23
Where Are We Now?

✦ Where we’ve been:
  • Cache Organization & Policies
  • System-Level Effects

✦ Where we’re going today:
  • How can you exploit memory hierarchy effectively?
  • How can you avoid being burned by it?

✦ Where we’re going next:
  • Main memory

Preview

✦ Review: interpret execution time charts to infer cache characteristics
  • Cache levels and size for each

✦ Tune software for cache memory performance
  • Cache size, associativity, block size, page size
  • Read vs. write behavior & policies

✦ Create blocked algorithms to improve locality
  • Matrix multiply as an example
Optimize For Cache Effects

- **Small Cache size**
  - Decompose large data sets into small ones
  - Encourage temporal & spatial locality with algorithm change

- **Low Associativity**
  - Remap conflicting instructions/data so as not to reside in same set
  - Intermix data so that related data loads into single cache block

- **Large Block Size**
  - Access data in sequential order
  - Attempt to modify all data in block at once (don’t mix “clean” and “dirty” words)

- **Write Policies**
  - Write back -- group writes to data
  - Write buffer -- smooth bursts of write traffic
  - Allocation -- force allocation if desirable

CACHE SIZE EFFECTS
Probing For Cache Size

- Cache size limits ability to sequentially re-touch array elements
  - Array < cache size all cache hits
  - cache size < array size < cache size * (1 + 1/assoc.) partial misses
  - cache size * (1 + 1/assoc.) < array size all misses (LRU)

- Cache Size Test Program
  - Using pointers in loops creates efficient inner loop

```c
int *p, *a, *limit; /* a points to malloc'ed array area */
...
limit = &(a[test_size]);
for (i = 0; i < NUM_TESTS; i++)
{ for (p = a;  p < limit; p++)
  { sum += *p; }
}
```

Cache Size Data

- Execution speed drops as array exceeds cache size
  - 1K - 8 KB all fits in L1 cache
  - 8K - 16KB increasing number of conflict misses as array wraps around L1 cache twice
  - Steady from 16 KB - 512KB as L2 cache holds entire array
  - Same pattern for L2 cache misses at 512KB+
Faster Alpha with L3 Cache

- Task interference is large “memory sweeper” running in background

Effects of Multi-Tasking On Cache Performance

EXAMPLE CODE OPTIMIZATION
Example: Optimizing 2-D Array Code

Running example:

```c
int a[N][N], b[N][N], c[N][N], d[N][N];

for (j = 0; j < N; j = j++)
    for (i = 0; i < N; i++)
        a[i][j] = b[i][j] * c[i][j];

for (j = 0; j < N; j = j++)
    for (i = 0; i < N; i = i++)
        d[i][j] = a[i][j] + c[i][j];
```

Example run multiple times for timing

- Optimistic, but representative for small arrays (results may be left in cache from a previous loop that produced them)
- Actual tested code uses pointers instead of array indexing to reduce overhead computations (aggressive compilers can do this automatically)
- Size of array, N, varied (results shown are total data set size for 4 arrays)

Unoptimized Performance

- Nested loop overhead amortizes over array size
- Conflicts occur with arrays that are perfect power of 2 sizes when L1 cache is exceeded
  - N=32; 64; 128; 256
  - N=32 is 16 KB total
Loop Interchange

int a[N][N], b[N][N], c[N][N], d[N][N];
for (j = 0; j < N; j = j++)
  for (i = 0; i < N; i = i++)
    a[i][j] = b[i][j] * c[i][j];
for (j = 0; j < N; j = j++)
  for (i = 0; i < N; i = i++)
    d[i][j] = a[i][j] + c[i][j];

Loop interchange reverses order of indexing

- Works when order of loop execution is unimportant
- After interchange arrays are accessed at sequential locations
- Improves locality at level of both page & block references

/* AFTER LOOP INTERCHANGE */
int a[N][N], b[N][N], c[N][N], d[N][N];
for (i = 0; i < N; i = i++)
  for (j = 0; j < N; j = j++)
    a[i][j] = b[i][j] * c[i][j];
for (i = 0; i < N; i = i++)
  for (j = 0; j < N; j = j++)
    d[i][j] = a[i][j] + c[i][j];

Loop Interchange Memory Access

- Loop interchange converts strided accesses into sequential accesses
  - Accesses with large stride only use one word per cache block
    - With arrays bigger than cache size, rest of fetch words are evicted before used
  - Access with “unit stride” (sequential access) use all words in cache block in consecutive iterations
Loop Interchange Performance

- Reduces overhead to single loop
  • With stride=1!
- Entire matrix row fits in L2 cache for N < 181
  • 128KB data per matrix
  • Loop interchange speedup is limited to avoided L1-miss/L2-hit delays

![ALPHA WORKSTATION PERFORMANCE](attachment:image)

Loop Fusion

```c
/* AFTER LOOP INTERCHANGE */
int a[N][N], b[N][N], c[N][N], d[N][N];
for (i = 0; i < N; i = i++)
  for (j = 0; j < N; j++)
    a[i][j] = b[i][j] * c[i][j];
for (i = 0; i < N; i = i++)
  for (j = 0; j < N; j++)
    d[i][j] = a[i][j] + c[i][j];
```

- Loop fusion places multiple array computations in the same loop
  • Increases temporal locality (c used twice; a used twice)
  • Reduces looping overhead computations
  • Must be careful of inter-loop data dependencies

```c
/* AFTER LOOP FUSION */
int a[N][N], b[N][N], c[N][N], d[N][N];
for (i = 0; i < N; i = i++)
  for (j = 0; j < N; j++)
  { a[i][j] = b[i][j] * c[i][j];
    d[i][j] = a[i][j] + c[i][j];
  }
```
Loop Fusion Performance

- Cuts loop overhead in half
- \(c[i][j]\) and \(a[i][j]\) stay in cache between two statements
  - \(a[i][j]\) stored after \(c[i][j]\) fetched for second time
  - Keeping \(c[i][j]\) in register might improve things further

![Alpha Workstation Performance Graph]

Array Merging

```c
/* AFTER LOOP FUSION */
int a[N][N], b[N][N], c[N][N], d[N][N];
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    { a[i][j] = b[i][j] * c[i][j];
      d[i][j] = a[i][j] + c[i][j];
    }
```

- Array merging intermingles array elements
  - Cache fetching of a block loads a set of related values at once
  - Eliminates accidental conflicts for arrays mapping into same block

```c
/* ARRAY MERGING */
struct merge { int a, int b, int c, int d; }
struct merge m[N][N];
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    { m.a[i][j] = m.b[i][j] * m.c[i][j];
      m.d[i][j] = m.a[i][j] + m.c[i][j];
    }
```
Array Merging Data Layout

- In example, each 4-int array element takes up 16 consecutive bytes
  - Touching any one of the elements loads all 4 into cache for block size of 16 bytes or greater

![Array Merging Diagram](image)

- Array merging works best when values are truly related, and usually fetched as a set
  - e.g., real and imaginary portions of a complex number

Array Merging Performance

- Eliminates vulnerabilities at power-of-2 boundaries
  - Guarantees spatial locality
  - No spikes due to conflict misses
  - Multiple data available for superscalar use when in L1 cache

- BUT, no free lunch
  - Modified data mingled with unmodified data increases traffic ratio
  - Lose ability to have two misses pending on non-blocking L1 cache miss
    - Non-merged data could overlap fetch of 2 data blocks on every miss

![Performance Graph](image)
If array data are unrelated and used in various places, array merging won’t be very helpful

Instead, lay out arrays so they map to different parts of cache, reducing conflict misses

• Optimal when cache size is known, but 8K is usually a good guess

/* note: N must be a power of 2 for this to work */
#define CACHESIZE 8192
#define OFFSET (CACHESIZE/(4*sizeof(int)))
a = (int *) malloc(4*N*N*sizeof(int)+CACHESIZE);
b = a + N*N + OFFSET; /* maps 25% into cache */
c = b + N*N + OFFSET; /* maps 50% into cache */
d = c + N*N + OFFSET; /* maps 75% into cache */
...

/* ARRAY MERGING */
struct merge { int a; int b; int c; int d; }
struct merge m[N][N];
for (i = 0; i < N; i = i++)
  for (j = 0; j < N; j++)
    { m.a[i][j] = m.b[i][j] * m.c[i][j];
      m.d[i][j] = m.a[i][j] + m.c[i][j];
    }

• Arrays placed so that they don’t conflict
  Alternate approach to array merging
  OFFSET must be selected with care so that corresponding [i][j] elements of the four matrices don’t map to the same cache set

/* ARRAY PLACEMENT */
int a[N][N], junka[OFFSET], b[N][N], junkb[OFFSET];
int c[N][N], junkc[OFFSET], d[N][N], junkd[OFFSET];
for (i = 0; i < N; i = i++)
  for (j = 0; j < N; j++)
    { a[i][j] = b[i][j] * c[i][j];
      d[i][j] = a[i][j] + c[i][j];
    }
Performance With Array Placement

- Array placement eliminates conflicts
  - Reads separated from writes
- Avoids write-back of unmodified data because arrays aren’t intermingled

**ALPHA WORKSTATION PERFORMANCE**

<table>
<thead>
<tr>
<th>TOTAL SIZE OF DATA ARRAYS</th>
<th>MB/SEC DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>0</td>
</tr>
<tr>
<td>10K</td>
<td>10</td>
</tr>
<tr>
<td>100K</td>
<td>20</td>
</tr>
<tr>
<td>1000K</td>
<td>30</td>
</tr>
<tr>
<td>64 KB</td>
<td>40</td>
</tr>
<tr>
<td>256 KB</td>
<td>50</td>
</tr>
<tr>
<td>1024 KB</td>
<td>60</td>
</tr>
</tbody>
</table>

**Write Merging**

/* ARRAY PLACEMENT */

```c
int a[N][N], junka[OFFSET], b[N][N], junkb[OFFSET];
int c[N][N], junkc[OFFSET], d[N][N], junkd[OFFSET];
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    { a[i][j] = b[i][j] * c[i][j];
      d[i][j] = a[i][j] + c[i][j];
    }
```

- Write merging exploits write assembly buffer
  - Equivalent to array merging only for modified data
  - Want number of blocks being written to fit into WAB
  - For this example, only a win if WAB is exactly 1 deep
  - Not important for copy-back caches in absence of conflicts

/* WRITE MERGING */

```c
int b[N][N], junkb[OFFSET], c[N][N], junkc[OFFSET];
struct merge { int a; int d; }    struct merge m[N][N];
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    { m.a[i][j] = b[i][j] * c[i][j];
      m.d[i][j] = m.a[i][j] + c[i][j];
    }
```
Write Merging Performance

- Write merging smoothes demand to write by writing combined blocks
  - 1 block instead of 2
  - Twice as frequently
- Benefit would be more pronounced on write-through machine with write assembly buffer of size 1 element
  - Merging of write values causes conflicts as arrays outgrow L2 cache
  - Not a huge win, but something to keep in mind

Review of Optimization Steps
BLOCKED ALGORITHMS

Blocked Algorithms

♦ Break problems up into cache-sized chunks
  • Simplifying assumption: no conflict misses
  • If conflict misses occur, use array placement

♦ 1-D blocking is called “strip mining”
  • Very important optimization for vector supercomputers
  • Straightforward to automate with compiler (in many cases)

♦ Multi-dimensional blocking gets harder
  • Often requires algorithmic transformations
  • May be best used as embedded in a library routing (e.g., matrix multiply)
Matrix Multiply

- Square Matrices: $X = Y \times Z$
  
  ```c
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      { r = 0;
        for (k = 0; k < N; k++)
          { r = r + y[i][k] * z[k][j]; }
        x[i][j] = r;
      }
  
  • One row of $Y$ tends to stay in cache if not too large
  
  Matrix Multiply in Block Form
  
  ```

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

(note: Hennessy & Patterson have bug on page 409 -- “+B-1” is not correct for j and k loops)
Blocked Matrix Multiply Performance

Optimization Effectiveness

(Hennessy & Patterson Figure 5.17)
OTHER OPTIMIZATIONS

Faked Write Allocation

- Forced write allocation
  - For write-followed-by-read behavior, force cache allocation by first reading the data
  - Speed-up of 20% on an (admittedly extreme) case for VAX 8800

```c
int a[100];
...
  a[i] = b[i] * c[i];
...
  d[i] = a[i] * 42;
```

BECOMES (with a compiler that respects the volatile keyword):

```c
volatile int a[100];
...
  foo = a[i];
  a[i] = b[i] * c[i];
...
  d[i] = a[i] * 42;
```
Program Mapping Optimization

- Compiler/Linker/Loader can minimize mapping/set conflicts
  - McFarling (1989) states that optimization can make direct caches more effective than unoptimized code on set associative caches
  - Same might be accomplished by operating system doing page mapping, especially for large L2 caches
**Optimize For Cache Effects**

- **Small Cache size**
  - Decompose large data sets into small ones
  - Encourage temporal & spatial locality with algorithm change

- **Low Associativity**
  - Remap conflicting instructions/data so as not to reside in same set
  - Intermix data so that related data loads into single cache block

- **Large Block Size**
  - Access data in sequential order
  - Attempt to modify all data in block at once (don’t mix “clean” and “dirty” words)

- **Write Policies**
  - Write back -- group writes to data
  - Write buffer -- smooth bursts of write traffic
  - Allocation -- force allocation if desirable

**Review**

- **Optimizing software for cache memory requires exploiting both organization & policy information**
  - Loop interchange to promote spatial locality at block & page level
  - Loop fusion to promote temporal locality (sometimes can hold all values in registers)
  - Array merging to promote spatial locality at block level (mostly for reads)
  - Separating reads from writes
    - Reduces traffic ratio with write back cache & large block sizes
    - Increases possibilities for write allocation buffer to merge writes

- **Blocked algorithms improve cache usage**
  - Intentionally wastes computations to reduce memory accesses
  - Want block size as big as will fit everything in cache for efficiency