Cache Performance

April 9, 1996

Topics:

- metrics
- cache performance analysis
- impact on C program performance

Basic performance metrics

hit rate:

- fraction of memory references found in cache

miss rate:

- fraction of memory references not found in cache

hit time:

- time to access cache line (includes time to determine hit or miss)

miss penalty:

- time to load cache line, plus time to deliver line to processor

Design space (structural issues)

cache data size

- **effect on hit time**
  - smaller is faster

line size

- **effect on miss rate**
  - tradeoff between better spatial locality and fewer blocks
- **effect on miss penalty**
  - longer transfer time
  - read/modify/write penalty for write misses to multibyte blocks

Write misses for multi-byte lines

If B>1, write misses can require read/modify/write cycle

Suppose:
line size is 4 bytes
x and y map to same direct mapped entry
before "store byte to x[0]" instruction entry contains:

tag[y]	y[0]	y[1]	y[2]	y[3]

Question:
What should cache line contain after "store byte" instruction?

Design space (operational issues)

line placement strategy

- **direct mapped, set associative, fully associative**
- **effect on total cache size (tags+data)**
  - higher associativity requires more tag bits
- **effect on miss rate**
  - higher associativity decreases miss rate
- **effect on hit time**
  - higher associativity decreases hit time

Design space (operational issues)

line replacement strategy

- **random, LRU**
- **effect on miss rate**
  - LRU has up to ~10% lower miss rate
- **effect on miss penalty**
  - LRU increases miss penalty

Design space (operational issues)

write strategy

- **write through**
  - data written to both cache and memory line
- **write back**
  - data written to memory only when cache line is replaced
- **advantages of write through**
  - read misses are cheaper. Why?
  - write through simpler to implement.
  - but requires a write buffer to pipeline writes.
- **advantages of write back**
  - reduced traffic to memory
  - individual writes at the processor rate

Qualitative cache performance model

Compulsory misses

- **first access to line not in cache**
- **"cold start" misses**

Capacity misses

- **active portion of memory exceeds cache size**

Conflict misses

- **active portion of address space fits in cache, but too many lines map to same cache entry**
- **direct mapped and set associative placement only**

Cache analysis (R2000)

C code (1-way interleaving)

```
ip1(double x[], double y[])
{
  int i;
  double result = 0.0
  for (i=0; i<512; i++) {
    double xt0 = x[i];
    result += xt0 * y[i];
  }
  return result;
}
```

MIPS assembly code

```
$L5:

l.d      $f0, 0($4)
l.d      $f2, 0($5)
mul.d    $f0,$f0,$f2
addu     $5,$5,8
addu     $4,$4,8
addu     $3,$3,1
slt      $2,$3,512
BNE      $2,$0,$L5
add.d    $f4,$f4,$f0
```


Cache analysis (R2000)

C code (2-way interleaving)

```
ip2(double x[], double y[])
{
  int i;
  double result = 0.0
  for (i=0; i<512; i+=2) {
    double xt0 = x[i];
    double xt1 = x[i+1];
    result += xt0 * y[i];
    result += xt1 * y[i+1];
  }
  return result;
}
```

C code (4-way interleaving)

```
ip4(double x[], double y[])
{
  int i;
  double result = 0.0
  for (i=0; i<512; i+=2) {
    double xt0 = x[i];
    double xt1 = x[i+1];
    double xt2 = x[i+2];
    double xt3 = x[i+3];
    result += xt0 * y[i];
    result += xt1 * y[i+1];
    result += xt2 * y[i+2];
    result += xt3 * y[i+3];
  }
  return result;
}
```

Cache analysis (R2000)

$L5:		start	end	comments
l.d	$f0, 0($4)	0	3	2 loads + bubble
l.d	$f2, 0($5)	2	5	2 loads + bubble
mul.d	$f0,$f0,$f2	5	10	load stall, 5 cycle exec
addu	$5,$5,8	6	7	overlaps mult
addu	$4,$4,8	7	8	overlaps mult
addu	$3,$3,1	8	9	overlaps mult
slt	$2,$3,512	9	10	overlaps mult
BNE	$2,$0,$L5	10	12	one slot delay
add.d	$f4,$f4,$f0	11	13	delay slot, 2 cycle exec

Next iteration can begin on cycle 12 with l.d overlapping
2nd cycle of add.d

Cache analysis (R2000)

Page size = 4KB

All data begins on page boundaries

64KB direct mapped data cache

Separate instruction cache

Experiment:

- **Run with different location for arguments x and y**

Cache analysis (R2000)

Measured performance (clock cycles per result):

Interleaving	Normal	Anomalous
1	12	68
2	10	38
4	9	37

From these numbers, determine:

- why the anomalous behavior?
- miss penalty?
- line size?

Cache analysis (R2000)

l.d	\$f0,0(\$4)	x[i] miss + hit
l.d	\$f2,0(\$5)	y[i] miss + hit

Thus, 2 misses per data value

(68 - 12) / 2 = 28 cycle read miss penalty (ouch!)

Hit rate = 50% (very poor)

Cache analysis (R2000)

```
l.d    $f0,0($4)      x[i] miss + hit
l.d    $f2,8($4)      x[i+1] 2 hits
l.d    $f4,0($5)      y[i] miss + hit
l.d    $f6,8($5)      y[i+1] 2 hits
```

Thus, 1 miss per data value

Hit rate = 75% (better, but not great)

Cache analysis (R2000)

```
l.d    $f0,0($4)      x[i] miss+hit
l.d    $f2,8($4)      x[i+1] 2 hits
l.d    $f0,16($4)     x[i+2] miss+hit
l.d    $f2,24($4)     x[i+3] 2 hits
l.d    $f4,0($5)      y[i] miss+hit
l.d    $f6,8($5)      y[i+1] 2 hits
l.d    $f4,16($5)     y[i+2] miss + hit
l.d    $f6,24($5)     y[i+3] 2 hits
```

Thus 1 miss per data value (no improvement)

Implies line size of 8

Impact of cache on C programs

C = A * B

```
void ijk(double a[][N], double b[][N], double c[][N], int n, int iters) {
   int i,j,k,iter;
   double sum;
   for (iter=0; iter<iters; iter++) {
      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;
         }
   }
}
```


Impact of cache on C programs

ijk (2 loads)

```
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;
   }
}
```

jik (2 loads)

```
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
   }
}
```

kij (2 loads/1 store)

```
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];
   }
}
```

ikj (2 loads/1 store)

```
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];
   }
}
```

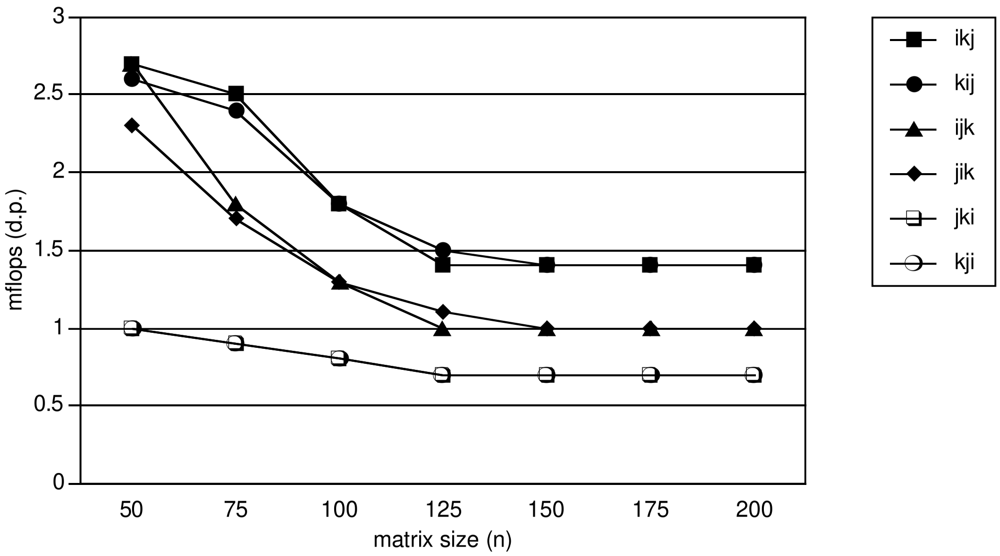
jki (2 loads/1 store)

```
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;
   }
}
```

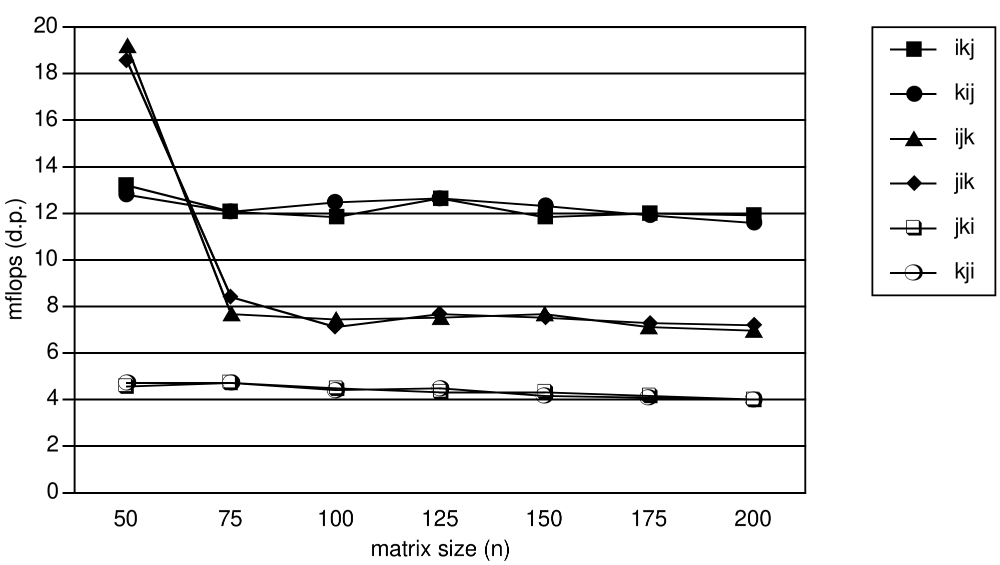
kji (2 loads/1 store)

```
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;
   }
}
```

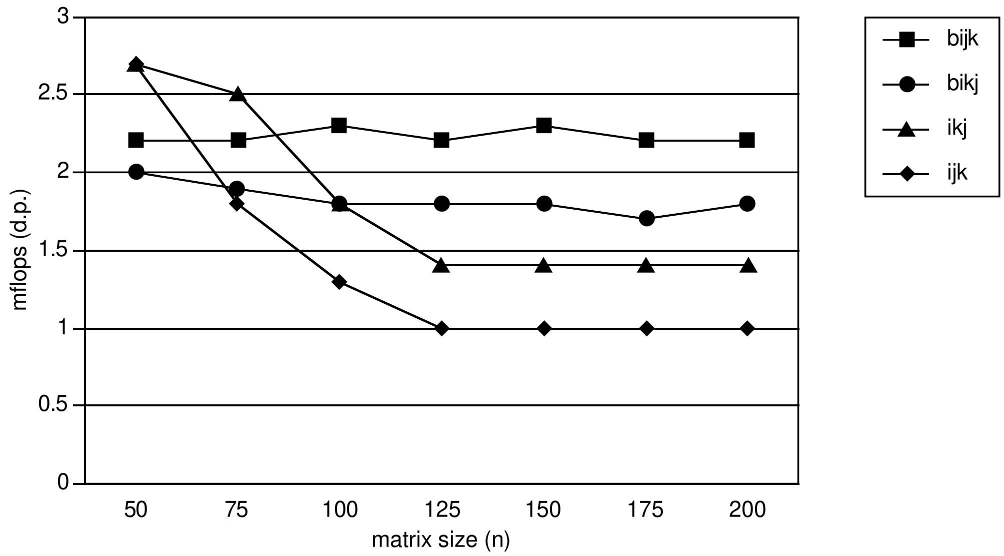
Matmult performance (DEC5000)



Matmult performance (Sparc20)



Blocked matmult perf (DEC5000)



Blocked matmult perf (Sparc20)