CACHE-CONSCIOUS ALGORITHMS

Daniel P. Bovet
University of Rome “Tor Vergata”

March 12, 2001
CACHES INCLUDED IN COMPUTER SYSTEMS

+ **hardware caches**: integrated into CPU chip, reduce average RAM access time
+ **software caches**: buffers handled by the kernel, reduce average disk access time
+ **disk controller caches**: integrated into disk controller, reduce average disk access time
IMPORTANCE OF HARDWARE CACHES

+ experiment on Intel 80x86 processors
  ? modify the kernel initialization routine by inserting an Assembly language instruction to set the CD flag of the cr0 control register
  ? boot the modified kernel: the system runs about 50 times slower
+ hw caches and multiple CPUs don’t mix well
MAIN INTEL P6 PROCESSOR
CACHES (1/2)

+ level 1 (L1) instruction and data caches (8-16 KB)
+ level 2 (L2) unified cache (128kb - 2 MB)
+ TLB instruction and data cache: reduce RAM accesses to page table entries (32 entries, 64 entries)
+ Write Buffer: reduce writes to RAM by optimizing bus accesses (12 entries)
Intel 80x86 processors make use of three kinds of addresses:

- **logical**: (16-bit segment selector + 32-bit offset), used in machine language instructions
- **linear**: (32-bit unsigned integer), the segmentation unit transforms logical addresses into linear ones
ADDRESS TERMINOLOGY (2/2)

+ **physical**: (electrical signals sent to pins A31-A0 of the chip), the paging unit transforms linear addresses into physical ones

+ *only logical addresses are accessible to programmers!*
BYPASSING THE SEGMENTATION UNIT (LINUX)

+ the segmentation unit can never be turned off
+ Linux bypasses the segmentation unit by using 2 pairs of segments having base address set to 0 and length set to the maximum length \((2^{32} - 1)\)
+ as a result, the offset of a logical address coincides with the linear address!
MAIN TYPES OF HARDWARE CACHING

+ **uncacheable**: data are always written to or read from RAM

+ **write-through**: read misses cause cachefills, write misses do not cause cache fills

+ **write-back**: read misses cause cache fills, write misses cause cache fills
SOFTWARE CONTROL ON HARDWARE CACHES (1/2)

+ on the whole cache system: CD flag of cr0 control register
+ on a range of contiguous physical addresses: Memory Type Range Registers (MTRR), used by BIOS
+ on a 4 KB linear address interval: flags PCD and PWT present in each page table entry
SOFTWARE CONTROL ON
HARDWARE CACHES (2/2)

+ encoding of hardware caching types for a 4 KB page frame:
  + PCD = 0, PWT = 0 ---> write-back
  + PCD = 0, PWT = 1 ---> write-through
  + PCD = 1, PWT = - ---> uncachable
PROGRAM LOADING WITH DEMAND PAGING

+ instead of getting page frames, the loaded program gets a set of intervals of linear addresses (process address space)

+ when a loaded program is executed, page fault exceptions occur: if the linear address that caused the exception belongs to the process address space, the kernel assigns a new page frame to it and forces re-execution of the instruction
IMPLEMENTATION OF LINEAR ADDRESS INTERVALS

+ each linear address interval owned by a process is managed by the kernel by means of one or more memory region descriptors

+ memory region descriptors are reserved to the kernel; User Mode processes ignore their existence

+ some requests for enlarging a process address space give raise to new memory regions, others increase the size of an existing memory region
HALF-BAKED IDEA

+ Linux does not take advantage of the PCD and PWT flags present in page table entries (they are always set to 0)

+ how about allowing programmers to get some control on the type of hardware caching associated with 4 KB page frames?
MOTIVATION

+ hardware caches have a limited size
+ instead of caching all data used by a program, avoid putting in the cache data that are written only once
+ by letting out of the cache write-once data, more cache space becomes available for data accessed repeatedly
RESEARCH PLAN

+ study the kind of linear address intervals that should be made *cache-sensible*
+ implement a new system call to set the cache mode of a linear address interval
+ study a way to get precise measurements of program execution times
+ write programs that make use of cache-sensible linear address intervals and test them
CACHE-SENSIBLE LINEAR ADDRESS INTERVALS (1/2)

+ the linear address interval must be created dynamically so that the program is aware of its length and initial address

+ the kernel must satisfy program requests for additional address space by creating new memory regions, instead of enlarging existing ones
CACHE-SENSIBLE LINEAR ADDRESS INTERVALS (2/2)

+ good candidates:
  ? memory mapping of a file
  ? IPC shared memory regions

+ bad candidates:
  ? malloc areas (they increase the size of the heap memory region)
  ? code region (created while loading the program)
THE mprotect() API

+ POSIX 1b standard specifies that mprotect() works with mapped files
+ Linux extends it to any kind of region
+ the kernel modifies the access right flags of all concerned regions
+ whenever a new page frame is granted to a region, the page table entry flags are set according to the memory region access rights
THE NEW `cprotect()` SYSTEM CALL

+ add to the flags of a memory region three mutually exclusive flags: `CACHE_NONE`, `CACHE_THROUGH`, `CACHE_BACK` (exactly one of them must be set)

+ since the linear address interval may span several memory regions, `cprotect()` acts iteratively on all regions included in the interval (same as `mprotect()`)

THE set_cache_mode() API

+ three input parameters:
  ? initial address of linear address interval
  ? length in bytes
  ? cache mode

+ performs the following tasks:
  ? rounds off first two parameters to a multiple of 4 KB
  ? checks the validity of third parameter
  ? invokes cprotect()
PROGRAM EXECUTION TIME

+ several factors make program execution times unpredictable
+ cache system: first execution of a program is slower than the following ones
+ demand paging: if short of memory, the kernel may stop the process while reclaiming memory
+ multitasking: program execution time-shared with other program executions
DRASTIC SOLUTION

+ cache system: clear hardware caches before performing measurement

+ demand paging: pre-allocate page frames by writing arbitrary values in all cache-sensitive linear address intervals used by the program

+ multitasking: disable maskable interrupts, including timer interrupts, while performing measurement
MEASURING TIME WITH INTERRUPTS DISABLED (1/2)

+ Linux relies on timer interrupts to keep track of elapsed time (kernel variable jiffies incremented once every 10 msec)
+ cannot use get_time_of_day() system call with interrupts disabled
+ rely on the 64-bit Time Stamp Counter (TSC) register that is incremented once every clock cycle
MEASURING TIME WITH INTERRUPTS DISABLED (2/2)

+ in order to transform TSC value into seconds, we must know the clock cycle frequency

+ in Linux this value is computed during system initialization and stored in the cpu_khz kernel variable

+ User Mode programs can retrieve the cpu_khz variable either via a new system call or via a proc file
two series of tests have been performed to evaluate the potential for cache conscious programs:

? tests on a classic matrix multiplication program

? tests on an artificial program that stretches the hardware cache
MATRICE MULTIPLICATION (1/2)

+ C = A * B

+ A, B, and C are memory mapped files

+ since we are not interested in average disk access times, A and B are read into memory in advance

+ in order to avoid page misses, C is also written in advance with dummy data

+ since we are not interested in saving the result on disk, PRIVATE memory mapping is used for C
for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
        x = 0;
        for (k=0; k<N; k++)
            x = x + M0[i,k] * M1[k,j];
        M2[i,j] = x;
    }

elements of M2 are written once, while elements of M0 and M1 are read many times!
## RESULTS ON INTEGER MATRIX MULTIPLICATION

<table>
<thead>
<tr>
<th>size</th>
<th>cache enabled for all matrices</th>
<th>cache disabled for M2 matrix</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>0.1484</td>
<td>0.1474</td>
<td>.67%</td>
</tr>
<tr>
<td>256</td>
<td>2.0184</td>
<td>2.0176</td>
<td>.04%</td>
</tr>
<tr>
<td>512</td>
<td>17.3225</td>
<td>17.3188</td>
<td>.02%</td>
</tr>
<tr>
<td>1024</td>
<td>146.4970</td>
<td>146.4336</td>
<td>.04%</td>
</tr>
</tbody>
</table>
## RESULTS ON FLOATING POINT MATRIX MULTIPLICATION

<table>
<thead>
<tr>
<th>size</th>
<th>cache enabled for all matrices</th>
<th>cache disabled for M2 matrix</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>0.0890</td>
<td>0.0890</td>
<td>0%</td>
</tr>
<tr>
<td>256</td>
<td>2.0199</td>
<td>2.0194</td>
<td>.05%</td>
</tr>
<tr>
<td>512</td>
<td>17.3417</td>
<td>17.3379</td>
<td>.02%</td>
</tr>
<tr>
<td>1024</td>
<td>146.8708</td>
<td>146.7691</td>
<td>.07%</td>
</tr>
</tbody>
</table>
A CACHE STRETCHING PROGRAM

+ matrix multiplication is not a good candidate since each entry $a[i,j]$ is read only $N$ times to compute element $c[i,*]$
+ long time interval between two consecutive read of $a[i,j]$
+ let us invent a “cache stretching” program
+ basic idea: scan portions of a sequence and write a lot of data once for each scanned portion
AN EXAMPLE OF CACHE STRETCHING PROGRAM

+ split a N*N sequence into N/M subsequences of length N*M
+ for each subsequence i:
  - scan randomly all N*M elements following p0[i]
  - select maximum number
  - copy N*N times maximum in randomly selected elements of output sequence p1[]
## RESULTS ON CACHE STRETCHING PROBLEM

<table>
<thead>
<tr>
<th>size</th>
<th>cache enabled for both arrays</th>
<th>cache disabled for output array</th>
<th>Cache Hit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>512/16</td>
<td>1.7897</td>
<td>0.9379</td>
<td>47%</td>
</tr>
<tr>
<td>512/32</td>
<td>0.9582</td>
<td>0.5004</td>
<td>48%</td>
</tr>
<tr>
<td>1024/16</td>
<td>13.7586</td>
<td>7.3103</td>
<td>47%</td>
</tr>
<tr>
<td>1024/32</td>
<td>6.8837</td>
<td>3.7841</td>
<td>45%</td>
</tr>
</tbody>
</table>