Caches
Virtual Memory
Parallel Computing: Why, and Why Now?

September 16, 2013
Before We Get Started…

● Last time
  ● Pipelining
  ● SRAM/DRAM

● Today
  ● Brief discussion of memory: caches and main memory
  ● Brief discussion of the Virtual Memory
  ● Parallel Computing: Why?, and Why Now?

● Miscellaneous
  ● First assignment, HW01, due on Monday at 11:59 PM
  ● Second assignment posted on the course website later today
  ● Read pages 28 through 56 of the primer available on the website
  ● Contact Andrew Seidl aaseidl@wisc.edu if you haven’t got an Euler account
  ● Make an early attempt to upload a file through Learn@UW. Check if it works for you
## Feature Comparison Between Memory Types

<table>
<thead>
<tr>
<th>Feature</th>
<th>SRAM</th>
<th>DRAM</th>
<th>Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speed</strong></td>
<td>Very fast</td>
<td>Fast</td>
<td>Very slow</td>
</tr>
<tr>
<td><strong>Density</strong></td>
<td>Low</td>
<td>High</td>
<td>Very high</td>
</tr>
<tr>
<td><strong>Endurance</strong></td>
<td>Good</td>
<td>Good</td>
<td>Poor</td>
</tr>
<tr>
<td><strong>Power</strong></td>
<td>Low</td>
<td>High</td>
<td>Very low</td>
</tr>
<tr>
<td><strong>Refresh</strong></td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Retention</strong></td>
<td>Volatile</td>
<td>Volatile</td>
<td>Non-volatile</td>
</tr>
<tr>
<td><strong>Mechanism</strong></td>
<td>Bi-stable Latch</td>
<td>Capacitor</td>
<td>Fowler-Nordheim tunneling</td>
</tr>
</tbody>
</table>
Cost and Speed Implications

- Since SRAM is expensive and bulkier, can’t have too much
  - Plagued by Space & Cost constraints

- Compromise:
  - Have some SRAM on-chip, making up what is called the “cache”
  - Have a lot of inexpensive DRAM off-chip, making up the “main memory”

- Hopefully your program has a low “average memory access time” by hitting the cache repeatedly instead of taking costly trips to main memory
You now have a “memory hierarchy”

Simplest memory hierarchy:
- Main Memory + One Cache (typically called L1 cache)

Today’s memory architectures typically have deeper hierarchy: L1+L2+L3
- L1 faster and smaller than L2
- L2 faster and smaller than L3

Note that all caches are typically on the chip
Example: Intel Chip Architecture

- Quad core Intel CPU die that illustrates L3 cache
- For Intel Core I7 975 Extreme, cache hierarchy is as follows
  - 32 KB L1 cache / core
  - 256 KB L2 (Instruction & Data) cache / core
  - 8 MB L3 (Instruction & Data) shared by all cores
Memory Hierarchy

- Memory hierarchy is deep:

Moving on to talk about caches
Cache Types

- Two main types of cache

- **Data** caches feed processor with data manipulated during execution
  - If processor would rely on data provided by main memory the execution would be pitifully slow
    - Processor Clock faster than the Memory Clock
    - Caches alleviate this memory pressure

- **Instruction** caches: used to store instructions
  - Much simpler to deal with compared to the data caches
    - Instruction use is much more predictable than data use

- In an ideal world, the processor would only communicate back and forth with the cache and avoid communication with the main memory
Split vs. Unified Caches

- Note that in the picture below L1 cache is split between data and instruction, which is typically the case.
- L2 and L3 (when present) typically unified.
How the Cache Works

- Assume simple setup with only one cache level L1

- Purpose of the cache: store for fast access a subset of the data stored in the main memory

- Data is moved at different resolutions between P ↔ C and between C ↔ M and
  - Between P and C: moved one word at a time
  - Between C and M: moved one block at a time (block called “cache line”)
Cache Hit vs. Cache Miss

- The processor typically agnostic about memory organization

- Middle man is the cache controller, which is an independent entity: it enables the “agnostic” attribute of the processor ↔ memory interaction
  - Processor requires data at some address
  - Cache Controller figures out if data is in a cache line
    - If yes: cache hit, processor served right away
    - If not: cache miss (data should be brought over from main memory → very slow)
  - Difference between cache hit and cache miss:
    - Performance hit related to SRAM vs. DRAM memory access
More on Cache Misses...

- A cache miss refers to a failed attempt to read/write a piece of data from/to the cache, which results in a main memory access with much longer latency.

- There are three kinds of cache misses:
  - **Cache read miss from an instruction cache**: generally causes the most delay, because the processor, or at least the thread of execution, has to wait (stall) until the instruction is fetched from main memory.
  - **A cache read miss from a data cache**: usually causes less delay, because instructions not dependent on the cache read can be issued and continue execution until the data is returned from main memory, and the dependent instructions can resume execution.
  - **A cache write miss to a data cache**: generally causes the least delay, because the write can be queued and there are few limitations on the execution of subsequent instructions. The processor can continue unless the queue is full and then it has to stall for the write buffer to partially drain.
Question:

- Can you control what’s in the cache and anticipate future memory requests?
  - Typically not…
    - Any serious system has a hardware implemented cache controller with a mind of its own
  - There are ways to increase your chances of cache hits by designing software for high degree of memory access locality
  - Two flavors of memory locality:
    - Spatial locality
    - Temporal locality
Spatial and Temporal Locality

- Spatial Locality for memory access by a program
  - A memory access pattern characterized by bursts of repeated requests for data that is physically located within the same memory region
  - “Bursts” because this accesses should happen in a sufficiently short interval of time (otherwise the cache line gets evicted)

- Temporal Locality for memory access by a program
  - Idea: If you access a variable at some time, then you’ll probably keep accessing the same variable for a while
  - Example: have a for loop with some variables inside the loop → you keep accessing those variables as long as you loop
Cache Characteristics

- Size attributes: absolute cache size and cache line size
- Strategy for mapping of memory blocks to cache lines
- Cache line replacement algorithms
- Write-back policies

NOTE: these characteristics carry over and become more convoluted when dealing with multilevel cache hierarchies
The Concept of Virtual Memory
Motivating Questions/Issues

- Assumption: we are not talking about embedded systems, which are running alone on a processor and basically do not require an operating system to play the role of the middle man.

- Question 1: On a 32 bit machine, how come you can have 512MB of main memory yet allocate an array of 1 GB?

- Question 2: How can you compile a program on a Windows workstation with 2 GB of memory and run it later on a different laptop with 512 MB of memory?

- Question 3: How can several processes run seemingly at the same time on a processor with one thread?
The three questions raised on previous slide answered by the interplay between the compiler, the operating system (OS), and the execution model adopted by the processor.

When you compile a program there is no way to know where in the physical memory the code will get its data allocated.
- There are other “tenants” that inhabit the memory, and they are there before you get there.

The solution is for the code to be compiled and assumed to lead to a process that executes in a virtual world in which it has access to 4 GB of memory (on 32 bit systems).
- The “virtual world” is called the virtual memory space.
Virtual vs. Physical Memory

- Virtual memory: this nice and immaculate space of $2^{32}$ addresses (on 32 bit architectures) in which a process sees its data being placed, the instructions stored, etc.

- Physical memory: a busy place that hosts at the same time data and instructions associated with tens of applications running on the system
Anatomy of the Virtual Memory

STACK segment
- [stores a collection of frames, each associated with one function call]
- [a stack frame stores function parameters, return addresses, local variables, etc.]
  - [last-in-first-out (LIFO) structure; push/pop managed]

STACK OVERFLOW
- [if top of stack reaches beyond this logical address]

HEAP segment
- [segment used when program allocates memory dynamically, at run time]
  - [managed by the OS in response to function calls like malloc, free, etc.]

BSS segment
- [stores uninitialized global and static variables]

DATA segment
- [stores static variables and initialized global variables]

TEXT segment
- [stores instructions associated with the program]
The Anatomy of the Stack

- Function `bar` and associated stack frame

```
float bar(int a, float b)
{
    int initials[2];
    float t1, t2, t3;
    //..code here..
    //..no other variables..
    return t1;
}
```
The Virtual Memory. The Page Table

- Virtual memory allows the processor to work in a virtual world in which each process, when run by the processor, seems to have exclusive access to a very large memory space.

- For 32 bits: memory space is 4 GB big.

- This virtual world is connected back to the physical memory through a Page Table.
Anatomy of a Virtual Memory Address

- A virtual address has two parts: the page number, and the offset
Anatomy of a Virtual Memory Address

- A page of virtual memory corresponds to a frame of physical memory.
- The size of a page (or frame, for that matter) is typically 4096 bytes.
- $2^{12} = 4096$: 12 address bits are sufficient to relatively position each byte in a page.
The Translation Process

- Example: imagine that your physical memory is 2 GB
- The physical address has 31 bits: $2^{31} = 2\text{GB}$
- Then the page table converts bits 12 through 31 of the virtual address into bits 12 through 30 of the physical address
Short Digression 1: The Unit of Address Resolution

- How many bits are available for data storage at each address?
- Example:
  - We have $2^{32}$ addresses that we can access
  - If each address points to a location that stores 8 bits (one byte) then we have 4 GB of addressable memory
  - However, if each address refers to a location that stores 2 bytes, we have 8 GB of addressable memory
- Intel and AMD CPUs: the unit of address resolution is 1 byte (8 bits)
- Consequence: the Intel 32 bit processors “see” a virtual memory space that can be 4 GB big
Short Digression 2: The 32 to 64 bit Migration

- If the architecture and OS have 32 bits to represent addresses, it means that $2^{32}$ addresses can be referenced.

- If unit of address resolution is 1 byte, that means that the size of the virtual memory space can be 4 GB.

- This is hardly enough today when programs are very large and the amounts of data they manipulate can be staggering.

- This motivated the push towards having addresses represented using 64 bits: the memory space balloons to $2^{64}$ bytes, that is 16 times 1152921504606846976 bytes.
Short Digression 3: The 32 to 64 Bit Migration

- Note that a 64 bit architecture typically calls for two things:
  - From a hardware perspective, the size of the registers, integer size, and word size is 64 bits
  - From a software perspective, the addresses are now 64 bits and therefore a program “operates” in a huge virtual memory space
    - The operating system (OS) is the party managing the execution of a program in the 64 bit universe
Comments on the Page Table Preamble to TLB.

- The page table is the key ingredient that allows the translation of virtual addresses into physical addresses.

- Every single process executing on a processor and managed by the OS has its own page table.

- Page table is stored in main memory.
  - For a 32 bit operating system size of a page table can be up to 4 MB in size.
Comments on the Page Table. The TLB

- If Page Table stored in main memory it means that each address translation would require a trip to main memory
  - This would be very costly

- There is a “cache” for this translation process: TLB
  - Translation lookaside buffer: holds the translation of a small collection of virtual page numbers into frame IDs

- Best case scenario: the TLB leads to a hit and allows for quick translation
- Bad scenario: the TLB doesn’t have the required information cached and a trip to main memory is in order
- Worst scenario: the requested frame is not in main memory and a trip to secondary memory is in order
  - Called “page fault”
Illustration: The Role of the TLB

- A TLB is just like a cache
- A TLB miss leads to substantial overhead in the translation of an address
Memory Access: The Big Picture

- A simplified version of how a memory request is serviced presented below