Projects

• Work together for the implementation
  – Discussion and debugging, but not the code itself
• Each submit your own implementation and report
• Presentation
  – One presentation
• Additional meeting time

<table>
<thead>
<tr>
<th>Name</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aditi Patil</td>
<td>Project 1</td>
</tr>
<tr>
<td>Aparna Puram</td>
<td>Project 1</td>
</tr>
<tr>
<td>Erik Hoggard</td>
<td>Project 1</td>
</tr>
<tr>
<td>Jacques Breaux</td>
<td>Project 1</td>
</tr>
<tr>
<td>Karam Abughalieh</td>
<td>Project 2</td>
</tr>
<tr>
<td>Kevin Weinert</td>
<td>Project 2</td>
</tr>
<tr>
<td>Mark Easterly</td>
<td>Project 2</td>
</tr>
<tr>
<td>Nathan Sketch</td>
<td>Project 2</td>
</tr>
<tr>
<td>Shayan Mukhtar</td>
<td>Project 2</td>
</tr>
</tbody>
</table>
Lecture 16: Parallel Architecture – Thread Level Parallelism

Concurrent and Multicore Programming

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan
Topics (Part 1)

• Introduction
• Principles of parallel algorithm design (Chapter 3)
• Programming on shared memory system (Chapter 7)
  – OpenMP
  – Cilk/Cilkplus
  – PThread, mutual exclusion, locks, synchronizations
• Analysis of parallel program executions (Chapter 5)
  – Performance Metrics for Parallel Systems
    • Execution Time, Overhead, Speedup, Efficiency, Cost
  – Scalability of Parallel Systems
  – Use of performance tools
Topics (Part 2)

- Parallel architectures and hardware
  - Parallel computer architectures
    - Thread level parallelism and data level parallelism
  - Memory hierarchy and cache coherency
- Manycore GPU architectures and programming
  - GPUs architectures
  - CUDA programming
  - Introduction to offloading model in OpenMP
- Programming on large scale systems (Chapter 6)
  - MPI (point to point and collectives)
  - Introduction to PGAS languages, UPC and Chapel
- Parallel algorithms (Chapter 8, 9 & 10)
Moore’s Law

- Long-term trend on the number of transistor per integrated circuit
- Number of transistors double every ~18 month

Binary Code and Instructions

loc_000000b1:

c9  
leave

c3  
ret

90  
nop

55  
push ebp

89e5  
mov ebp, esp

83e4f0  
and esp, 0xfffffffff0

83ec20  
sub esp, 0x20

dd05f0840408  
fld QWORD PTR ds:0x80484f0

dd5c2418  
fstp QWORD PTR [esp+0x18]

dd442418  
fld QWORD PTR [esp+0x18]

dd5c2404  
fstp QWORD PTR [esp+0x4]

c70424e0840408  
mov DWORD PTR [esp], 0x80484e0

e8f5feffff  
call func_feffffd0

c9  
leave

c3  
ret

90  
nop

90  
nop

55  
push ebp
Stages to Execute an Instruction
Pipeline
Pipeline and Superscalar

<table>
<thead>
<tr>
<th>Instr. No.</th>
<th>Pipeline Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>2</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>3</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>4</td>
<td>IF ID EX MEM</td>
</tr>
<tr>
<td>5</td>
<td>IF ID EX</td>
</tr>
</tbody>
</table>

Clock Cycle: 1 2 3 4 5 6

Diagram showing pipeline stages.
What do we do with that many transistors?

- Optimizing the execution of a single instruction stream through
  - **Pipelining**
    - Overlap the execution of multiple instructions
    - Example: all RISC architectures; Intel x86 underneath the hood
  - **Out-of-order execution:**
    - Allow instructions to overtake each other in accordance with code dependencies (RAW, WAW, WAR)
    - Example: all commercial processors (Intel, AMD, IBM, SUN)
  - **Branch prediction and speculative execution:**
    - Reduce the number of stall cycles due to unresolved branches
    - Example: (nearly) all commercial processors
What do we do with that many transistors? (II)

- **Multi-issue processors:**
  - Allow multiple instructions to start execution per clock cycle
  - Superscalar (Intel x86, AMD, ...) vs. VLIW architectures

- **VLIW/EPIC architectures:**
  - Allow compilers to indicate independent instructions per issue packet
  - Example: Intel Itanium

- **Vector units:**
  - Allow for the efficient expression and execution of vector operations
  - Example: SSE - SSE4, AVX instructions
Limitations of optimizing a single instruction stream (II)

• Problem: within a single instruction stream we do not find enough independent instructions to execute simultaneously due to
  – data dependencies
  – limitations of speculative execution across multiple branches
  – difficulties to detect memory dependencies among instruction (alias analysis)

• Consequence: significant number of functional units are idling at any given time

• Question: Can we maybe execute instructions from another instructions stream
  – Another thread?
  – Another process?
The “Future” of Moore’s Law

- The chips are down for Moore’s law
  - [http://www.nature.com/news/the-chips-are-down-for-moores-law-1.19338](http://www.nature.com/news/the-chips-are-down-for-moores-law-1.19338)

- Special Report: 50 Years of Moore's Law

- Moore’s law really is dead this time
  - [http://arstechnica.com/information-technology/2016/02/moores-law-really-is-dead-this-time/](http://arstechnica.com/information-technology/2016/02/moores-law-really-is-dead-this-time/)

- Rebooting the IT Revolution: A Call to Action (SIA/SRC, 2015)
  - [https://www.semiconductors.org/clientuploads/Resources/RITR%20WEB%20version%20FINAL.pdf](https://www.semiconductors.org/clientuploads/Resources/RITR%20WEB%20version%20FINAL.pdf)
Thread-level parallelism

- Problems for executing instructions from multiple threads at the same time
  - The instructions in each thread might use the same register names
  - Each thread has its own program counter
- Virtual memory management allows for the execution of multiple threads and sharing of the main memory
- When to switch between different threads:
  - Fine grain multithreading: switches between every instruction
  - Course grain multithreading: switches only on costly stalls (e.g. level 2 cache misses)
Simultaneous Multi-Threaded (SMT)

• Convert Thread-level parallelism to instruction-level parallelism
Simultaneous multi-threading (II)

• Dynamically scheduled processors already have most hardware mechanisms in place to support SMT (e.g. register renaming)

• Required additional hardware:
  – Register file per thread
  – Program counter per thread

• Operating system view:
  – If a CPU supports $n$ simultaneous threads, the Operating System views them as $n$ processors
  – OS distributes most time consuming threads ‘fairly’ across the $n$ processors that it sees.
Example for SMT architectures (I)

• Intel Hyperthreading:
  – First released for Intel Xeon processor family in 2002
  – Supports two architectural sets per CPU,
  – Each architectural set has its own
    • General purpose registers
    • Control registers
    • Interrupt control registers
    • Machine state registers
  – Adds less than 5% to the relative chip size

Example for SMT architectures (II)

• IBM Power 5
  – Same pipeline as IBM Power 4 processor but with SMT support
  – Further improvements:
    • Increase associativity of the L1 instruction cache
    • Increase the size of the L2 and L3 caches
    • Add separate instruction prefetch and buffering units for each SMT
    • Increase the size of issue queues
    • Increase the number of virtual registers used internally by the processor.
Simultaneous Multi-Threaded Systems

- Works well if
  - Number of compute intensive threads does not exceed the number of threads supported in SMT
  - Threads have highly different characteristics (e.g. one thread doing mostly integer operations, another mainly doing floating point operations)

- Does not work well if
  - Threads try to utilize the same function units
  - Assignment problems:
    - e.g. a dual processor system, each processor supporting 2 threads simultaneously (OS thinks there are 4 processors)
    - 2 compute intensive application processes might end up on the same processor instead of different processors (OS does not see the difference between SMT and real processors!)
Synchronization between processors

• Required on all levels of multi-threaded programming
  – Lock/unlock
  – Mutual exclusion
  – Barrier synchronization

• Key hardware capability: *
  – Uninterruptable instruction capable of automatically retrieving or changing a value
### Race Condition

```c
int count = 0;
int * cp = &count;
....
*cp++; /* by two threads */
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Integer value</th>
</tr>
</thead>
<tbody>
<tr>
<td>read value</td>
<td>←</td>
<td>0</td>
</tr>
<tr>
<td>increase value</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>write back</td>
<td>→</td>
<td>1</td>
</tr>
<tr>
<td>read value</td>
<td>←</td>
<td>1</td>
</tr>
<tr>
<td>increase value</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>write back</td>
<td>→</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Integer value</th>
</tr>
</thead>
<tbody>
<tr>
<td>read value</td>
<td>←</td>
<td>0</td>
</tr>
<tr>
<td>increase value</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>write back</td>
<td>→</td>
<td>1</td>
</tr>
<tr>
<td>increase value</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>write back</td>
<td>→</td>
<td>1</td>
</tr>
</tbody>
</table>

void *thread_func (void *arg){
    int * cp (int *) arg;

    pthread_mutex_lock (&mymutex);
    *cp++;      // read, increment and write shared variable
    pthread_mutex_unlock (&mymutex);

    return NULL;
}
Synchronization

• Lock/unlock operations on the hardware level, e.g.
  – Lock returning 1 if lock is free/available
  – Lock returning 0 if lock is unavailable

• Implementation using atomic exchange (compare and swap)
  – Process sets the value of a register/memory location to the required operation
  – Setting the value must not be interrupted in order to avoid race conditions
  – Access by multiple processes/threads will be resolved by write serialization
Synchronization (II)

• Other synchronization primitives:
  – Test-and-set
  – Fetch-and-increment

• Problems with all three algorithms:
  – Require a read and write operation in a single, uninterruptable sequence
  – Hardware can not allow any operations between the read and the write operation
  – Complicates cache coherence
  – Must not deadlock
Load linked/store conditional

• Pair of instructions where the second instruction returns a value indicating, whether the pair of instructions was executed as if the instructions were atomic

• Special pair of load and store operations
  – Load linked (LL)
  – Store conditional (SC): returns 1 if successful, 0 otherwise

• Store conditional returns an error if
  – Contents of memory location specified by LL changed before calling SC
  – Processor executes a context switch
• Assembler code sequence to atomically exchange the contents of register R4 and the memory location specified by R1

try: MOV R3, R4
    LL R2, 0(R1)
    SC R3, 0(R1)
    BEQZ R3, try
    MOV R4, R2
Load linked/store conditional (III)

- Implementing fetch-and-increment using load linked and conditional store

``` assembly
try: LL    R2, 0(R1)
    DADDUI R3, R2, #1
    SC    R3, 0(R1)
    BEQZ  R3, try
```

- Implementation of LL/SC by using a special Link Register, which contains the address of the operation
Spin locks

- A lock that a processor continuously tries to acquire, spinning around in a loop until it succeeds.
- Trivial implementation

```
DADDUI    R2, R0, #1
lockit:   EXCH    R2, 0(R1)   !atomic exchange
           BNEZ    R2, lockit
```

- Since the EXCH operation includes a read and a modify operation
  - Value will be loaded into the cache
    - Good if only one processor tries to access the lock
    - Bad if multiple processors in an SMP try to get the lock (cache coherence)
  - EXCH includes a write attempt, which will lead to a write-miss for SMPs
Spin locks (II)

- For cache coherent SMPs, slight modification of the loop required

lockit:  
LD  R2, 0(R1)  !load the lock
BNEZ  R2, lockit !lock available?
DADDUI R2, R0, #1  !load locked value
EXCH  R2, 0(R1)  !atomic exchange
BNEZ  R2, lockit  !EXCH successful?
Spin locks (III)

• ...or using LL/SC

lockit:   LL  R2, 0(R1)  !load the lock
          BNEZ  R2, lockit !lock available?
          DADDUI R2, R0, #1  !load locked value
          SC   R2, 0(R1)   !atomic exchange
          BNEZ  R2, lockit  !SC successful?