Concurrency

Hardware Trends

Moore’s law (attributed to Gordon Moore, Founder of Intel): Circuit complexity double every 18 months (revised 1975)

Image from http://www.intel.com/technology/silicon/mooreslaw/pix/mooreslaw_chart.gif

Leveraging Moore’s Law

• More transistors – opportunities for exploiting parallelism
  – Implicit parallelism
    • Pipelining
    • Superscalar
  – Explicit parallelism
    • Streaming and multimedia processor extensions
      – E.g., MMX, Altivec
    • Very long instruction words (VLIW)

Uniprocessor Limits

The power problem!

Current Trends

• Problems:
  – Fundamental circuit delay and heat limitations
  – Limited amount of instruction-level parallelism
• Solutions: proliferation of (from Sun, IBM, Intel)
  – Multithreading
  – Multicore
  – Multiprocessors

Explicit Parallelism is here to stay!

Other examples: niagara, montecito
Challenge: Ease the task of writing multithreaded code

Sun’s Niagara/Rock

Rock projected to Have 16 cores, 8 threads each

Intel’s 80-core Prototype - Polaris

1 Teraflop
Previously requiring 2,500 sq. ft!

http://techresearch.intel.com/articles/Tera-Scale/1449.htm
Top 500 List of Supercomputers (www.top500.org)

- Top 5 from the list
  - BlueGene/L – 131072 processors
  - Red Storm (2.4 GHz dual core opterons from Cray) – 26544 processors
  - eServer BlueGene Solution – 40960 processors
  - ASC Purple – eServer pSeries from IBM – 12208
  - IBM Myrinet cluster in Barcelona – 10240 processors

Shared Memory: A Look Underneath

Data Sharing: OS support

- Processes: default shared nothing (copy-on-write); change by
  - mmap
  - shmget/attach
- Threads: default shared everything (except stack); change by
  - Data structure conventions
  - Support for special private regions in thread libraries (with higher overhead)
Data Sharing: CPU and Cache Support

- Special atomic read-modify-write instructions
  - Test-and-set, fetch-and-increment, load-linked/store conditional
- Coherent caches
  - Ensure that modifications propagate to copies

Snoop-Based Coherence

- Makes use of a shared broadcast medium to serialize events (all transactions visible to all controllers and in the same order)
  - Write update-based protocol
  - Write invalidate-based (e.g., basic MSI, MESI protocols)
- Cache controller uses a finite state machine (FSM) with a handful of stable states to track the status of each cache line
- Consists of a distributed algorithm represented by a collection of cooperating FSMs

A Simple Invalidate-Based Protocol
- State Transition Diagram

Synchronization

- Basic types
  - Mutual exclusion (only one process/thread allowed to be in the critical section)
  - Events (e.g., waiting for a condition)
Peterson’s Algorithm (Using Reads and Writes)

Process $P_i$

do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn==j) ;
    critical section
    flag[i] = false;
    remainder section
} while (1);

Solves the critical-section problem for two processes

Basic Hardware Mechanisms for Synchronization

- Test-and-set – atomic exchange
- Fetch-and-op (e.g., increment) – returns value and atomically performs op (e.g., increments it)
- Compare-and-swap – compares the contents of two locations and swaps if identical
- Load-locked/store conditional – pair of instructions – deduce atomicity if second instruction returns correct value

Test and Set Lock (Mutual Exclusion)

Lock (*lock) {
    while (test&set(lock) == 1);
}

UnLock (*lock) {
    *lock = 0;
}

Implementing Locks Using Test&Set

- On the x86 xchgl swaps the contents of a register and a memory location atomically

  _Lock_acquire:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %ecx
  Loop:
  movl $1, %edx
  xchgl %edx, (%ecx)
  cmpl $0, %edx
  jne Loop
  popl %ebp
  ret

  _Lock_release:
  movl $0, 4(%esp)
  ret
Implementing Locks Using Test&Set

• On the SPARC ldstub moves an unsigned byte into the destination register and rewrites the same byte in memory to all 1s

  _Lock_acquire:
  ldstub [%o0], %o1
  addcc %g0, %o1, %g0
  bne _Lock
  nop
  fin:
  jmpl %r15+8, %g0
  nop

  _Lock_release:
  st %g0, [%o0]
  jmpl %r15+8, %g0
  nop

Concurrency Issues

• Thread safety
• Reentrancy
• Races
• Deadlocks

Summary

• Exploiting parallelism requires/benefits from
  – Algorithm development
  – Operating system support
  – ISA support
  – Coherent cache support

Other Topics

• System-level I/O
  – Unix I/O: the file system interface
    • All I/O devices – networks, disks, and terminals – are modeled as files
    • Operations include open, close, read, write
  – OS support include
    • Descriptor table – each process’s open file descriptors
  – File table – set of open files in the system (reference count, current position)

• Networks
  – Client-server programming model
  – Sockets: OS abstraction for communication end-points with associated addresses and ports
  – Addressing:
    • Internet domain names
    • IP (Internet Protocol) addresses, ports
Socket System Calls for a Connection-Oriented Protocol

Data Representation

- Memory: a large single-dimensional, conventionally byte-addressable, untyped array
- Byte ordering – big versus little endian
- Possible common interpretations
  - Instruction
  - Integer
  - Floating point
  - character

Number Representation

An n digit number can be represented in any base as

MSD \[ \cdots \] LSD

\[ d_i \times base^i \]

The value of the \( i \)th digit \( d \) is \( d \times base^i \), where \( i \) starts at 0 and increases from right to left.

Decimal (base 10) is the natural human representation, binary (base 2) is the natural computer representation.

E.g. \( 1100_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 12_{10} \)
Assembly Characteristics

- **Minimal Data Types**
  - "Integer" data of 1, 2, or 4 bytes
  - Addresses (untyped pointers)
- **Data values**
  - Floating point data of 4, 8, or 10 bytes
  - No aggregate types such as arrays or structures
- **Primitive Operations**
  - Perform arithmetic function on register or memory data
  - Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
  - Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Summary: Abstract Machines

<table>
<thead>
<tr>
<th>Machine Models</th>
<th>Data</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1) char</td>
<td>1) loops</td>
</tr>
<tr>
<td>mem</td>
<td>2) int, float</td>
<td>2) conditionals</td>
</tr>
<tr>
<td>proc</td>
<td>3) double</td>
<td>3) switch</td>
</tr>
<tr>
<td>mem</td>
<td>4) struct, array</td>
<td>4) call</td>
</tr>
<tr>
<td>Stack</td>
<td>5) pointer</td>
<td>5) Proc. return</td>
</tr>
<tr>
<td>Cond. Codes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>alu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>processor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CISC vs. RISC

- **Original Debate**
  - Strong opinions!
  - CISC proponents---easy for compiler, fewer code bytes
  - RISC proponents---better for optimizing compilers, can make run fast with simple chip design
- **Current Status**
  - For desktop processors, choice of ISA not a technical issue
    - With enough hardware, can make anything run fast
    - Code compatibility more important
  - For embedded processors, RISC makes sense
    - Smaller, cheaper, less power

Computing with Logic Gates

- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs
  - With some, small delay

Voltage

Time

\[ a \text{AND} b = a \land b \]
\[ a \text{OR} b = a \lor b \]
\[ \neg a = \lnot a \]
Combinational Circuits

- Acyclic Network of Logic Gates
  - Continuously responds to changes on primary inputs
  - Primary outputs become (after some delay) Boolean functions of primary inputs

Arithmetic Logic Unit

- Combinational logic
  - Continuously responding to inputs
  - Control signal selects function computed
    - Corresponding to 4 arithmetic/logical operations in Y86
  - Also computes values for condition codes

Sequential Logic: Memory and Control

- Sequential:
  - Output depends on the current input values and the previous sequence of input values.
- Are Cyclic:
  - Output of a gate feeds its input at some future time.
- Memory:
  - Remember results of previous operations
  - Use them as inputs.
- Example of use:
  - Build registers and memory units.

Clocks

- Signal used to synchronize activity in a processor
- Every operation must be completed in the time between two clock pulses (or rising edges) --- the cycle time
- Maximum clock rate (frequency) determined by the slowest logic path in the circuit (the critical path)
Processor Summary

- **Design Technique**
  - Create uniform framework for all instructions
  - Want to share hardware among instructions
  - Connect standard logic blocks with bits of control logic
- **Operation**
  - State held in memories and clocked registers
  - Computation done by combinational logic
  - Clocking of registers/memories sufficient to control overall behavior
- **Enhancing Performance**
  - Pipelining increases throughput and improves resource utilization
  - Must make sure maintains ISA behavior

Code Optimization Summary

- **Loop fusion and unrolling**
- **Code Motion**
  - *Compilers are good at this for simple loop/array structures*
  - *Don’t do well in presence of procedure calls and memory aliasing*
- **Reduction in Strength**
  - Shift, add instead of multiply or divide
    - *compilers are (generally) good at this*
    - *Exact trade-offs machine-dependent*
  - Keep data in registers rather than memory
    - *compilers are not good at this, since concerned with aliasing*
- **Share Common Subexpressions**
  - *compilers have limited algebraic reasoning capabilities*

Locality

- **Principle of Locality:**
  - Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - **Temporal locality:** Recently referenced items are likely to be referenced in the near future.
  - **Spatial locality:** Items with nearby addresses tend to be referenced close together in time.

Locality Example:

- **Data**
  - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
  - Reference sum each iteration: **Temporal locality**
- **Instructions**
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**

Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte and have less capacity.
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.
- These fundamental properties complement each other beautifully.
- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.
An Example Memory Hierarchy

- L0: registers
- L1: on-chip L1 cache (SRAM)
- L2: on/off-chip L2/L3 cache (SRAM)
- L3: main memory (DRAM)
- L4: local secondary storage (local disks)
- L5: remote secondary storage (distributed file systems, Web servers)

Caches

- Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy:
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- Why do memory hierarchies work?
  - Programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
  - Net effect: A large pool of memory that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

Cache Organization

- Location – how do you locate a block
- Placement – where is a block placed
- Re-placement – which block do you replace
  - Least recently used (LRU)
  - FIFO
  - Random
- Write policy – what happens on a write
  - Write back
  - Write through no write allocate
  - Write through write allocate

Exceptions

- An exception is a transfer of control to the OS in response to some event (i.e., change in processor state)

Event current next

User Process OS

exception processing by exception handler

exception return (optional)
Interrupt Vectors

Each type of event has a unique exception number k
Index into jump table (a.k.a., interrupt vector)
Jump table entry k points to a function (exception handler).
Handler k is called each time exception k occurs.

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor’s interrupt pin
  - Handler returns to “next” instruction.
- Examples:
  - I/O interrupts
    - hitting ctrl-c at the keyboard
    - arrival of a packet from a network
    - arrival of a data sector from a disk
  - Hard reset interrupt
    - hitting the reset button
  - Soft reset interrupt
    - hitting ctrl-alt-delete on a PC

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - Traps
    - Intentional
    - Examples: system calls, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - Faults
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable).
    - Either re-executes faulting (“current”) instruction or aborts.
  - Aborts
    - Unintentional and unrecoverable
    - Examples: parity error, machine check.
    - Aborts current program

Exceptional Control Flow

- Mechanisms for exceptional control flow exists at all levels of a computer system
- Low level Mechanism
  - exceptions
    - change in control flow in response to a system event (i.e., change in system state)
  - Combination of hardware and OS software
- Higher Level Mechanisms
  - Process context switch
  - Signals
  - Nonlocal jumps (setjmp/longjmp)
  - Implemented by either:
    - OS software (context switch and signals)
    - C language runtime library: nonlocal jumps
Processes

- **Def:** A *process* is an instance of a running program.
  - One of the most profound ideas in computer science.
  - Not the same as "program" or "processor"
- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU.
  - Private address space
    - Each program seems to have exclusive use of main memory.
- How are these illusions maintained?
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system

Virtual Memory

- **Programmer’s View**
  - Large "flat" address space
    - Can allocate large blocks of contiguous addresses
  - Processor "owns" machine
    - Has private address space
    - Unaffected by behavior of other processes
- **System View**
  - User virtual address space created by mapping to set of pages
    - Need not be contiguous
    - Allocated dynamically
    - Enforce protection during address translation
  - OS manages many processes simultaneously
    - Continually switching among processes
    - Especially when one must wait for resource
      - E.g., disk I/O to handle page fault

Harsh Reality

- **Memory Matters**
  - Memory is not unbounded
    - It must be allocated and managed
    - Many applications are memory dominated
      - Especially those based on complex, graph algorithms
  - Memory referencing bugs especially pernicious
    - Effects are distant in both time and space
  - Memory performance is not uniform
    - Cache and virtual memory effects can greatly affect program performance
    - Adapting program to characteristics of memory system can lead to major speed improvements

Dynamic Memory Allocation

- **Explicit vs. Implicit Memory Allocator**
  - Explicit: application allocates and frees space
    - E.g., `malloc` and `free` in C
  - Implicit: application allocates, but does not free space
    - E.g., garbage collection in Java, ML or Lisp
- **Allocation**
  - In both cases the memory allocator provides an abstraction of memory as a set of blocks
  - Doles out free memory blocks to application
Memory-Related Bugs

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks