COSC 6385
Computer Architecture
- Multi-Processors (II)
Simultaneous multi-threading and multi-core processors

Edgar Gabriel
Fall 2008

Moore’s Law

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

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: all commercial processors (Intel, AMD, IBM, SUN)

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 series
- Vector units:
  - Allow for the efficient expression and execution of vector operations
  - Example: SSE, SSE2, SSE3, 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 instructions (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?

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-Threading (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:
  - Registerfile 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

ftp://download.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf

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-Threading

- 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!)

Multi-Core processors

- Next step in the evolution of SMT: replicate not just the architectural state, but also the functional units
- Compute cores on a multi-core processor share the same main memory -> SMP system!
- Difference to previous multi-processor systems:
  - compute cores are on the same chip
  - Multi-core processors typically connected over a cache, while previous SMP systems were typically connected over the main memory
    - Performance implications
    - Cache coherence protocol
Multi-core processors: Example (I)

- Intel X7350 quad-core (Tigerton)
  - Private L1 cache: 32 KB instruction, 32 KB data
  - Shared L2 cache: 4 MB unified cache

Multi-core processors: Example (I)

- Intel X7350 quad-core (Tigerton) multi-processor configuration
Multi-core processors: Example (II)

- AMD 8350 quad-core Opteron (Barcelona)
  - Private L1 cache: 32 KB data, 32 KB instruction
  - Private L2 cache: 512 KB unified
  - Shared L3 cache: 2 MB unified

Multi-core processors: Example (IV)

- AMD 8350 quad-core Opteron (Barcelona): multi-processor configuration
  - It’s a NUMA!
### Comparison Intel Tigerton vs. AMD Barcelona

<table>
<thead>
<tr>
<th>Chip</th>
<th>Speed (GHz)</th>
<th>Peak (GFlops)</th>
<th>L1 (KB)</th>
<th>L2 (MB)</th>
<th>L3 (MB)</th>
<th>Power (W)</th>
<th>No. of transistors</th>
<th>Mem. control llers</th>
<th>Peak (GFlops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Tigerton</td>
<td>2.93</td>
<td>46.9</td>
<td>64</td>
<td>4</td>
<td>-</td>
<td>130</td>
<td>582</td>
<td>1</td>
<td>187.6</td>
</tr>
<tr>
<td>AMD Barcelona</td>
<td>2.0</td>
<td>32</td>
<td>64</td>
<td>0.5</td>
<td>2</td>
<td>75</td>
<td>463</td>
<td>4</td>
<td>128.0</td>
</tr>
</tbody>
</table>


### Programming for multi-core

- Programmers must use threads or processes
- Spread the workload across multiple cores
- Write parallel algorithms
- OS will map threads/processes to cores

- True concurrency, not just uni-processor time-slicing
  - Pre-emptive context switching: context switch can happen at any time
  - Concurrency bugs exposed much faster with multi-core

Programming for multi-core (II)

- Each thread/process has an affinity mask
  - Affinity mask specifies what cores the thread is allowed to run on
  - Different threads can have different masks
  - Affinities are inherited across fork()
- Example: 4-way multi-core, without SMT

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>

core 3  core 2  core 1  core 0

- Process/thread is allowed to run on cores 0,2,3, but not on core 1

Affinity masks when multi-core and SMT combined

- Separate bits for each simultaneous thread
- Example: 4-way multi-core, 2 threads per core

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>

thread 1  thread 0  thread 1  thread 0  thread 1  thread 0  thread 1  thread 0  thread 1  thread 0  thread 1  thread 0

- Core 2 can’t run the process
- Core 1 can only use one simultaneous thread
Process migration is costly

- Need to restart the execution pipeline
- Cached data is invalidated
- OS scheduler tries to avoid migration as much as possible: it tends to keeps a thread on the same core
- This is called *soft affinity*
- Default Affinities
  - Default affinity mask is all 1s: all threads can run on all processors
  - Then, the OS scheduler decides what threads run on what core
  - OS scheduler detects skewed workloads, migrating threads to less busy processors

Hard affinities

- The programmer can prescribe her own affinities (hard affinities)
- Rule of thumb: use the default scheduler unless a good reason not to
- When to set your own affinity
  - Two (or more) threads share data-structures in memory -> map to same core or to cores that are close to each other, so that they can share cache
  - Real-time threads: e.g. running controller thread which must not be context switched
#include <sched.h>

int sched_getaffinity(pid_t pid, 
    unsigned int len, unsigned long * mask);

Retrieves the current affinity mask of process ‘pid’ and 
stores it into space pointed to by ‘mask’.  
‘len’ is the system word size: sizeof(unsigned int long)

---

#include <sched.h>

int sched_setaffinity(pid_t pid, 
    unsigned int len, unsigned long * mask);

Sets the current affinity mask of process ‘pid’ to *mask 
‘len’ is the system word size: sizeof(unsigned int long)

To query affinity of a running process:  
[barbic@bonito ~]$ taskset -p 3935  
pid 3935's current affinity mask: f
Windows APIs

- BOOL WINAPI GetProcessAffinityMask(
  HANDLE hProcess,
  PDWORD_PTR lpProcessAffinityMask,
  PDWORD_PTR lpSystemAffinityMask);
- BOOL WINAPI SetProcessAffinityMask(
  HANDLE hProcess,
  DWORD_PTR dwProcessAffinityMask);


Legal licensing issues

- Will software vendors charge a separate license per each core or only a single license per chip?
- Microsoft, Red Hat Linux, Suse Linux will license their OS per chip, not per core