COSC 4397

Simultaneous multi-threading and multi-core processors

Edgar Gabriel
Spring 2010

Moore’s Law

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

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 series
- Vector units:
  - Allow for the efficient expression and execution of vector operations
  - Example: SSE, SSE2, SSE3, SSE4 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?

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/ijt/2002/volume06issue01/vol6iss1_hyp

er_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

![Diagram of multi-core processors with Intel X7350 quad-core (Tigerton)]
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): multiprocessor 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 nodes</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>


Memory Bandwidth

(a) Aggregate memory bandwidth
(b) Bandwidth per core

Processor locality

(a) Intel Xeon
(b) AMD Barcelona

Figure 4. Observed latency from any core to any other core in a node


Single core performance comparison

Single node application performance

![Bar chart comparing Intel Tigerton and AMD Barcelona performance across different applications.]


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

- 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

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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

Process migration is costly

- Default Affinities
  - Default affinity mask is all 1s: all threads can run on all processors
  - OS scheduler decides what threads run on what core
  - OS scheduler detects skewed workloads, migrating threads to less busy processors
- Soft affinity:
  - tendency of a scheduler to try to keep processes on the same CPU as long as possible
- Hard affinity:
  - Affinity information has been explicitly set by application
  - OS has to adhere to this setting
Retrieve the current affinity mask of a process

```c
#include <sys/types.h>
#include <sched.h>
#include <unistd.h>
#include <errno.h>

unsigned int len = sizeof(cpu_set_t);
cpu_set_t mask;
pid_t pid = getpid(); /* get the process id of this app */

ret = sched_getaffinity (pid, len, &mask);
if ( ret != 0 )
    printf("Error in getaffinity %d (%s)\n", errno, strerror(errno));
for (i=0; i<NUMCPUS; i++) {
    if ( CPU_ISSET(i, &mask) )
        printf("Process could run on CPU %d\n", i);
}
```

Set the affinity mask of a process

```c
/* clear the mask */
CPU_ZERO (&mask);

/* set the mask such that the process is only allowed to
execute on the desired CPU */
CPU_SET ( cpu_id, &mask);

ret = sched_setaffinity (pid, len, &mask);
if ( ret != 0 ) {
    printf("Error in setaffinity %d (%s)\n", errno, strerror(errno));
}
```
Linux Kernel scheduler API (III)

- Setting thread-related affinity information
  - Use sched_setaffinity with a pid = 0
    - Changes the affinity settings for this thread only
  - Use libnuma functionality

```c
numa_run_on_node();
uma_run_on_node_mask();
```

- Modifying affinity information based on CPU sockets, not on cores
  - Use pthread functions on most Linux systems

```c
#define __USE_GNU
pthread_setaffinity_np(thread_t t, len, mask);
pthread_attr_setaffinity_np ( thread_attr_t a, len, mask);
```

Windows APIs

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