多线程技术 Multithreading

• Difficult to continue to extract ILP from a single thread
• Many workloads can make use of thread-level parallelism
  - TLP from multiprogramming (run independent sequential jobs, process level parallel)
  - TLP from multithreaded applications (run one job faster using parallel threads)
• Multithreading uses TLP to improve utilization of a single processor

Greatest trend in VLSI generation is increase in parallelism

- Up to 1985: bit level parallelism: 4-bit -> 8 bit -> 16-bit
  - slows after 32 bit
  - adoption of 64-bit now under way, 128-bit far (not performance issue)
  - great inflection point when 32-bit micro and cache fit on a chip
- Mid 80s to mid 90s: instruction level parallelism, ILP
  - pipelining and simple instruction sets, + compiler advances (RISC)
  - on-chip caches and functional units => superscalar execution
  - greater sophistication: out of order execution, speculation, prediction
    - to deal with control transfer and latency problems
- Next step: thread level parallelism, TLP
- And process-level parallelism

指令集并行的上限研究

- 无限的资源和取指宽度，完美的指令预测和寄存器重命名
- 理想的Cache，不为0的失效开销
Commercial Computing

- Relies on parallelism for high end
  - Computational power determines scale of business that can be handled
- Databases, online-transaction processing, decision support, data mining, data warehousing ...
- TPC benchmarks (TPC-C order entry, TPC-D decision support)
  - Explicit scaling criteria provided
  - Size of enterprise scales with size of system
  - Problem size not fixed as p increases.
  - Throughput is performance measure (transactions per minute or tpm)

Multimedia user interfaces
- 3D graphics
- Full-motion video
- Image recognition
- Voice generation
- Voice recognition

Future applications on physical modeling
- Computational chemistry
- Fluid dynamics

Compiler: loop-level to thread-level [SUIF, 1996]
Loop-level and thread-level parallelism

Multiprocessor-aware operating systems and environments
- Microsoft Windows NT and Unix
- Execute separate applications in parallel to increase throughput and provide a more responsive computing environment.

Multimedia-like user environments have enabled users to run more applications simultaneously

I/O Management: Interrupt, DMA, I/O processor

Process-level parallelism is increasing as a result
Process-level and thread-level parallelism is the future

Several reasons for the decreasing processor utilization in parallel machine and pipeline microprocessor
- I/O latency
- Memory latency
- Network transaction and synchronization latency
- Functional unit latency because of hazards

Timing-sharing
Thread-level Overlap
多线程解决方案

- How can we guarantee no dependencies between instructions in a pipeline?
  - One way is to interleave execution of instructions from different program threads on same pipeline

Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe

**CDC 6600 Peripheral Processors (Cray, 1964)**

- First multithreaded hardware
- 10 "virtual" I/O processors
- Fixed interleave on simple pipeline
- Pipeline has 100ns cycle time
- Each virtual processor executes one instruction every 1000ns
- Accumulator-based instruction set to reduce processor state
多线程软件

- 进程 Process
  - 每个进程都有其唯一的地址空间
  - 可以由几个线程组成
- 线程 Thread
  - 每个线程有自己的 PC + 寄存器 + 栈
  - 所有线程（在同一个进程中）共享相同的地址空间
  - 私有堆栈是可选的
- Multithreaded app’s: process broken into threads

多线程体系结构

- 进程 Process
  - 每个进程都有其唯一的地址空间
- 线程 Thread
  - 每个线程都有自己的执行上下文 - PC 为每个线程
    - 分离的 PC + 寄存器
- Multithreaded Architecture: processor capable of executing multiple software threads
  - 可以同时执行
    - 线程可以是 HW 切换，无需 OS 控制
- 共享资源
  - 分配 - 更好的资源利用率 - 更好的吞吐量
- 可以属于同一个进程 - 但不必属于!

多线程的代价

- 每个线程都需要自己的用户状态
  - PC
  - 寄存器
  - 状态寄存器
- 也需要自己的系统状态
  - 虚拟内存页面表基寄存器
  - 异常处理寄存器
- Other costs?
**Thread Creation and Scheduling**

**Explicit Multithreading**
- Created by the program
- Fine-grained/coarse-grained/SMT

**Implicit Multithreading**
- Automatic parallelization of serial algorithms on run time
- Efficient hardware support for solving problems associated with the efficient extraction of multiple thread execution
- Multiscalar
- Dynamic multithreading
- Speculative multithreading

---

**Denelcor HEP (Burton Smith, 1982)**

First commercial machine to use hardware threading in main CPU
- 120 threads per processor
- 10 MHz clock rate
- Up to 8 processors
- Precursor to Tera MTA (Multithreaded Architecture)

**Tera MTA (1990-97)**

- Up to 256 processors
- Up to 128 active threads per processor
- Processors and memory modules populate a sparse 3D torus interconnection fabric
- Flat, shared main memory
  - No data cache
  - Sustains one main memory access per cycle per processor
- GaAs logic in prototype, 1KW/processor @ 260MHz
  - CMOS version, MTA-2, 50W/processor

---

**Thread Creation and Scheduling**

- **Fixed Interleave (CDC 6600 PPU, 1964)**
  - Each of N threads executes one instruction every N cycles
  - If thread not ready to go in its slot, insert pipeline bubble

- **Software-controlled Interleave (TI ASC PPU, 1971)**
  - OS allocates S pipeline slots amongst N threads
  - Hardware performs fixed interleave over S slots, executing whichever thread is in that slot

- **Hardware-controlled Thread Scheduling (HEP, 1982)**
  - Hardware keeps track of which threads are ready to go
  - Picks next thread to execute based on hardware priority scheme
Fine-Grain and Coarse-Grain Multithreading

- Fine-Grained Multithreading designed for applications with large data sets and low locality
  - Many parallel threads needed to hide large memory latency
  - No control hazard and Or don’t need branch prediction
- Coarse-Grained Multithreading for Other applications are more cache friendly,
  - Few pipeline bubbles when cache getting hits
  - Just add a few threads to hide occasional cache miss latencies
  - Swap threads on cache misses

MIT Alewife (1990)
- Modified SPARC chips
  - register windows hold different thread contexts
- Up to four threads per node
- Thread switch on local cache miss

IBM PowerPC RS64-IV (2000)
- Commercial coarse-grain multithreading CPU
- Based on PowerPC with quad-issue in-order five-stage pipeline
- Each physical CPU supports two virtual CPUs
- On L2 cache miss, pipeline is flushed and execution switches to second thread
  - short pipeline minimizes flush penalty (4 cycles), small compared to memory access latency
  - flush pipeline to simplify exception handling

SMT, Simultaneous Multithreading
- A technique that permits several independent threads to issue to multiple functional units each cycle
- The objective of SMT is to substantially increase processor utilization in the face of both long memory latencies and limited available parallelism per thread
- Combine superscalar processors with the latency-hiding ability of multithreaded architectures.
- Challenges
  - achieving high register file bandwidth
  - supporting high memory access demands
  - meeting large forwarding requirements
  - scheduling instructions onto functional units
Scalar Execution

- Dependencies reduce throughput/utilization

Superscalar Execution

- Generally increases throughput, but decreases utilization

Predication

- Generally increases utilization, increases throughput less
- Much of the utilization is thrown away

CMP – Chip Multi-Processor

- Low utilization / higher throughput
Coarse-grained Multithreading

- May increase utilization and throughput, but must switch when current thread goes to low utilization/throughput section (e.g. L2 cache miss)
- Advantage: small changes to existing hardware
- Drawback: high single thread perf. requires long thread switch

Fine-grained Multithreading

- Increases utilization/throughput by reducing impact of dependences
- Drawback:
  - Complicated hardware, Multiple contexts (states)
  - (w/ inflexible interleave:) limits single thread performance

Simultaneous Multithreading, SMT

- Increases utilization/throughput
- Complicated hardware, Multiple contexts (states)

IBM Power4/Power5

- 2 fetch (PC), 2 initial decodes
- 2 commits (architected register sets)
Pentium4 HyperThreading (2002)

- First commercial SMT design (2-way SMT)
  - Hyperthreading == SMT
- Logical processors share nearly all resources of the physical processor
  - Caches, execution units, branch predictors
- Die area overhead of hyperthreading ~ 5%
- When one logical processor is stalled, the other can make progress
  - No logical processor can use all entries in queues when two threads are active
- Processor running only one active software thread runs at approximately same speed with or without hyperthreading
**Execution Pipeline**

- I-Fetch
- Fetch Queue
- Rename
- Uop Queue
- Sched
- Register Read
- Execute
- D-Cache
- Store Buffer
- Register
- ROB
- LT D-Cache
- Retire Queue

**Sun OpenSparc T1/T2**

- Through-put computing
- Simple pipeline architecture

**Design challenges in SMT**

- Not to gain high single-thread performance
- Large register file
- Frequency
- Cache and TLB
- Enough Performance vs. High Performance
- Low utilization of Superscalar

**Initial Performance of SMT**

- Pentium 4 Extreme SMT yields 1.01 speedup for SPECint_rate benchmark and 1.07 for SPECfp_rate
  - Pentium 4 is dual threaded SMT
  - SPECRate requires that each SPEC benchmark be run against a vendor-selected number of copies of the same benchmark
- Running on Pentium 4 each of 26 SPEC benchmarks paired with every other (26^2 runs) speed-ups from 0.90 to 1.58; average was 1.20
- Power 5, 8-processor server 1.23 faster for SPECint_rate with SMT, 1.16 faster for SPECfp_rate
- Power 5 running 2 copies of each app speedup between 0.89 and 1.41
  - Most gained some
  - Fl.Pt. apps had most cache conflicts and least gains
Intel® Xeon™ Processor MP

- Performance varies as expected with:
  - Number of parallel threads
  - Resource utilization
- Less aggressive MT than EV8 -> Less impressive scalability
  - Machine optimized for single-thread

Power 5 thread performance ...

Relative priority of each thread controllable in hardware.

For balanced operation, both threads run slower than if they “owned” the machine.

Multithreaded Programming Models

- **Boss / Worker I** – all tasks come into a single “Boss Thread” who passes the tasks off to worker threads that it creates on the fly.
- **Boss / Worker II** – all tasks come into a single “Boss Thread” who passes the tasks off to worker threads from a “thread pool” that the Boss created up front. (Avoids overhead of thread creation and destruction.)
- **Peer** – specific-task threads are created up front, all tasks come into the correct thread
- **Pipeline** – all tasks come into the first thread, who does a specific operation then passes them onto the second thread, etc.

多线程编程

- “Concurrent Programming”: Tasks can occur in any order
- “Parallel Programming”: Simultaneous execution of concurrent tasks on different processors or cores

When is it good to use Multithreading?

- Where specific tasks can become blocked
- Where specific tasks can be CPU-intensive
- Where specific tasks must respond to asynchronous I/O, including the UI
- Where specific tasks have higher or lower priority than other tasks
- Where performance can be gained by overlapping I/O
- To manage independent behaviors in interactive simulation
- For use with the new multicore CPU chips
### pthreads Multithreaded Programming

- Pthreads is short for “Posix Threads”
- Posix is an IEEE standard for a Portable Operating System (section 1003.1c)
- The pthread paradigm is to spawn an application’s key procedures as separate threads
- All threads share a single global heap (malloc, new)
- Each thread has its own stack (subroutine arguments, local variables)

### pThreads multithreaded programming

- Creating pthreads: `pthread_create()`
- Waiting for pthreads to Finish: `pthread_join()`
- Synchronizing pthreads:
  - `pthread_mutex_t` `Sync`
  - `pthread_mutex_init(&Sync, NULL)`
  - `pthread_mutex_lock(&Sync)`
  - `pthread_mutex_unlock(&Sync)`
  - `pthread_mutex_trylock(&Sync)`
- Letting a pthread Identify Itself
  - `pthread_self()`
  - `pthread_equal()`
- Canceling another pthread
  - `pthread_cancel()`

### OpenMP multithreaded programming

- OpenMP is a multi-vendor standard
- The OpenMP paradigm is to issue C/C++ “pragmas” to tell the compiler how to build the threads into the executable

  ```
  #pragma omp directive [clause]
  ```

- All threads share a single global heap (malloc, new)
- Each thread has its own stack (subroutine arguments, local variables)

### OpenMP multithreaded programming

- Creating OpenMP threads in Loops
  ```
  #pragma omp parallel for private(i)
  ```
- Creating Sections of OpenMP Threads
  ```
  #pragma omp section
  ```
- Number of OpenMP threads
  ```
  omp_get_num_threads()
  omp_get_thread_num()
  ```
- Synchronizing OpenMP threads
  ```
  omp_lock_t `Sync`;
  omp_init_lock(&Sync);
  omp_set_lock(&Sync);
  omp_unset_lock(&Sync);
  omp_test_lock(&Sync);
  ```
**MultiScalar**

- Definition: A processor that internally breaks a single software thread into multiple hardware threads

- Goal: Use MT to Pursue Single Thread Performance

- Very hot topic in last few years

- But no machine exists, yet

- Examples:
  - MultiScalar
  - Dynamic Multithreading
  - Speculative Multithreading

---

**MultiScalar**

- Communication of data/control information - a key issue
  - Create/accumulate masks
  - Register values dynamically forwarded to later threads
    - May use value speculation for early computations
  - Relies on compiler/SW to help

- Address Resolution Buffer (ARB) performs dynamic memory disambiguation between threads
  - Holds speculative stores
  - Tracks speculative loads for dependence violation

- Thread squash

---

**Definition:**

"A processor that internally breaks a single software thread into multiple hardware threads"

**Goal:**

Use MT to Pursue Single Thread Performance

**Examples:**

- MultiScalar
- Dynamic Multithreading
- Speculative Multithreading

**Control Dependences**

- Joins in the control flow graph->future thread
- Disjoint Eager Execution
- Out-of-Order Thread execution

**Data Dependences**

- Intrathread/Interthread dependences

**Memory Dependences**

- Conventional load/store queues, Dynamic Multithreading
- Address resolution buffer (ARB), Multiscalar
- Cache Coherence, Speculative Multithreading

**MultiScalar**

- Threads are serial portions of Control Flow Graph (CFG) (Basic Block (BB), multiple BBs)
  - e.g. a full procedure or loop
  - Determined by compiler or at runtime

- Sequencer uses CFG to dynamically assign threads to PU
  - Threads are assigned to PUs and retired in sequential order
    - Head and tail PU
  - Threads try to execute simultaneously

---

**MultiScalar**

- Communication of data/control information - a key issue
  - Create/accumulate masks
  - Register values dynamically forwarded to later threads
    - May use value speculation for early computations
  - Relies on compiler/SW to help

- Address Resolution Buffer (ARB) performs dynamic memory disambiguation between threads
  - Holds speculative stores
  - Tracks speculative loads for dependence violation

- Thread squash

---

**MultiScalar**

- Threads are serial portions of Control Flow Graph (CFG) (Basic Block (BB), multiple BBs)
  - e.g. a full procedure or loop
  - Determined by compiler or at runtime

- Sequencer uses CFG to dynamically assign threads to PU
  - Threads are assigned to PUs and retired in sequential order
    - Head and tail PU
  - Threads try to execute simultaneously

---

**MultiScalar**

- Communication of data/control information - a key issue
  - Create/accumulate masks
  - Register values dynamically forwarded to later threads
    - May use value speculation for early computations
  - Relies on compiler/SW to help

- Address Resolution Buffer (ARB) performs dynamic memory disambiguation between threads
  - Holds speculative stores
  - Tracks speculative loads for dependence violation

- Thread squash

---

**MultiScalar**

- Threads are serial portions of Control Flow Graph (CFG) (Basic Block (BB), multiple BBs)
  - e.g. a full procedure or loop
  - Determined by compiler or at runtime

- Sequencer uses CFG to dynamically assign threads to PU
  - Threads are assigned to PUs and retired in sequential order
    - Head and tail PU
  - Threads try to execute simultaneously

---

**MultiScalar**

- Communication of data/control information - a key issue
  - Create/accumulate masks
  - Register values dynamically forwarded to later threads
    - May use value speculation for early computations
  - Relies on compiler/SW to help

- Address Resolution Buffer (ARB) performs dynamic memory disambiguation between threads
  - Holds speculative stores
  - Tracks speculative loads for dependence violation

- Thread squash
**Dynamic Multithreading**

- Concept similar to MultiScalar, but simpler
  - Fetch and execute speculatively beyond specific SW structures
    - procedure call
    - loop exit
    - Potentially: Loop body
    - Less dependencies
  - Easy identified control flow
  - Can use value prediction to resolve some of the data dependencies
  - Any good job done is a gain
    - Can use history/profile to guide where to do it
  - But can be wrong
- Basically can
  - Advance some computations
  - Or at least prefetch data
- Microarchitecture only - no compiler needed
- Need recovery mechanism

---

**Speculative Multithreading**

- Try to run ahead so to start memory accesses and resolve branches early (Memory prefetch, Branch preprocessing)
- Control-driven and Data-driven threads

---

**Data-driven thread**

- **Target**
  - Problem instruction
- **Body**
  - Dependence chain of target
  - No explicit control, branches executed but not “followed”
  - Implicit control: multiple-paths, embedded loops, etc.
- **Trigger**
  - Dynamically launches DDT
  - All external dependences of DDT satisfied by or prior to launch
- **Real Challenge**
  - Determine which DDT to build
  - Integrate DDT results with main thread
Disjoint Eager Execution (DEE)

- Eager Execution
- Disjoint Eager Execution
  - Eager Execution Three
  - Cumulative branch mispredictions
  - The branch path with highest cumulative prediction rate

多线程小结

- 显式多线程
  - Fine-grained
  - Coarse-grained
  - SMT
- 隐式多线程
  - Multiscalar
  - Dynamic multithreading
  - Speculative multithreading
- 多线程编程

嵌入式计算 embedded computing

A computer not used to run general-purpose programs, but instead used as a component of a larger system. Usually, user does not change the computer program (except for manufacturer upgrades).

Example applications:
- Toasters
- Cellphone
- Digital camera (some have several processors)
- Games machines
- Set-top boxes (DVD players, personal video recorders, ...)
- Televisions
- Dishwashers
- Car (some have dozens of processors)
- Internet router (some have hundreds to thousands of processors)
- Cellphone basestation
- .... many more
嵌入式应用的需求

- Real-time performance
  - *hard real-time*: if deadline missed, system has failed (car brakes!)
  - *soft real-time*: missing deadline degrades performance (skipping frames on DVD playback)

- Real-world I/O with multiple concurrent events
  - sensor and actuators require continuous I/O (can’t batch process)
  - non-deterministic concurrent interactions with outside world

- Cost
  - includes cost of supporting structures, particularly memory
  - static code size very important (cost of ROM/RAM)
  - often ship millions of copies (worth engineer time to optimize cost down)

- Power
  - expensive package and cooling affects cost, system size, weight, noise, temperature

支持实时软件的处理器

- Simpler pipelines and memory hierarchies make it easier (possible?) to determine the worst-case execution time (WCET) of a piece of code
  - Would like to guarantee task completed by deadline

- Out-of-order execution, caches, prefetching, branch prediction, make it difficult to determine worst-case run time
  - Have to pad WCET estimates for unlikely but possible cases, resulting in over-provisioning of processor (wastes resources)

降低开关功耗 switching power

Power $\propto$ activity * $1/2 CV^2$ * frequency

- Reduce activity
- Reduce switched capacitance C
- Reduce supply voltage V
- Reduce frequency

低功耗设计技术

Clock Gating
- *don’t clock flip-flop if not needed*
- *avoids transitioning downstream logic*
- *Pentium-4 has hundreds of gated clocks*

Bus Encodings
- choose encodings that minimize transitions on average (e.g., Gray code for address bus)
- compression schemes (move fewer bits)

Remove Glitches
- balance logic paths to avoid glitches during settling
- use monotonic logic (domino)
“Just Enough” Performance

- Run fast then stop
- Run slower and just meet deadline

- Save energy by reducing frequency and voltage to minimum necessary (usually done in O.S.)

Embedded memory hierarchies
- Scratchpad RAMs often used instead, or as well as, caches
  - RAM has predictable access latency, simplifies execution time analysis for real-time applications
  - RAM has lower energy/access (no tag access or comparison/multiplexing logic)
  - RAM is cheaper than same size cache (no tags or cache logic)
- Typically no memory protection or translation
  - Code uses physical addresses
- Often embedded processors will not have direct access to off-chip memory (only on-chip RAM)
- Often no disk or secondary storage (but printers, iPods, digital cameras, sometimes have hard drives)
  - No swapping or demand-paged virtual memory
  - Often, flash EEPROM storage of application code, copied to system RAM/DRAM at boot

System-on-a-Chip Environment
- Often, a single chip will contain multiple embedded cores with multiple private or shared memory banks, and multiple hardware accelerators for application-specific tasks
  - Multiple dedicated memory banks provide high bandwidth, predictable access latency
  - Hardware accelerators can be ~100x higher performance and/or lower power than software for certain tasks
- Off-chip I/O ports have autonomous data movement engines to move data in and out of on-chip memory banks
- Complex on-chip interconnect to connect cores, RAMs, accelerators, and I/O ports together

Block Diagram of Cellphone SoC (TI OMAP 2420)
“Classic” DSP Processors

- Embedded applications usually involve many concurrent processes and handle multiple concurrent I/O streams
- Microcontrollers, DSPs, network processors, media processors usually have complex, non-orthogonal instruction sets with specialized instructions and special memory structures
  - poor compiled code quality (% peak with compiled code)
  - high static code efficiency
  - high MIPS/$ and MIPS/W
  - usually assembly-coded in critical routines
- Worth one engineer year in code development to save $1 on system that will ship 1,000,000 units
- Assembly coding easier than ASIC chip design
- But much room for improvement…

Intel IXP Network Processors

- Embedded systems have specific constraints and characteristics:
  - Realtime constraints
  - low-power requirements
  - cost limitations
- Unpredictable components, such as caches and branch predictors
- Simultaneous multithreading (SMT) processor
  - shares many resources between several threads
  - good cost-performance tradeoff
- The key problem with using SMTs in embedded real-time systems
  - collaboration between the operating system (OS) and the SMT