Multithreaded Architectures

Multiprocessors

- multiple threads execute on *different* processors

Uniprocessors

- multiple threads execute on *the same* processor if they are context switched in and out

Multithreaded processors

- multiple threads execute on *the same* processor *without* context switches

Motivation for Multithreaded Architectures

Performance, again.

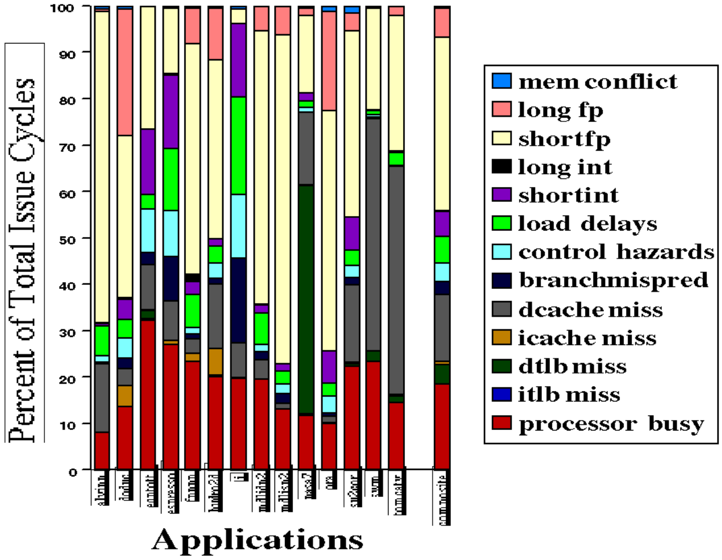
Past: performance suffered from a particular source of latency

Today: all sources of latency

Individual processors not executing code at their hardware potential despite increasingly complex parallel hardware

- increase in instruction issue bandwidth & number of functional units
- out-of-order execution
- techniques for decreasing/hiding branch & memory latencies
- for example:
  - processor utilization was decreasing
  - instruction throughput not increasing in proportion to the increase in issue width

Motivation for Multithreaded Architectures



Motivation for Multithreaded Architectures

Major *cause* of low instruction throughput:

- more complicated than a particular source of latency
- the lack of instruction-level parallelism in a single executing thread

Therefore the *solution*:

- has to be more general than building a smarter cache or a more accurate branch predictor
- has to involve more than one thread

Multithreaded Processors

Multithreaded processors

- execute instructions from multiple threads
- execute multiple threads without ***software*** **context switching**
- hardware support
  - holds processor state for more than one thread of execution
    - registers
    - PC
    - each thread's state is a **hardware context**

Multithreaded Processors

Effect on performance: higher instruction throughput

- threads hide latencies for each other
  - utilize **thread-level parallelism** (TLP) to compensate for low single-thread ILP
- may degrade latency of individual threads (but improves the execution time of all threads by increasing instruction throughput)

Traditional Multithreading

Traditional multithreaded processors ***hardware*** switch to a different context to avoid processor stalls

Two styles of traditional multithreading

Each trades off single thread latency vs. multiple thread throughput in a different way

1. **coarse-grain** multithreading
2. **fine-grain** multithreading

Traditional Multithreading

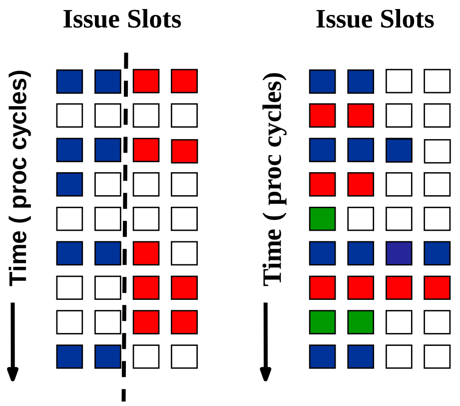
Coarse-grain multithreading

- switch on a long-latency operation (e.g., L2 cache miss)
- another thread executes while the miss is handled
- *modest* increase in instruction throughput
  - doesn't hide latency of short-latency operations
  - no switch if no long-latency operations
  - need to fill the pipeline on a switch
- potentially no slowdown to the thread with the miss, if stall is long, pipeline is short & switch back fairly promptly
- Denelcor HEP, IBM RS64 III, IBM Northstar/Pulsar

Traditional Multithreading

Fine-grain multithreading
- can switch to a different thread each cycle (usually round robin)
- hides latencies of all kinds
- larger increase in instruction throughput but slows down the execution of each thread
- Cray MTA

Simultaneous Multithreading



CMP reduces **horizontal** waste

FGMT reduces **vertical** waste

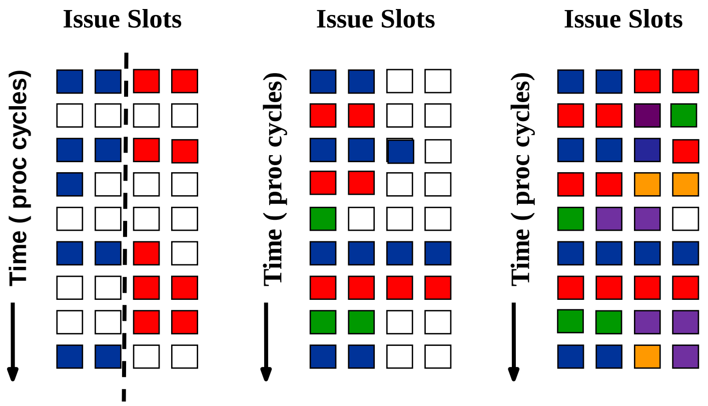
Simultaneous Multithreading (SMT)

Third style of multithreading, different concept

3. simultaneous multithreading (SMT)

- no hardware context switching
- **same-cycle multithreading**: can issue multiple instructions from multiple threads each cycle
- huge boost in instruction throughput with less degradation to individual threads
- Intel Core i7 (Hyperthreading); IBM Power7, BlueGene/Q

Simultaneous Multithreading



CMP reduces **horizontal** waste

FGMT reduces **vertical** waste

SMT reduces **both**

Cray (Tera) MTA

Goals

- uniform memory access
- lightweight synchronization
- heterogeneous parallelism

Cray MTA

Fine-grain multithreaded processor

- can switch to a different thread each cycle
  - switches to ready threads only
- up to 128 hardware contexts/processor
  - lots of latency to hide, mostly from the multi-hop interconnection network
  - average instruction latency for computation: 22 cycles (i.e., 22 instruction streams needed to keep functional units busy)
  - average instruction latency including memory: 120 to 200-cycles (i.e., 120 to 200 instruction streams needed to hide all latency, on average)
- processor state for all 128 contexts
  - GPRs (total of 4K registers!)
  - status registers (includes the PC)
  - branch target registers

Cray MTA

Interesting features

- **No processor-side data caches**
  - increases the maximum latency for data accesses but reduces the variation between memory ops
  - to avoid having to keep caches coherent
  - memory-side buffers instead
- L1 & L2 instruction caches
  - instructions have more locality & have no coherency problem
  - prefetch fall-through & target code

Cray MTA

Interesting features

- **no paging**
  - want pages pinned down in memory for uniform latency
  - page size is 256MB

- **VLIW instructions**
  - memory/arithmetic/branch
  - load/store architecture
  - need a good code scheduler

Cray MTA

Interesting features

- **Trade-off between avoiding memory bank conflicts & exploiting spatial locality for data**

- conflicts:
  - memory distributed among processing elements (PEs)
  - memory addresses are randomized to avoid conflicts
    - want to fully utilize all memory bandwidth
- locality:
  - run-time system can confine consecutive virtual addresses to a single (close-by) memory unit

Cray MTA

Interesting features

- **tagged memory**, i.e., **full/empty bits**
  - indirectly set full/empty bits to prevent **data races**
    - prevents a consumer from loading a value before a producer has written it
    - prevents a producer from overwriting a value before a consumer has read it
  - example for the consumer:
    - set to empty when producer instruction starts executing
    - consumer instructions block if try to read the producer value
    - set to full when producer writes value
    - consumers can now read a valid value

Cray MTA

Interesting features

- **tagged memory**, i.e., **full/empty bits**
  - explicitly set full/empty bits for cheap thread synchronization
    - primarily used accessing shared data
    - very fine-grain synchronization (on the level of a data word)
      - locking: read memory location & set to empty
      - other readers are blocked
      - unlocking: write memory location & set to full

SMT: The Executive Summary

Simultaneous multithreaded (SMT) processors combined designs from:

- traditional multithreaded processors
  - multiple per-thread hardware contexts
- out-of-order superscalar processors
  - wide instruction issue
  - out-of-order execution
  - hardware register renaming

SMT: The Executive Summary

The combination was a processor with two important capabilities.

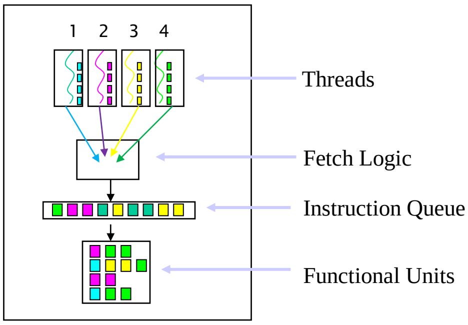
1) **same-cycle multithreading**: issues & executes instructions from multiple threads each cycle
   => converting thread-level parallelism (TLP) to cross-thread instruction-level parallelism (ILP)

Functional Units

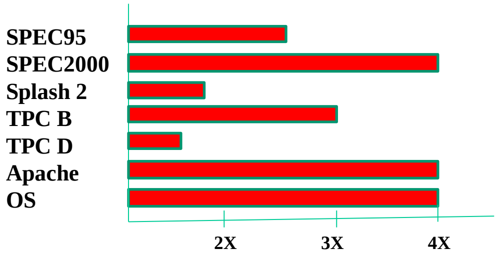
SMT: The Executive Summary

The combination was a processor with two important capabilities.

2) **thread-shared hardware resources**, both logic & memories



Performance Implications



Does this Processor Sound Familiar?

Technology transfer =>

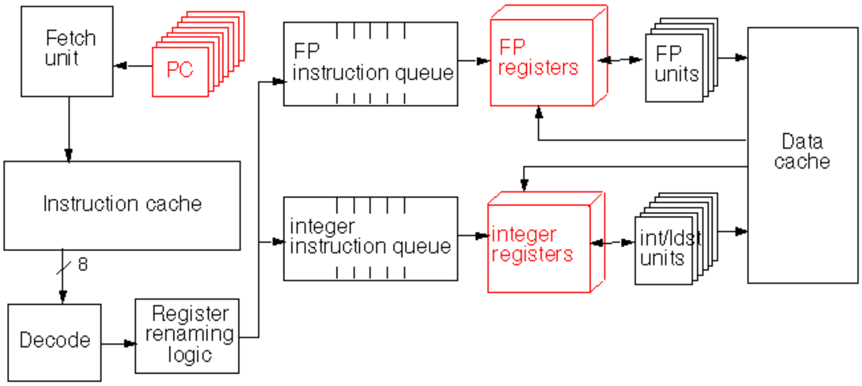
- 2-context Intel Pentium 4; Xeon; Core i5, i7; Atom (Hyperthreading)
- 2-context IBM Power5 & Power6; 4-context IBM Power7 (8 cores) & BlueGene/Q (16 cores)
- 4-context Compaq 21464

An SMT Architecture

Three primary **goals** for this architecture:

1. Achieve significant throughput gains with multiple threads
2. Minimize the performance impact on a single thread executing alone
3. Minimize the microarchitectural impact on a conventional out-of-order superscalar design

Implementing SMT



Implementing SMT

No special hardware for scheduling instructions from multiple threads

- use the hardware register renaming & dynamic instruction scheduling mechanisms as a superscalar
- register renaming hardware eliminates false dependences both within a thread (just like a superscalar) & also between threads

How it works:

- map *thread-specific* architectural registers onto a pool of *thread-independent* physical registers
  - for example: A3 in T1 onto P5; A3 on T2 onto P6
- operands are thereafter called by their physical names
- an instruction is issued when its operands become available & a functional unit is free
- instruction scheduler not have to consider thread IDs when dispatching instructions to functional units
  (unless threads have different priorities)

From Superscalar to SMT

Extra pipeline stages for accessing thread-shared register files

- 8 hardware contexts * 32 registers + renaming registers

SMT instruction fetcher (ICOUNT chooser)

- fetch from 2 threads each cycle
  - count the number of instructions for each thread in the pre-execution stages
  - pick the 2 threads with the lowest number
- in essence fetching from the two highest throughput threads

From Superscalar to SMT

Per-thread hardware

- small stuff
- all part of current out-of-order processors
- none endangered the cycle time

1. other per-thread processor state, e.g.,
   - program counters
   - return stacks
   - thread identifiers, e.g., with BTB entries, TLB entries
2. per-thread bookkeeping for, e.g.,
   - instruction queue flush on branch mispredictions
   - instruction commit
   - trapping

This is why there is only a 15% increase in chip area on a 4 hardware-context Alpha 21464.

Implementing SMT

Thread-shared hardware:

- fetch buffers
- branch target buffer
- instruction queues
- functional units
- all caches (physical tags)
- TLBs
- store buffers & MSHRs

Thread-shared hardware is why there is little single-thread performance degradation (~1.5%).

What hardware might you not want to share?

Implementing SMT

Does thread-shared hardware cause more conflicts?

- 2X more data cache misses

Does it matter?

- threads hide miss latencies for each other
- data sharing

SMT

Interesting features

- **thread-blind instruction scheduling**
- **thread chooser** for instruction fetching
- **hardware queuing locks** for cheap synchronization
  - orders of magnitude faster because does not access memory
  - can parallelize previously unparallelizable codes
- **software-directed register deallocation**
  - communicate **last-use information to HW** for early register deallocation
  - now need fewer renaming registers

What does SMT change?

1. Costs of data sharing

CMPs

- Threads reside on distinct processors & inter-thread communication is a big overhead.
- Parallelizing compilers attempt to decompose applications to minimize inter-processor communication.
- Disjoint set of data & iterations for each thread

SMT

- Threads execute on the same processor with thread-shared hardware.
- Inter-thread communication incurs no overhead.

SMT Compiler Strategy

No special SMT-centered compilation is necessary

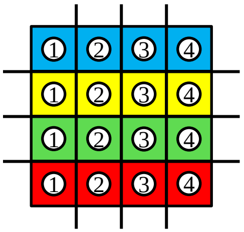
However, if optimizations focused on data *sharing*, not data *isolation*, might SMT do better?

Tiling Example

```
/* matrix multiple before */
for (i=0; i<n; i=i+1)
       for (j=0; j<n; j=j+1){
              r = 0;
              for (k=0; k<n; k=k+1) {
                     r = r + y[i,k] * z[k,j]; }
              x[i,j] = r;
              };
/* matrix multiply after tiling */
for (jj=0; jj<n; jj=jj+T)
for (kk=0; kk<n; kk=kk+T)

   for (i=0; i<n; i=i+1)
       for (j=jj; j<min(jj+T-1,n); j=j+1) {
              r = 0;
              for (k=kk; k<min(kk+T-1,n); k=k+1)
                     {r = r + y[i,k] * z[k,j]; }
              x[i,j] = x[i,j] + r;
              };
```

Tiling



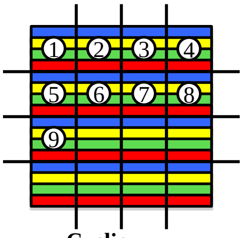
Blocked

The Normal Way (blocked):

Tiled to exploit data reuse, separate tiles/thread

Often works, except when: large number of threads, large number of arrays, small data cache

Issue of tile size sweet spot

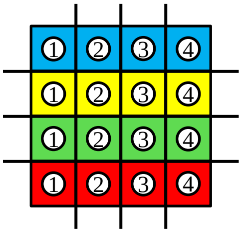


Cyclic

The SMT-friendly Way (cyclic)

Threads share a tile so there is less pressure on the data cache

Tiling



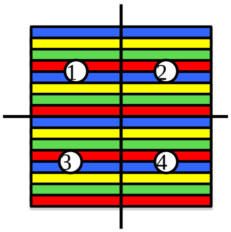
Blocked

The Normal Way (blocked):

Tiled to exploit data reuse, separate tiles/thread

Often works, except when: large number of threads, large number of arrays, small data cache

Issue of tile size sweet spot



Cyclic

The SMT-friendly Way (cyclic)

Threads share a tile so there is less pressure on the data cache

Less sensitive to tile size

- tiles can be large to reduce loop control overhead
- cross-thread latency hiding hides misses
- more adaptable to different cache configurations

Multicore vs. Multithreading

If you wanted to execute multiple threads, would you build a:

- Multicore with multiple, simple pipelines?
- SMT with a single, higher performance pipeline?
- Both together?

Multicore vs. Multithreading

If you wanted to execute multiple threads, would you build a:

- Multicore with multiple, separate pipelines?
  - simple, easy to design, build, test
  - probably faster clock
  - power? turn off unused cores
- SMT with a single, larger pipeline?
  - better performance from same-cycle multithreading
  - better power/performance ratio

- Both together?

  Intel Nehalem (Core-i7): up to 8 cores, 16 SMT threads

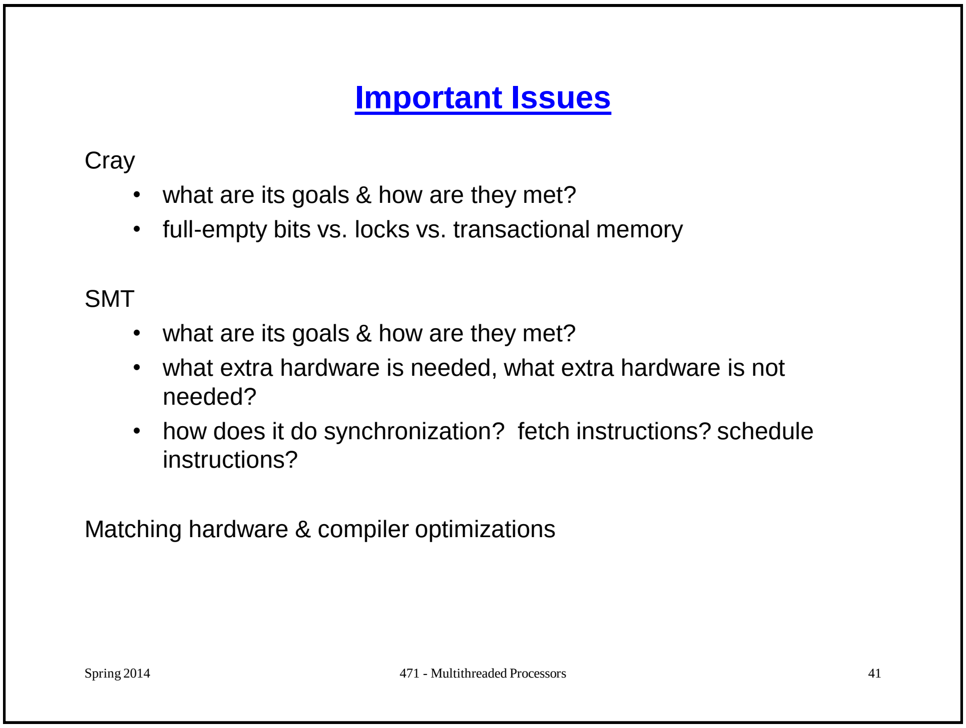
  4-context IBM Power7 (8 cores)

Important Issues

Multithreaded processors

- what are they?
- what problem do they solve?
- hardware support
- 4th through-put vs. latency trade-off

Coarse-grain vs. fine-grain vs. simultaneous multithreading



Important Issues

Cray
- what are its goals & how are they met?
- full-empty bits vs. locks vs. transactional memory

SMT
- what are its goals & how are they met?
- what extra hardware is needed, what extra hardware is not needed?
- how does it do synchronization? fetch instructions? schedule instructions?

Matching hardware & compiler optimizations