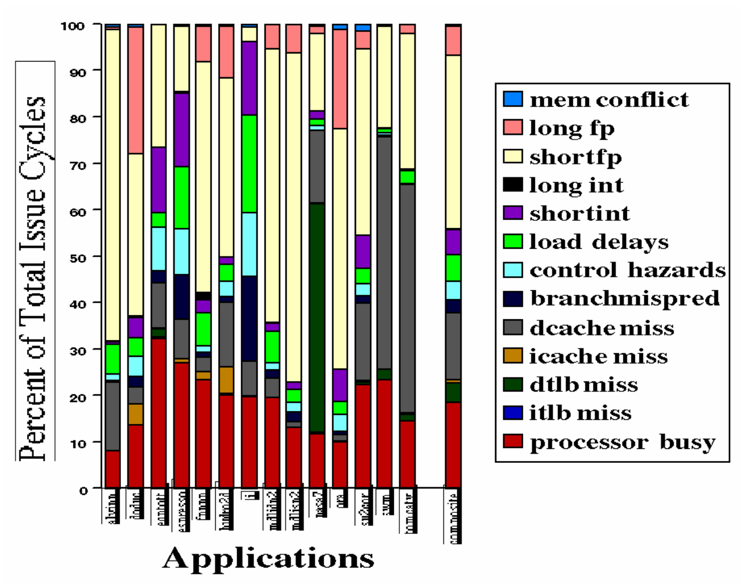
Motivation for Multithreaded Architectures

Processors not executing code at their hardware potential

- late 70's: performance lost to memory latency
- 90's: performance not in line with the increasingly complex parallel hardware
  - Increase in instruction issue bandwidth
  - Increase in number of functional units
  - execute out-of-order execution
  - techniques for decreasing/hiding branch & memory latencies
  - Still, processor utilization was **decreasing** & instruction throughput not increasing in proportion to the issue width

Motivation for Multithreaded Architectures



Motivation for Multithreaded Architectures

Major cause is 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

Multithreaded Processors

Multithreaded processors can increase the pool of independent instructions & consequently address multiple causes of processor stalling

- holds processor state for more than one thread of execution
  - registers
  - PC
  - each thread's state is a **hardware context**
- execute the instruction stream from multiple threads without software context switching
- utilize thread-level parallelism (TLP) to compensate for a lack in ILP

Multithreading

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

2 styles of traditional multithreading

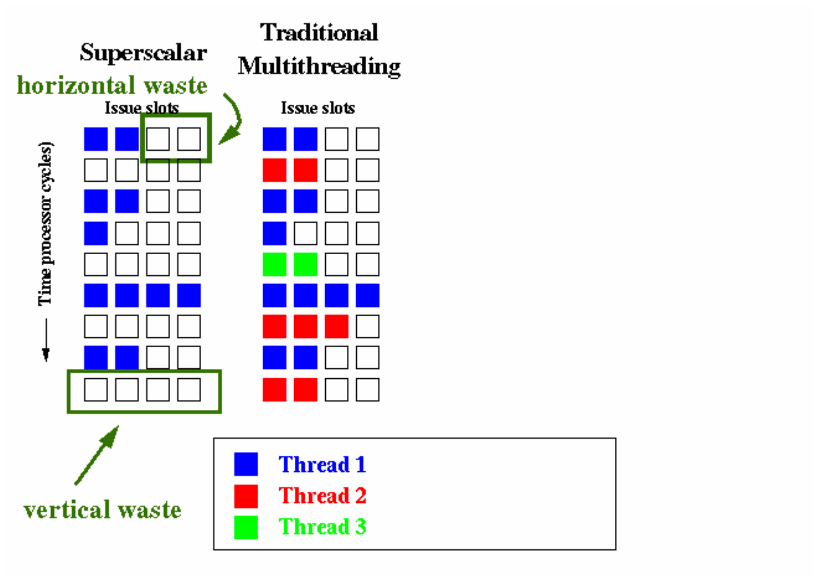
1. **coarse-grain** multithreading
    - switch on a long-latency operation (e.g., L2 cache miss)
    - another thread executes while the miss is handled
    - modest increase in instruction throughput
        - doesn't hide latency of short-latency operations
        - no switch if no long-latency operations
        - need to fill the pipeline on a switch
    - potentially no slowdown to the thread with the miss
        - doesn't add much to an already long stall
    - HEP, IBM RS64 III

Traditional Multithreading

2 styles of traditional multithreading

2. **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 (Tera) MTA

Comparison of Issue Capabilities

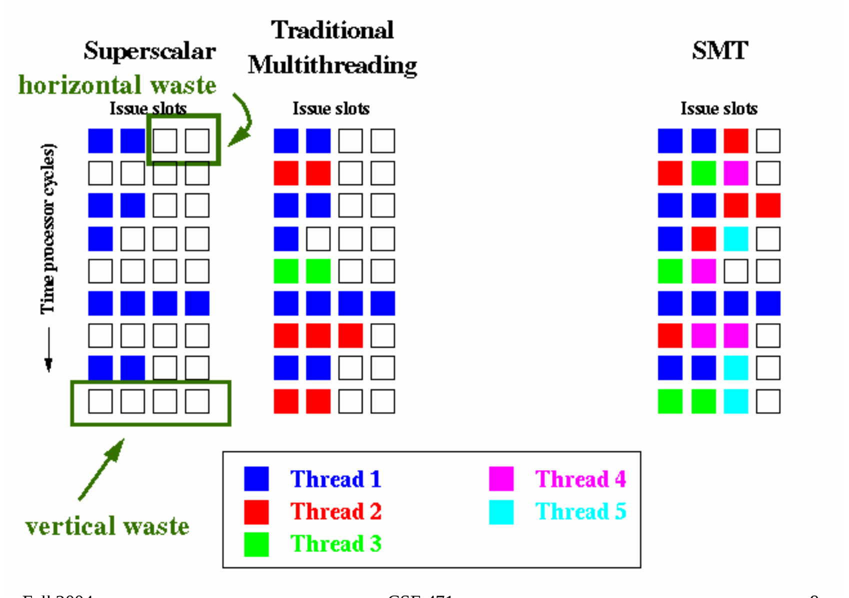


Simultaneous Multithreading (SMT)

3rd style of multithreading, different concept

3. **simultaneous multithreading (SMT)**
    - issues multiple instructions from multiple threads each cycle
    - no hardware context switching
    - huge boost in instruction throughput with less degradation to individual threads

Comparison of Issue Capabilities



Cray (Tera) MTA

Fine-grain multithreaded processor

- can switch to a different thread each cycle
  - switches to ready threads only
  - allows execution to remain with the current thread for a specific number of cycles (discussed under compiler support)
- up to 128 hardware contexts
  - lots of latency to hide, mostly from the multi-hop interconnection network
  - average instruction latency is 70-cycles (at one point) (i.e., 70 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 (Tera) MTA

Interesting features

- Originally no data caches
  - to avoid having to keep caches coherent (topic of the next lecture section)
  - increases the latency for data accesses but reduces the variation
- L1 & L2 instruction caches
  - instruction accesses are more predictable & have no coherency problem
  - prefetch straight-line & target code

Cray (Tera) MTA

Interesting features

- Trade-off between avoiding memory bank conflicts & exploiting spatial locality
  - memory distributed among hardware contexts (processors)
  - memory addresses are randomized to avoid conflicts
    - want to fully utilize all memory bandwidth
  - run-time system can confine consecutive virtual addresses to a single (close-by) memory unit
    - reduces latency
    - used mainly for instructions

Cray (Tera) MTA

Interesting features

- tagged memory for synchronization
  - indirectly set **full/empty bits** to prevent data races
    - prevents a consumer/producer from loading/overwriting a value before a producer/consumer has written/read it
      - set to empty when producer instruction starts
      - consumer instructions block if try to read the producer value
      - set to full when producer value is written
      - consumers can now read a valid value
  - explicitly set full/empty bits for synchronization
    - primarily used to synchronize threads that are accessing shared data (topic of the next lecture)
      - lock: read memory location & set to empty
      - other readers are blocked
      - unlock: write & set to full

Cray (Tera) MTA

Interesting features

- virtual memory system
  - no paging: want pages pinned down in memory
  - page size is 256MB
- user-mode trap handlers
  - fatal exceptions, normalizing floating point numbers

Cray (Tera) MTA

Compiler support

- each instruction is a VLIW instruction
  - memory/arithmetic/branch
  - load/store architecture
  - need a good code scheduler
- explicit dependence look-ahead
  - field in a memory instruction that specifies the number of independent (of the memory op) LIW instructions that follow
  - deviation from fine-grain multithreading
- handling branches
  - instruction to store a branch target in a register before the branch is executed
  - can start prefetching the target code

Cray (Tera) MTA

Run-time support

- thread manipulation
  - protection domains: group of threads executing in the same virtual address space
  - OS sets the maximum number of thread contexts (instruction streams) a domain is allowed
  - domain can create & kill threads within that limit, depending on its need for them

- OS bases resource allocations on resource usage

SMT: The Executive Summary

Simultaneous multithreaded (SMT) processors combine designs from:

- superscalar processors
- traditional multithreaded processors

The combination:

- SMT issues & executes instructions from multiple threads simultaneously
  => converting TLP to ILP
- threads share almost all hardware resources

Performance Implications

Multiprogramming workload

- 2.5X on SPEC95, 4X on SPEC2000

Parallel programs

- ~.7 on SPLASH2

Commercial databases

- 2-3X on TPC B; 1.5 on TPC D

Web servers & OS

- 4X on Apache and Digital Unix

Does this Processor Sound Familiar?

Huge performance boost + straightforward implementation =>

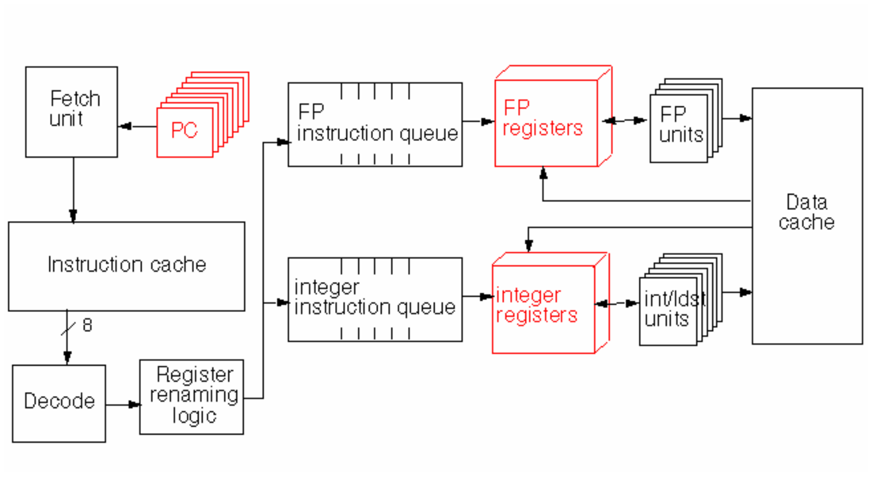
- 2-context Intel Hyperthreading
- 4-context IBM Power5
- 2-context Sun UltraSPARC on 4-processor CMP
- 4-context Compaq 21464
- network processor & mobile device start-ups
- others in the wings

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

Can **use as is** most hardware on current out-or-order processors

Out-of-order renaming & instruction scheduling mechanisms

- physical register pool model
- renaming hardware eliminates false dependences both within a thread (just like a superscalar) & between threads
- map thread-specific architectural registers onto a pool of thread-independent physical registers
- 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 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 threads * 32 registers + renaming registers

SMT instruction fetcher (ICOUNT)

- 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 endangers the cycle time

- other per-thread processor state, e.g.,
  - program counters
  - return stacks
  - thread identifiers, e.g., with BTB entries, TLB entries
- per-thread bookkeeping for
  - instruction queue flush
  - instruction retirement
  - trapping

This is why there is only a 10% increase to Alpha 21464 chip area.

Implementing SMT

Thread-shared hardware:

- fetch buffers
- branch prediction structures
- instruction queues
- functional units
- active list
- all caches & TLBs
- MSHRs
- store buffers

This is why there is little single-thread performance degradation (~1.5%).

Architecture Research

Concept & potential of Simultaneous Multithreading: ISCA ’95 & ISCA 25th Anniversary Anthology

Designing the **microarchitecture**: ISCA ’96

- straightforward extension of out-of-order superscalars

I-fetch **thread chooser**: ISCA ’96

- 40% faster than round-robin

The **lockbox** for cheap synchronization: HPCA ’98

- orders of magnitude faster
- can parallelize previously unparallelizable codes

Architecture Research

Software-directed **register deallocation**: TPDS '99

- large register-file performance w. small register file

Mini-threads: HPCA '03

- large SMT performance w. small SMTs

SMT instruction **speculation**: TOCS '03

- speculation keeps a good thread mix in the IQ
- most important performance factor

Compiler Research

Tuning compiler optimizations for SMT: Micro '97 & IJPP '99

- data decomposition: use cyclic iteration scheduling
- tiling: use cyclic tiling; no tile size sweet spot
- software speculation: generates too much overhead code on integer programs

Communicate **last-use info to HW** for early register deallocation: TPDS '99

- now need ¼ the renaming registers

Compiling for fewer registers/thread: HPCA '03

- surprisingly little additional spill code (avg. 3%)

OS Research

Analysis of **OS behavior** on SMT: ASPLOS ‘00

- Kernel-kernel conflicts in I$ & D$ & branch mispredictions ameliorated by SMT instruction issue + thread-sharing in HW

OS/runtime support for mini-threads: HPCA ’03

- dedicated server: recompile for fewer registers
- multiprogrammed environment: multiple versions

OS/runtime support for executing threaded programs: ISCA ’98 & PPoPP ’03

- dynamic memory allocation, synchronization, stack offsetting, page mapping

Others are Now Carrying the Ball

Fault detection & recovery
Thread-level speculation
Instruction & data prefetching
Instruction issue hardware design
Thread scheduling
Single-thread execution
Profiling executing threads
SMT-CMP hybrids
Power considerations

SMT Collaborators

UW

Hank Levy
Steve Gribble

Dean Tullsen (UCSD)
Jack Lo (VMWare)
Sujay Parekh (IBM Yorktown)
Brian Dewey (Microsoft)
Manu Thambi (Microsoft)
Josh Redstone (Google)
Mike Swift
Luke McDowell (Naval Academy)
Steve Swanson
Aaron Eakin (HP)
Dimitriy Portnov (Google)

DEC/Compaq

Joel Emer (now Intel)
Rebecca Stamm
Luiz Barroso (now Google)
Kourosh Gharachorloo (now Google)

For more info on SMT:
http://www.cs.washington.edu/research/smt