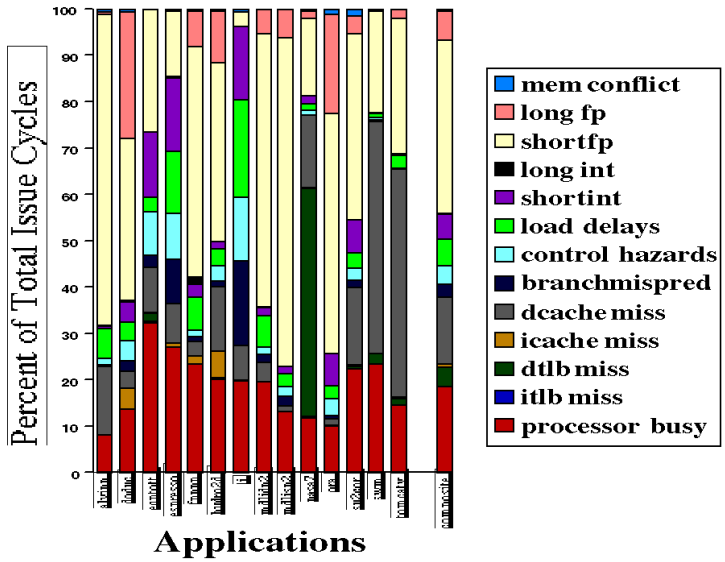
Motivation for Multithreaded Architectures

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 as well
  - increase in instruction issue bandwidth
  - increase in number of functional units
  - 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
- has to involve more than one thread

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 single-thread ILP
  - Improve hardware utilization
  - May degrade latency of individual threads, but improves latency of all threads together (by increasing instruction throughput)

Multithreading

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

Two styles of traditional multithreading

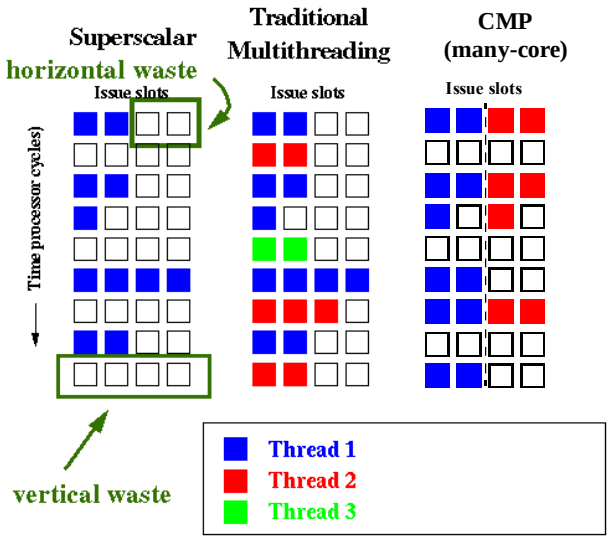
1. **coarse-grain** multithreading
   - switch on a long-latency operation (e.g., L2 cache miss)
   - another thread executes while the miss is handled
   - modest increase in instruction throughput
     - doesn't hide latency of short-latency operations
     - no switch if no long-latency operations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

Traditional Multithreading

Two 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

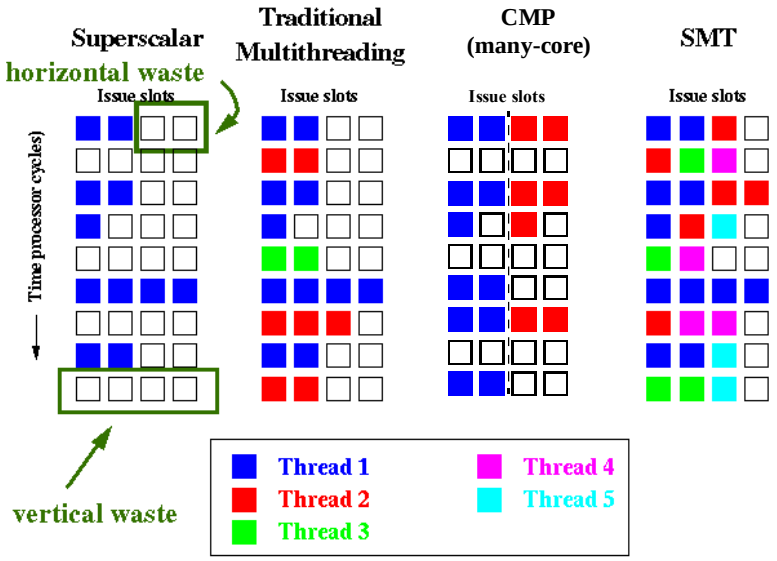


Simultaneous Multithreading (SMT)

Third style of multithreading, different concept

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

Comparison of Issue Capabilities



Cray (Tera) MTA

Goals

- the appearance of uniform memory access
- lightweight synchronization
- heterogeneous parallelism

Cray (Tera) MTA

Fine-grain multithreaded processor

- can switch to a different thread each cycle
  - switches to ready threads only
- up to 128 hardware contexts
  - 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/stream

Cray (Tera) MTA

Interesting features

- **No processor-side data caches**
  - increases the 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
  - instruction accesses are more predictable & have no coherency problem
  - prefetch fall-through & target code

Cray (Tera) 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 (Tera) MTA

Interesting features

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

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

Cray (Tera) MTA

Interesting features

- **tagged memory**, i.e., **full/empty bits**
  - 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
    - 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
  - explicitly set full/empty bits for cheap thread synchronization
    - primarily used accessing shared data
      - lock: read memory location & set to empty
      - other readers are blocked
      - unlock: write & set to full

SMT: The Executive Summary

Simultaneous multithreaded (SMT) processors combine designs from:

- out-of-order superscalar processors
- traditional multithreaded processors

The combination enables a processor

- that issues & executes instructions from multiple threads simultaneously, same-cycle multithreading

  => *converting* TLP to ILP
- in which threads share almost all hardware resources

Performance Implications

Multiprogramming workload
- 2.5X on SPEC95, 4X on SPEC2000

Parallel programs
- ~1.7X on SPLASH2

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

Web servers & OS
- 4X on Apache and Unix

Does this Processor Sound Familiar?

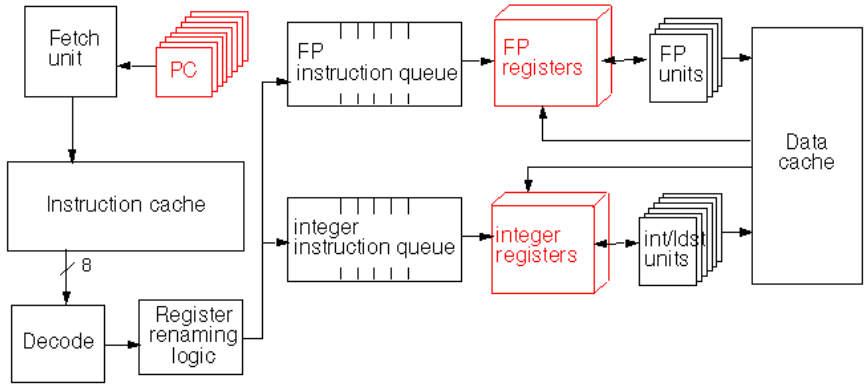
Technology transfer =>
- 2-context Intel Xeon w/Hyperthreading (Sun Fire servers)
- 4-context IBM Power7 (8 cores)
- 2-context Sun UltraSPARC on a 4-processor CMP
- 4-context Compaq 21464

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 out-of-order renaming & instruction scheduling mechanisms as a superscalar
- physical register pool
- hardware renaming hardware eliminates false dependences both within a thread (just like a superscalar) & between threads

How it works:

- 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 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

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

This is why there is only a 15% increase to Alpha 21464 chip area.

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

What hardware might you not want to share?

Implementing SMT

Does sharing hardware cause more conflicts?

– 2X more data cache misses

\+ other threads hide the miss latency

\+ data sharing

Bottom line is huge overall performance boost

Managing Thread-shared Hardware

Concept & potential of Simultaneous Multithreading

Designing the **microarchitecture**

- straightforward extension of out-of-order superscalars

Thread chooser for instruction fetching (ICOUNT) (pipeline)

- 40% faster than round-robin

Hardware queuing locks for cheap synchronization

- orders of magnitude faster
- can parallelize previously unparallelizable codes

Managing Thread-shared Hardware

Mini-threads (registers)

- large SMT performance w. small SMTs
- compiling for fewer registers/thread; surprisingly little additional spill code (avg. 3%)

SMT instruction **speculation** (pipeline)

- don't execute as far down a wrong path
- speculative instructions don't get as far down the pipeline
- speculation keeps a good thread mix in the IQ
  - most important factor for performance

Managing Thread-shared Hardware

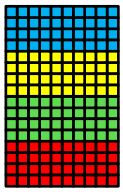
Tuning compiler optimizations for SMT (data caches, TLB)

- data decomposition: use cyclic iteration scheduling
- tiling: use cyclic tiling; no tile size sweet spot

Software-directed **register deallocation** (registers)

- communicate **last-use information to HW** for early register deallocation
- now need fewer renaming registers
- large register-file performance with a small register file

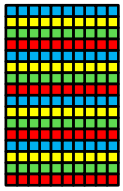
Loop distribution



Blocked

Distributes iterations & contiguous data across threads, separate data/thread

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

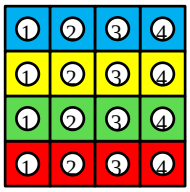


Cyclic

Clusters threads' data, intra-page thread sharing

Speedups from break-even to 4X

Tiling

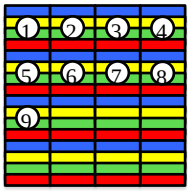


Blocked

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 tiling sweet spot



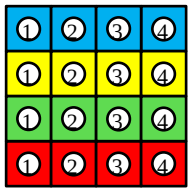
Cyclic

Threads share a tile & iterations are cyclically distributed across threads

Better performance for cache hierarchies of different sizes

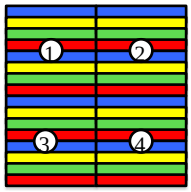
Insensitive to tile size

Tiling



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 tiling sweet spot



- Threads share a tile & iterations are cyclically distributed across threads
- Better performance for cache hierarchies of different sizes
- Insensitive to tile size
- Tiles can be large to reduce loop control overhead

Others Picked Up the Ball

- Fault detection & recovery from transient errors (run 2 copies of a thread)
- Thread-level speculation (speculatively parallelizing loops, conditional code, function calling)
- Instruction prefetching
- Data prefetching
- Single-thread execution
- Profiling executing threads
- Instruction issue hardware design
- Thread scheduling & thread priority
- SMT-CMP hybrids
- Power considerations

I've stopped keeping track

Multicore vs. Multithreading

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

- Multicore with multiple, separate pipelines?
- SMT with a single larger pipeline?
- Both together?
  - Sun Niagra: 8 in-order, short-pipelined processors, each with 4 threads (fine-grained multithreading)
  - Intel Nehalem: up to 8 cores, 16 SMT threads
  - 4-context IBM Power7 (8 cores)

SMT Collaborators

UW

Hank Levy
Steve Gribble

Dean Tullsen (UC San Diego)
Jack Lo (VMWare)
Sujay Parekh (IBM Yorktown)
Brian Dewey (Microsoft)
Manu Thambi (Microsoft)
Josh Redstone (Google)
Mike Swift (U. Wisconsin)
Luke McDowell (Naval Academy)
Steve Swanson (UC San Diego)
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