Computer Architecture: Multithreading

Prof. Onur Mutlu (Editted by Seth)
Carnegie Mellon University

Multithreading (Outline)

- Multiple hardware contexts
- Purpose
- Initial incarnations
  - CDC 6600
  - HEP
- Levels of multithreading
  - Fine-grained (cycle-by-cycle)
  - Coarse grained (multitasking)
    - Switch-on-event
  - Simultaneous
- Uses: traditional + creative (now that we have multiple contexts, why do we not do …)

Multithreading: Basics

- Thread
  - Instruction stream with state (registers and memory)
  - Register state is also called "thread context"
- Threads could be part of the same process (program) or from different programs
  - Threads in the same program share the same address space (shared memory model)
- Traditionally, the processor keeps track of the context of a single thread
- Multitasking: When a new thread needs to be executed, old thread's context in hardware written back to memory and new thread's context loaded

Hardware Multithreading

- General idea: Have multiple thread contexts in a single processor
  - When the hardware executes from those hardware contexts determines the granularity of multithreading
- Why?
  - To tolerate latency (initial motivation)
    - Latency of memory operations, dependent instructions, branch resolution
    - By utilizing processing resources more efficiently
  - To improve system throughput
    - By exploiting thread-level parallelism
    - By improving superscalar/OoO processor utilization
  - To reduce context switch penalty

Initial Motivations

- Tolerate latency
  - When one thread encounters a long-latency operation, the processor can execute a useful operation from another thread

- CDC 6600 peripheral processors
  - I/O latency: 10 cycles
  - 10 I/O threads can be active to cover the latency
  - Pipeline with 100ns cycle time, memory with 1000ns latency
  - Idea: Each I/O "processor" executes one instruction every 10 cycles on the same pipeline
  - Thornton, "Design of a Computer: The Control Data 6600," 1970.
  - Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.

Hardware Multithreading

- Benefit
  + Latency tolerance
  + Better hardware utilization (when?)
  + Reduced context switch penalty

- Cost
  - Requires multiple thread contexts to be implemented in hardware (area, power, latency cost)
  - Usually reduced single-thread performance
    - Resource sharing, contention
    - Switching penalty (can be reduced with additional hardware)

Types of Hardware Multithreading

- Fine-grained
  - Cycle by cycle

- Coarse-grained
  - Switch on event (e.g., cache miss)
  - Switch on quantum/timeout

- Simultaneous
  - Instructions from multiple threads executed concurrently in the same cycle

Fine-grained Multithreading

- Idea: Switch to another thread every cycle such that no two instructions from the thread are in the pipeline concurrently

- Improves pipeline utilization by taking advantage of multiple threads
- Alternative way of looking at it: Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads

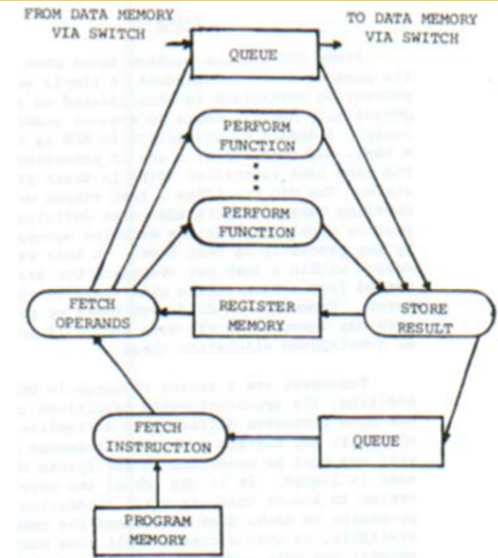
- Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
- Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.

Fine-grained Multithreading

- CDC 6600's peripheral processing unit is fine-grained multithreaded
  - Processor executes a different I/O thread every cycle
  - An operation from the same thread is executed every 10 cycles
- Denelcor HEP
  - Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.
  - 120 threads/processor
    - 50 user, 70 OS functions
  - available queue vs. unavailable (waiting) queue
  - each thread can only have 1 instruction in the processor pipeline; each thread independent
  - to each thread, processor looks like a sequential machine
  - throughput vs. single thread speed

Fine-grained Multithreading in HEP

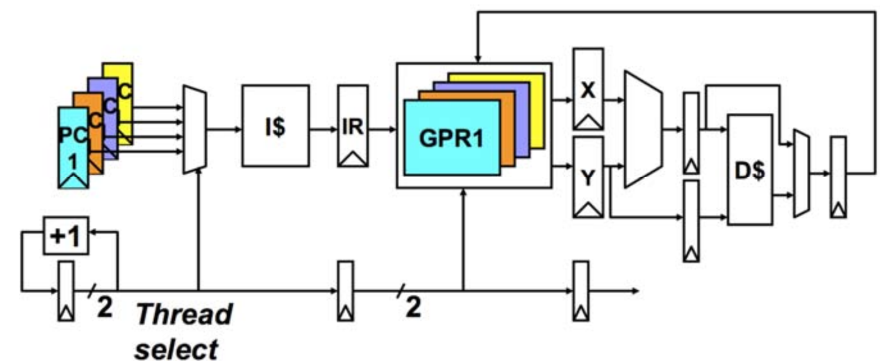
- Cycle time: 100ns
- 8 stages → 800 ns to complete an instruction
  - assuming no memory access



Fine-grained Multithreading

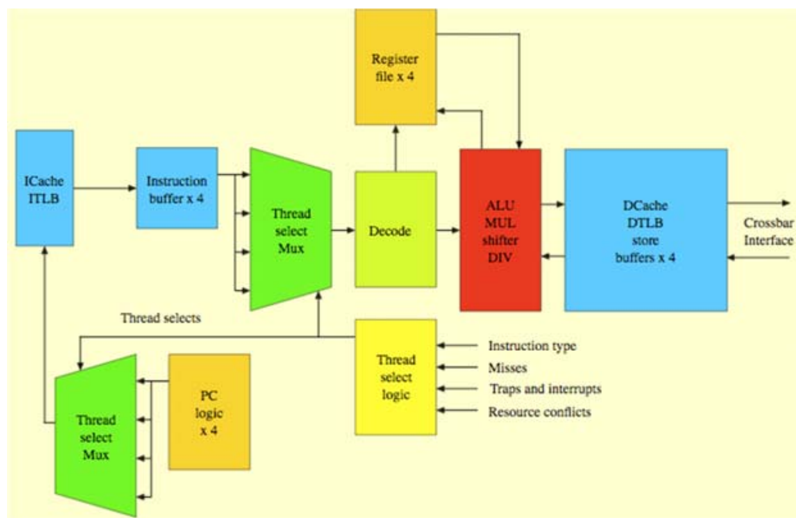
- Advantages
  + No need for dependency checking between instructions (only one instruction in pipeline from a single thread)
  + No need for branch prediction logic
  + Otherwise-bubble cycles used for executing useful instructions from different threads
  + Improved system throughput, latency tolerance, utilization
- Disadvantages
  - Extra hardware complexity: multiple hardware contexts, thread selection logic
  - Reduced single thread performance (one instruction fetched every N cycles)
  - Resource contention between threads in caches and memory
  - Dependency checking logic between threads remains (load/store)

Multithreaded Pipeline Example



- Slide from Joel Emer

Sun Niagara Multithreaded Pipeline



Tera MTA Fine-grained Multithreading

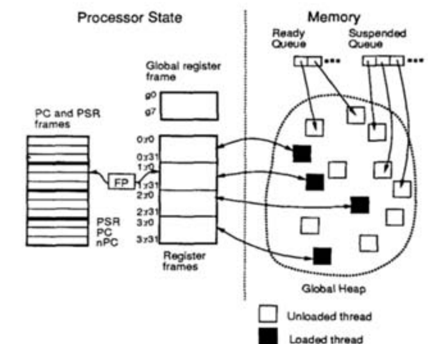
- 256 processors, each with a 21-cycle pipeline
- 128 active threads
- A thread can issue instructions every 21 cycles
  - Then, why 128 threads?
- Memory latency: approximately 150 cycles
  - No data cache
  - Threads can be blocked waiting for memory
  - More threads → better ability to tolerate memory latency
- Thread state per processor
  - 128 x 32 general purpose registers
  - 128 x 1 thread status registers

Coarse-grained Multithreading

- Idea: When a thread is stalled due to some event, switch to a different hardware context
  - Switch-on-event multithreading
- Possible stall events
  - Cache misses
  - Synchronization events (e.g., load an empty location)
  - FP operations
- HEP, Tera combine fine-grained MT and coarse-grained MT
  - Thread waiting for memory becomes blocked (un-selectable)
- Agarwal et al., "APRIL: A Processor Architecture for Multiprocessing," ISCA 1990.
  - Explicit switch on event

Coarse-grained Multithreading in APRIL

- Agarwal et al., "APRIL: A Processor Architecture for Multiprocessing," ISCA 1990.
- 4 hardware thread contexts
  - Called "task frames"
- Thread switch on
  - Cache miss
  - Network access
  - Synchronization fault



- How?
  - Empty processor pipeline, change frame pointer (PC)

Fine-grained vs. Coarse-grained MT

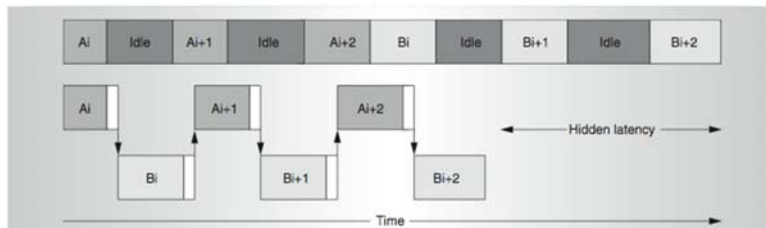
- Fine-grained advantages
  - \+ Simpler to implement, can eliminate dependency checking, branch prediction logic completely
  - \+ Switching need not have any performance overhead (i.e. dead cycles)
    - \+ Coarse-grained requires a pipeline flush or a lot of hardware to save pipeline state
      - → Higher performance overhead with deep pipelines and large windows
- Disadvantages
  - \- Low single thread performance: each thread gets 1/Nth of the bandwidth of the pipeline

IBM RS64-IV

- 4-way superscalar, in-order, 5-stage pipeline
- Two hardware contexts
- On an L2 cache miss
  - Flush pipeline
  - Switch to the other thread
- Considerations
  - Memory latency vs. thread switch overhead
  - Short pipeline, in-order execution (small instruction window) reduces the overhead of switching

Intel Montecito

- McNairy and Bhatia, "Montecito: A Dual-Core, Dual-Thread Itanium Processor," IEEE Micro 2005.



- Thread switch on
  - L3 cache miss/data return
  - Timeout – for fairness
  - Switch hint instruction
  - ALAT invalidation – synchronization fault
  - Transition to low power mode
- <2% area overhead due to CGMT

Fairness in Coarse-grained Multithreading

- Resource sharing in space and time always causes fairness considerations
  - Fairness: how much progress each thread makes
- In CGMT, the time allocated to each thread affects both fairness and system throughput
  - When do we switch?
  - For how long do we switch?
    - When do we switch back?
  - How does the hardware scheduler interact with the software scheduler for fairness?
  - What is the switching overhead vs. benefit?
    - Where do we store the contexts?

Fairness in Coarse-grained Multithreading

- Gabor et al., "Fairness and Throughput in Switch on Event Multithreading," MICRO 2006.
- How can you solve the below problem?

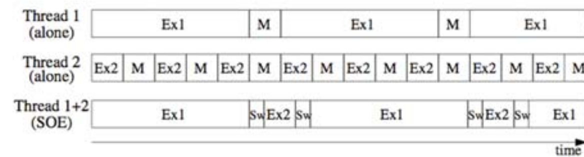


Figure 1. Intuitive example of unfair execution in SOE. *Ex1* **marks execution of instructions from thread 1,** *Ex2* **from thread 2,** *M* **marks last level cache misses and** *Sw* **denotes thread switch overheads. When both threads run together using SOE (bottom), the 2nd thread runs extremely slowly while the 1st thread's performance is hardly affected by the multithreading.**

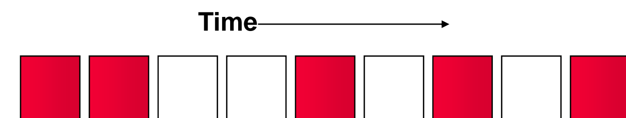
Fairness vs. Throughput

- Switch not only on miss, but also on data return
- Problem: Switching has performance overhead
  - Pipeline and window flush
  - Reduced locality and increased resource contention (frequent switches increase resource contention and reduce locality)
- One possible solution
  - Estimate the slowdown of each thread compared to when run alone
  - Enforce switching when slowdowns become significantly unbalanced
  - Gabor et al., "Fairness and Throughput in Switch on Event Multithreading," MICRO 2006.

Simultaneous Multithreading

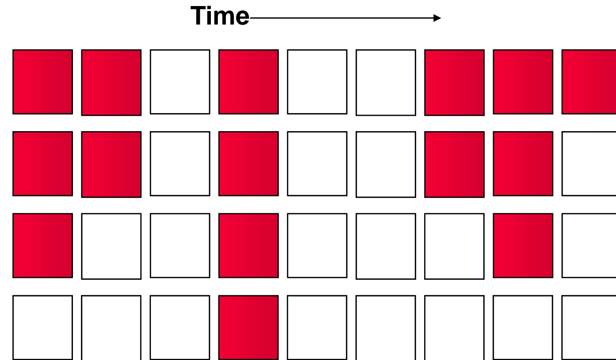
- Fine-grained and coarse-grained multithreading can start execution of instructions from *only* a single thread at a given cycle
- Execution unit (or pipeline stage) utilization can be low if there are not enough instructions from a thread to "dispatch" in one cycle
  - In a machine with multiple execution units (i.e., superscalar)
- Idea: Dispatch instructions from multiple threads in the same cycle (to keep multiple execution units utilized)
  - Hirata et al., "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," ISCA 1992.
  - Yamamoto et al., "Performance Estimation of Multistreamed, Superscalar Processors," HICSS 1994.
  - Tullsen et al., "Simultaneous Multithreading: Maximizing On-Chip Parallelism," ISCA 1995.

Functional Unit Utilization



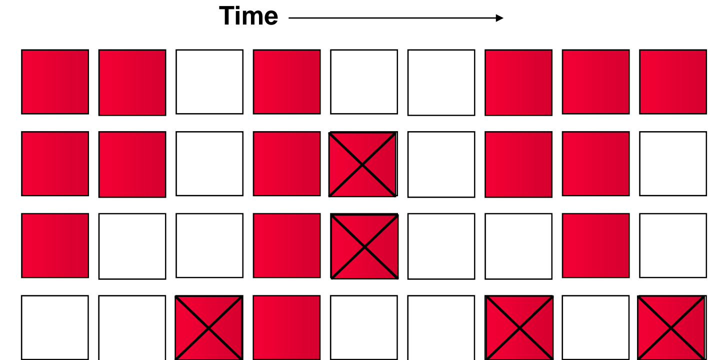
- Data dependencies reduce functional unit utilization in pipelined processors

Functional Unit Utilization in Superscalar



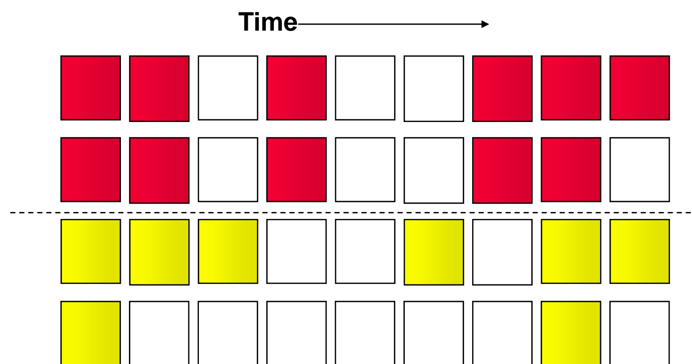
- Functional unit utilization becomes lower in superscalar, OoO machines. Finding 4 instructions in parallel is not always possible

Predicated Execution



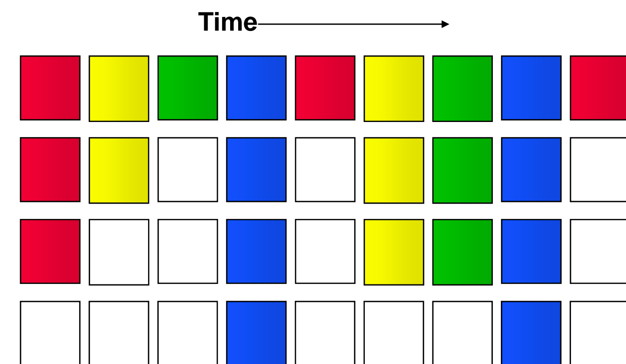
- Idea: Convert control dependencies into data dependencies
- Improves FU utilization, but some results are thrown away

Chip Multiprocessor



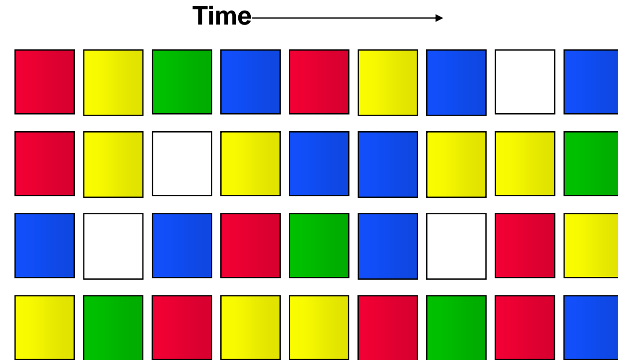
- Idea: Partition functional units across cores
- Still limited FU utilization within a single thread; limited single-thread performance

Fine-grained Multithreading



- Still low utilization due to intra-thread dependencies
- Single thread performance suffers

Simultaneous Multithreading

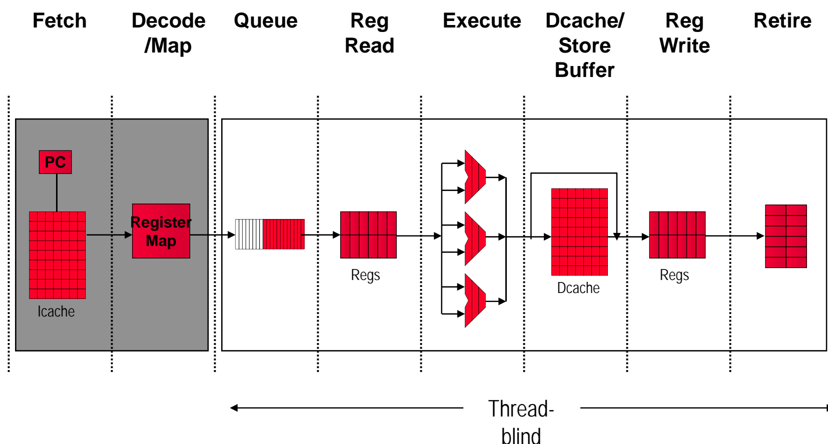


- Idea: Utilize functional units with independent operations from the same or different threads

Simultaneous Multithreading

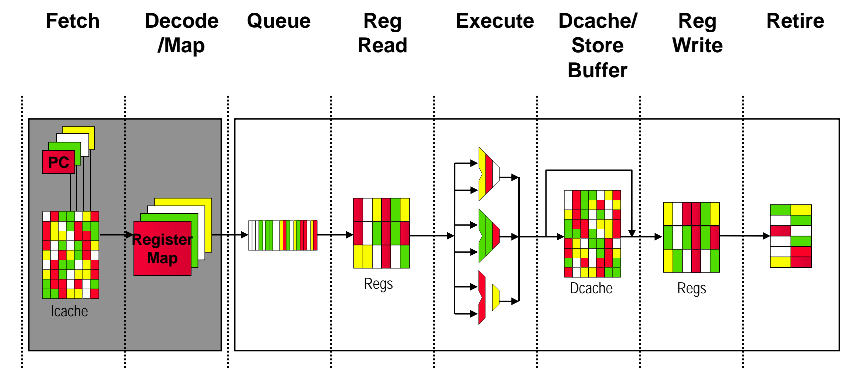
- Reduces both horizontal and vertical waste
- Required hardware
  - The ability to dispatch instructions from multiple threads simultaneously into different functional units
- Superscalar, OoO processors already have this machinery
  - Dynamic instruction scheduler searches the scheduling window to wake up and select ready instructions
  - As long as dependencies are correctly tracked (via renaming and memory disambiguation), scheduler can be thread-agnostic

Basic Superscalar OoO Pipeline



SMT Pipeline

- Physical register file needs to become larger. Why?



Changes to Pipeline for SMT

- Replicated resources
  - Program counter
  - Register map
  - Return address stack
  - Global history register

- Shared resources
  - Register file (size increased)
  - Instruction queue (scheduler)
  - First and second level caches
  - Translation lookaside buffers
  - Branch predictor

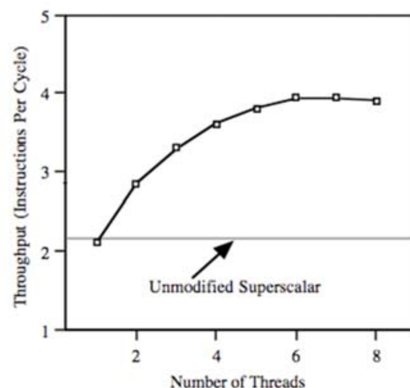
Changes to OoO+SS Pipeline for SMT

- multiple program counters and some mechanism by which the fetch unit selects one each cycle,
- a separate return stack for each thread for predicting subroutine return destinations,
- per-thread instruction retirement, instruction queue flush, and trap mechanisms,
- a thread id with each branch target buffer entry to avoid predicting phantom branches, and
- a larger register file, to support logical registers for all threads plus additional registers for register renaming. The size of the register file affects the pipeline (we add two extra stages) and the scheduling of load-dependent instructions, which we discuss later in this section.

Tullsen et al., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," ISCA 1996.

SMT Scalability

- Diminishing returns from more threads. Why?



SMT Design Considerations

- Fetch and prioritization policies
  - Which thread to fetch from?
- Shared resource allocation policies
  - How to prevent starvation?
  - How to maximize throughput?
  - How to provide fairness/QoS?
  - Free-for-all vs. partitioned
- How to measure performance
  - Is total IPC across all threads the right metric?
- How to select threads to co-schedule
  - Snavely and Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor," ASPLOS 2000.

Which Thread to Fetch From?

- (Somewhat) Static policies
  - Round-robin
  - 8 instructions from one thread
  - 4 instructions from two threads
  - 2 instructions from four threads
  - ...
- Dynamic policies
  - Favor threads with minimal in-flight branches
  - Favor threads with minimal outstanding misses
  - Favor threads with minimal in-flight instructions
  - ...

Which Instruction to Select/Dispatch?

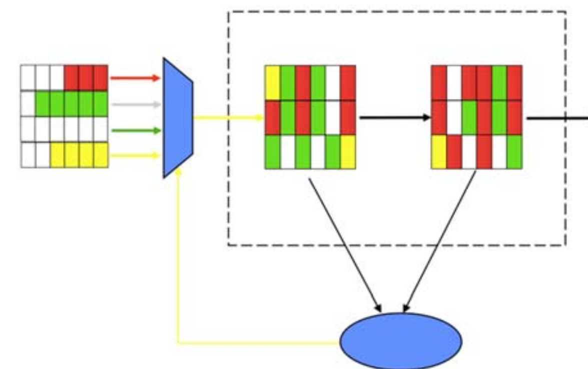
- Can be thread agnostic.
- Why?

SMT Fetch Policies (I)

- Round robin: Fetch from a different thread each cycle
- Does not work well in practice. Why?
- Instructions from slow threads hog the pipeline and block the instruction window
  - E.g., a thread with long-latency cache miss (L2 miss) fills up the window with its instructions
  - Once window is full, no other thread can issue and execute instructions and the entire core stalls

SMT Fetch Policies (II)

- ICOUNT: Fetch from thread with the least instructions in the earlier pipeline stages (before execution)



- Why does this improve throughput?

Slide from Joel Emer

SMT ICOUNT Fetch Policy

- Favors faster threads that have few instructions waiting

- Advantages over round robin
  - + Allows faster threads to make more progress (before threads with long-latency instructions block the window fast)
    - + Higher IPC throughput

- Disadvantages over round robin
  - \- Is this fair?
  - \- Prone to short-term starvation: Need additional methods to ensure starvation freedom

Some Results on Fetch Policy

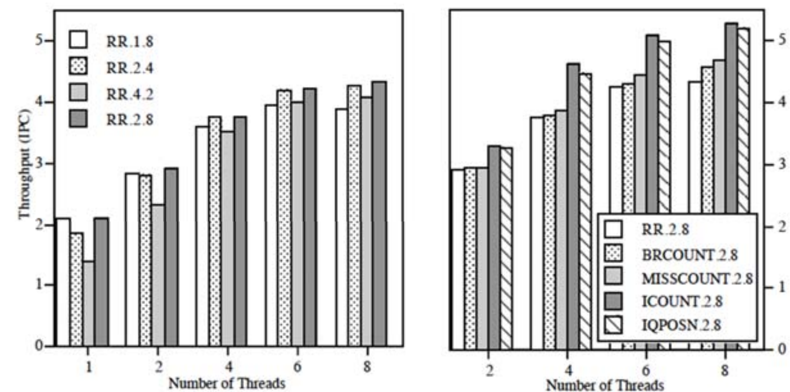
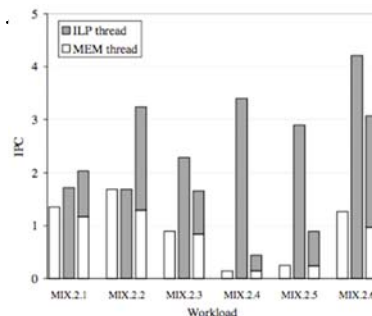


Figure 4: Instruction throughput for the different instruction cache interfaces with round-robin instruction scheduling.

Handling Long Latency Loads

- Long-latency (L2/L3 miss) loads are a problem in a single-threaded processor
  - Block instruction/scheduling windows and cause the processor to stall
- In SMT, a long-latency load instruction can block the window for ALL threads
  - i.e. reduce the memory latency tolerance benefits of SMT
- Brown and Tullsen, "Handling Long-latency Loads in a Simultaneous Multithreading Processor," MICRO 2001.



Proposed Solutions to Long-Latency Loads

- Idea: Flush the thread that incurs an L2 cache miss
  - Brown and Tullsen, "Handling Long-latency Loads in a Simultaneous Multithreading Processor," MICRO 2001.

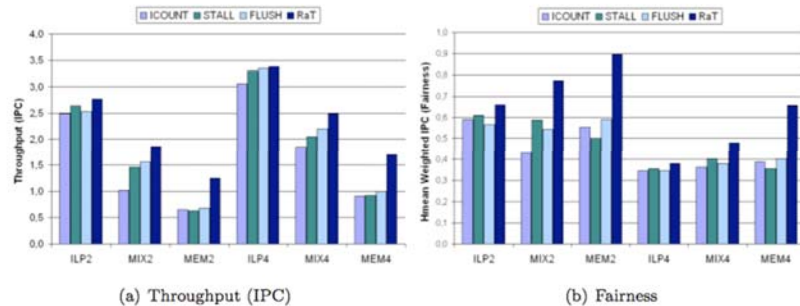
- Idea: Predict load miss on fetch and do not insert following instructions from that thread into the scheduler
  - El-Moursy and Albonesi, "Front-End Policies for Improved Issue Efficiency in SMT Processors," HPCA 2003.

- Idea: Partition the shared resources among threads so that a thread's long-latency load does not affect another
  - Raasch and Reinhardt, "The Impact of Resource Partitioning on SMT Processors," PACT 2003.

- Idea: Predict if (and how much) a thread has MLP when it incurs a cache miss; flush the thread after its MLP is exploited
  - Eyerman and Eeckhout, "A Memory-Level Parallelism Aware Fetch Policy for SMT Processors," HPCA 2007.

Runahead Threads

- Idea: Use runahead execution on a long-latency load
+ Improves both single thread and multi-thread performance
- Ramirez et al., "Runahead Threads to Improve SMT Performance," HPCA 2008.



(a) Throughput (IPC) (b) Fairness

Commercial SMT Implementations

- Intel Pentium 4 (Hyperthreading)
- IBM POWER5
- Intel Nehalem
- ...

SMT in IBM POWER5

- Kalla et al., "IBM Power5 Chip: A Dual-Core Multithreaded Processor," IEEE Micro 2004.

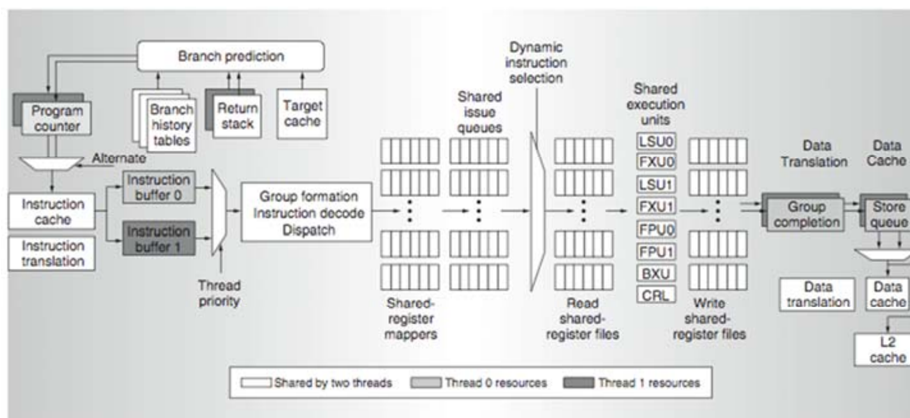
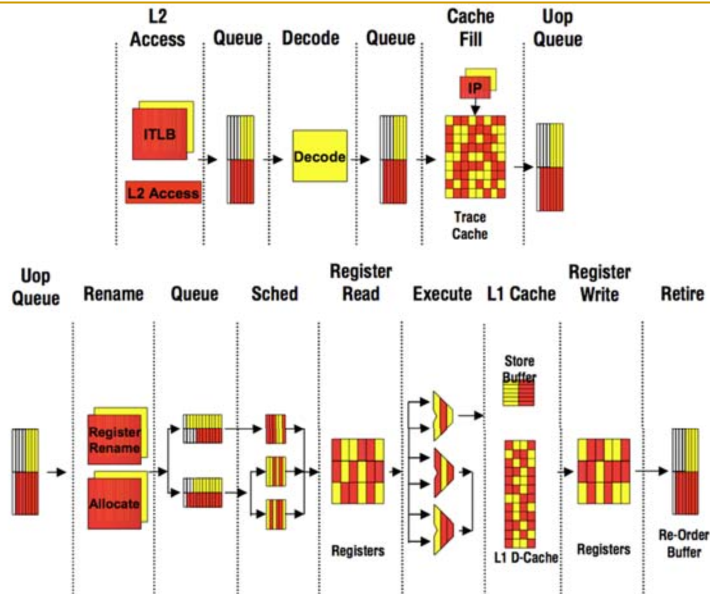


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

IBM POWER5 Thread Throttling

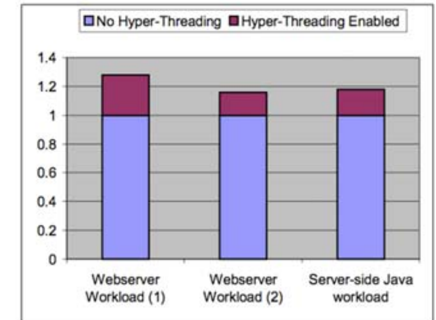
- Throttle under two conditions:
  - Resource-balancing logic detects the point at which a thread reaches a threshold of load misses in the L2 cache and translation misses in the TLB.
  - Resource-balancing logic detects that one thread is beginning to use too many GCT (i.e., reorder buffer) entries.
- Throttling mechanisms:
  - Reduce the priority of the thread
  - Inhibit the instruction decoding of the thread until the congestion clears
  - Flush all of the thread's instructions that are waiting for dispatch and stop the thread from decoding additional instructions until the congestion clears

Intel Pentium 4 Hyperthreading



Intel Pentium 4 Hyperthreading

- Long latency load handling
  - Multi-level scheduling window
- More partitioned structures
  - I-TLB
  - Instruction Queues
  - Store buffer
  - Reorder buffer
- 5% area overhead due to SMT



- Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal 2002.

Other Uses of Multithreading

Now that We Have MT Hardware …

- … what else can we use it for?
- Redundant execution to tolerate soft (and hard?) errors
- Implicit parallelization: thread level speculation
  - Slipstream processors
  - Leader-follower architectures
- Helper threading
  - Prefetching
  - Branch prediction
- Exception handling

SMT for Transient Fault Detection

- Transient faults: Faults that persist for a “short” duration
  - Also called “soft errors”
- Caused by cosmic rays (e.g., neutrons)
- Leads to transient changes in wires and state (e.g., 0→1)

- Solution
  - no practical absorbent for cosmic rays
  - 1 fault per 1000 computers per year (estimated fault rate)
- Fault rate likely to increase in the feature
  - smaller feature size
  - reduced voltage
  - higher transistor count
  - reduced noise margin

Need for Low-Cost Transient Fault Tolerance

- The rate of transient faults is expected to increase significantly → Processors will need some form of fault tolerance.

- However, different applications have different reliability requirements (e.g. server-apps vs. games) → Users who do not require high reliability may not want to pay the overhead.

- Fault tolerance mechanisms with low hardware cost are attractive because they allow the designs to be used for a wide variety of applications.

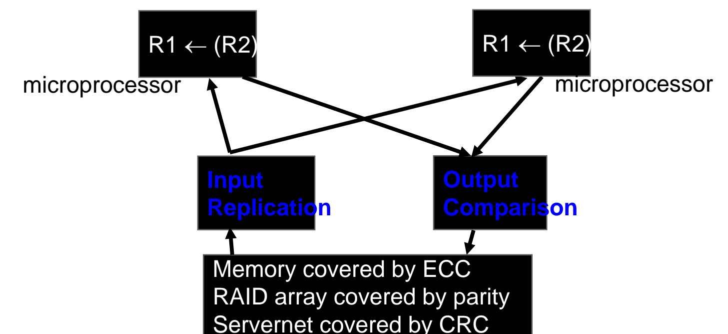
Traditional Mechanisms for Transient Fault Detection

- Storage structures
  - Space redundancy via parity or ECC
  - Overhead of additional storage and operations can be high in time-critical paths

- Logic structures
  - Space redundancy: replicate and compare
  - Time redundancy: re-execute and compare

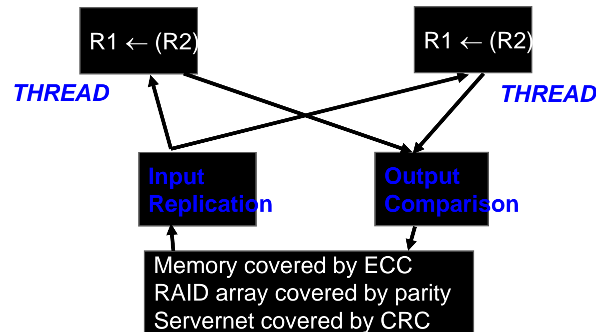
- Space redundancy has high hardware overhead.
- Time redundancy has low hardware overhead but high performance overhead.
- What additional benefit does space redundancy have?

Lockstepping (Tandem, Compaq Himalaya)



- Idea: Replicate the processor, compare the results of two processors before committing an instruction

Transient Fault Detection with SMT (SRT)



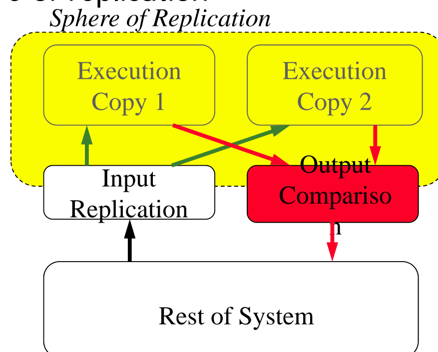
- Idea: Replicate the threads, compare outputs before committing an instruction
- Reinhardt and Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," ISCA 2000.
- Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," FTCS 1999.

Sim. Redundant Threading vs. Lockstepping

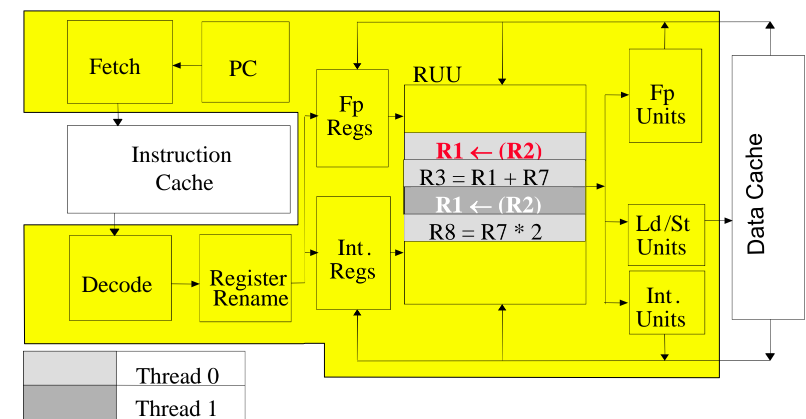
- SRT Advantages
  - \+ No need to replicate the processor
  - \+ Uses fine-grained idle FUs/cycles (due to dependencies, misses) to execute the same program redundantly on the same processor
  - \+ Lower hardware cost, better hardware utilization
- Disadvantages
  - \- More contention between redundant threads → higher performance overhead (assuming unequal hardware)
  - \- Requires changes to processor core for result comparison, value communication
  - \- Must carefully fetch & schedule instructions from threads
  - \- Cannot easily detect hard (permanent) faults

Sphere of Replication

- Logical boundary of redundant execution within a system
- Need to replicate input data from outside of sphere of replication to send to redundant threads
- Need to compare and validate output before sending it out of the sphere of replication

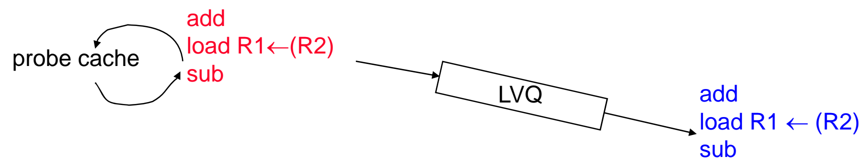


Sphere of Replication in SRT



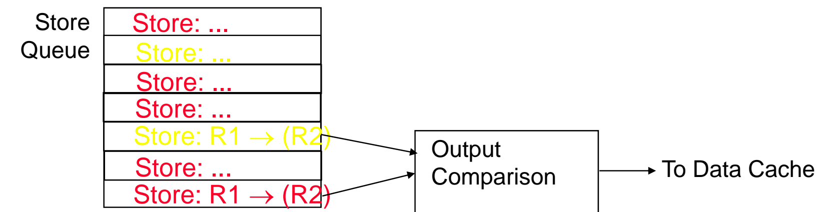
Input Replication

- How to get the load data for redundant threads
  - pair loads from redundant threads and access the cache when both are ready: too slow – threads fully synchronized
  - allow both loads to probe cache separately: false alarms with I/O or multiprocessors
- Load Value Queue (LVQ)
  - pre-designated leading & trailing threads



Output Comparison

- <address, data> for stores from redundant threads
  - compare & validate at commit time



- How to handle cached vs. uncacheable loads
- Stores now need to live longer to wait for trailing thread
- Need to ensure matching trailing store can commit

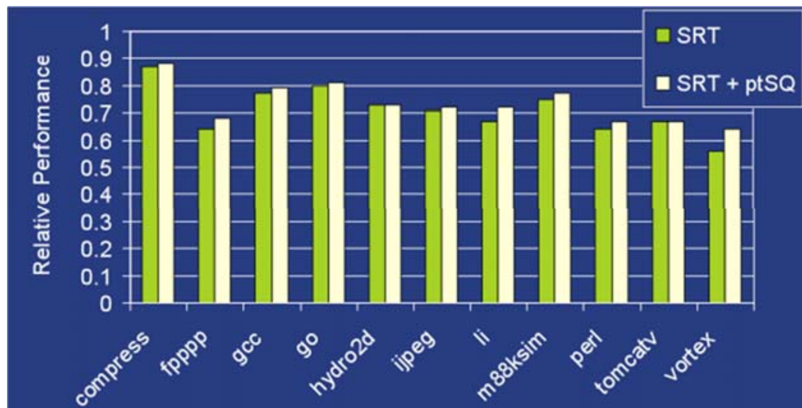
Handling of Permanent Faults via SRT

- SRT uses time redundancy
  - Is this enough for detecting permanent faults?
  - Can SRT detect some permanent faults? How?
- Can we incorporate explicit space redundancy into SRT?
- Idea: Execute the same instruction on different resources in an SMT engine
  - Send instructions from different threads to different execution units (when possible)

SRT Evaluation

- SPEC CPU95, 15M instrs/thread
  - Constrained by simulation environment
  - → 120M instrs for 4 redundant thread pairs
- Eight-issue, four-context SMT CPU
  - Based on Alpha 21464
  - 128-entry instruction queue
  - 64-entry load and store queues
    - Default: statically partitioned among active threads
  - 22-stage pipeline
  - 64KB 2-way assoc. L1 caches
  - 3 MB 8-way assoc L2

Performance Overhead of SRT

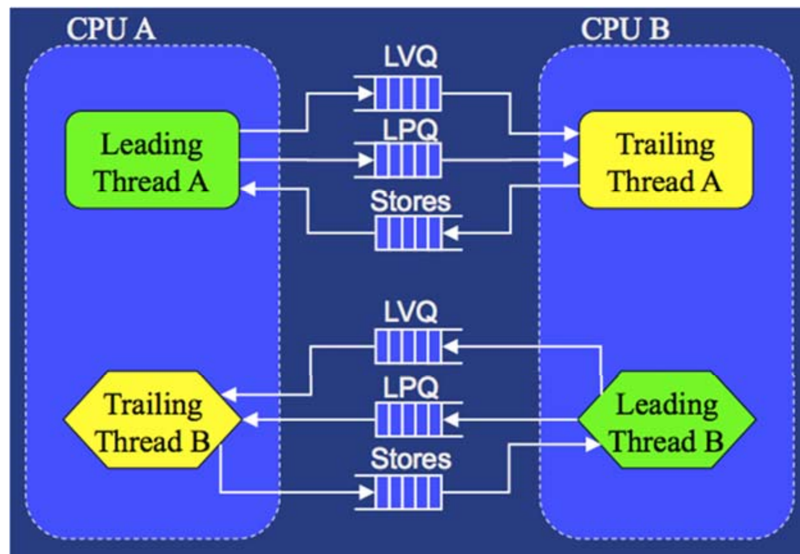


- Performance degradation = 30% (and unavailable thread context)
- Per-thread store queue improves performance by 4%

Chip Level Redundant Threading

- SRT typically more efficient than splitting one processor into two half-size cores
- What if you already have two cores?
- Conceptually easy to run these in lock-step
  - Benefit: full physical redundancy
  - Costs:
    - Latency through centralized checker logic
    - Overheads (e.g., branch mispredictions) incurred twice
- We can get both time redundancy and space redundancy if we have multiple SMT cores
  - SRT for CMPs

Chip Level Redundant Threading



Some Other Approaches to Transient Fault Tolerance

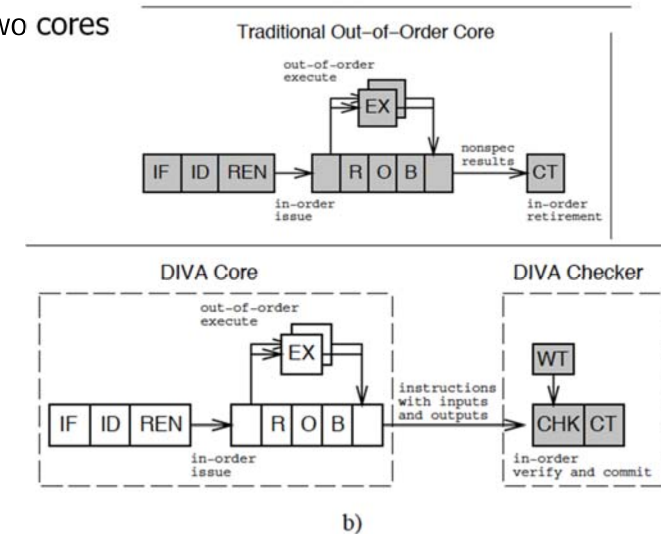
- Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," MICRO 1999.
- Qureshi et al., "Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors," DSN 2005.

DIVA

- Idea: Have a "functional checker" unit that checks the correctness of the computation done in the "main processor"
- Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," MICRO 1999.
- Benefit: Main processor can be prone to faults or sometimes incorrect (yet very fast)
- How can checker keep up with the main processor?
  - Verification of different instructions can be performed in parallel (if an older one is incorrect all later instructions will be flushed anyway)

DIVA (Austin, MICRO 1999)

- Two cores



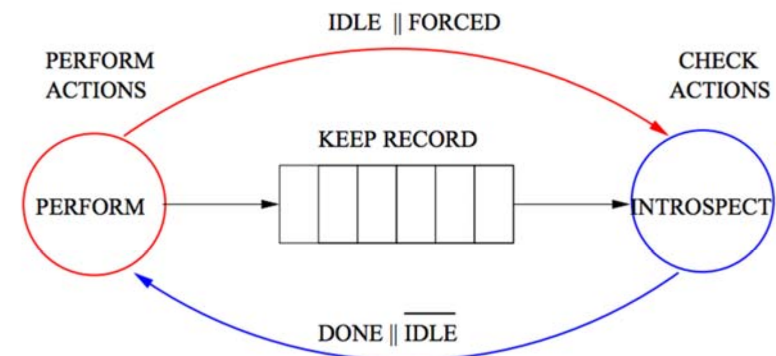
b)

Microarchitecture Based Introspection

- Idea: Use cache miss stall cycles to redundantly execute the program instructions
- Qureshi et al., "Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors," DSN 2005.
- Benefit: Redundant execution does not have high performance overhead (when there are stall cycles)
- Downside: What if there are no/few stall cycles?

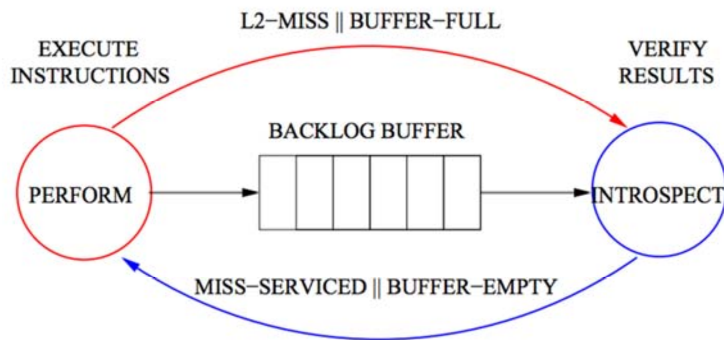
Introspection

An example from the human domain:



MBI (Qureshi+, DSN 2005)

Extending it to the microarchitecture domain:



Microarchitecture-Based Introspection (MBI)

MBI Microarchitecture

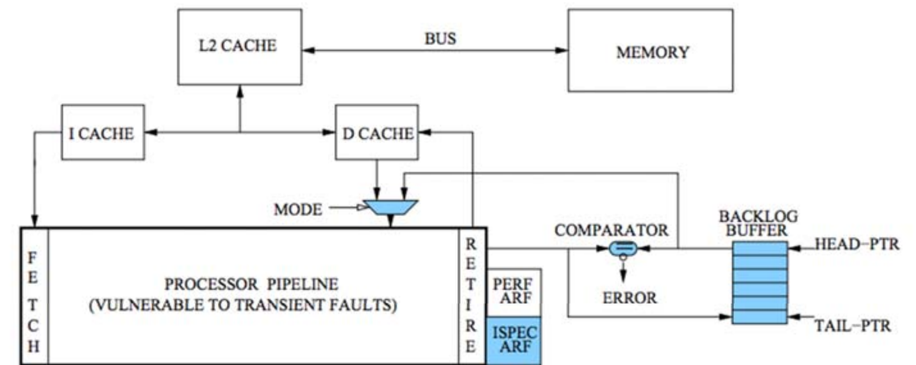


Figure 3. Microarchitecture support for MBI.

Performance Impact of MBI

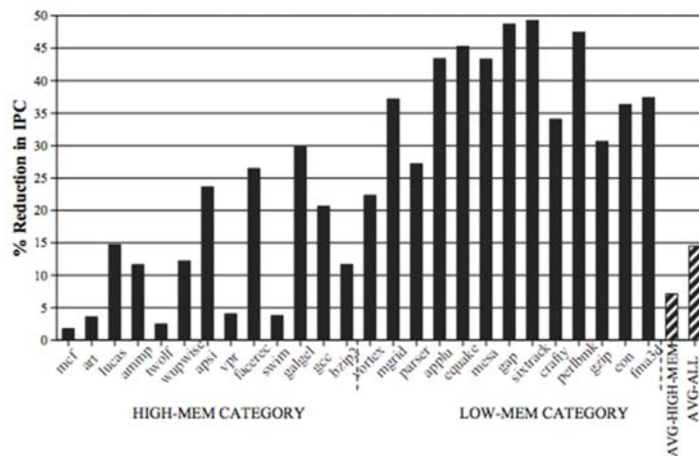


Figure 4. IPC reduction due to the MBI mechanism.

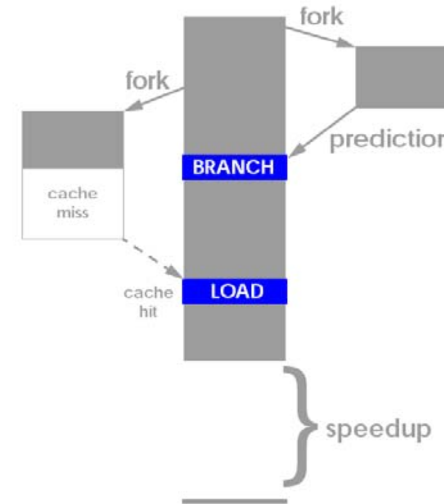
Helper Threading for Prefetching

- Idea: Pre-execute a piece of the (pruned) program solely for prefetching data
  - Only need to distill pieces that lead to cache misses
- Speculative thread: Pre-executed program piece can be considered a "thread"
- Speculative thread can be executed
- On a separate processor/core
- On a separate hardware thread context
- On the same thread context in idle cycles (during cache misses)

Helper Threading for Prefetching

- How to construct the speculative thread:
  - Software based pruning and "spawn" instructions
  - Hardware based pruning and "spawn" instructions
  - Use the original program (no construction), but
    - Execute it faster without stalling and correctness constraints
- Speculative thread
  - Needs to discover misses before the main program
    - Avoid waiting/stalling and/or compute less
  - To get ahead, uses
    - Branch prediction, value prediction, only address generation computation

Generalized Thread-Based Pre-Execution



- Dubois and Song, "Assisted Execution," USC Tech Report 1998.
- Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," ISCA 1999.
- Zilles and Sohi, "Execution-based Prediction Using Speculative Slices", ISCA 2001.

Thread-Based Pre-Execution Issues

- Where to execute the precomputation thread?
  1. Separate core (least contention with main thread)
  2. Separate thread context on the same core (more contention)
  3. Same core, same context
     - When the main thread is stalled
- When to spawn the precomputation thread?
  1. Insert spawn instructions well before the "problem" load
     - How far ahead?
       - Too early: prefetch might not be needed
       - Too late: prefetch might not be timely
  2. When the main thread is stalled
- When to terminate the precomputation thread?
  1. With pre-inserted CANCEL instructions
  2. Based on effectiveness/contention feedback

Slipstream Processors

- Goal: use multiple hardware contexts to speed up single thread execution (implicitly parallelize the program)
- Idea: Divide program execution into two threads:
  - Advanced thread executes a reduced instruction stream, speculatively
  - Redundant thread uses results, prefetches, predictions generated by advanced thread and ensures correctness
- Benefit: Execution time of the overall program reduces
- Core idea is similar to many thread-level speculation approaches, except with a reduced instruction stream
- Sundaramoorthy et al., "Slipstream Processors: Improving both Performance and Fault Tolerance," ASPLOS 2000.

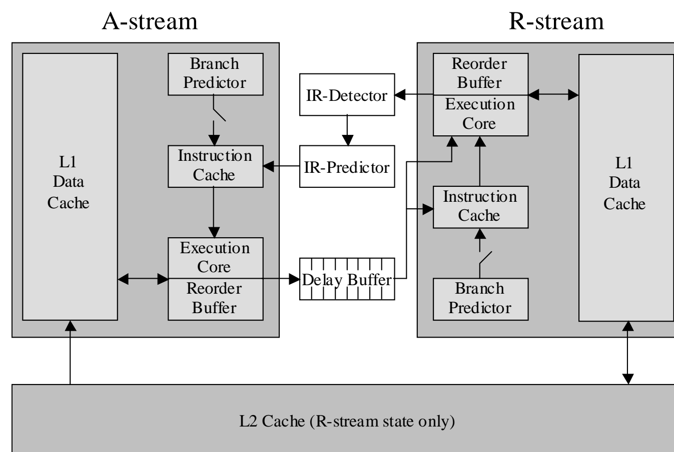
Slipstreaming

- "At speeds in excess of 190 m.p.h., high air pressure forms at the front of a race car and a partial vacuum forms behind it. This creates drag and limits the car's top speed.
- A second car can position itself close behind the first (a process called *slipstreaming* or *drafting*). This fills the vacuum behind the lead car, reducing its drag. And the trailing car now has less wind resistance in front (and by some accounts, the vacuum behind the lead car actually helps pull the trailing car).
- As a result, both cars speed up by several m.p.h.: the two combined go faster than either can alone."

Slipstream Processors

- Detect and remove ineffectual instructions; run a shortened "effectual" version of the program (Advanced or A-stream) in one thread context
- Ensure correctness by running a complete version of the program (Redundant or R-stream) in another thread context
- Shortened A-stream runs fast; R-stream consumes near-perfect control and data flow outcomes from A-stream and finishes close behind
- Two streams together lead to faster execution (by helping each other) than a single one alone

Slipstream Idea and Possible Hardware

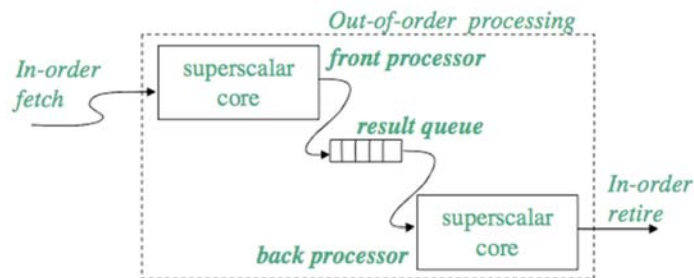


Slipstream Questions

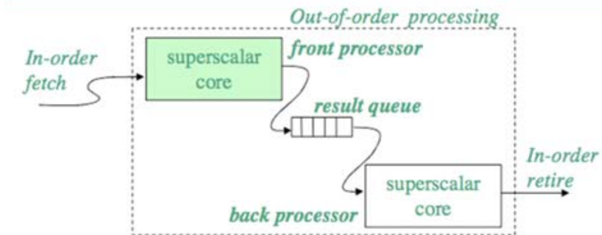
- How to construct the advanced thread
  - Original proposal:
    - Dynamically eliminate redundant instructions (silent stores, dynamically dead instructions)
    - Dynamically eliminate easy-to-predict branches
  - Other ways:
    - Dynamically ignore long-latency stalls
    - Static based on profiling
- How to speed up the redundant thread
  - Original proposal: Reuse instruction results (control and data flow outcomes from the A-stream)
  - Other ways: Only use branch results and prefetched data as predictions

Dual Core Execution

- Idea: One thread context speculatively runs ahead on load misses and prefetches data for another thread context
- Zhou, "Dual-Core Execution: Building a Highly Scalable Single- Thread Instruction Window," PACT 2005.

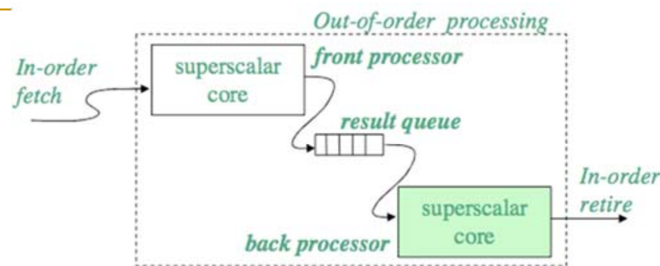


Dual Core Execution: Front Processor



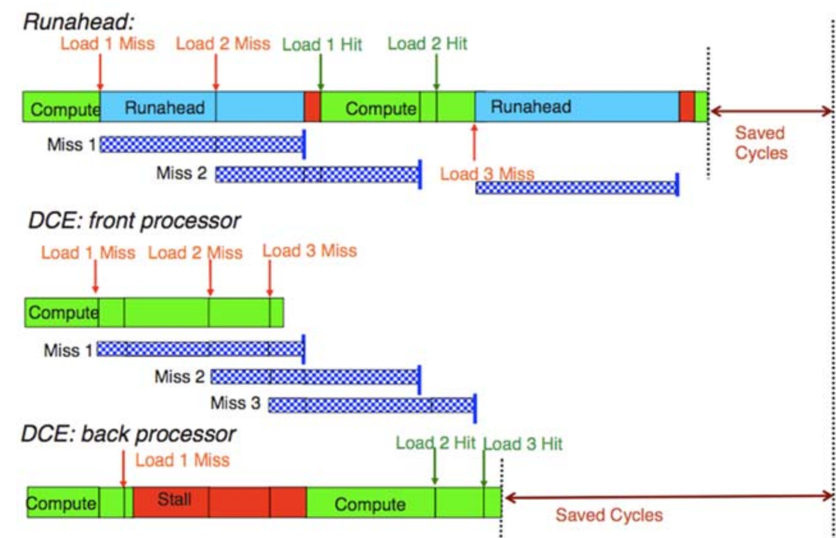
- The front processor runs faster by invalidating long-latency cache-missing loads, same as runahead execution
  - Load misses and their dependents are invalidated
  - Branch mispredictions dependent on cache misses cannot be resolved
- Highly accurate execution as independent operations are not affected
  - Accurate prefetches to warm up caches
  - Correctly resolved independent branch mispredictions

Dual Core Execution: Back Processor



- Re-execution ensures correctness and provides precise program state
  - Resolve branch mispredictions dependent on long-latency cache misses
- Back processor makes faster progress with help from the front processor
  - Highly accurate instruction stream
  - Warmed up data caches

Dual Core Execution



DCE Microarchitecture

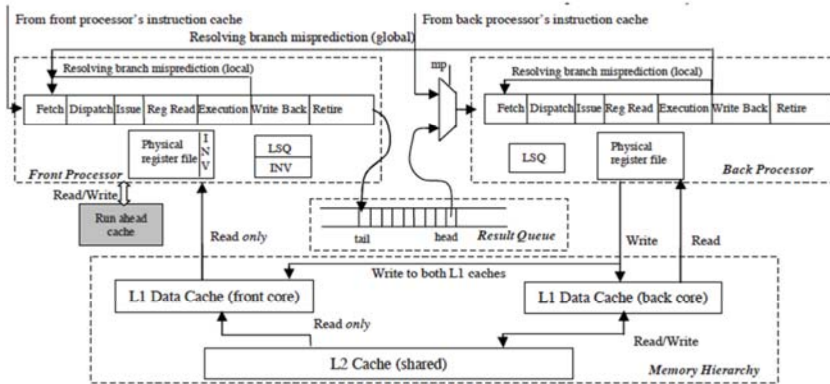


Figure 3. The design of DCE architecture.

Dual Core Execution vs. Slipstream

- Dual-core execution does not
  - remove dead instructions
  - reuse instruction register results
  - uses the "leading" hardware context solely for prefetching and branch prediction

+ Easier to implement, smaller hardware cost and complexity

- "Leading thread" cannot run ahead as much as in slipstream when there are no cache misses

- Not reusing results in the "trailing thread" can reduce overall performance benefit

Some Results

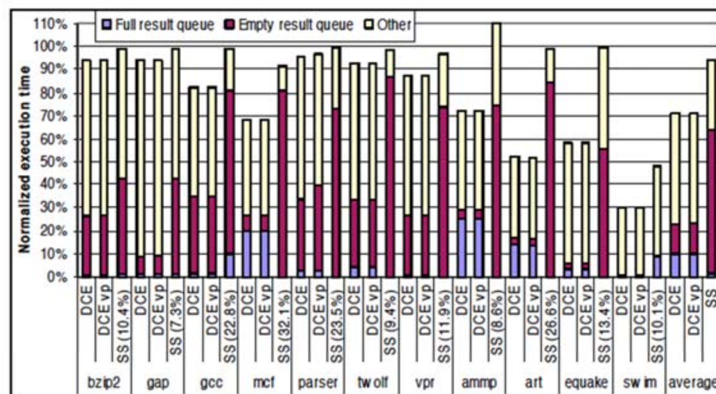


Figure 6. Normalized execution time of DCE, DCE with value prediction (DCE vp), and slipstreaming processors (SS).

Thread Level Speculation

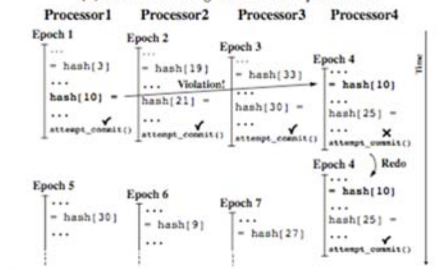
- Speculative multithreading, dynamic multithreading, etc…

- Idea: Divide a single instruction stream (speculatively) into multiple threads at compile time or run-time
  - Execute speculative threads in multiple hardware contexts
  - Merge results into a single stream

- Hardware/software checks if any true dependencies are violated and ensures sequential semantics
- Threads can be assumed to be independent
- Value/branch prediction can be used to break dependencies between threads
- Entire code needs to be correctly executed to verify such predictions

Thread Level Speculation Example

- Colohan et al., "A Scalable Approach to Thread-Level Speculation," ISCA 2000.



TLS Conflict Detection Example

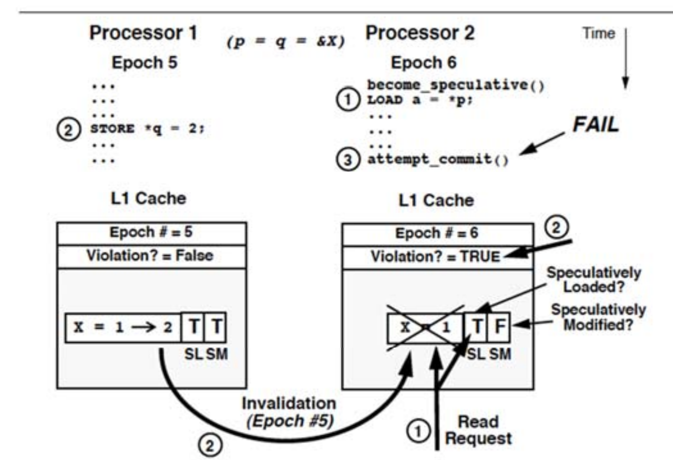
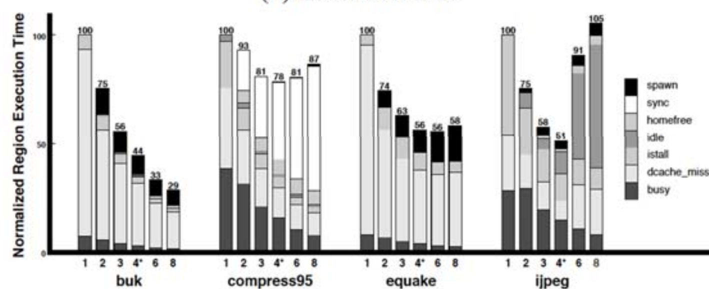


Figure 2. Using cache coherence to detect a RAW dependence violation.

Some Sample Results [Colohan+ ISCA 2000]

Table 3. Performance impact of TLS on our baseline architecture (a four-processor single-chip multiprocessor).

Application	Overall Region Speedup	Parallel Coverage	Program Speedup
buk	2.26	56.6%	1.46
compress95	1.27	47.3%	1.12
equake	1.77	39.3%	1.21
ijpeg	1.94	22.1%	1.08



Other MT Issues

- How to select threads to co-schedule on the same processor?
  - Which threads/phases go well together?
  - This issue exists in multi-core as well
- How to provide performance isolation (or predictable performance) between threads?
  - This issue exists in multi-core as well
- How to manage shared resources among threads
  - Pipeline, window, registers
  - Caches and the rest of the memory system
  - This issue exists in multi-core as well

Why These Uses?

- What benefit of multithreading hardware enables them?

- Ability to communicate/synchronize with very low latency between threads
  - Enabled by proximity of threads in hardware
  - Multi-core has higher latency to achieve this