Architectural and Implementation Tradeoffs for Multiple-Context Processors

Motivation

- **Latency is a serious problem for modern processors**
  - → wide gap between processor and memory speeds
  - → deeply pipelined
  - → multiprocessors
- **Three major forms of latency**
  - → memory
  - → instruction
  - → synchronization

Coping with Latency

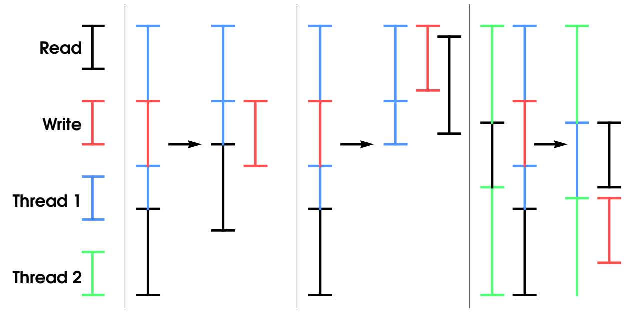
- **Two-step approach to managing latency**
  - → First, reduce latency
    - coherent caches
    - locality optimizations
    - pipeline bypassing
  - → Then, tolerate remaining latency
    - relaxed memory consistency
    - prefetch
    - multiple context

Multiple Context Processors

- **Multiple context processors address latency by:**
  - → switching to another thread whenever one thread performs a long latency operation
  - → making sure that context switch overhead is low, so that thread switches can be done often

Latency Tolerance Techniques

	Relaxed Consistency	Prefetch	Multiple Contexts
Parallelism	Single-thread	Single-thread	Multiple-thread
Latency Tolerated	Write	Memory	Memory Instruction Synchronization



Trends

- **What are the microprocessor trends?**
  - → relaxed memory consistency
  - → prefetch
- **Why hasn't multiple contexts been included?**
  - → multiple contexts is thought to be expensive
  - → performance benefits are relatively unknown
  - → existing multiple context designs do not help uniprocessors

⇒ **For multiple contexts to gain acceptance, all these issues must be addressed**

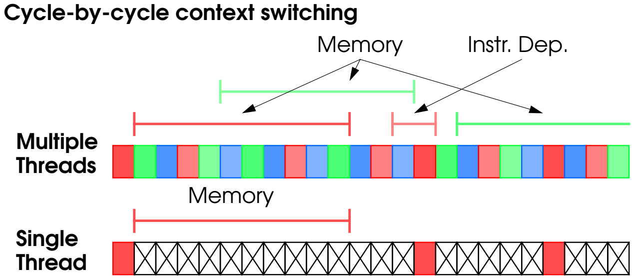
Outline

⇒ **Multiple-Context Approaches**

- **Performance Results**
- **Implementation Issues**
- **Conclusions**

HEP and TERA Approach

- **HEP was first machine to use mult. ctxts. to hide latency**
- **Processor architecture:**
  - → pipelined but no interlocks, and no caches
  - → large # of registers (2K), cheap thread creation, F/E bits
- **Cycle-by-cycle context switching**



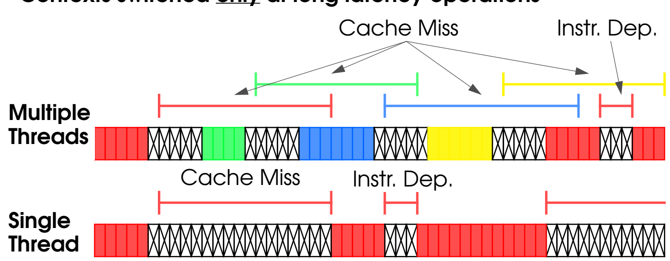
HEP and TERA Approach (cont.)

- **Multiple context used to hide two kinds of latency:**
  - → pipeline latency (8 cycles) and memory latency
  - → 128 contexts per processor (total focus on toleration)
- **Drawbacks of HEP approach**
  - → Low single context performance (bad for applns with limited parallelism)
  - → Lots of contexts implied lots of hardware resources and high cost
  - → No caches implied high memory bandwidth requirements and high cost

TERA System Pipeline

Blocked Scheme

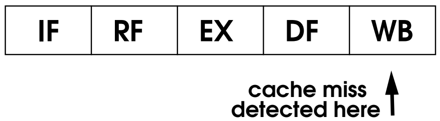
- **Combine multiple contexts with latency reduction (caches)**
  - → Assumes a base cache-coherent system
  - → Assumes pipelined processor with interlocks
- **Contexts switched <u>only</u> at long latency operations**



Design Considerations

- **Issues:**
  - → number of contexts per PE
  - → context switch overhead
  - → effect of memory latency
  - → cache interference effects
  - → when to switch contexts
  - → implementation issues
- **Advantage of blocked scheme**
  - → small number of contexts suffice to hide memory latency
- **Disadvantage of blocked scheme**
  - → switch overhead still quite large to hide pipeline latency

Context Switch Cost



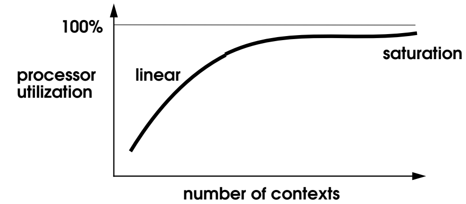
- **Cache miss detected late in pipeline**

  → squash partially executed instructions

  → start fetching instructions from next context

Swicth Cost, Latency, and Num Ctxts

- **Given #-of-ctxts = k, avg-run-length = R, switch-cost = C, avg-latency = L**



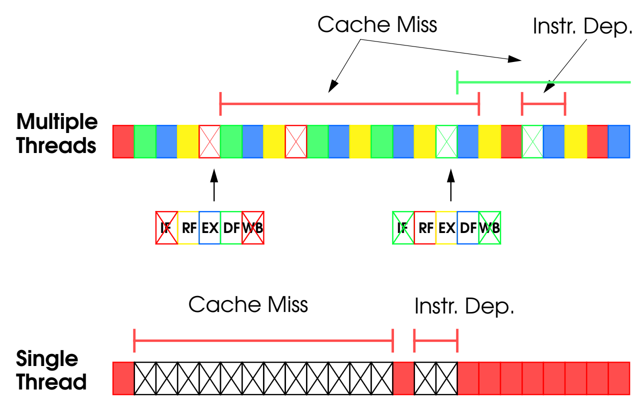
- **In linear region, proc. utilization = (k x R) / (L + C)**
  - → Knee of curve is close to k = L/ (R + C)
  - → Cost(n) = $shared + n x $inc + $ovhd
- **In saturation region, max proc. utilization = R / (R + C)**
  - → Efficiency increases only marginally with more contexts
  - → Important to keep C small if R is going to be low

Interleaved Scheme

- **Assumptions**
  - → coherent caches
  - → parallelism available, but not necessarily abundant
- **Full single-thread support**
- **Cycle-by-cycle interleaving**
  - → lowers switch cost
  - → instruction latency tolerance

⇒ **Combines best of HEP-like and blocked approaches**

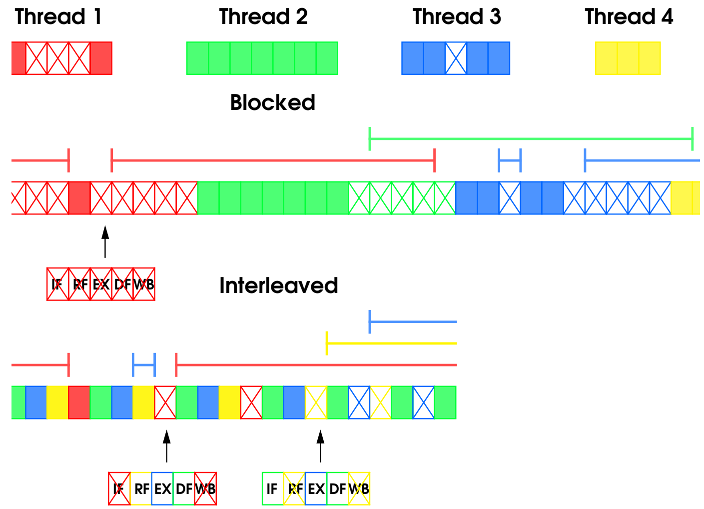
Interleaved Scheme



Interleaved vs. Blocked

Type of Latency	Blocked	Interleaved
Memory	Context Switch	Make Context Unavailable
Synchronization	Context Switch	Make Context Unavailable
Instruction (Long)	Context Switch	Make Context Unavailable
Instruction (Short)	STALL!	Rely on Context Interleaving

Interleaved vs. Blocked

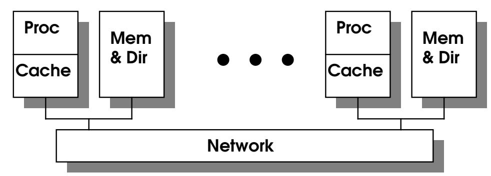


Outline

- Current Multiple-Context Approaches
- ⇒ Performance Results
- Implementation Issues
- Conclusions

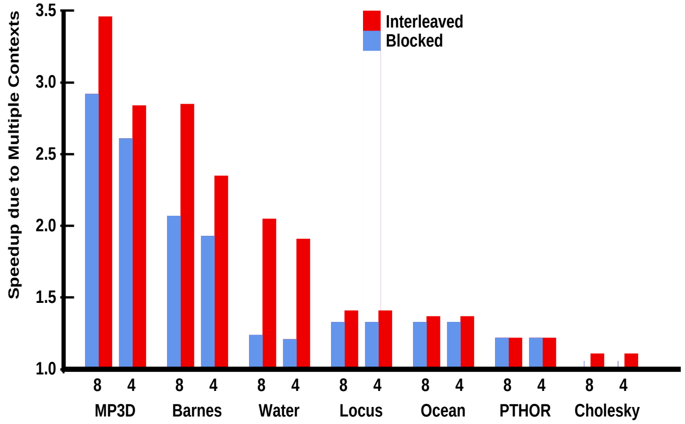
Methodology

- **Shared-memory multiprocessor**

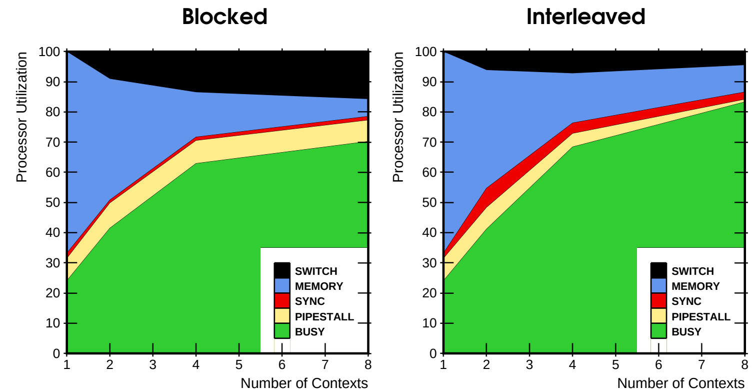


  - → 16 processors, 1-8 contexts per processor
  - → pipeline based on R4000 (pipelined floating-point)
  - → ideal instruction cache, 64K data cache
  - → memory latencies (1:35:105:135)
- **Event-driven simulation**
  - → optimized code, scheduled for pipeline
- **Parallel application suite (SPLASH)**

Simulation Results

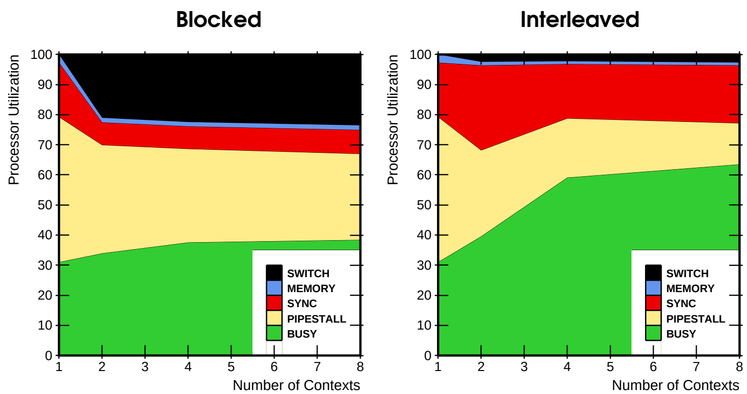


MP3D: Memory Latency



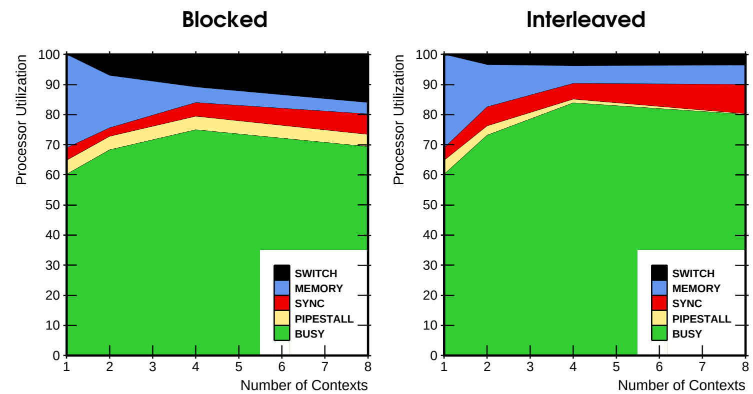
- Both schemes effective in tolerating memory latency
- Interleaved has lower context switch overhead

Water: Instruction Latency



Floating Point Operation	Latency	Pipelined
Division	60	No
Add, Convert, Multiply	5	Yes

LocusRoute: Limited Latency



→ latency is a problem

→ application has additional parallelism

Performance Summary

- **Multiple contexts works well when extra parallelism is available**
- **Interleaved scheme has performance advantage**
  - → mean speedup for blocked scheme: 1.61
  - → mean speedup for interleaved scheme: 1.93

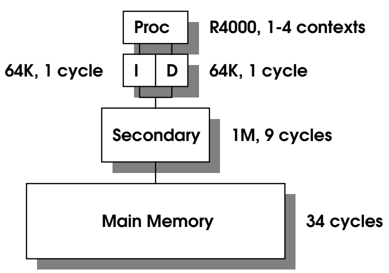
Uniprocessor Issues

- **More difficult environment for multiple contexts**

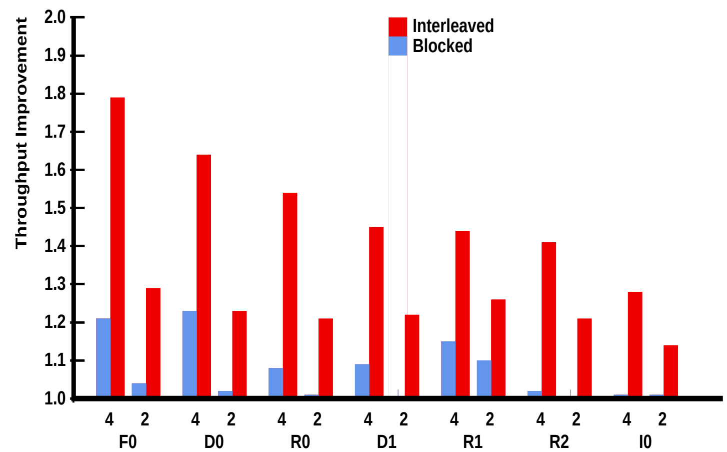
  - → greater cache interference
  - → needs to tolerate shorter latencies

Uniprocessor Study

- **Several workloads composed of SPLASH and SPEC applications**
  - → Three random (R0-R2)
  - → Two stress the data cache (D0-D1)
  - → One stresses the instruction cache (I0)
  - → One is floating point intensive (F0)
- **Simulation parameters**



Simulation Results



Uniprocessor Summary

- **Blocked scheme unable to address uniprocessor needs**
- **Interleaved scheme able to improve uniprocessor throughput**

  → mean improvement of 50% for our workloads

Outline

- **Current Multiple-Context Approaches**
- **Performance Results**

⇒ **Implementation Issues**

- **Conclusions**

Implementation Study

- **Single data point in existence for blocked scheme (MIT APRIL)**
- **Explored implementation issues for both schemes**

  → enough detail to expose major issues
  - RTL diagrams
  - transistor level + layout + Spice

Basic Implementation Needs

- **Cache capable of multiple outstanding requests (lockup-free)**
- **Replication of key hardware state**
- **Context scheduling logic**

Lockup-free Cache

- **Required for all latency tolerance schemes**

State Replication

- **Register File**
- **Program Counter Related**
- **Process Status Word**

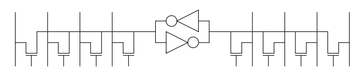
State Replication

- **Blocked scheme - single active context**
  - → single set of active state
    - master copy plus backup copies
    - swap master and backup during context switch
- **Interleaved - all contexts active**
  - → all sets of state active
  - → state used changes each cycle

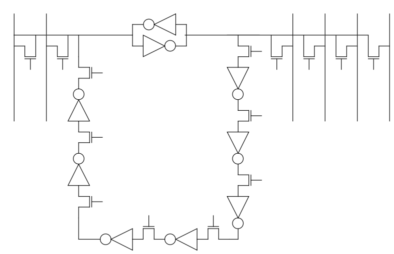
State Replication Example

- **Optimizing the four-context register file**

Single-context

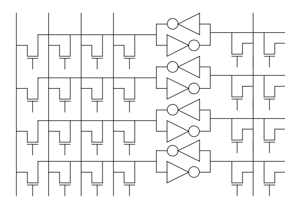


Blocked



2x single-context area
15% longer access time

Interleaved



4x single-context area
35% longer access time

Context Control

- **Blocked Scheme**
  - → provide context switch signal
  - → global context identifier (CID)
  - → change CID and state during switch cycles
- **Interleaved Scheme**
  - → provide selective squash signal
  - → CID associated with each instruction (CID chain)
  - → CID becomes another pipeline control signal
    - state used depends on CID value

Concluding Remarks

- **Multiple contexts work well when combined with caches**
  - → better at handling unstructured programs
- ***Interleaved* multiple context architecture offers**
  - → better multiprocessor performance than blocked approach
  - → can improve uniprocessor throughput
- **Implementation of blocked and interleaved multiple-context architectures is manageable**
  - → more flexibility in implementing blocked scheme
  - → increase in area is small for both schemes
  - → should not impact cycle time

Implementation Summary

- **More implementation flexibility for blocked scheme**
  - → most of the time looks like a single context processor
  - → context control is simpler
- **Implementation cost and complexity is manageable for both schemes**
  - → small area overhead (e.g. register file is 2% of the R4000 die)
  - → extra delays not in critical path