

Informed Caching Environment

Alvin R. Lebeck

Computer Science Department
Duke University
<http://www.cs.duke.edu/ari/ice>
alvy@cs.duke.edu

Cache Memory Review

Very fast, 1ns clock,
Multiple Instructions
per cycle

P

\$

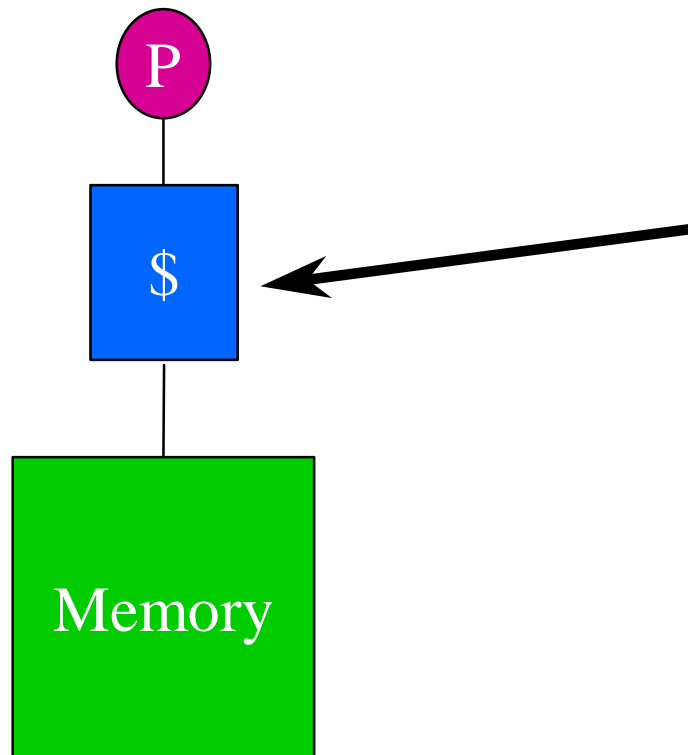
SRAM, Fast,
Expensive, Small

Memory

DRAM, Slow, Big,
Cheap

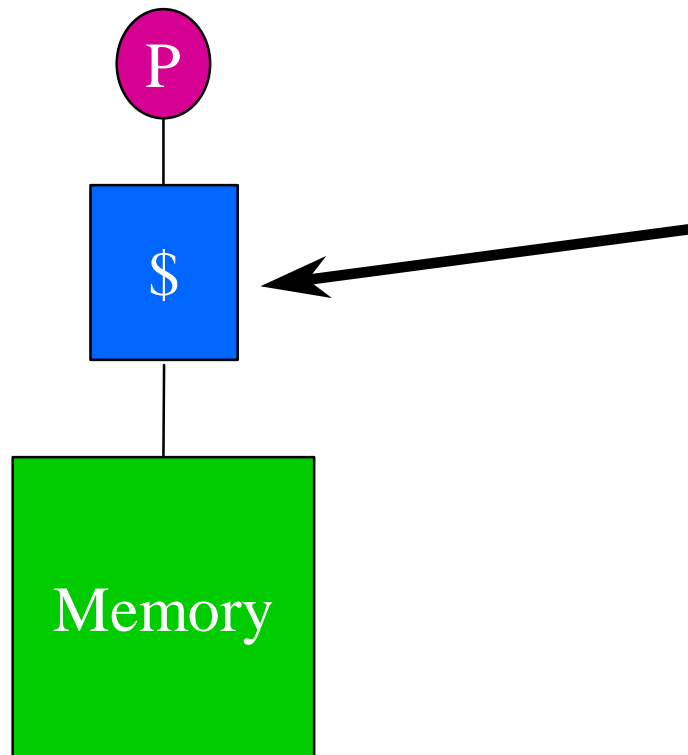
=> Cost Effective Memory System (Price/Performance)

Current Caches Are Naive



- Hardware Managed
- Use **naive history** of past references to manage content
- **No information from Program!**

Informed Caching Environment



- Augment conventional cache with set of sophisticated **mechanisms**
- Exploit **information** from program to improve cache management, thus overall performance

Outline

- Motivation
- ICE Overview
- Annotated Memory References
- Exploiting Information on Temporal and Spatial Locality
- Latency Tolerance in Dynamically Scheduled Processors
- Conclusion

Three Aspects of ICE

- How to obtain information?
Profiling, user directives, compiler, HW gadgets
- How to convey information?
Instructions, TLB, HW gadgets
- How to exploit the information?
New mechanisms, HW gadgets

What Information?

- **Locality**
 - Temporal: reuse same data items
 - Spatial: use nearby data
 - Use for cache replacement or fetch size
- **Latency tolerance in dynamically scheduled processors (e.g., Alpha 21264)**
 - can tolerate some long latency loads
- **Pointer Chasing**
- **Hints, not required for correct execution**

What Mechanisms?

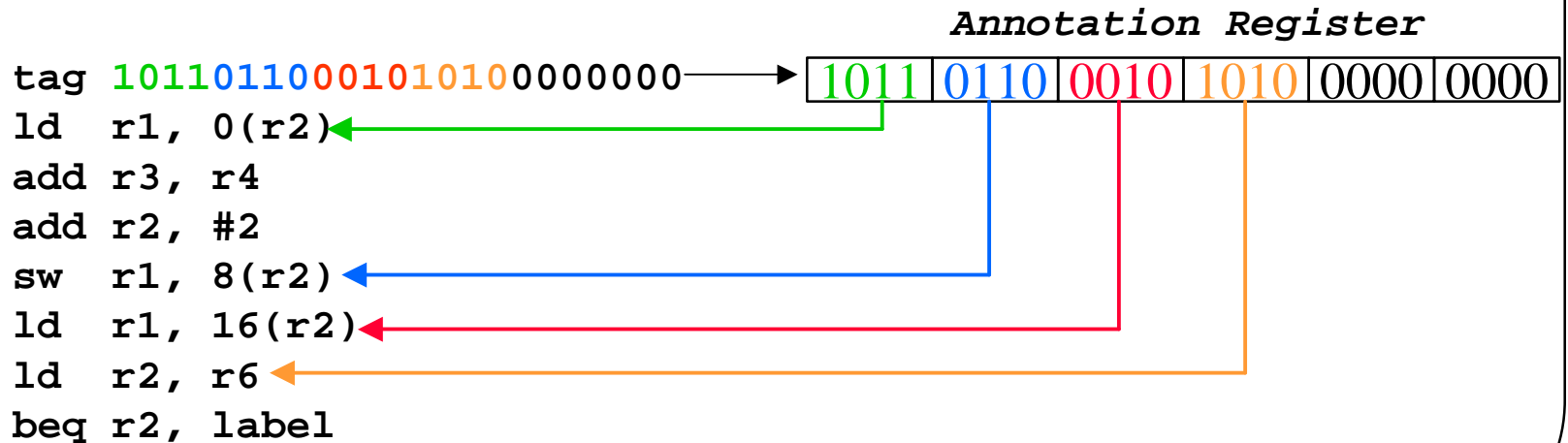
- Annotated memory references to convey information
 - Can annotate either instruction (PC) or data (effective address)
 - **New Instruction**, Compiler or programmer inserted
 - Hardware Gadgets
- Locality
 - Retain / Release Operations
 - Variable block size
- Latency tolerance
 - Keep **tolerant** data in slower (L2) cache
 - Keep **intolerant** data in faster (L1) cache
- Pointer Chasing
 - Push data up to processor

Annotated Memory References

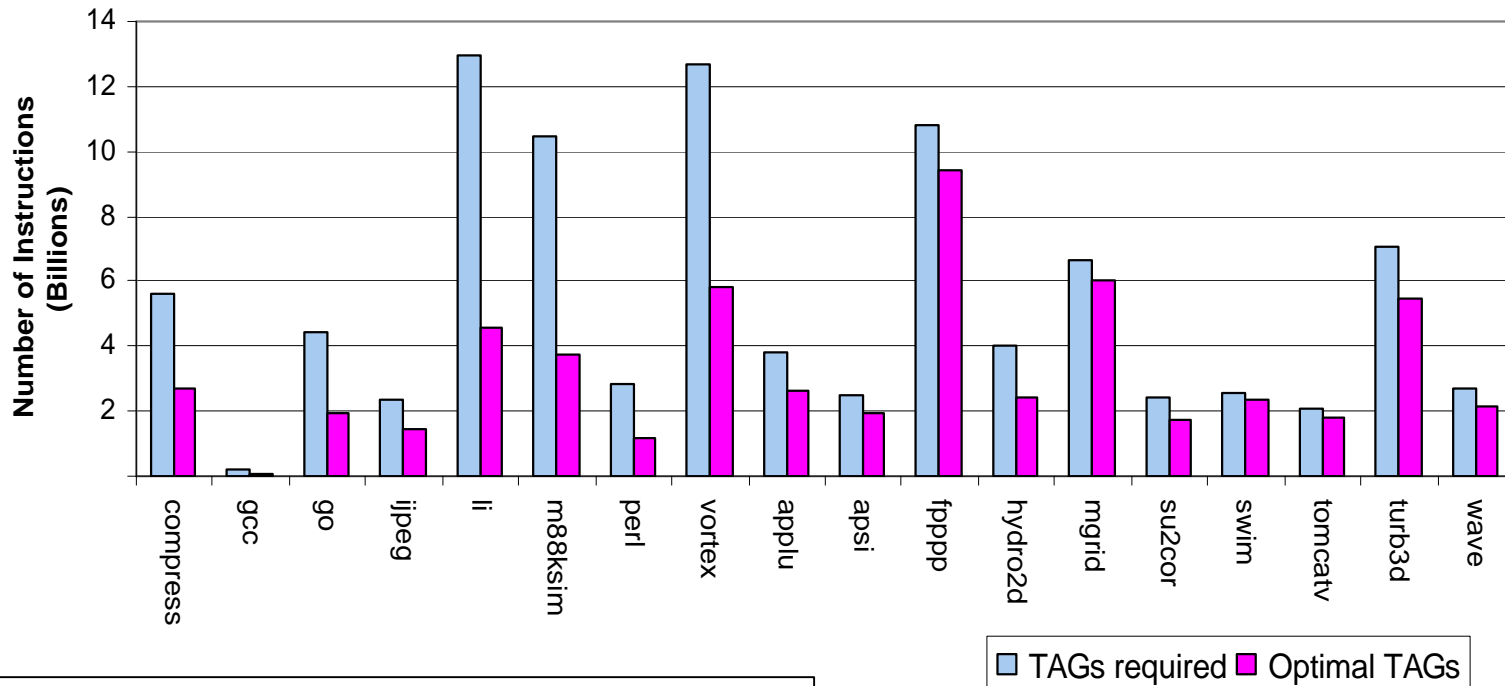
- New instruction (**TAG**)
- New hardware register -- the **annotation register**
 - Bits are extracted to annotate references
- TAG instruction provides annotations for the next several memory references
 - We can TAG 6 refs
- TAG required for each basic block

Annotation Example

<u>Instructions</u>	<u>Annotations</u>
ld r1, 0(r2)	1011
add r3, r4	
add r2, #2	
sw r1, 8(r2)	0110
ld r1, 16(r2)	0010
ld r2, r6	1010
beq r2, label	



TAG Coverage



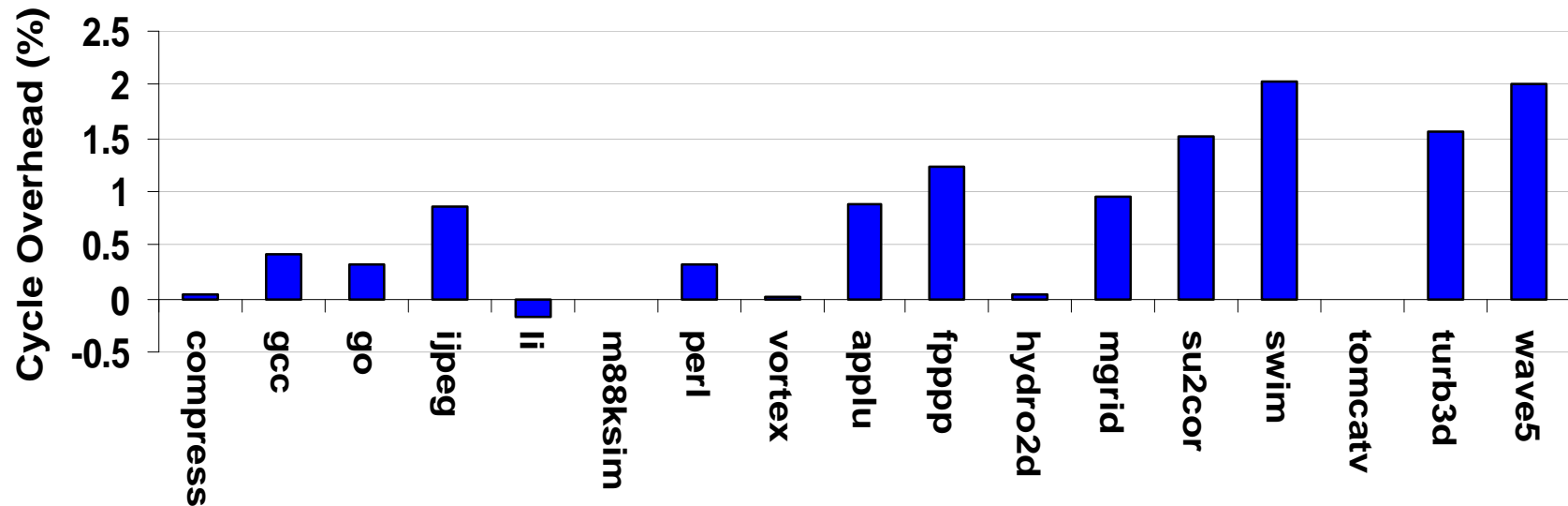
$$\text{Optimal TAGs} = \frac{\text{total memory references}}{6}$$

- Integer benchmarks have shorter basic blocks with fewer memory references

Cycle Overheads - Experiments

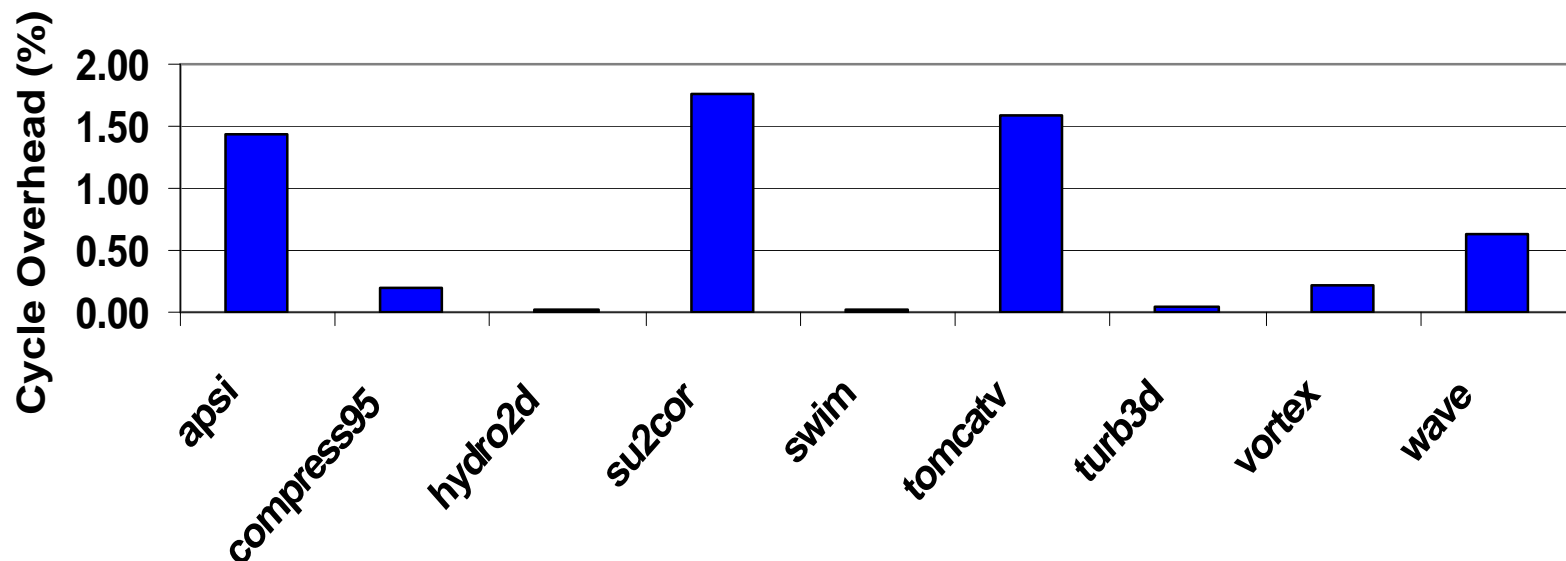
- **Statically scheduled processors**
 - ATOM based
 - 21164 issue policies
 - Perfect branch prediction
 - Ideal memory system
 - No inter-block dependencies
- **Dynamically scheduled processors**
 - SimpleScalar simulator
 - 4-way issue, 64 RUU, 32 LSQ
 - Simple overhead computation

Statically Scheduled Processor



- All memory references annotated
 - 4 bits/annotation; Tag coverage of 6
- Cycle overhead
 - Integer Codes: 0% to 0.85%
 - Floating Point Codes: 0% to 2%

Dynamically Scheduled Processor



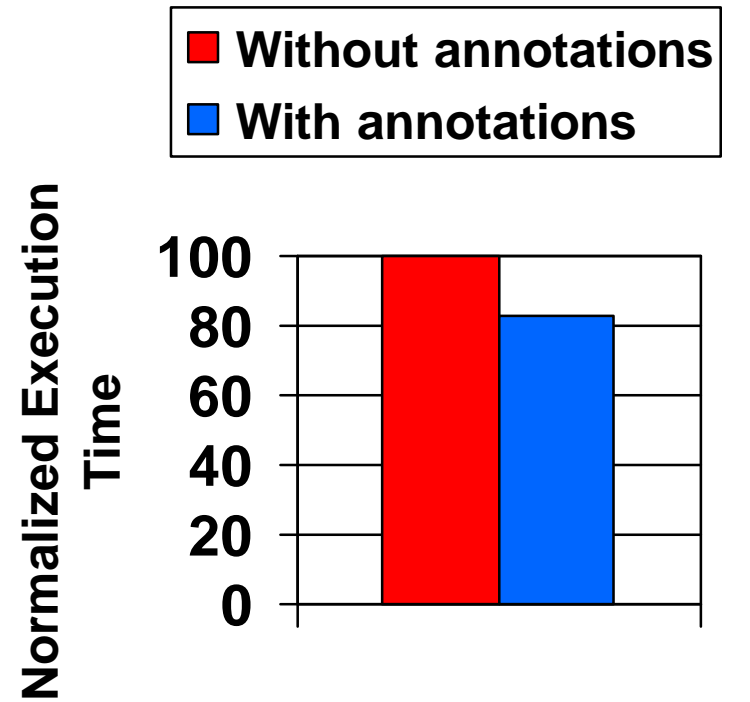
- All memory references annotated
 - 4 bits/annotation; Tag coverage of 6
- Cycle overhead
 - Integer Codes: 0% to 0.2%
 - Floating Point Codes: 0% to 1.76%

Utilizing Annotated Memory References

- Code inspection and manual insertion of annotations
- CProf tool to give insights of code operation
- Multimedia applications
 - epic, jpeg, pegwit
- Assume small simple cache

Better Block Replacement (epic)

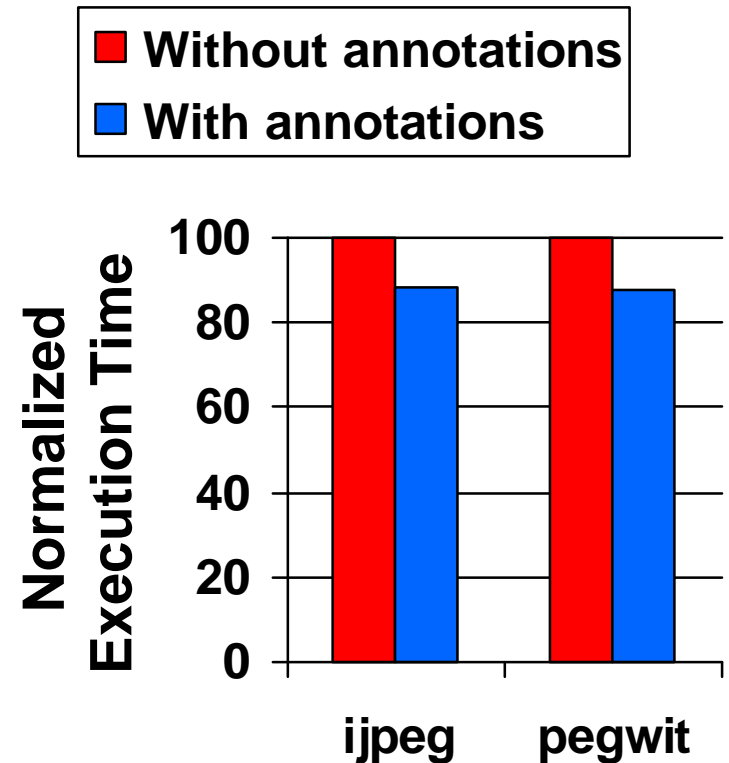
- Insight: some blocks should be retained even if LRU block
- Retain/Release annotations
- A block marked Retain cannot be replaced unless Released
- Bypass cache if no replacement candidate
- 17% improvement
- Would need 32-way associativity



4-way issue, OoO processor
64 RUU, 32 LSQ entries
8KB, 32 Byte block, Direct Mapped

Better Block Sizes (pegwit & ijpeg)

- Insight: Implicit prefetch of larger blocks hurts performance
- WordMode/BlockMode annotations
- WordMode annotated references bring in only a word and not the whole block
- 12% improvement
- Design dynamically reallocates space in cache



4-way issue, OoO processor
64 RUU, 32 LSQ entries
8KB, 32 Byte block, Direct Mapped

Related Work

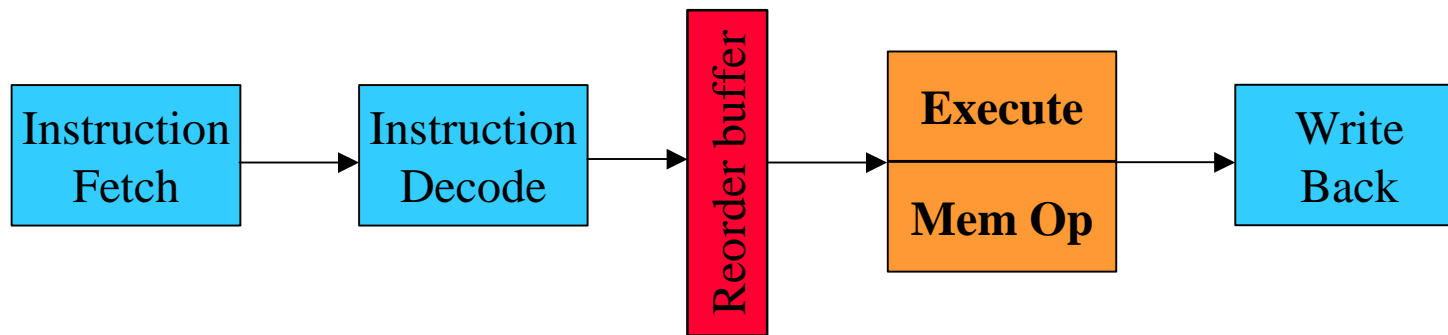
Tyson <i>et al</i> (95)	Temporal locality analysis	- 30% BW (up to 60%) - 27% data \$ misses
Temam <i>et al</i> (95)	Temporal & spatial tags	- 60% AMAT - 60% data \$ misses
Johnson/Hwu (97)	Adaptive cache hierarchy managment	- 15% execution time
ICE	Locality analysis	- 52% mem sys cycles - 11% -17% execution time

Outline

- Motivation / Background
- ICE Overview
- Annotated Memory References
- Exploiting Information on Temporal Locality
- Latency Tolerance in Dynamically Scheduled Processors
- Conclusions and Future Work

A Dynamically Scheduled Super-Scalar Processor

Out-of-Order Pipeline



- Dynamically schedule instructions from reorder buffer
 - issue and execute out-of-order
- Commit Instructions in-order
 - buffer results
- Speculative execution, branch prediction

Instruction Level Parallelism

- Mechanisms

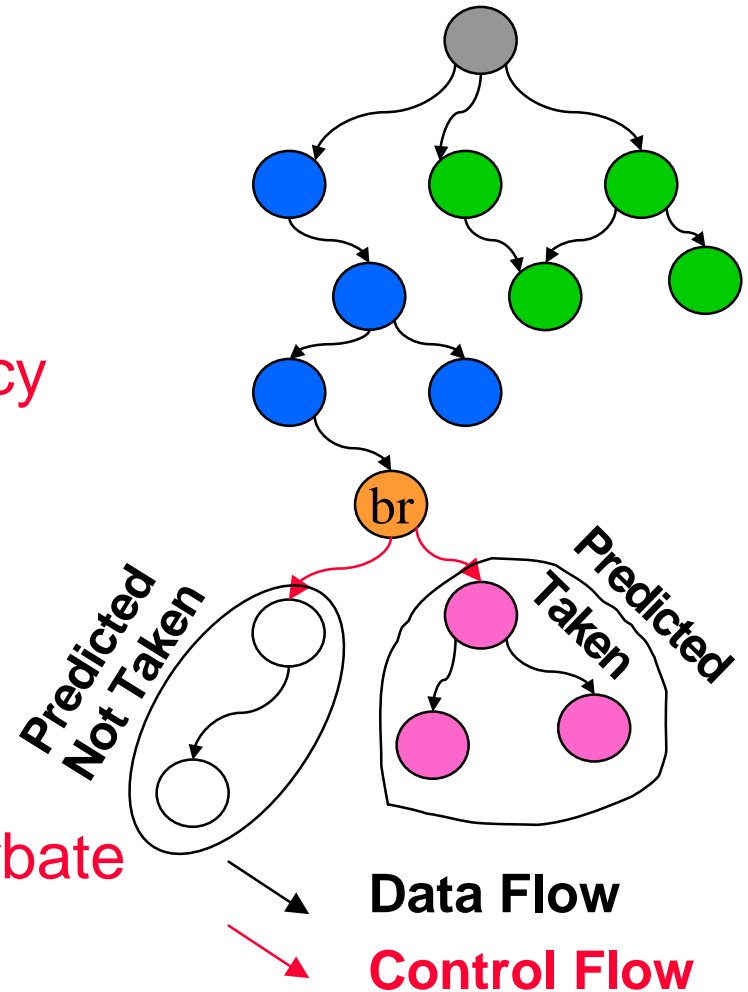
- Dynamic Scheduling
- Branch Prediction
- Speculative Execution

- Mechanisms help tolerate Latency

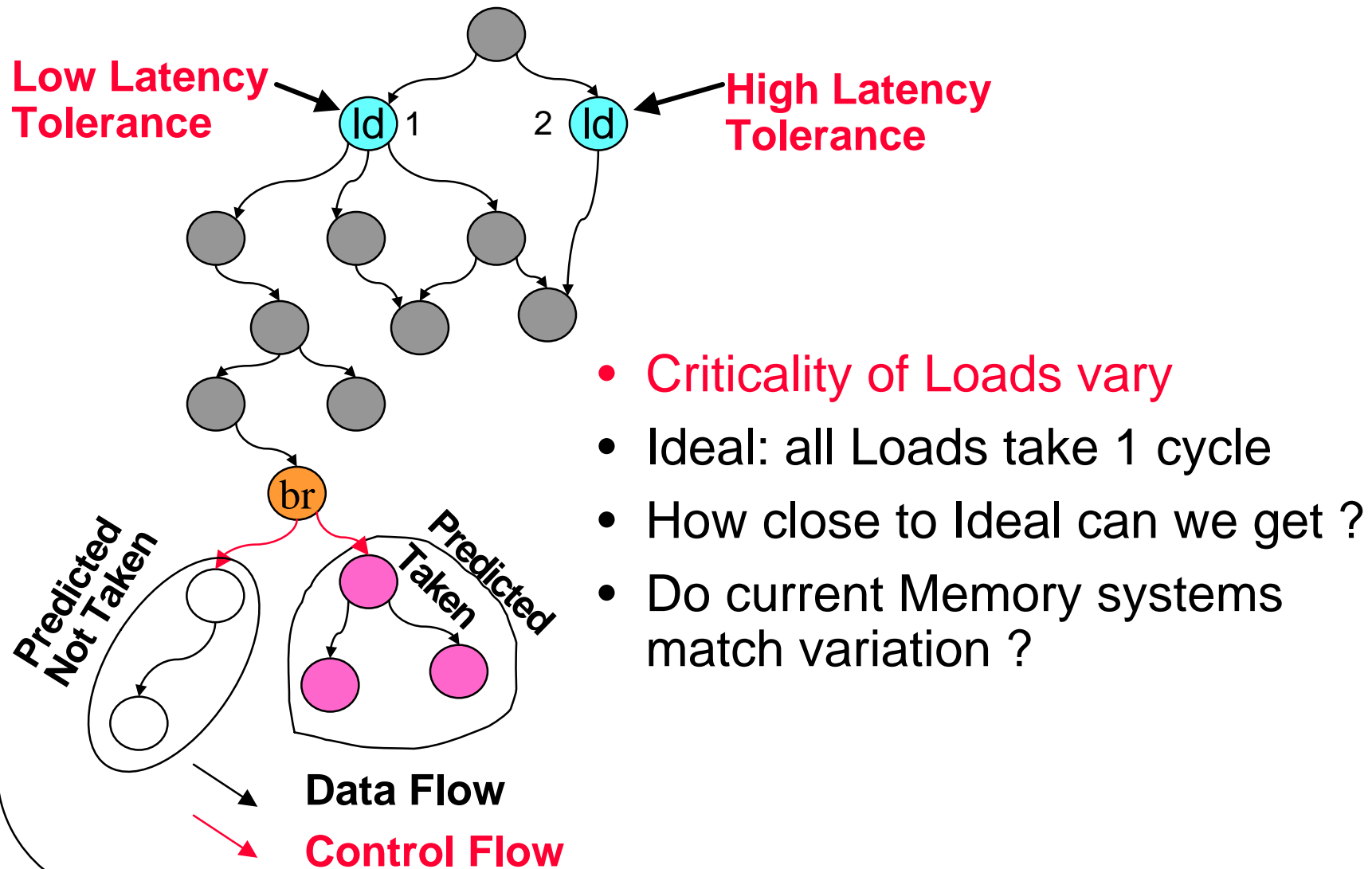
- Limitations

- Data Dependencies
- Finite Resources
- Branch Mispredictions

- Long Latency Operations exacerbate Limitations



Motivation



Outline

- Motivation
- Quantifying Load Latency Tolerance
- Experimental Setup
- Results
- Conclusion

Measuring Load Latency Tolerance

- Increase Latency
 - Uniform [Lizy Kurian, ISCA92]
 - » All or Nothing approach
 - Miss Penalties
 - » Cache Organization dependent

**No insight on individual
Load Latency Tolerance**

Our Approach

- How long can each Load wait to complete ?
- Remove Memory hierarchy
- Performance level controls Load completion
- Achieve IPC close to Ideal memory system
- Simulate processor with constrained resources

Simulation Methodology

- Delay Load completion
- Every cycle, our simulator asks
 - Should Loads complete?
 - Which Loads should complete?
 - When should Loads complete?
 - How many Loads should complete ?
- Latency Tolerance = Completion time - Issue time
- Goal: Maximize IPC and Load Latency Tolerance

Should Loads Complete?

Branch - based

- Minimize speculative execution
- Mispredicted Branches alone ?

Performance - based

- Free dependent instructions
- Release buffer space
- Metrics
 - Instruction Issue Rate
 - Functional Unit Utilization

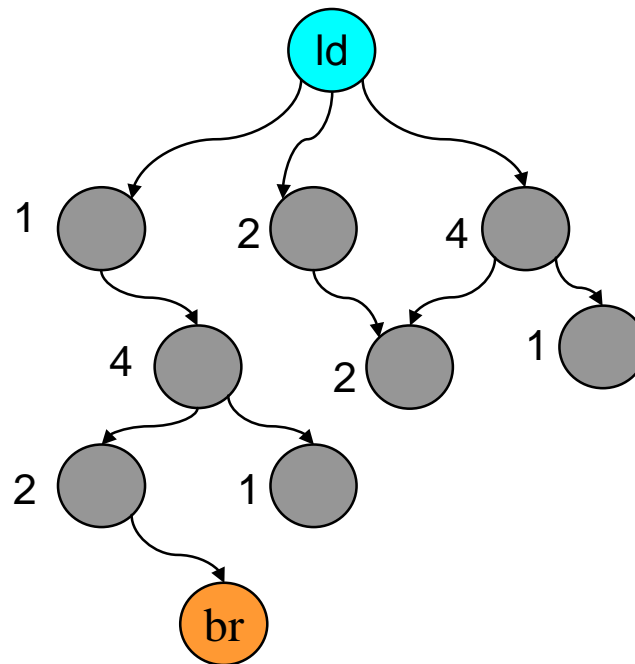
Limit - based

- 32 cycle limit

Branch-based Load Completion

- **Which ?** All Loads on which Branch is dependent
- **When ?** Avoid need for Branch prediction
 - Need for Rollback

Load completion time = $t - 8$



Branch execution time = t

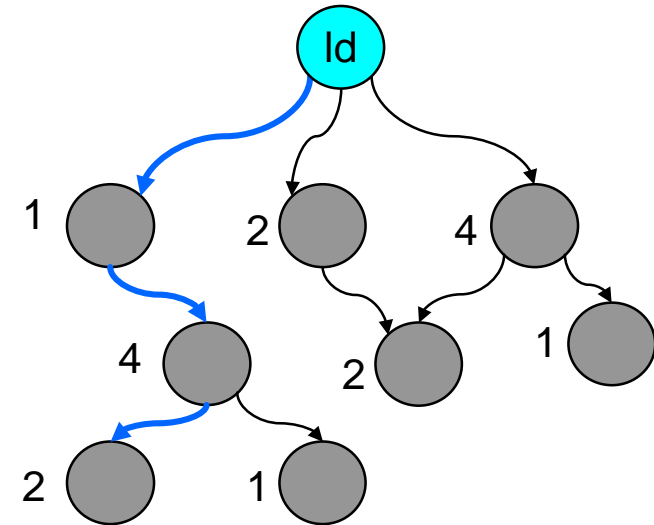
Performance-based Load Completion

- Which?

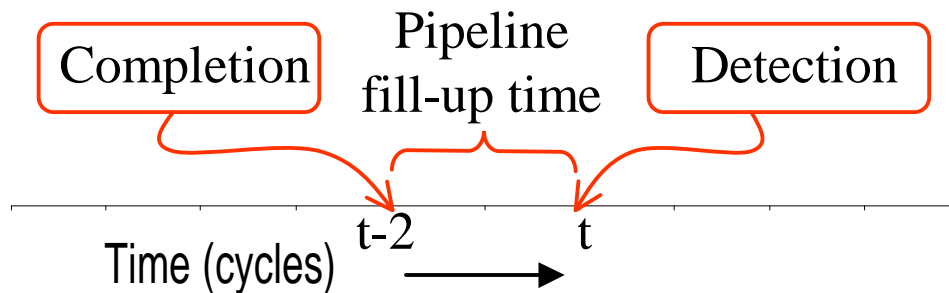
- Program order
- Dependence Graph Depth based

- When?

- Allow pipeline to fill-up with ready instructions



Dependence Graph Depth = 7



Outline

- Motivation
- Measuring Load Latency Tolerance
- **Experimental Setup**
- Results
- Conclusion

Experimental Setup

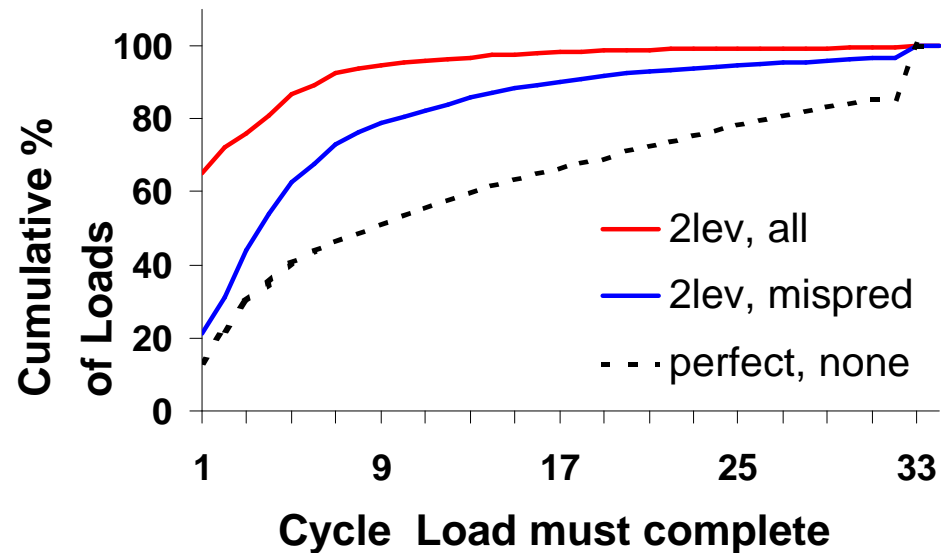
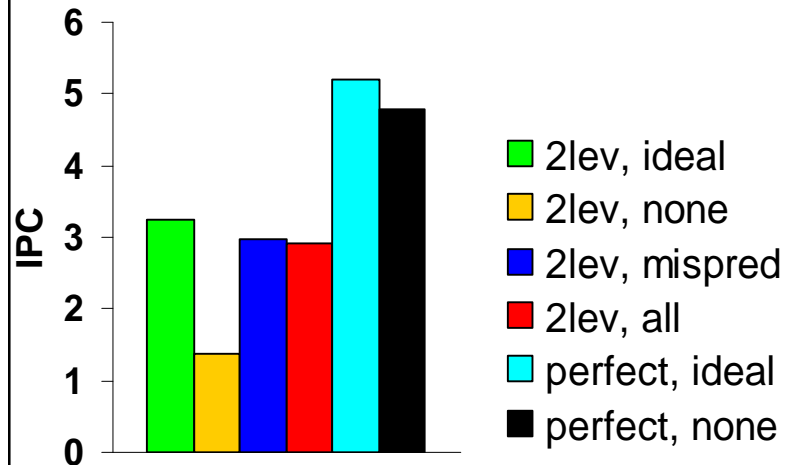
- SimpleScalar
- Checkpointing, Rollback and Sampling
- Benchmarks - Spec95
 - compress, gcc, li, **vortex**, hydro2d, **swim**, tomcatv, wave
- Reference data set
 - up to 10 billion instructions with 1% sampling
 - IPC values within 5% of complete simulations
- Baseline processor
 - 8 issue, out-of-order processor
 - 256 RUU entries, 128 LSQ entries
 - 2-level branch predictor with a total of 8192 entries

Goals Revisited

- **Maximize IPC and Latency Tolerance**
 - Evaluate Load Completion Parameters
- Variation in Load Latency Tolerance
- Computed Tolerance vs. Actual Latency

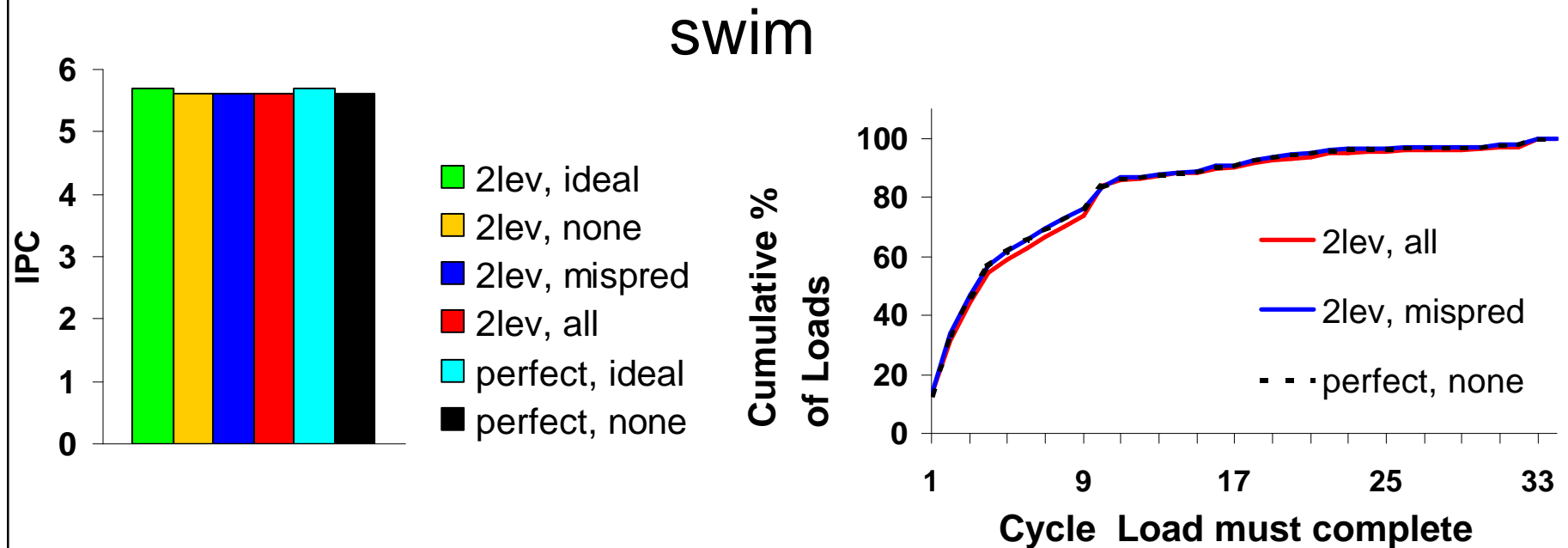
Branch-based Load Completion

vortex



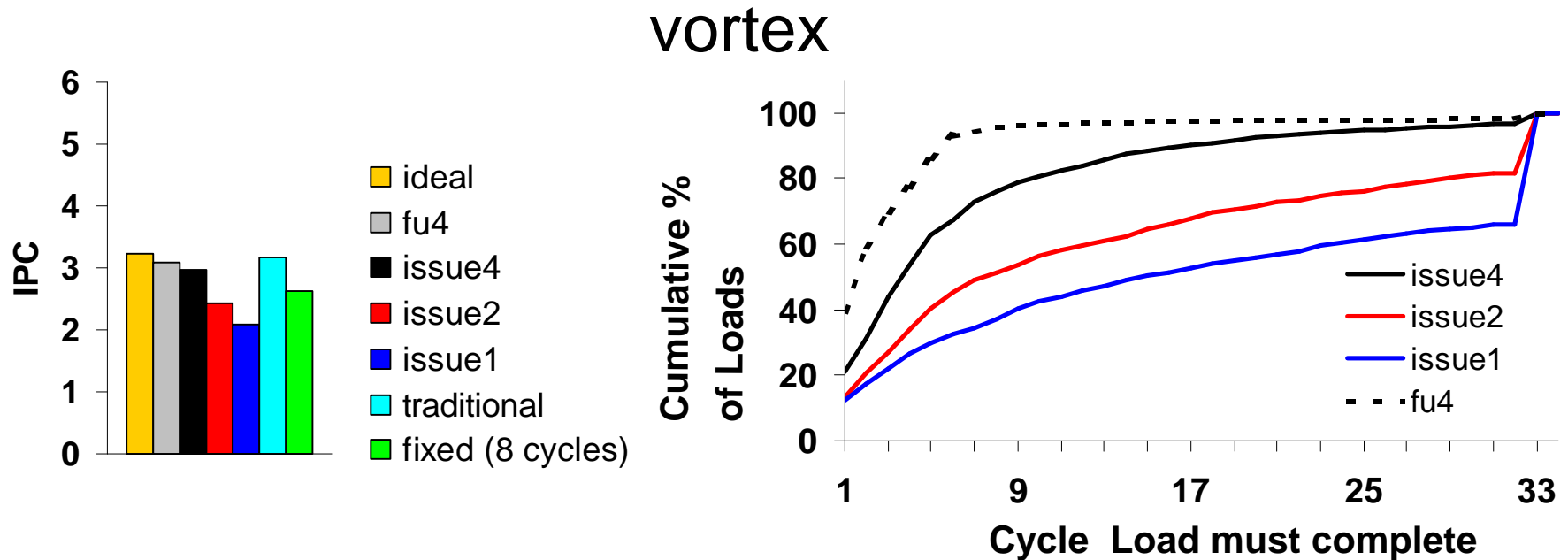
- Performance decreases if Branches not considered
- IPC: mispred == all
- Latency Tolerance: mispred >> all
- Latency Tolerance increases with prediction accuracy

Branch-based Load Completion



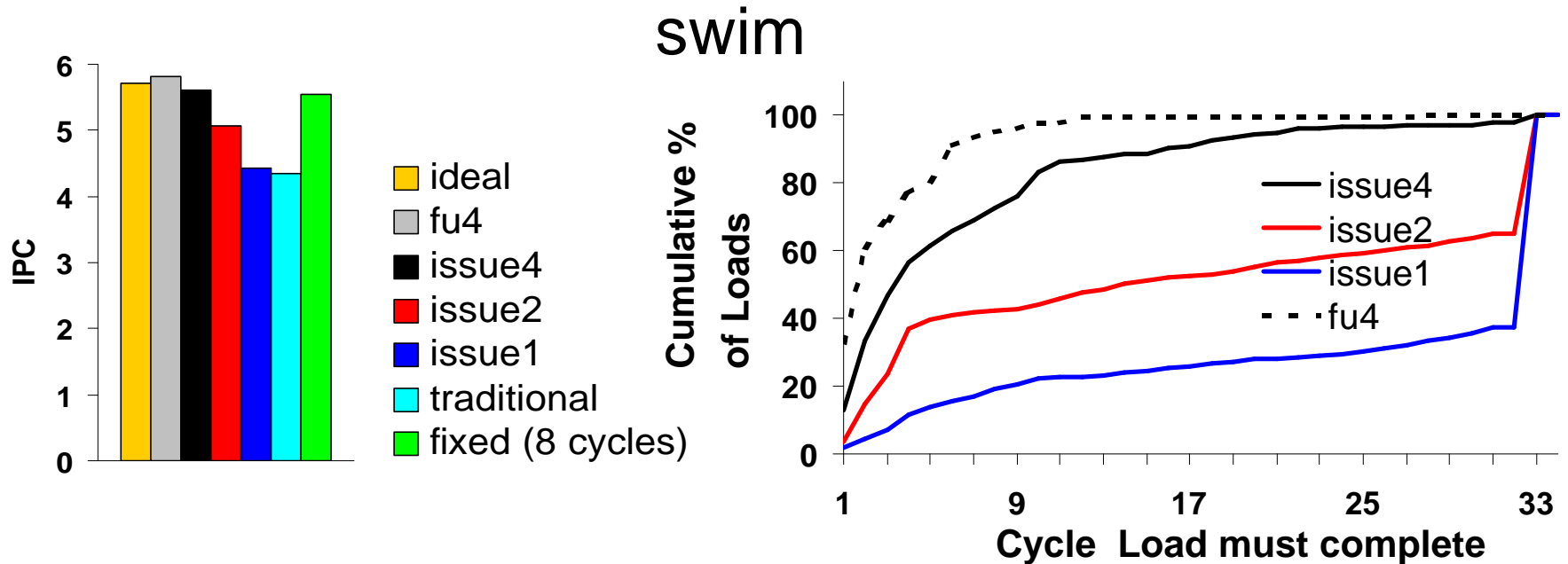
- Less than 2% of Loads completed due to Branches
- No effect on IPC or Latency Tolerance

Performance-based Load Completion



- IPC decreases with threshold
- Latency Tolerance increases as threshold decreases
- IPC: issue4 == fu4
- Latency Tolerance: issue4 >> fu4

Performance-based Load Completion

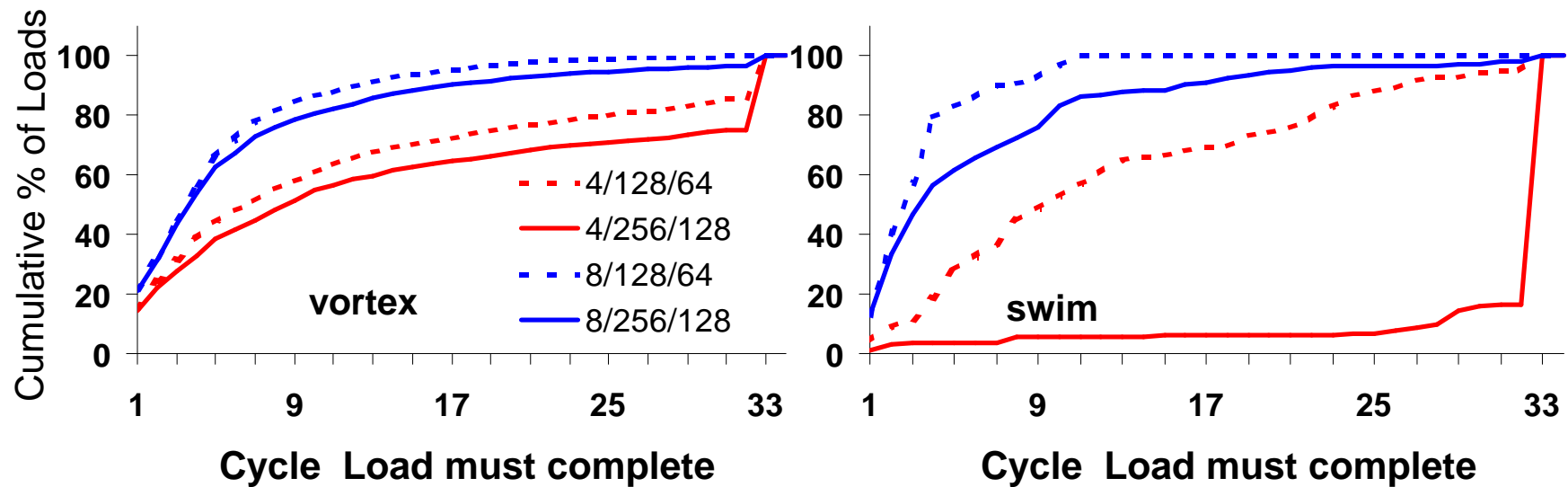


- IPC: issue4 == fu4
- Latency Tolerance: issue4 >> fu4

Load Completion Parameters - Summary

- Should Loads complete?
 - Loads with dependent mispredicted Branches
 - Issue Rate threshold of 4
- Which Loads should complete?
 - Dependence Graph based
 - Satisfying Loads in program order not always best
- When should Loads complete?
 - Pipeline fill-up time of 2 cycles
- How long can Loads wait?
 - 13% to 62% must complete in 1 cycle
 - 2% to 42% can wait for 8 cycles or more
 - IPC within 8% of ideal memory system

Processor Microarchitecture and Latency Tolerance



- IPC within 11% of ideal
- Decreasing issue-width or increasing buffer-space increases Load Latency Tolerance
- 4/256/128 swim: 84% Loads can complete in 32 cycles

Traditional Memory Hierarchies and Latency Tolerance

vortex

Measured Tolerance	Where Satisfied ?	
	L1 Cache	L2 Cache
Low	53%	3%
High	21%	1%

swim

Measured Tolerance	Where Satisfied ?	
	L1 Cache	L2 Cache
Low	45%	16%
High	23%	7%

- **Measured Tolerance does not match actual Latencies**
- Caches must differentiate between critical and non-critical Loads

Conclusion

- **Informed Caching Environment**
 - exploit information from executing program for better cache management
- **Annotated Memory References**
 - efficient mechanism for passing information from SW to memory hierarchy
- **Exploiting Locality**
 - improvements ~15%
- **Load Latency Tolerance**
 - Quantified an intuitive metric
 - Variation in latency tolerance exists
 - Current systems do not account for this variation
- **Future Work**
 - Use Latency Tolerance information in Cache replacement decisions
 - Give priority to critical Loads in Memory System Queues