Quantifying Instruction Criticality for Shared Memory Multiprocessors

Tong Li and Alvin R. Lebeck

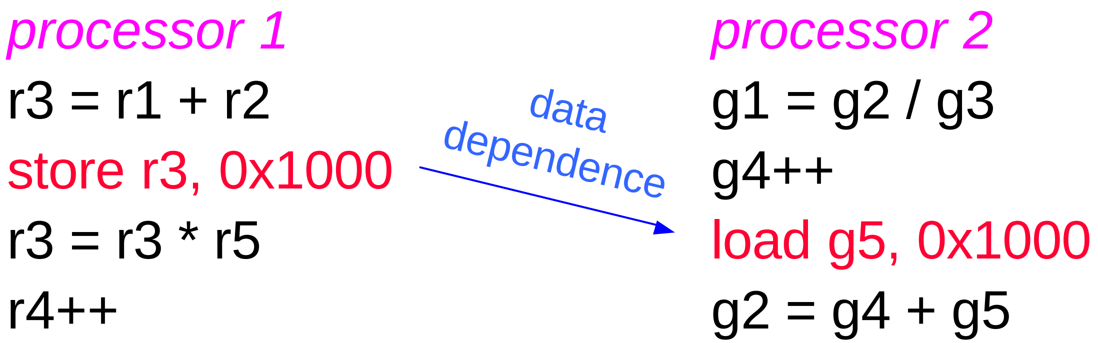
Department of Computer Science
Duke University
{tongli,alvy}@cs.duke.edu

Daniel J. Sorin

Department of Electrical and
Computer Engineering
Duke University
sorin@ee.duke.edu

Overview

- All instructions are NOT created equal
  - With respect to impact on performance → criticality
- Example (a 2-processor shared memory system):



processor 1	*processor 2*
r3 = r1 + r2	g1 = g2 / g3
store r3, 0x1000	g4++
r3 = r3 * r5	load g5, 0x1000
r4++	g2 = g4 + g5

- Contributions of this work
  - Create model for determining criticality in MP systems
  - Devise algorithm for computing criticality
  - Evaluate criticality of real MP workloads
- But why do we care about criticality?

Multiprocessor Control Policies

If the system knew instruction criticality dynamically, how could this be helpful?

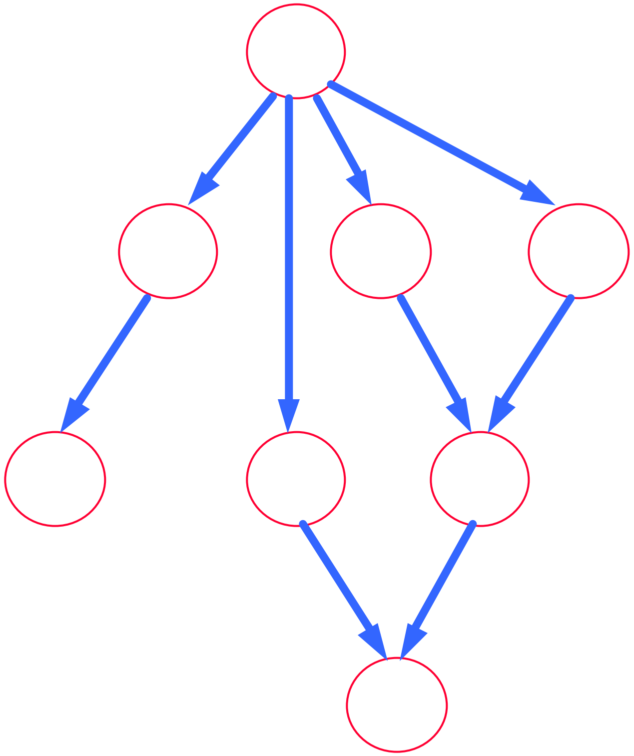
- Power efficiency
  - Less critical instructions can run more slowly
- Resource utilization
  - Critical-instruction-first resource allocations
- Misspeculation reduction
  - Turn off speculation for less critical instructions

Outline

- Motivation
- A directed acyclic graph (DAG) model for execution
  - Critical path and slack
  - Mapping DAGs to multiprocessor systems
  - Computing slack
- Graph Reduction
- Evaluation
- Related work
- Conclusions and future work

A DAG Model for Program Execution

- Node: dynamic event during execution (e.g., fetching an instruction, executing a task)
- Edge: dependence between source and sink nodes (e.g., data dependence)
  - Weighted by the time to resolve the dependence
- Critical path: longest weighted path in the DAG
  (CP length = runtime)



We study *spectrum of criticality*, not just on or not on the critical path

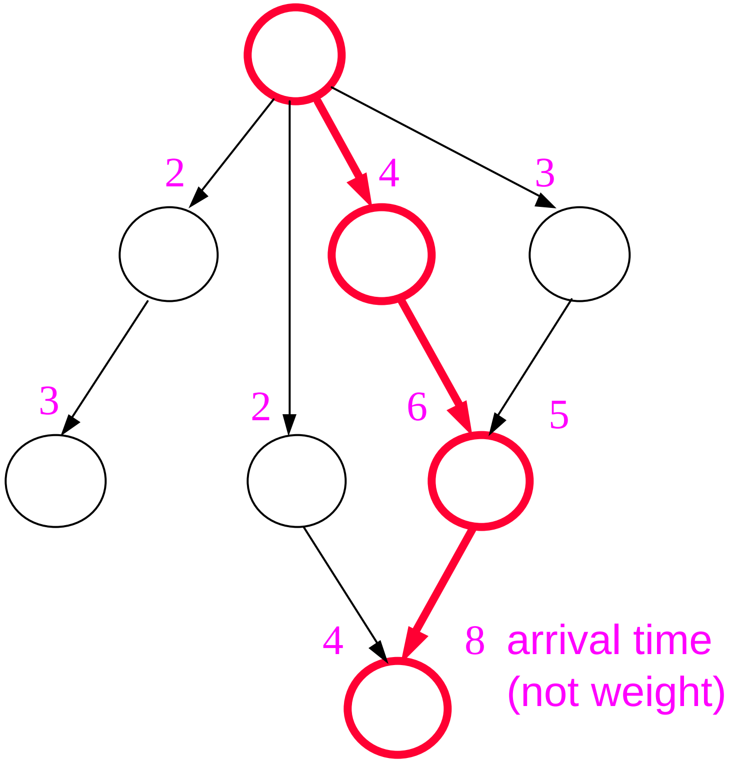
Criticality

- Criticality: importance level of event to overall performance

Fields et al. (ISCA ’02):

- Global slack: how long the start time of an event (node) can be delayed without affecting program runtime (criticality!)
- Edge arrival time: time *at which* the represented dependence is resolved during execution
- Last arriving edge: edge that arrives last at the sink node

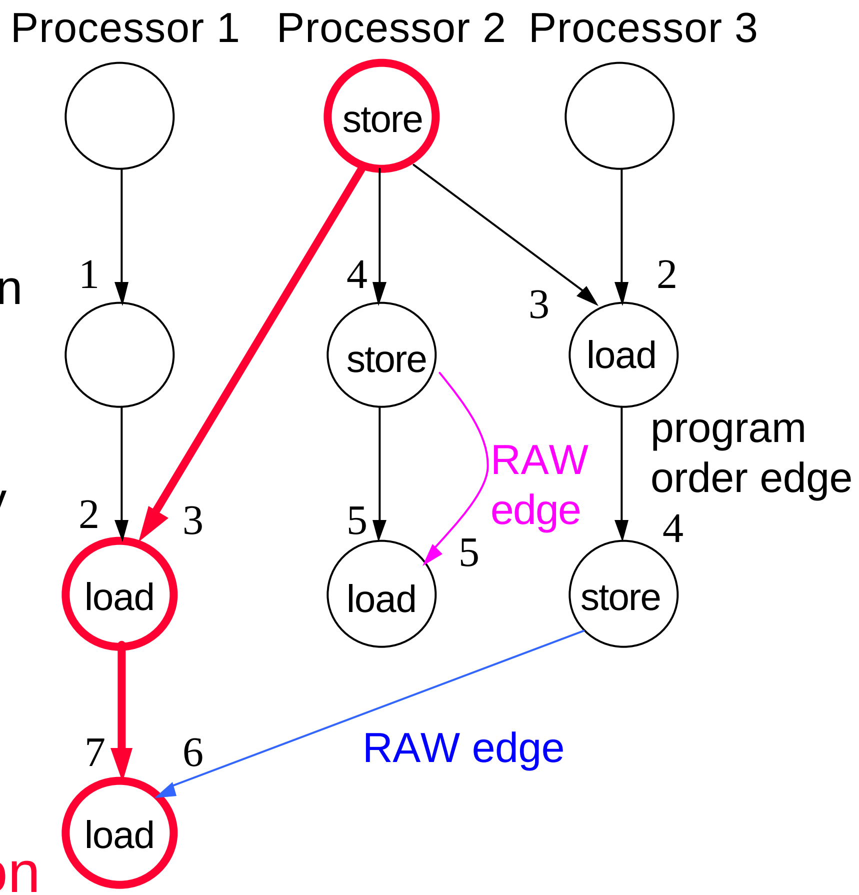
Previous work applies criticality to uniprocessors. We extend it to multiprocessors



An edge on a critical path must be a last-arriving edge; A non-last-arriving edge must not be on a critical path

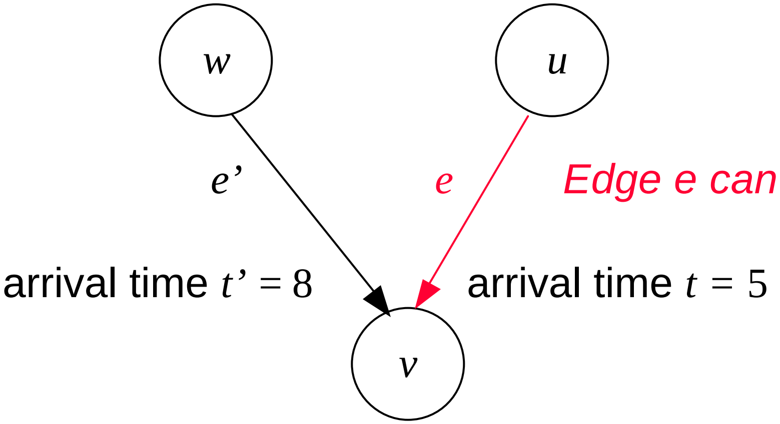
Multiprocessor Criticality

- Extension of uniprocessor DAG model (Fields et al. ISCA'01, ISCA'02)
- In-order processors
  - Each node represents an instruction
- Shared memory system
  - Processors communicate only via loads and stores to shared memory
- Two types of dependence (edges)
  - *Program order*
  - *Read-after-write* (*RAW*)
- Global slack quantifies instruction criticality, but how to compute it?



Local Slack: A Tool for Global Slack

- The *local slack* of an edge $e = (u, v)$, denoted by $L(e)$, is the time that the latency of e can be increased without delaying its sink node v. (Fields et al. ISCA 2002)
- Properties
  - If an edge is not last-arriving, then it can be delayed
  - If an edge is last-arriving, then it cannot be delayed



Edge e can be delayed for 3 time units

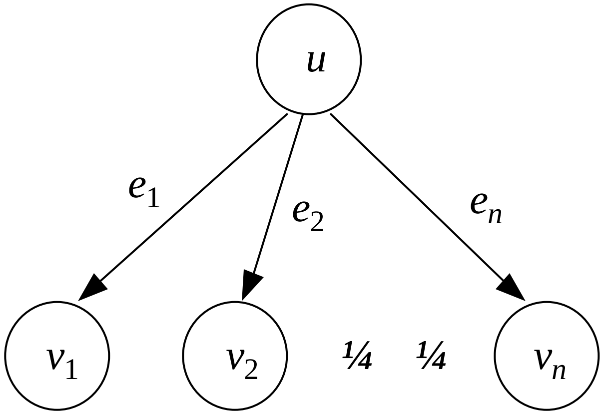
$$L(e) = \max(t, t') - t$$

$$L(e') = \max(t, t') - t'$$

Based on local slack, we can compute global slack

Computing Global Slack

- The *global slack* of a node u, denoted by $G(u)$, is the maximum time u can be delayed without extending the critical path of the DAG (Fields et al. ISCA 2002)
- An instruction's global slack quantifies its criticality
- A node's global slack depends on local slack of its outgoing edges and global slack of its children
- To compute global slack for all nodes, we need to process the entire DAG



$$G(u) = \min_i \left(L(e_i) + G(v_i) \right)$$

Outline

- Motivation
- A directed acyclic graph (DAG) model for execution
- Graph Reduction
- Evaluation
- Related work
- Conclusions and future work

Graph Reduction

- We compute global slack offline, but processing DAGs requires large amounts of storage and time
  - Programs have billions of instructions
- We propose graph reduction to reduce DAGs
- Graph reduction dynamically removes DAG nodes and edges that don’t change the critical path and global slack of all nodes
- Three theorems describe when a reduction can be performed dynamically during a program’s execution
  - Details of theorems and proofs are in the paper

Graph Reduction – Theorem 1

Program situation: Many instructions are neither loads nor stores. We can remove all of them!

If

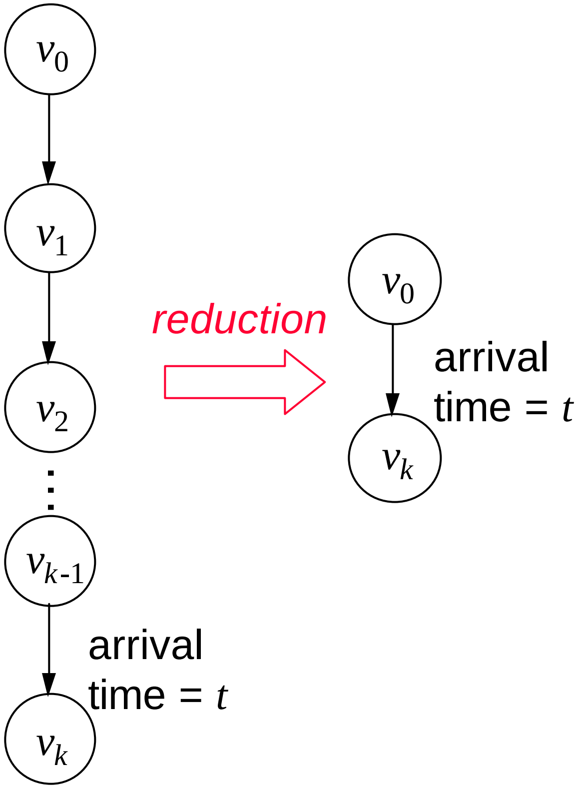
- $v_0, \ldots, v_k$ are on the same processor
- $v_1, \ldots, v_{k-1}$ are neither loads nor stores

Then

- The DAG can be reduced by removing $v_1, \ldots, v_{k-1}$ and retaining arrival time t

Why ?

- $G(v_1) = G(v_2) = \ldots = G(v_{k-1}) = G(v_k)$
- If $v_1, \ldots, v_{k-1}$ are on the critical path, then v_0 and v_k must be on the critical path of the reduced DAG



Graph Reduction – Theorem 2

Program situation: A sequence of loads on the same processor read the same value written by a store. We could remove all these RAW edges except the first one!

If

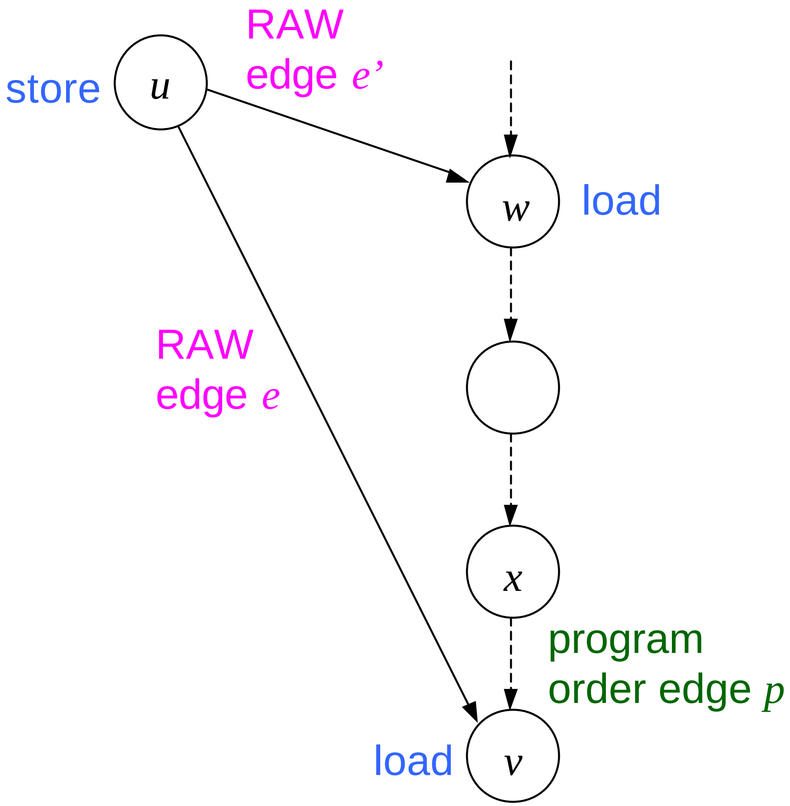
- Arrival time of e is less than arrival time of p
- No node between w and v is the sink of a RAW edge that is last-arriving at the node

Then

- RAW edge e can be removed

Why ?

- e must not be on the critical path
- e does not contribute to computing $G(u)$ and $G(x)$



Graph Reduction – Theorem 3

Program situation: A load reads a value written by a store on the same processor. We could remove this RAW edge!

If

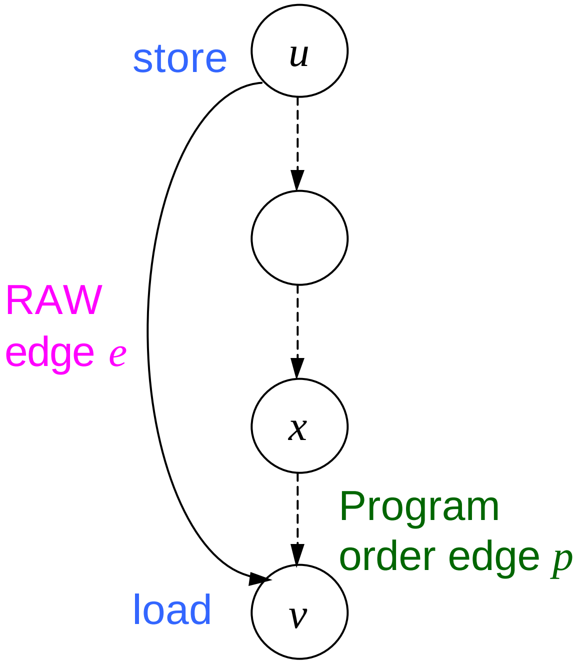
- Arrival time of e is less than arrival time of p
- No node between u and v is the sink of a RAW edge that is last-arriving at the node

Then

- RAW edge e can be removed

Why ?

- e must not be on the critical path
- e does not contribute to computing $G(u)$ and $G(x)$



Outline

- Motivation
- A directed acyclic graph (DAG) model for execution
- Graph Reduction
- Evaluation
  - Methodology
  - Results
- Related work
- Conclusions and future work

Experiments

- Do instructions really have global slack? How much?
- How critical is an entire processor in a program’s execution?
- How do different cache coherence protocols affect global slack of instructions?
- How effective is graph reduction?

Methodology – Simulator

- Simics
  - Full-system multiprocessor simulator
  - Functional simulator, can boot unmodified Solaris 8
  - A detailed memory hierarchy timing module
- Processor model
  - In-order processor core
  - Blocking cache requests
- Memory model
  - MOSI broadcast snooping cache coherence protocol
  - Sequential consistency

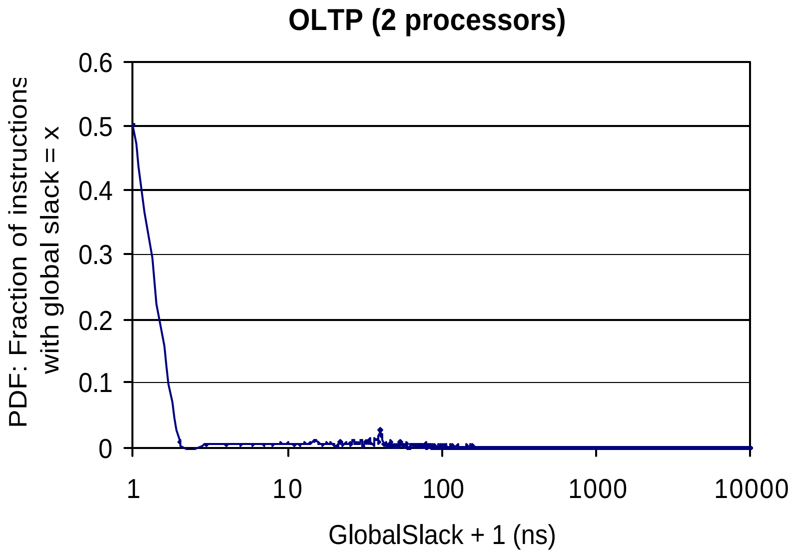
Methodology – Workloads

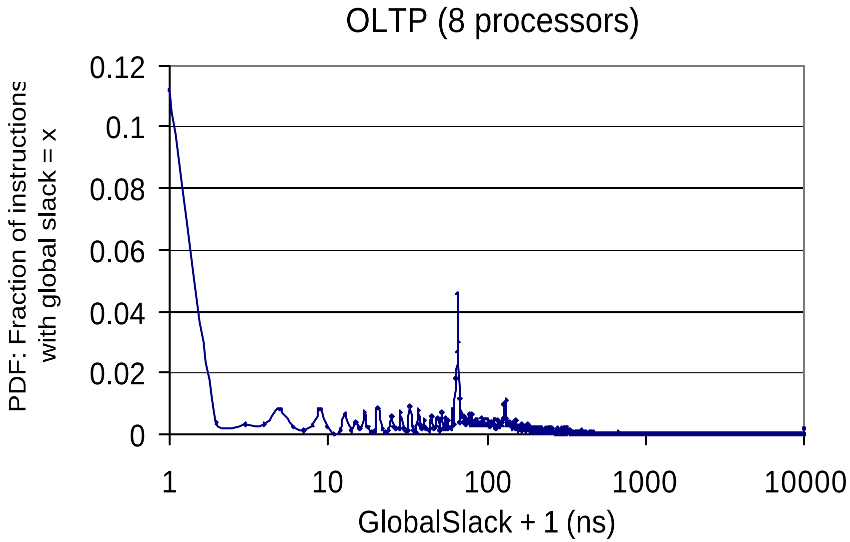
- Commercial workloads (Wisconsin suite)
  - *OLTP*: online transaction processing
  - *Java server*: SPECjbb2000 server-side java benchmark
  - *Static web server*: web server with static content
  - *Dynamic web server*: web server with dynamic content
- Scientific workloads (Stanford SPLASH-2)
  - *Barnes-Hut*: simulates the interactions of a system of bodies using the Barnes-Hut hierarchical N-body method
  - *Ocean*: simulates ocean movements using Gauss-Seidel multi-grid equation solver

Methodology – Data Acquisition and Analysis

- Warm up simulated system for each workload
- Log dependences (edges) into files during execution
- Dynamically apply graph reduction during execution
- Construct DAG from log files
- Offline compute global slack for each instruction

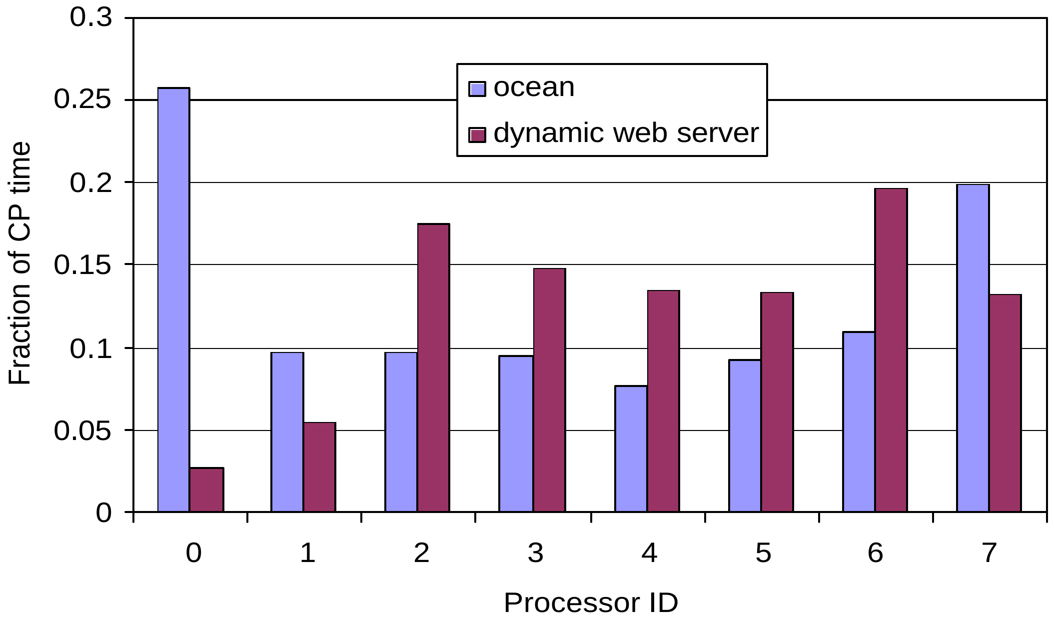
How Much Global Slack Exists?





- x-axis: global slack plus one in log scale
- y-axis: fraction of instructions that have global slack x
- Most instructions have global slack < 100 ns
- Spikes between 100 and 200 ns correspond to inter-processor communication latency
- Other workloads have similar results

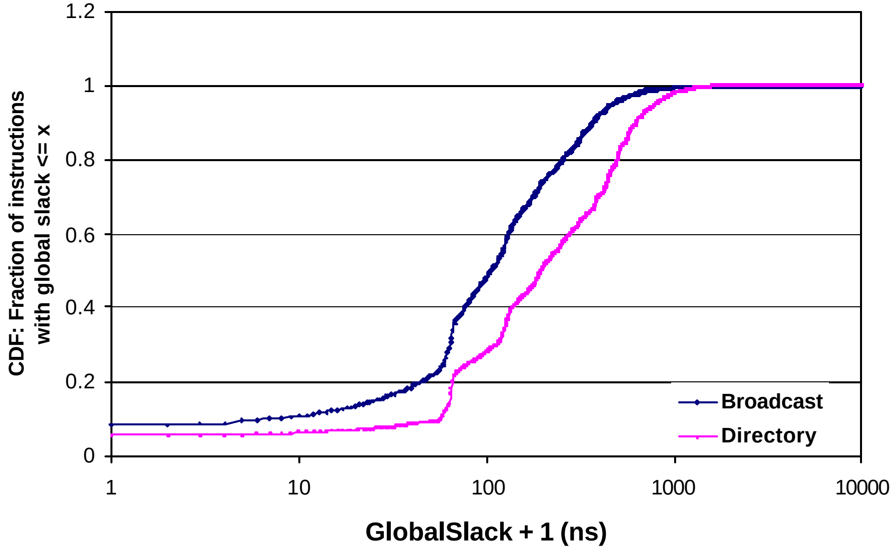
Insight into Processor Criticality



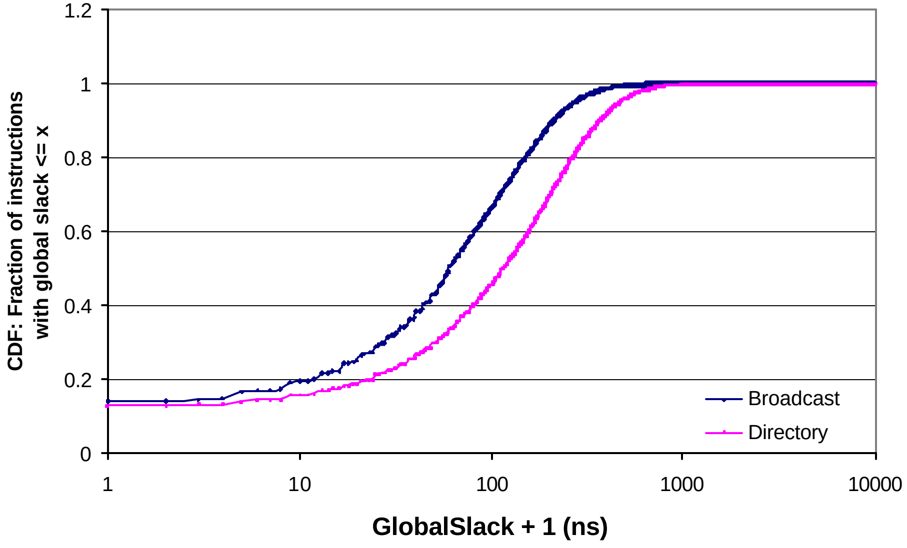
- x-axis: each processor in an 8-processor system
- y-axis: fraction of critical path's time spent on processor x
- Critical path time breakdowns closely correspond with processor L2 cache miss rates
- Other workloads have critical path evenly distributed

DUKE Systems & Architecture

Broadcast vs. Directory Protocols



Static Web Server

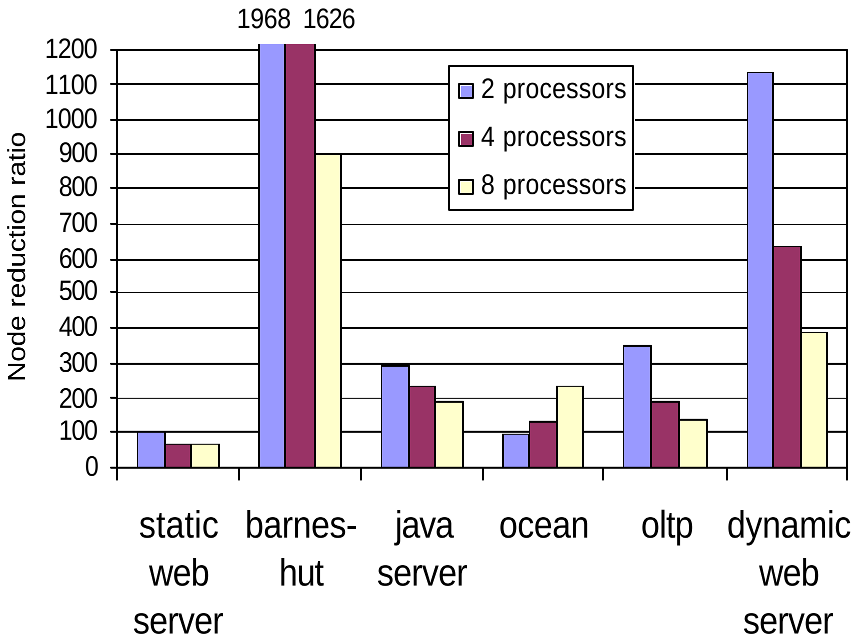


Java Server

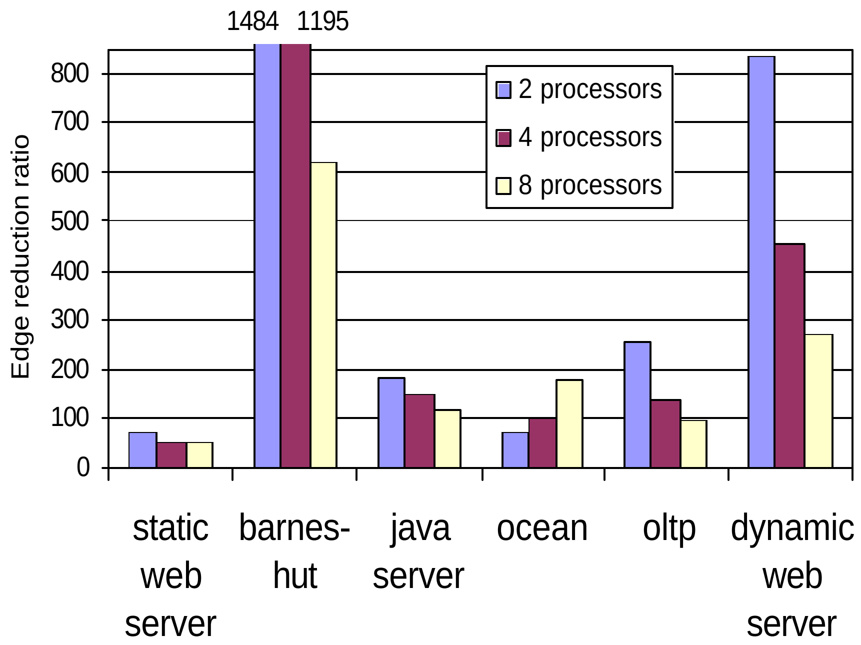
- x-axis: global slack plus one in log scale
- y-axis: fraction of instructions that have global slack = x
- More global slack in directory system
- Directory protocol has higher L2 miss latency because of indirections
- Other workloads have similar results

Effectiveness of Graph Reduction

Node reduction ratios



Edge reduction ratios



- Reduction ratios range from 66 to 1968
- Average node reduction ratio 485, edge ratio 352
- Maximum node reduction ratio 1968, edge ratio 1484

Experiments

- Do instructions really have global slack? How much?
  - Most have global slack < 100 ns, some spikes between 100 and 200 ns
- How critical is an entire processor in a program’s execution?
  - A processor’s time on critical path closely corresponds with its L2 cache miss rates
- How do different cache coherence protocols affect global slack of instructions?
  - Directory protocol has more global slack
- How effective is graph reduction?
  - Reduction ratios range from 66 to 1968

Related Work

- Uniprocessor DAG model and critical path and slack analysis (Fields 2001, 2002)
- Critical path and slack analysis at the procedure level or above for performance bottlenecks (Hollingsworth 1994, 1998, and Yang 1998)
- Multiprocessor scheduling
- DAG reduction (Beckmann 1994, Netzer 1993)

Conclusions and Future Work

- We can construct a DAG model for multiprocessor slack
- We can determine criticality by computing global slack in the DAG model
- Experiments show global slack exists and graph reduction effectively reduces DAG size
- Future research will study online algorithms for predicting global slack and design criticality-based processor control policies