Focusing Processor Policies via Critical-Path Prediction

ISCA 2001

Brian Fields* Shai Rubin* Rastislav Bodík*

*University of Wisconsin-Madison

Presented by

Jan Mantsch

Executive Summary

Motivation:

- Egalitarian scheduling policies in processors waste resources
- Increasing parallelism and sophistication in processors justify critical path analysis

Key Idea:

- Predict whether an instruction is on the critical path in hardware while keeping cost for graph model as low as possible

Challenges:

- Compile-time optimizations only consider data dependences
- Processor only ever sees fraction of program
  - -> How to optimize for global critical path?

Key Mechanism:

- Token-passing algorithm to estimate criticality of nodes

Results:

- Better scheduling improves CPU performance up to 21%
- Optimized prediction improves CPU performance up to 5%

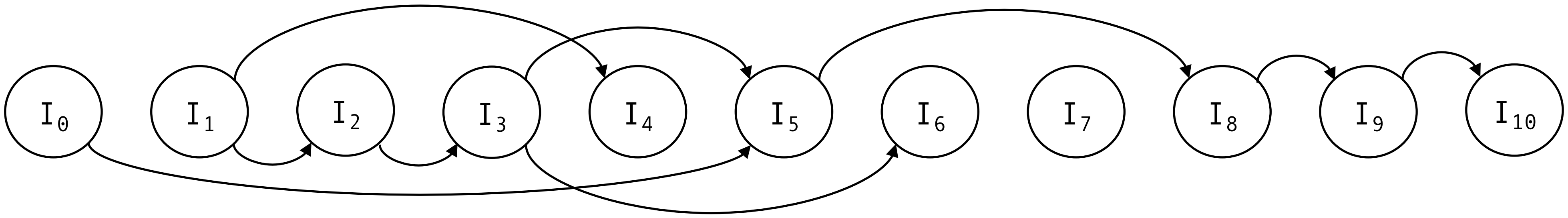
Overview

- The Model of the Critical Path
- Predicting the Critical Path in Hardware
- Applications of the Critical Path Detection
- Conclusion
- Paper Analysis
- Discussion

Compiler Model of Dependences

```
   I0:   r5 = 0
   I1:   r3 = ld[r2]
L1 I2:   r1 = r3*6
   I3:   r6 = ld[r1]
   I4:   r3 = r3+1
   I5:   r5 = r6+r5
   I6:   cmp R6,0
   I7:   br L1
   I8:   r5 = r5+100
   I9:   r0 = r5/3
   I10:  ret r0
```

Compiler optimizes execution by analysing data dependences



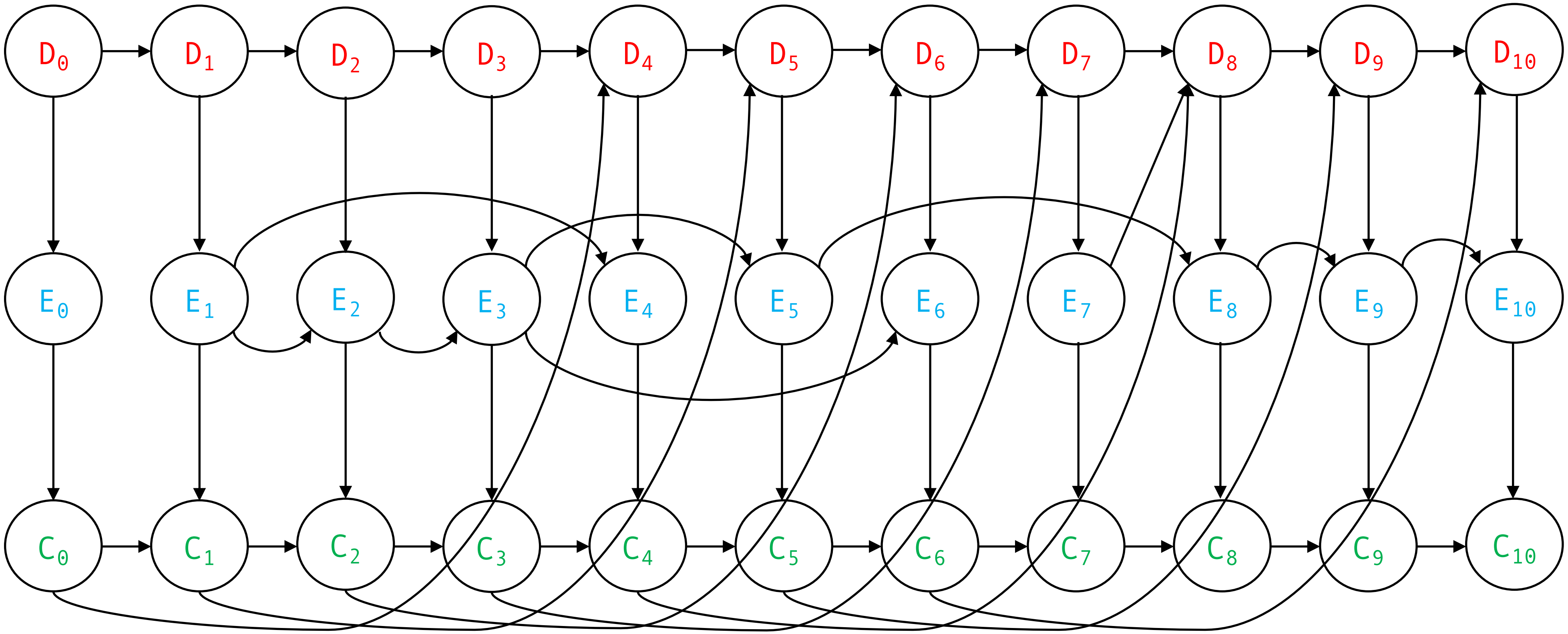
Weaknesses of Compiler Based Approaches

Compiler models critical path solely based on data-dependece

-> Other dependences and hardware constraints are not considered, i.e:

- Control dependences
- In-Order dependences
- Re-order buffer limitations
- ...

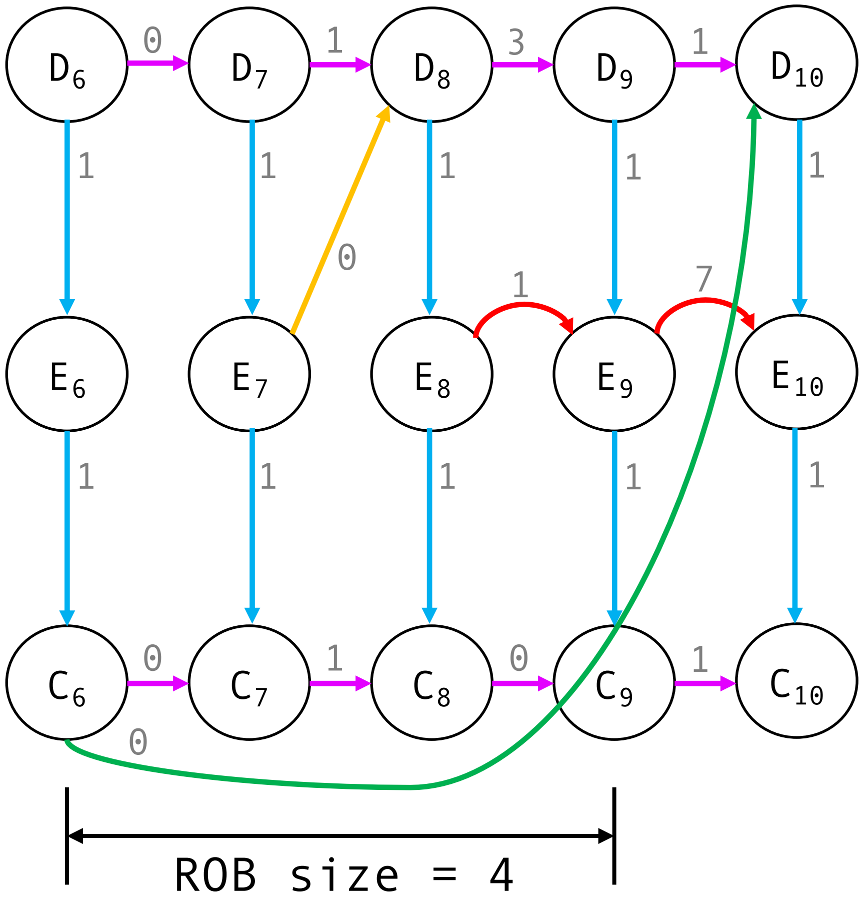
Model of the Critical Path



Idea: Model each dynamic instruction with 3 nodes

Dispatch Node	D_i
Execute Node	E_i
Commit Node	C_i

Classification of Edges



DE: Execution follows dispatch

EC: Commit follows execution

DD: In-order dispatch

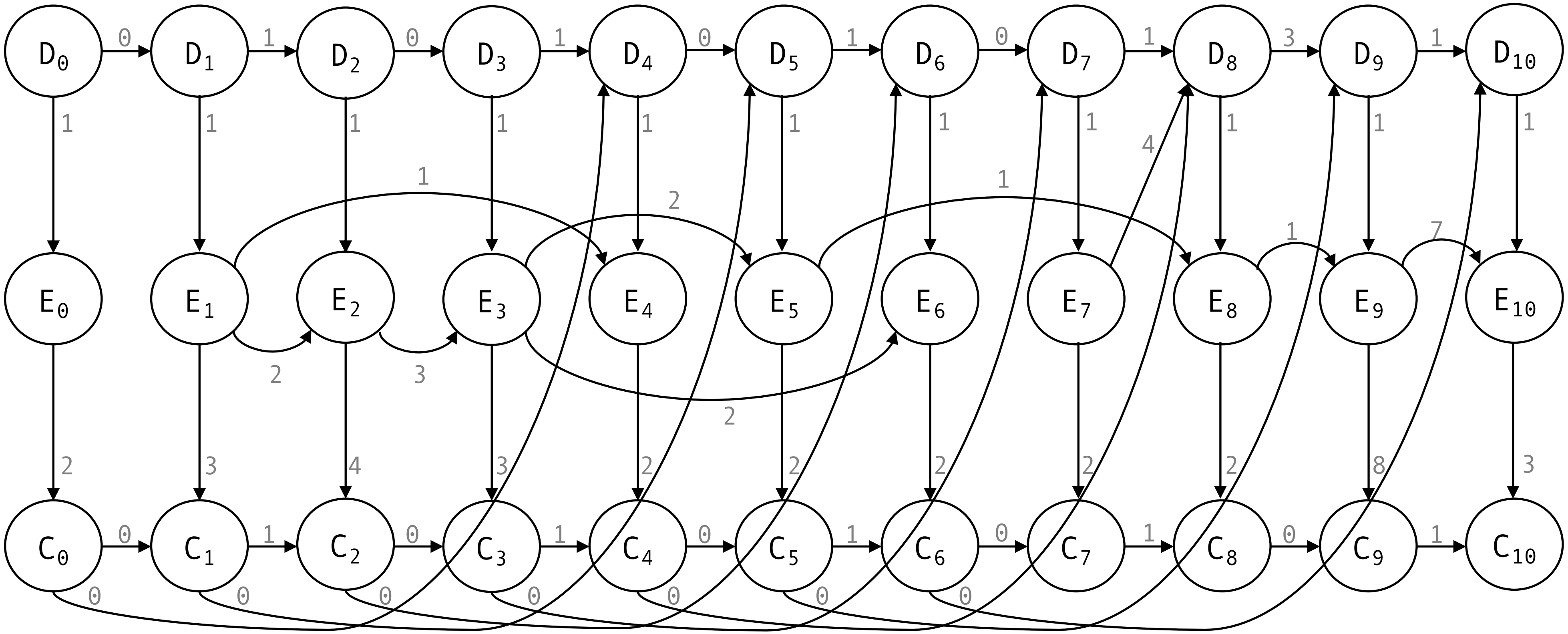
CC: In-order commit

EE: Data dependences

CD: Finite re-order buffer

ED: Control dependence

Critical Instructions



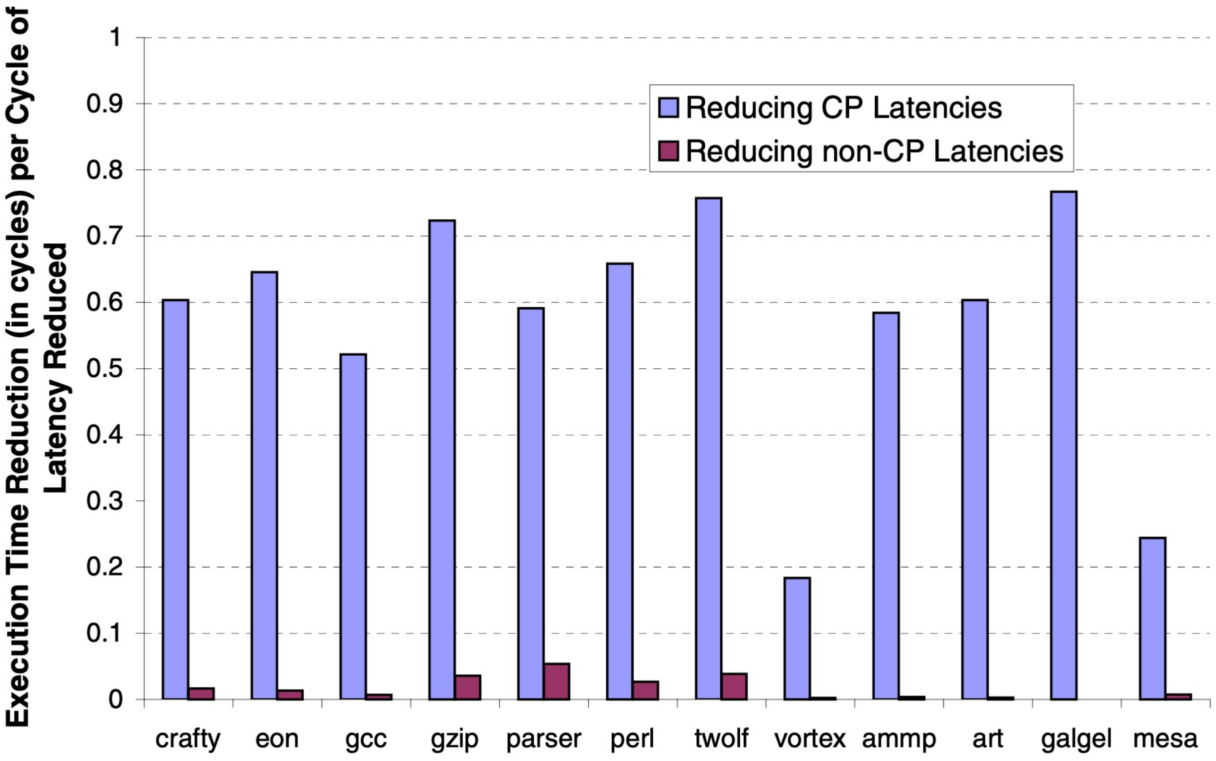
-> Critical Instructions: I_0, $I_{1,}$ I_2, I_3, I_7, I_8, I_9, I_{10}

Validation of Critical Path Model: Methodology

1. Run simulation with benchmark workloads as baseline.
2. Build critical path graph from baseline run.
3. Run two comparison simulations:
   1. All critical path latencies decreased by 1
   2. All non-critical path latencies decreased by 1

-> Idea: If latencies on critical path are reduced, overall execution time must be reduced too

Validation of Critical Path Model: Results

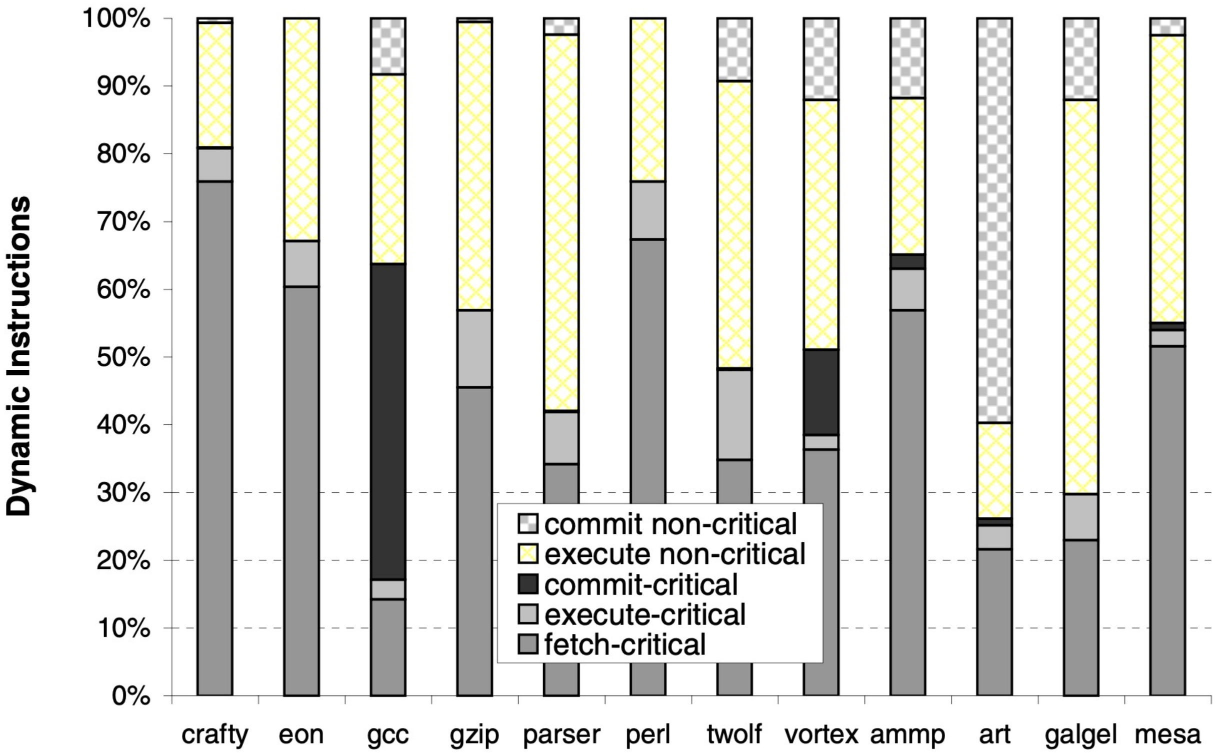


(a) Validation of the critical Path

- Execution time reduction for reduced CP latencies suggests model is good at identifying critical instructions

- Nice insight: Reduction ratio can be used as measure of critical path dominance

Validation of Critical Path Model: Results



(b) Breakdown of the dynamic instruction count

- Only 26-80% of instructions are critical

- More specifically, only 2-13% of instructions are execute critical

Overview

- The Model of the Critical Path
- Predicting the Critical Path in Hardware
- Applications of the Critical Path Detection
- Conclusion
- Paper Analysis
- Discussion

The *Last-Arriving* Rules

Observation: Critical Path can be computed solely by observing the arrival order of instruction operands

-> If a dependence between nodes i and j (i<j) is the last to be resolved for node j, the according edge from i to j is called *last-arriving*

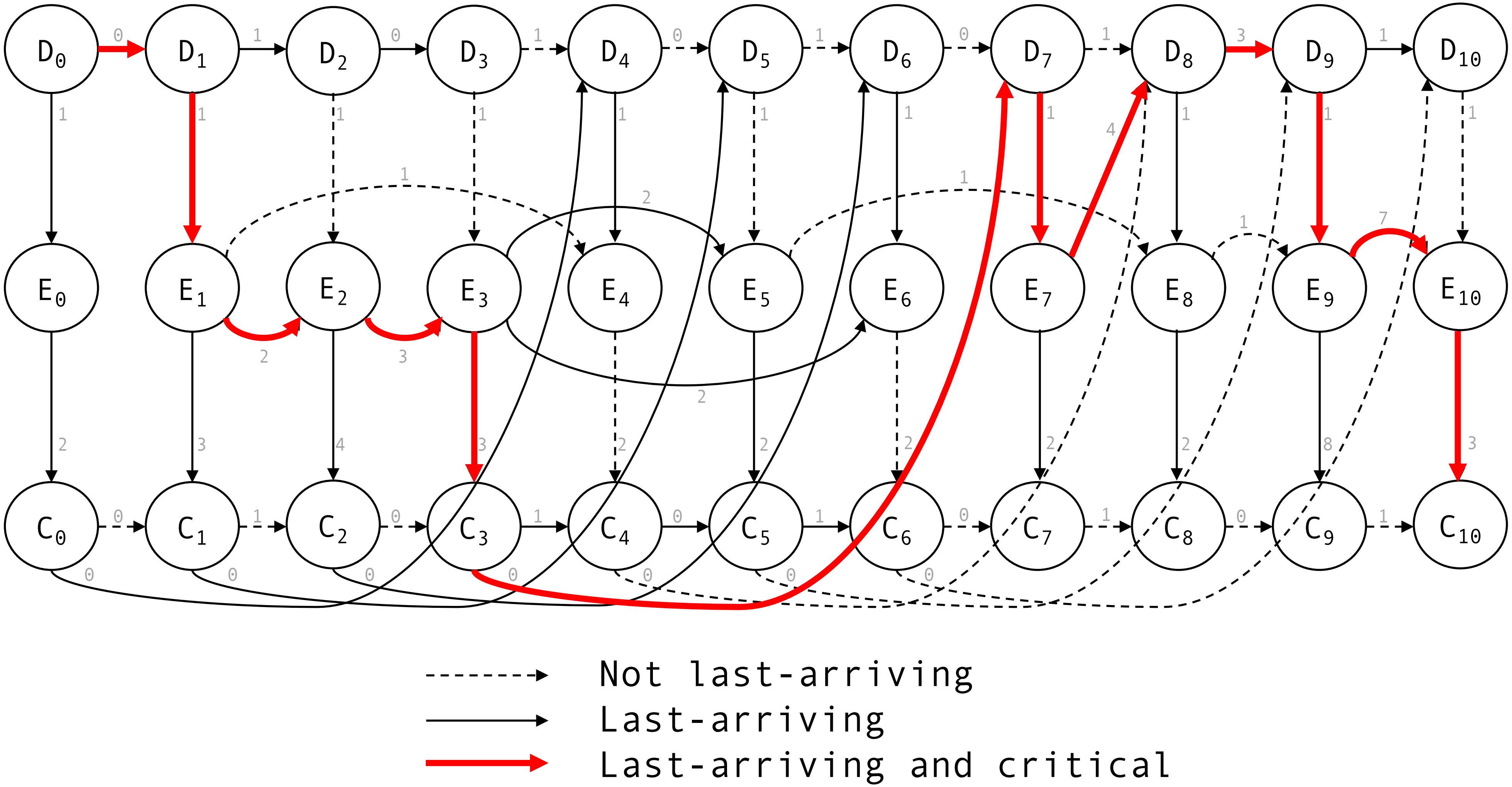
-> Each edge on the critical path is last-arriving edge

-> If an edge is not last-arriving, it is not critical

Building the Critical Path (in Simulation)

- Start at commit node of last instruction
- Traverse graph backwards along last-arriving edges
- Done when arrived at dispatch node of first instruction -> Critical path complete!

Critical Path Model with Last-Arriving Edges



⇢ Not last-arriving

→ Last-arriving

➡ Last-arriving and critical

Building the Critical Path

This approach works for simulations, but is too expensive to implement in hardware

- We would have to save almost the entire graph
- Backwards traversal not trivial

Key Mechanism: The Token-Passing Critical Path Predictor

Approximate CP with following intuiton:

Critical path is chain of last arriving edges through entire graph -> long last-arriving chain is *likely* to be part of critical path

Such a chain can be found with *forward* propagation of tokens

-> Requires no graph building at all!

Token-Passing Algorithm

1. Plant token at node n

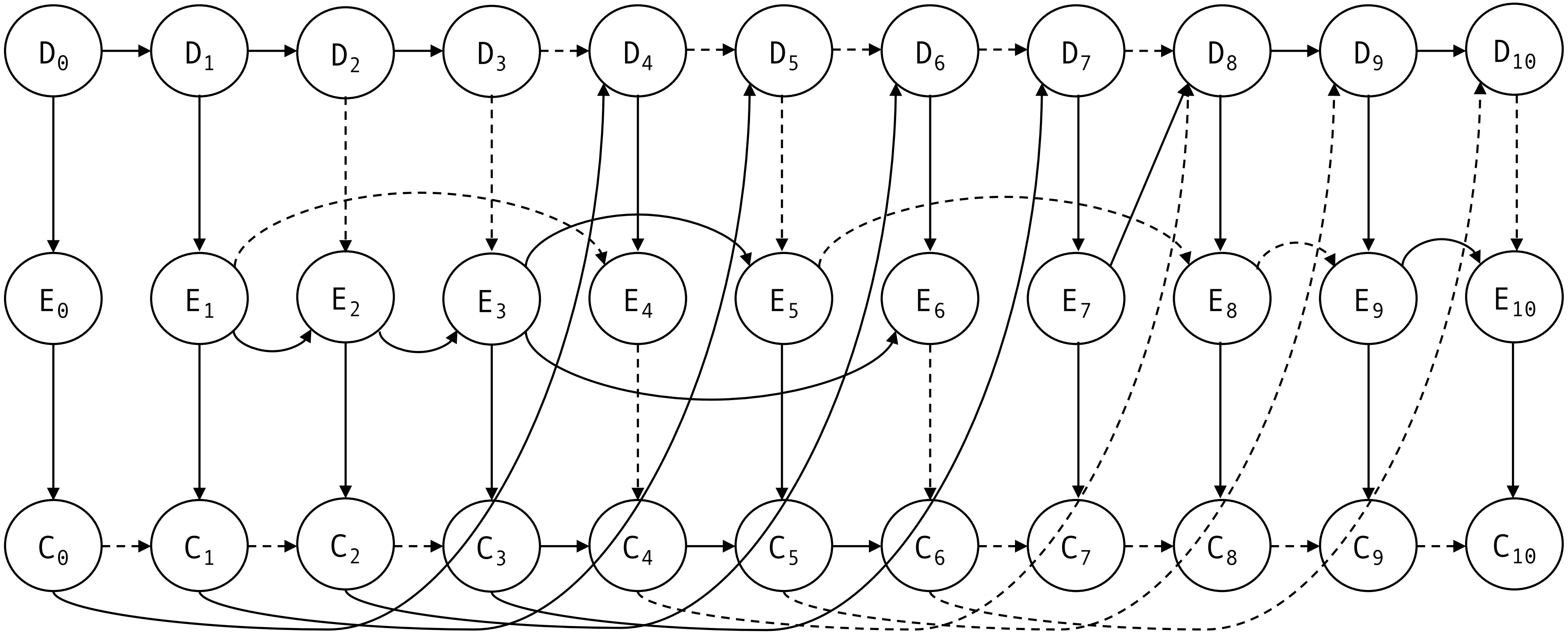
2. Propagate token forward along last—arriving edges.

   -> If a node doesn't have an outgoing last-arriving edge, the token dies.

3. After allowing token to propagate for some time, check if token is still alive

4. If token is alive, train node as critical; otherwise, train n as non-critical

Token-Passing Algorithm: Visualization



Prediction: C_3 is critical

Hardware Implementation: Specs

Critical path table is conventional array indexed by the PC of the instruction

Trainer is implemented as a small token array. It stores information about the [ROB_size] most recent instructions committed

-> No critical path dependence can span more than [ROB_size] entries

Hardware Implementation: Training Parameters

Critical path prediction table: 12 kilobytes (16K entries * 6 bit hysteresis)

Token propagation Distance: 1012 Dynamic instructions (500 + ROB size)

Maximum #tokens in flight: 8

Hysteresis: Saturate at 63, increment by 8 when training critical, decrement by 1 when training non-critical. Instruction is predicted critical if hysteresis is above 8.

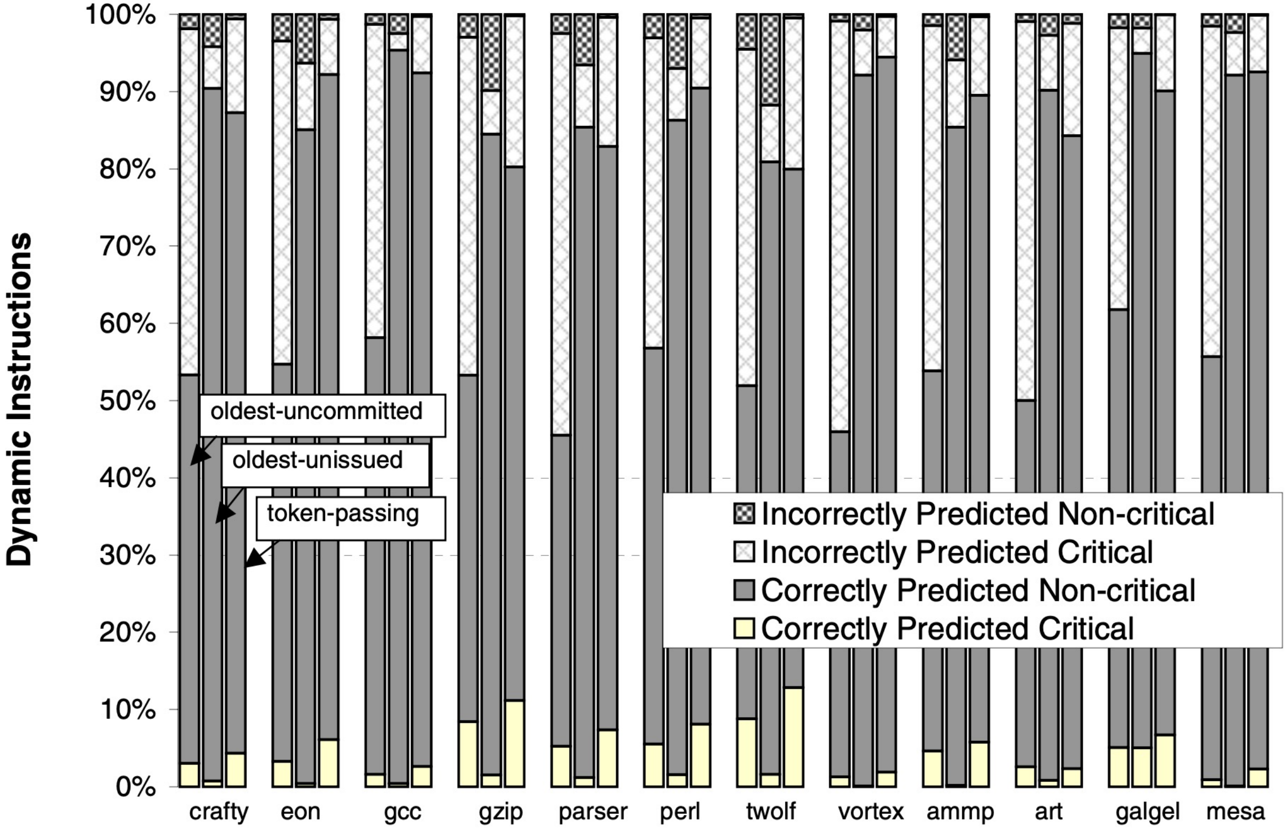
Planting Tokens: A token is planted randomly in the next 10 instructions after it becomes available

Evaluating the CP Predictor: Methodology

We want to evaluate the accuracy of the proposed predictor

- Compare predictions to ”ideal” critical path model
- Comparison of latency reduction against two heuristics:
  - oldest-unissued instruction is critical
  - oldest-uncommitted instruction is critical

Evaluating the CP Predictor: Results

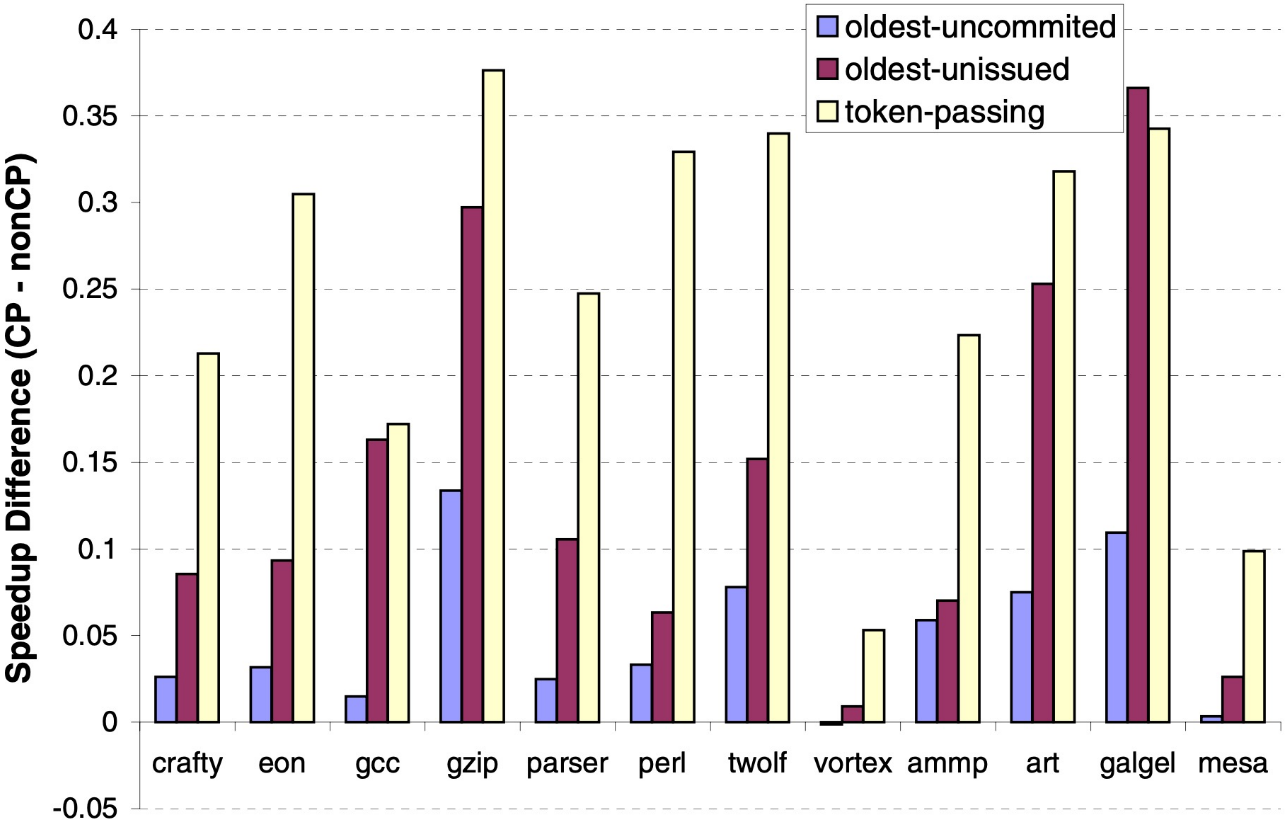


(a) Comparison against ideal CP trace

- Up to 88% accuracy (avg. 80%)

- Especially good at correctly predicting critical instructions

Evaluating the CP Predictor: Results



(b) Comparison via latency reduction

- Comparison against heuristics for evaluation independent of critical path model

- Heuristics work nicely for some workloads

- But CP Predictor is most robust across different workloads

Overview

- The Model of the Critical Path
- Predicting the Critical Path in Hardware
- Applications of the Critical Path Detection
- Conclusion
- Paper Analysis
- Discussion

Applications of the Critical Path

The following applications are examined in the paper:

- Focused cluster instruction scheduling and steering

- Focused value prediction

Focused Cluster Instruction Scheduling and Steering

Complexity of increasingly large instruction windows has prompted proposals of clustering (partitioning) instruction windows and functional units. This introduced new challenges:

-> Latency to bypass results increased
-> Instruction steering

-> Functional unit contention increased due to smaller issue width -> Instruction scheduling

-> Steering policies have conflicting goals:
Good load balancing might increase inter-cluster bypass latency

Decreasing Inter-Cluster Bypass Latency

Baseline policy: Register-dependence steering

Assign instruction to cluster that procuces one of its operands.

If there is more than one producing cluster (tie), choose cluster with fewest instructions.

Decreasing Inter-Cluster Bypass Latency

Modified policy: Focused instruction steering

Assign instruction to cluster that produces one of its operands.

If there is more than one producing cluster (tie) and instruction is critical, it is placed into the cluster of its critical predecessor.

Decreasing Functional Unit Contention

Baseline policy: Prioritize long latency instructions

Modified policy: Schedule critical instructions before non-critical ones

-> Goal: Add contention only to non-critical instructions

Evaluating Proposed Policies - Methodology

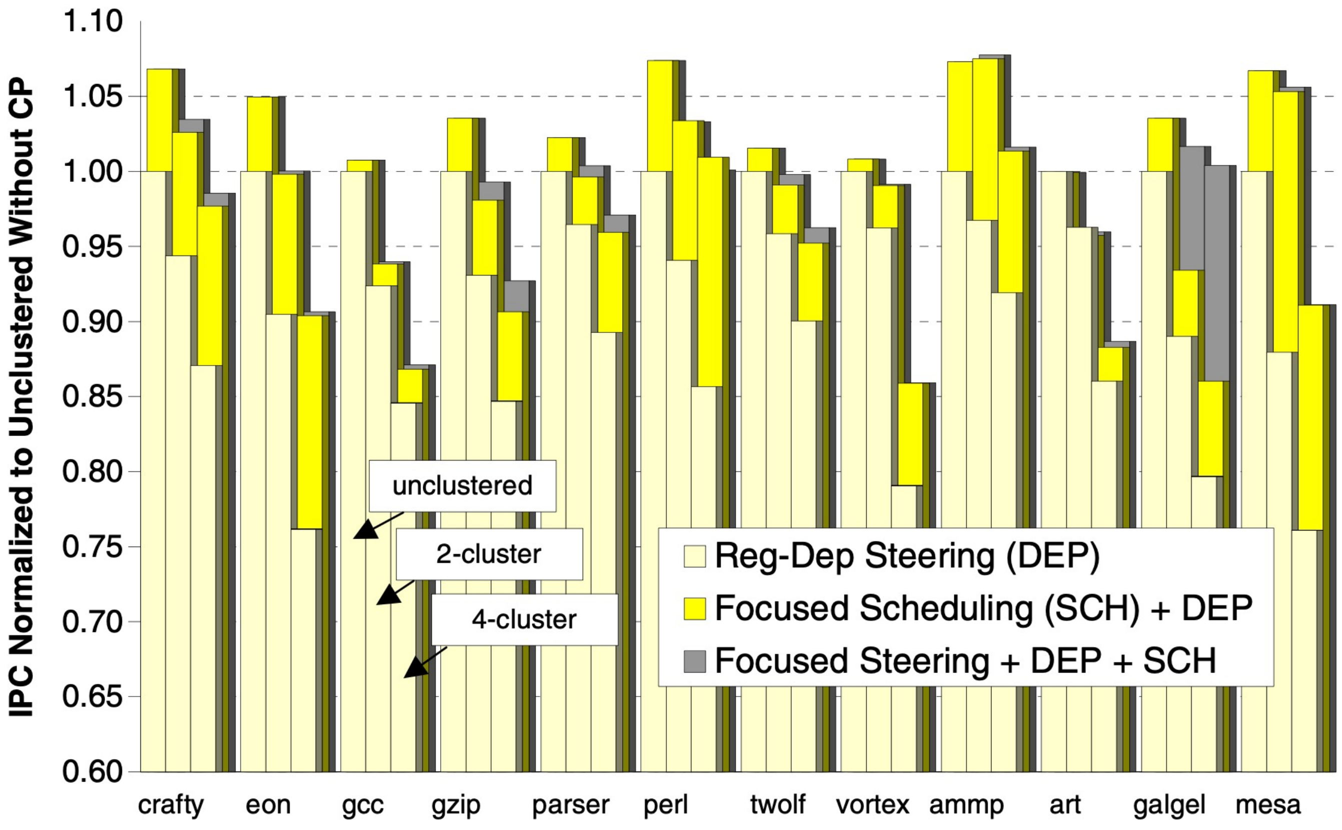
- Same workloads as before

- Observing performance degradation of
  - 2-clustered 4-way issue architecture
  - 4-clustered 2-way issue architecture

  compared to unclustered architecture

- Further comparison against heuristics seen before

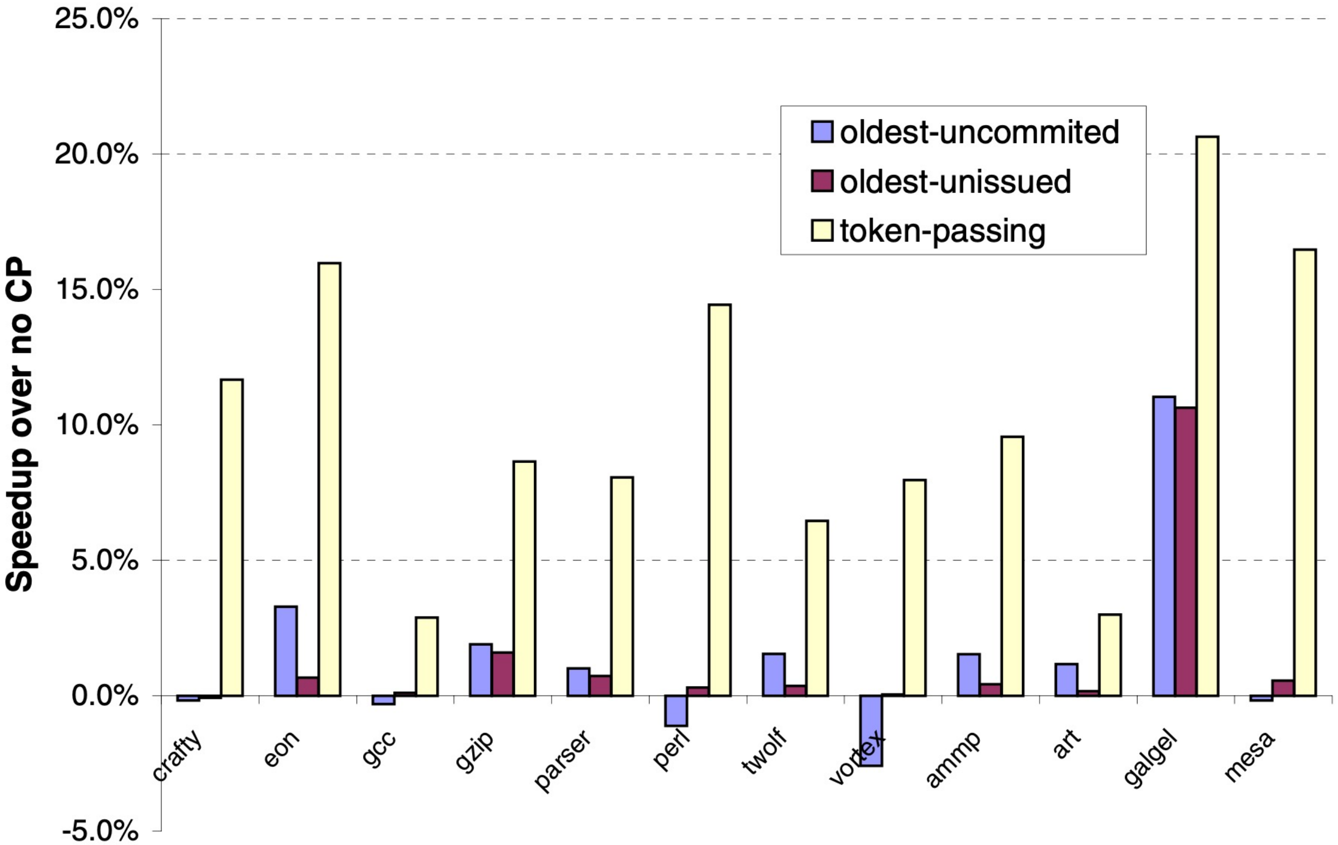
Evaluating Proposed Policies – Results



(a) Scheduling in clustered architectures

- Unclustered: Speedup of up to 7% (Average 3.5%)
- 2-cluster: Average slowdown from 7% improved to slight speedup of 1%
- 4-cluster: Degradation improved from 19% to 6%

Evaluating Proposed Policies – Results



(b) Comparison to heuristics based predictors

- Token-passign algorithm clearly more effective

Overview

- The Model of the Critical Path
- Predicting the Critical Path in Hardware
- Applications of the Critical Path Detection
- Conclusion
- Paper Analysis
- Discussion

Conclusion

Motivation:

- Egalitarian scheduling policies in processors waste resources
- Increasing parallelism and sophistication in processors justify critical path analysis

Key Idea:

- Predict whether an instruction is on the critical path in hardware while keeping cost for graph model as low as possible

Challenges:

- Compile-time optimizations only consider data dependences
- Processor only ever sees fraction of program
  - -> How to optimize for global critical path?

Key Mechanism:

- Token-passing algorithm to estimate criticality of nodes

Results:

- Better scheduling improves CPU performance up to 21%
- Optimized prediction improves CPU performance up to 5%

Questions about the paper?

Overview

- The Model of the Critical Path
- Predicting the Critical Path in Hardware
- Applications of the Critical Path Detection
- Conclusion
- Paper Analysis
- Discussion

Paper Strengths

- Fundamentally novel approach with global critical path prediction at little hardware cost
- Many possible applications for critical path
- Nice insight from validation approach

  i.e. dominance of critical path

Paper Weaknesses

- Validation method not really sound. Increasing all edge weights by one can have unwanted consequences.
  -> Solid proof not possible?
- Hardware implementation ambiguous
- No sensitivity analysis for training parameters

Overview

- The Model of the Critical Path
- Predicting the Critical Path in Hardware
- Applications of the Critical Path Detection
- Conclusion
- Paper Analysis
- Discussion

Expanding the Critical Path Model

- -> How could the critical path model be expanded to capture more dependences and increase precision?
  - -> Cache-line-sharing and other memory dependences are not captured by model
  - -> What are other dependences you can think of that are not captured?

Training parameters

-> How could we tweak the training parameters to increase performance / adapt to usecase?

-> Interesting parameters:

- Token propagation distance
- Maximum number of tokens in flight
- Hysteresis
- Token-planting heuristic

Applications

-> What other applications can you imagine for critical path analysis?

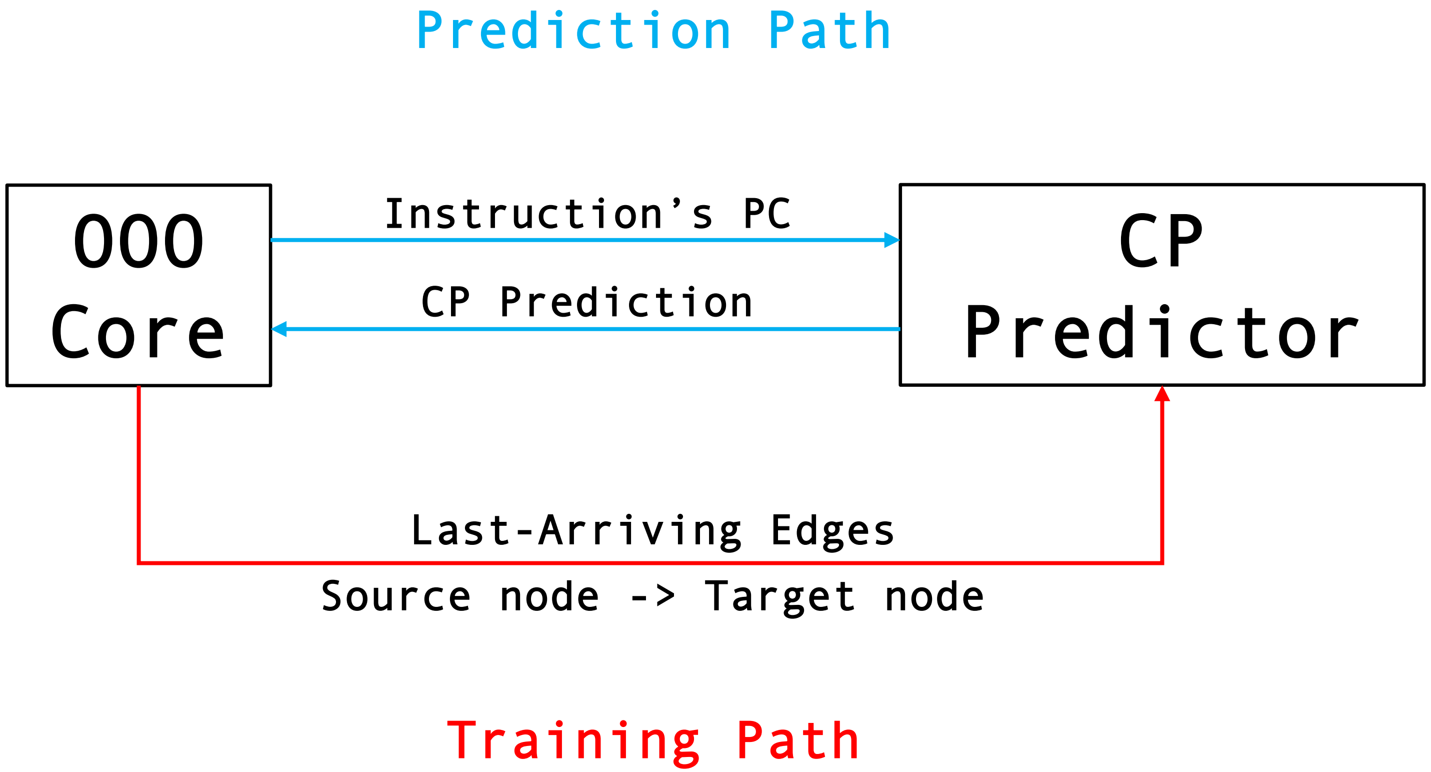
-> Some ideas:

- Scheduling memory accesses in GPUs by criticality (Adwait Jog, et al. SIGMETRICS 2016)
- Optimizing cache prefetching (Anant Vithal Nori, et al. ISCA 2018)
- Focused Value Prediction (Summet Bandishte, et al. ISCA 2020)

Thank you for your attention!

Back-Up Slides

Hardware Implementation



Prediction Path

OOO Core

Instruction’s PC

CP Prediction

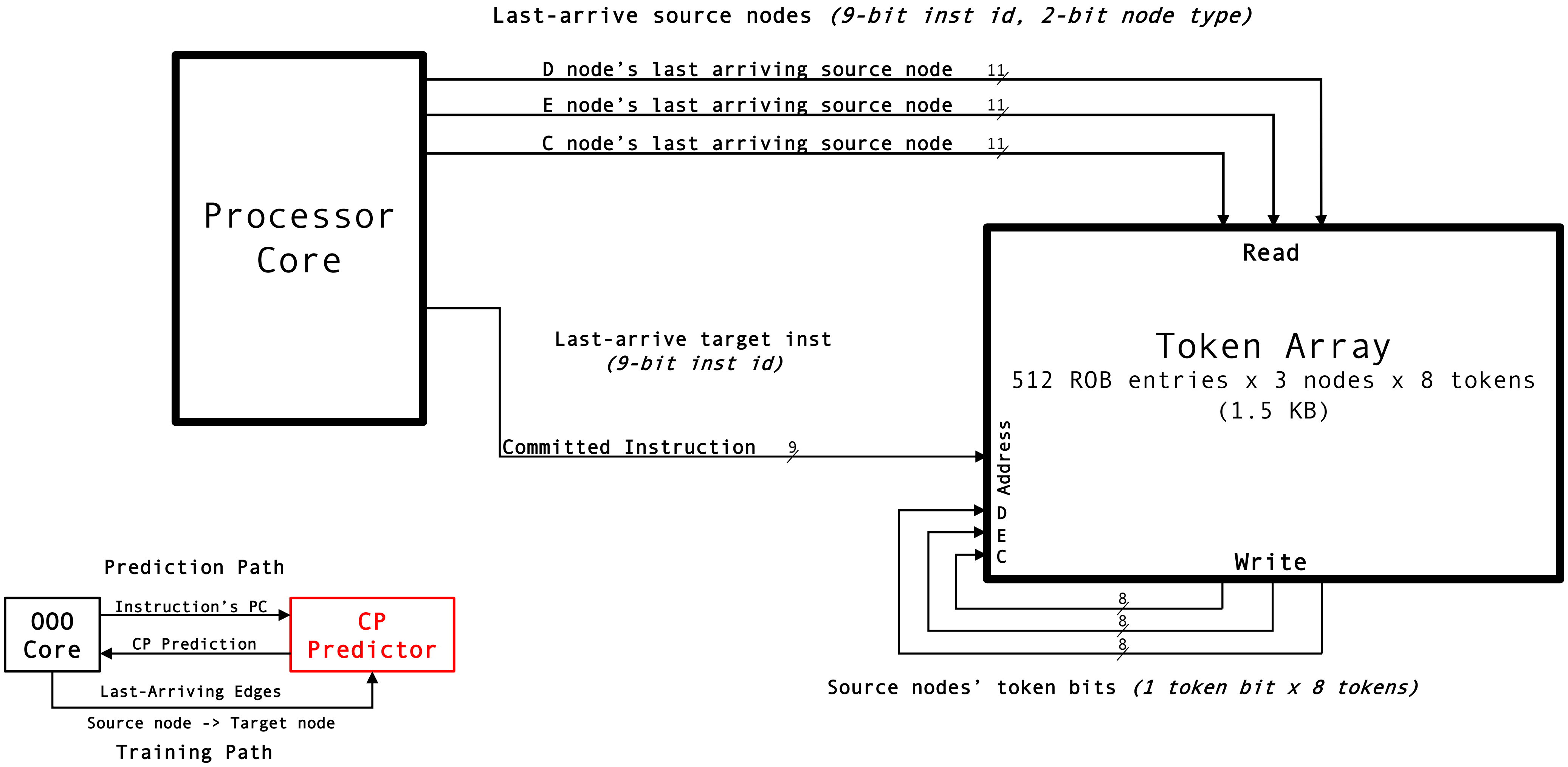
CP Predictor

Last-Arriving Edges

Source node -> Target node

Training Path

Hardware Implementation: The Critical Path Predictor



Last-arrive source nodes *(9-bit inst id, 2-bit node type)*

Processor Core

D node's last arriving source node 11

E node's last arriving source node 11

C node's last arriving source node 11

Last-arrive target inst
(9-bit inst id)

Committed Instruction 9

Read

Token Array
512 ROB entries x 3 nodes x 8 tokens
(1.5 KB)

Address

D
E
C

Write

8
8
8

Source nodes' token bits *(1 token bit x 8 tokens)*

Prediction Path

OOO Core

Instruction's PC

CP Prediction

CP Predictor

Last-Arriving Edges

Source node -> Target node

Training Path

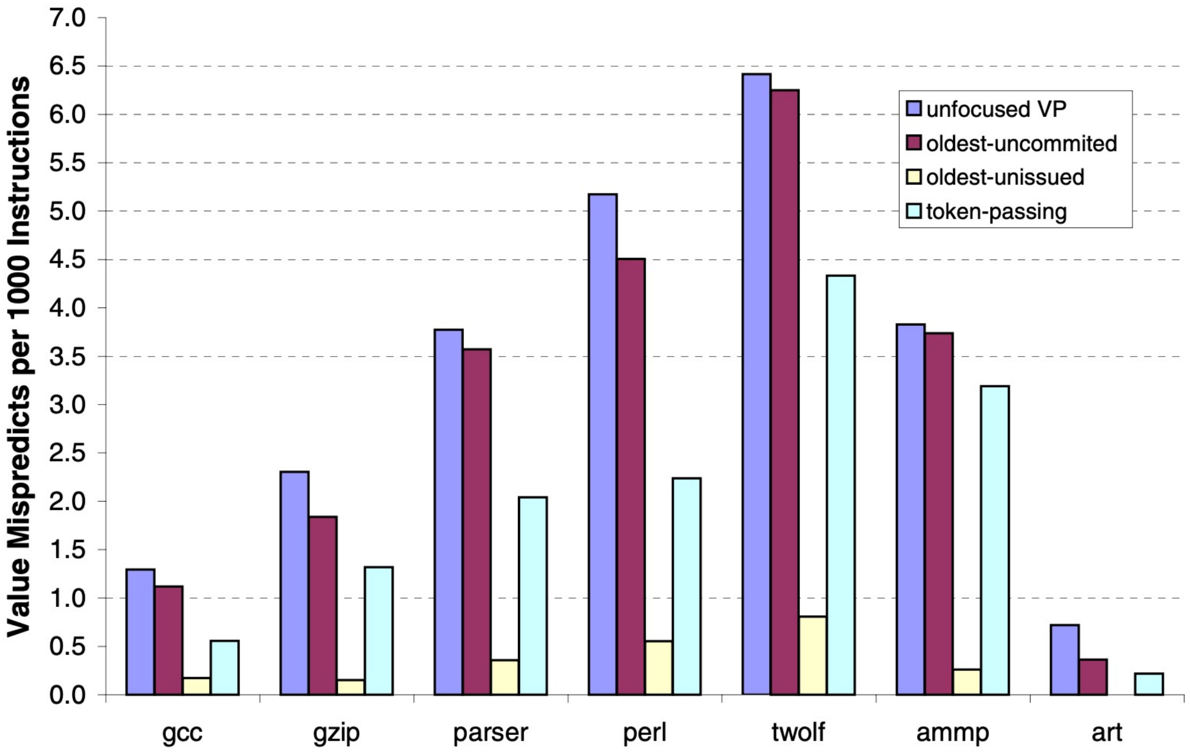
Focused Value Prediction

Idea: Try to predict result of calculation to break data-flow dependences

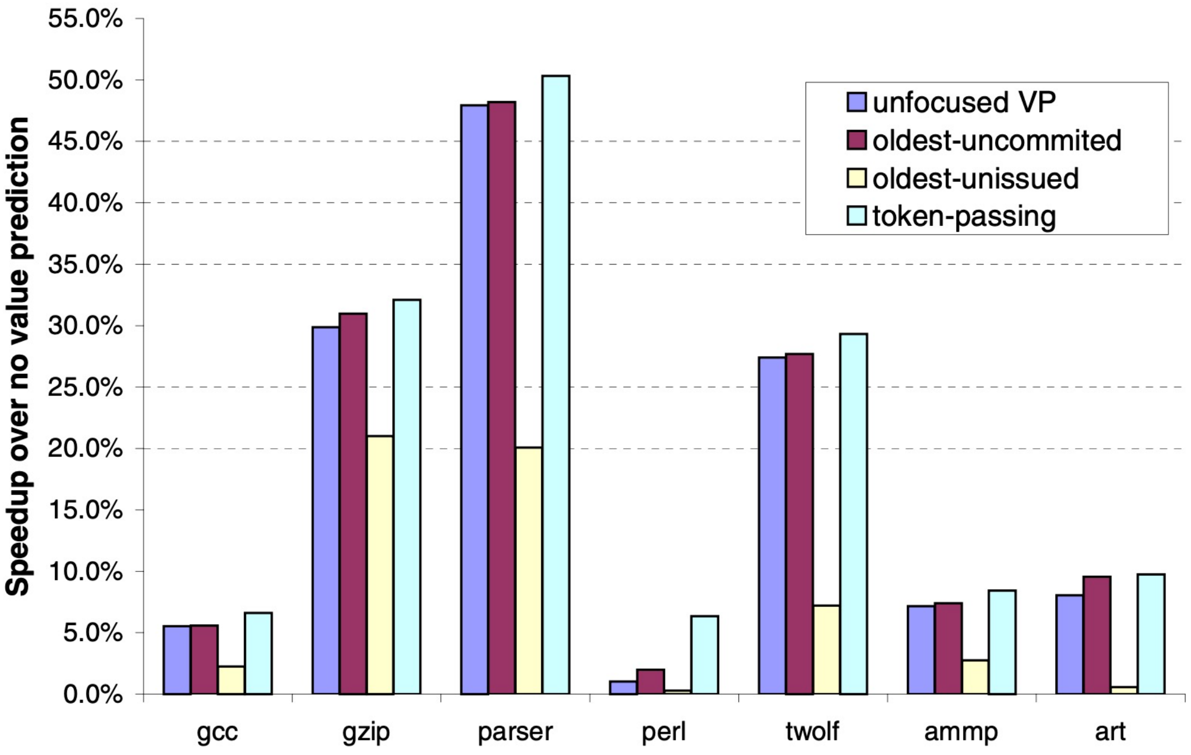
Problem: If non-critical instructions are mispredicted, it might severely degrade performance. If correct, nothing gained.

-> Only make predictions for critical instructions

Focused Value Prediction: Evaluation



(a) Value misspeculations



(b) Speedup of focused value prediction