BAFL: Bottleneck Analysis of Fine-grain Parallelism

Prof. Rastislav Bodík

with Brian Fields
in part with Shai Rubin, Prof. Mark Hill, Prof. Mary Vernon

University of Wisconsin

A tour of a microprocessor museum

Tour theme:

μ-architectural parallelism complicates performance understanding.

Tour game: "Bottleneck Hunt"

Which instruction slowed down the execution, and by how much?

More specifically, why the following model fails?

execution time =
cost(instruction$_1$) + … + cost(instruction$_n$) [cycles]

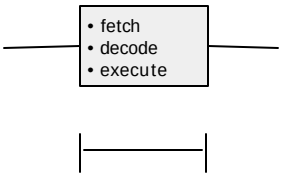
The computer system

- **Many levels of granularity, each with unique performance problems**
  - internet WANS
  - servers
  - microprocessors
- **Our goal:**
  - quantitative approach for modern (out-of-order) processors

A tour of a microprocessor museum (0)

no parallelism

Intel 80386



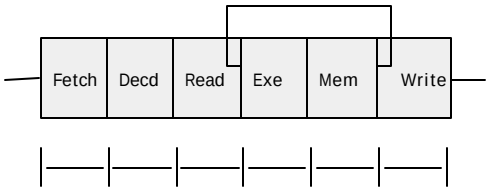
Who cares?

- Architects:
  - circuit complexity
  - power consumption
- Software engineers:
  - performance-critical software
- Students:
  - intuition how processors work
- Processors:
  - understand themselves

A tour of a microprocessor museum (1)

scalar pipeline parallelism

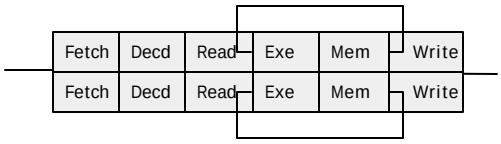
Intel 80486



A tour of a microprocessor museum (2)

in-order superscalar pipeline

Intel Pentium



The Bill Cosby Rule:*

"You're not a parent if you only have one child."

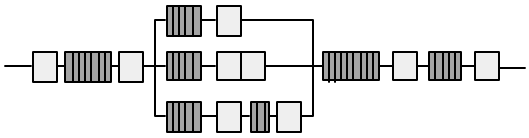
*rule named by Amir Roth

Why critical path?

☛ Microprocessors are fine-grain parallel systems

like wide-area networks:

- queues are like routers, pipelines are like communication links
- many (bad) events going on in parallel, their latency tolerated



A tour of a microprocessor museum (3)

out-of-order superscalar

Intel Pentium 4

Outline

The model of micro-execution

- capture both program and processor constraints

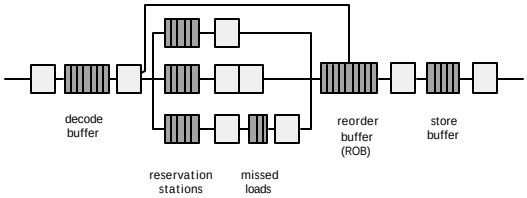
Four metrics:

- criticality
- slack
- execution modes
- cost

A tour of a microprocessor museum (end)

out-of-order superscalar

typical buffers, queues, windows



☞ processors are good at tolerating latency, but poor at deciding what to tolerate.

Critical path of a *microexecution*

Critical path misconceptions:

- "Every 'bad event' is critical."
  - branch misprediction
  - reorder-buffer stall
  - L1 cache miss
  - L2 cache miss
- "Critical path is obvious … "
  … it contains instructions providing data for 'bad events'

Modeling: why hard?

Critical path consists of:

1. instructions and data dependences
   - as in a traditional "compiler" view
2. microarchitectural resource constraints
   - branch mispredictions, finite fetch b/w, etc.

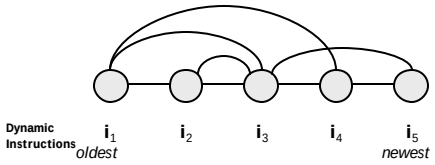
Together describe the *microexecution* of a given program executing on a given machine

How to model in a uniform way?

Critical Path Models (1)

First, for a simple in-order machine

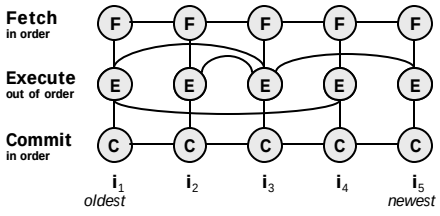
- Data dependencies
- Resource dependencies



Resources constrain the dataflow execution

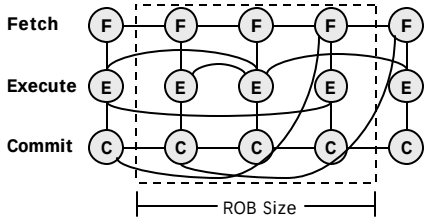
Critical Path Models (2)

For an out-of-order machine



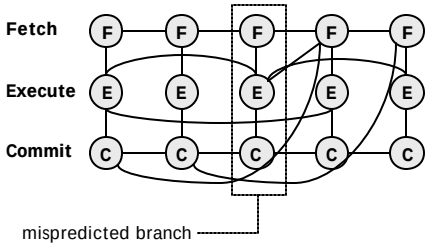
Critical Path Models (3)

OOO + finite re-order buffer

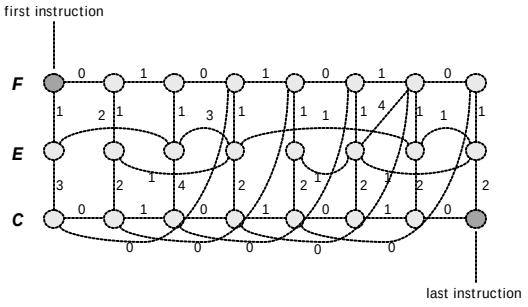


Critical Path Models (4)

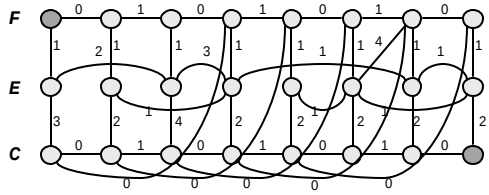
OOO + finite ROB + branch misp



Example



Example

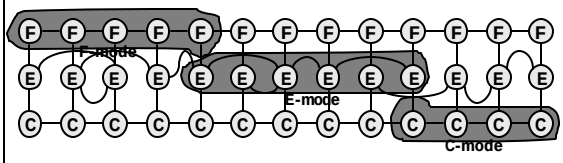


CP Length = 16 cycles Þ Exe Time = 16 cycles

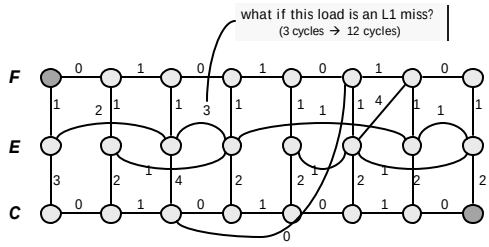
Execution Modes

Three modes of execution

fetch	limited (F-mode)
execute	limited (E-mode)
commit	limited (C-mode)

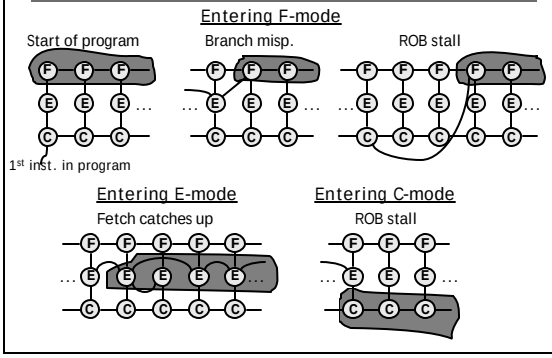


Example

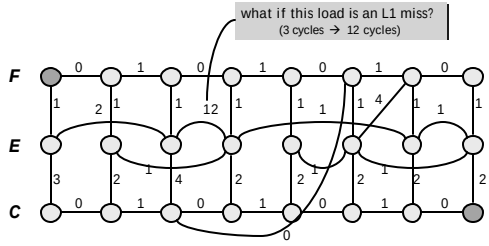


CP Length = 16 cycles Þ Exe Time = 16 cycles

Execution Modes



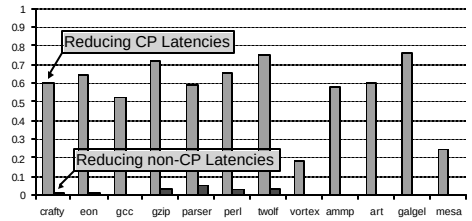
Example



CP Length = 19 cycles Þ Exe Time = 19 cycles

Validation: can we trust our model?

Execution Time Reduction (in cycles) per Cycle of Latency Reduced



Outline

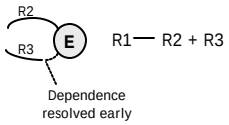
The model of micro-execution

- capture both program and processor constraints

Four metrics:

- **criticality**
- slack
- execution modes
- cost

Step 1. Observing



If dependence into R2 is on critical path,
then value of R2 arrived last.

critical Þ arrives last

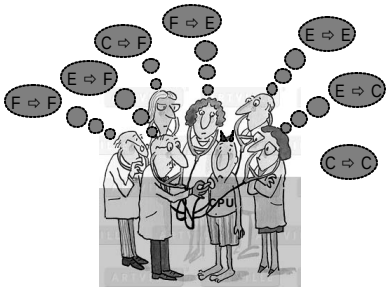
arrives last ÞÞ critical

Why criticality predictor? Policies!

mechanism	policy:	current	better
OOO execution	how to schedule?	oldest first	critical first
prediction and speculation	when to speculate?	on each prediction	only critical
non-blocking caches	how to serve mem requests?	FIFO	critical first
pre-fetch, pre-execute	what to prefetch?	all misses	prefetch critical

☞ **Current policies are egalitarian: all "bad" events equally harmful.**

Last-arrive edges: a CPU stethoscope



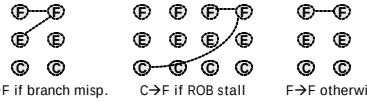
Prediction: why hard?

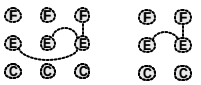
Three steps:

1. **observe the microexecution Þ hard!**
   - **measuring edge latencies is intrusive**
2. **analyze to find critical path Þ hard!**
   - **graph too large to buffer**
   - **and topological sort too complex**
3. **store prediction for later use Þ easy!**
   - **store in table indexed by PC**

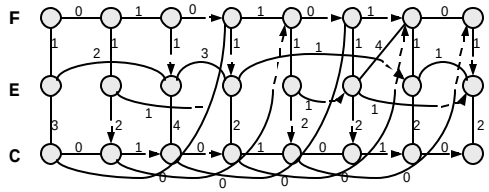
Implementing last-arrive edges

Observe events within the machine



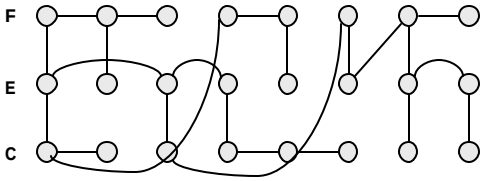


Last-arrive edges



…and we've found the critical path!

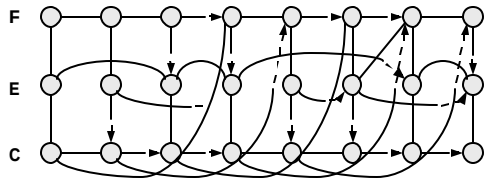
Backward propagate along last-arrive edges



☞ Found CP by **only observing last-arrive edges**
☞ **but still requires constructing entire graph**

Remove latencies

Do not need explicit weights



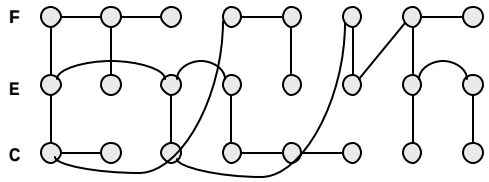
Prediction: why hard?

Three steps:

1. **observe the microexecution** Þ **solved!**
   - **measuring edge latencies is intrusive**
2. **analyze to find critical path**
   - **graph too large to buffer** Þ **hard!**
   - **and topological sort too complex** Þ **solved!**
3. **store prediction for later use** Þ **easy!**
   - **store in table indexed by PC**

Prune the graph

Only last-arrive edges needed
(other edges must be non-critical)



Step 2. Efficient analysis (predictor training)

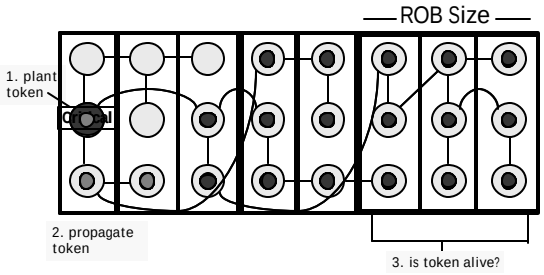
CP is a "long" chain of last-arrive edges.

⇒ the longer a given chain of last-arrive edges, the more likely it is part of the CP

Algorithm: *find sufficiently long last-arrive chains*

1. Plant token into a node **n**
2. Propagate forward, only along last-arrive edges
3. Check for token after several hundred cycles
4. If token alive, **n** is assumed critical

Token-passing example



☞ Found CP **without constructing entire graph**

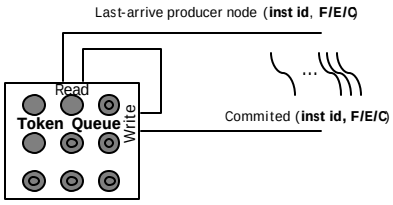
Steps to exploiting critical path

- ✓ **modeling**
- ✓ **predicting**

⇨ **applying**

1. resource arbitration
   - case study: cluster scheduling
2. speculation control
   - case study: value prediction

Implementation: a small SRAM array



Size of SRAM: 3 bits x ROB size < 200 Bytes

can simply replicate for additional tokens

Experiment Setup

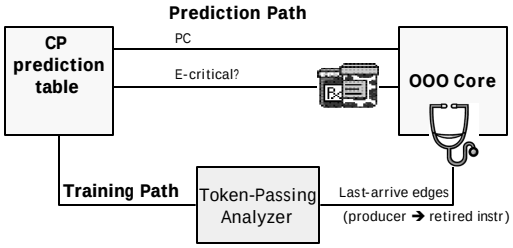
Aggressive Core

- 8-way issue, 256-entry window,
- three configurations: core split into 1, 2, or 4 clusters:
  - unclustered, 8-way, 256-entry
  - 2 clusters, each 4-way, 128-entry
  - 4 clusters, each 2-way, 64-entry

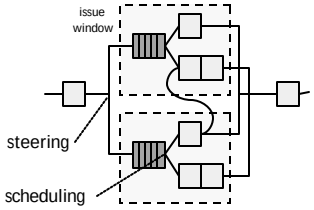
CP Predictor

- 8 tokens (1.5 KB token-passing array)
- 16K-entry array for storing predictions (12 KB)
- 6-bit hysteresis

Putting it all together



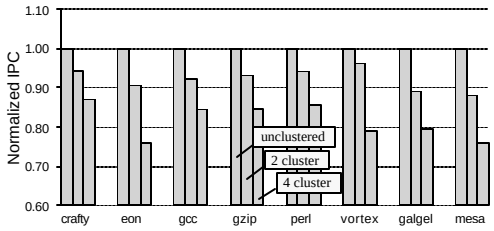
Case Study #1: Clustered architectures



1. current state of art (Base)
2. base + CP Scheduling
3. base + CP Scheduling + CP Steering

Current State of the Art

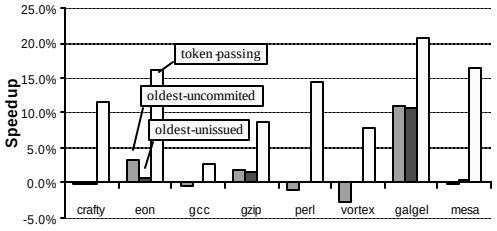
Constant issue width, clock frequency



☞ Avg. clustering penalty for 4 clusters: 19%

Local vs. Global Analysis

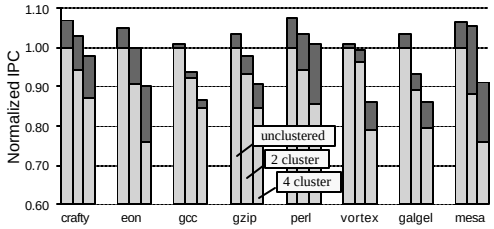
Previous CP predictors:
local stall-based predictions (HPCA 01, ISCA 01 by others)



☞ CP exploitation seems to require global analysis

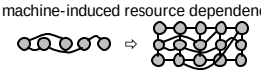
CP Optimizations

Base + CP Scheduling



Criticality: contributions

1. Critical Path of a *microexecution* consists of :
   - program-induced data dependences
   - machine-induced resource dependences

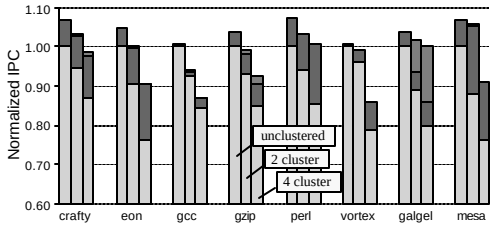


2. CP Prediction = global run-time analysis
   - observe last arrive ⇨ analyze ⇨ apply predictions

3. One predictor ⇨ many optimizations
   - schedule critical instructions first ⇨ up to 20% speedup
   - value predict only critical ⇨ up to 5% speedup

CP Optimizations

Base + CP Scheduling + CP Steering



☞ Avg. clustering penalty reduced from 19% to 6%

Outline

The model of micro-execution

- capture both program and processor constraints

Four metrics:

- criticality
- **slack**
- execution modes
- cost

Beyond criticality

- **Slack (definition):**
  number of cycles an instruction can be slowed down before it becomes critical.
- **Slack is prevalent:**
  75% dynamic instructions can be delayed by at least 5 cycles without impact on performance (no slow down)
- **How to compute slack?**
  - in simulator
  - in hardware

Experiments

- **Power-aware machine:**
  - two clusters:
    - fast: full frequency
    - slow: half frequency (consumes ¼ power)
  - three non-uniformities
    1.
    2.
    3.
  - results:
    - 3% within performance of two fast clusters
    - existing techniques: 10% slowdown

Why is slack useful?

- **Non-uniform machines**
  - resources at multiple levels of quality
  - to deal with technological constraints
    - to save power: slow/fast clusters of ALUs
    - wire delay: some caches further away
- **Problem boils down to controlling non-uniform machines**
  - goal: hide the (longer) latency of low-quality resources
  - can do this with slack

Outline

The model of micro-execution

- capture both program and processor constraints

Four metrics:

- criticality
- slack
- **execution modes**
- cost

How to compute slack?

- **On the graph**
  - two-pass topological sort
- **In the processor**
  - *delay and observe:* by reduction to criticality analysis
    - delay instruction *i* by *n* cycles
    - if *i* is not critical,
      then *i* did have at least *n* cycles of slack.

Reconfigurable machines

- **Imagine that, to save power, you can dynamically:**
  1. turn on/off some ALUs
  2. change their frequency

Problem: how to adapt the machine configuration to the program needs?

Outline

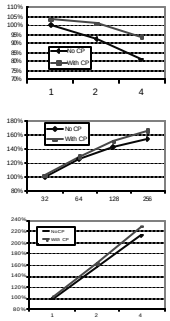
The model of micro-execution

- capture both program and processor constraints

Four metrics:

- criticality
- slack
- execution modes
- **cost**

Effect of CP scheduling on future designs?



1) Cluster the machine:

- 8-way machine split into 1, 2, 4 clusters
- as in the previous experiment

2) Enlarge scheduling window:

- 4 cluster x 2-way machine
- vary the window size in each cluster

3) Add clusters:

- each cluster is 2-way, 64-entry window

Finally, the quantitative approach

- **All boils down to computing a cost of instruction:**
  - can easily compute from the graph, if the graph is available (in the simulator)
  - can compute in hardware? a new version of the randomized algorithm?

This talk is about:

Making processors smarter

- **a modern processor:** strong body, weak mind
- **example:** can execute instructions out of order, but does so without considering instruction cost

Making smarter = teach how to find bottlenecks

- instructions whose latency hurts
- resources whose contention hurts

I will show how to

- find bottlenecks (at run-time, with simple hardware), and
- alleviate them (using existing resources, retrofitting)

The future

Superscalar complexity haunts

- not only circuit designers
- and verification engineers
- but also performance engineers
- and hence also architects themselves

Critical-path instruction processing helps

- understand performance complexity
- and hence also
  - exploit better existing designs
  - lead to simpler designs

Our solution:

Critical-Path Instruction Processing

- critical-path analysis of μ-exe performance
- critical-path prediction
- critical-path hardware optimizations

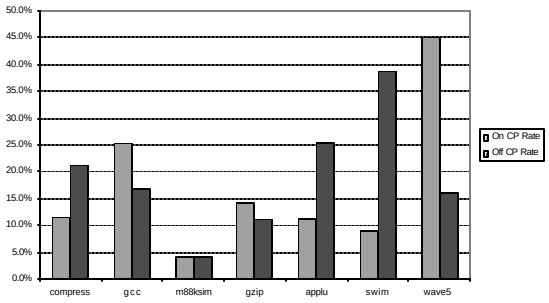
critical-path instruction processing
is
taming the m-architectural evil

which is

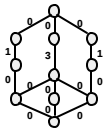
taming the superscalar performance complexity:

- find execution bottlenecks, and
- alleviate them.

Are critical instructions value predictable?



Why critical path?

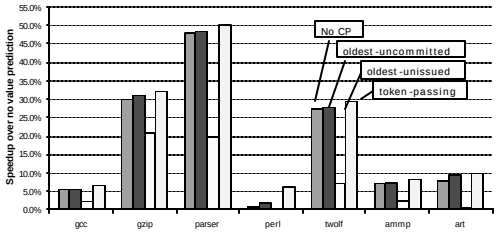


call
arrive at barrier
leave barrier

CP used to understand bottlenecks in large-scale systems:

- message passing, and locking in shared-memory systems
  Hollingsworth [IEEE PDS '98]
- TCP transactions
  Barford and Crovella [SIGCOMM '00]

Focused Value Prediction



Speculation Control: Value prediction

Optimization:
Value predict only critical instructions

- removes speculations that
  - have no benefit, but
  - may have high misspeculation recovery cost

Focused Value Prediction (2)

less confident value predictor → more value mispredictions: