New Methods for Exploiting Program Structure and Behavior in Computer Architecture

Guri Sohi

sohi@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison

Contributors: Andreas Moshovos, Avinash Sodani, Amir Roth, Andy Glew, Craig Zilles, Harit Modi, Adam Butts, Param Oberoi
Outline

- Growth in processor performance
- Concepts and performance refiners
- Existing basis: observed events
- Potential future basis: causal relationships
- Examples
- Handling information about causal relationships
Growth in Microprocessor Performance

-- Graph showing the growth in microprocessor performance over time, with various models and their performance metrics.

Thanks to Doug Burger and Steve Keckler

New Methods for Exploiting Program Structure and Behavior in Computer Architecture
Contributors to Performance Growth

• Semiconductor technology
• Architectural and microarchitectural concepts
  ◦ memory hierarchies
  ◦ pipelining
  ◦ out-of-order execution with speculation
  ◦ speculative multithreading
• Concept Enablers/Performance Refinners
  ◦ novel cache elements
  ◦ cache management
  ◦ branch predictors
  ◦ dependence predictors
Software/Hardware Interface and Hardware Role

- Interface is imperative (vs. declarative)
  + results in repeatable execution state
  - limited flexibility for changing technology parameters

- Hardware tries to determine program’s future actions
Hardware Performance Refiners

- Hardware tries to learn program’s intentions
  - Observes program actions (statistical)
  - Uses other information
- Uses information to keep ahead of program’s execution
  - Make predictions about future outcomes
- Limited amount of hardware limits ability to learn about events
Observed Events: Secondary Information

**Secondary information**: instruction inputs/outputs

- Examples: branch outcomes, addresses, values
- Properties: spatial/temporal locality, patterns

**Current mechanisms almost exclusively based on secondary information and its properties**

**Problem I**: secondary properties may not hold all the time

**Problem II**: Hard to determine program’s future actions
Observations

• Programs have structure (relationships amongst operations)

• Program structure causes the observed program behavior

• Can we exploit primary information, i.e., causal relationships in architecture/microarchitecture?

• Good model for predicting about future outcomes might be selected parts of the program!
Primary Information: Program Structure

**Primary information**: relationships amongst operations

- Examples: control dependences, data dependences
- Properties:
  - **temporal stability**: program is invariant (strong)
  - **causality**: causes all observed secondary behavior
Problems Considered

• Branch Prediction
• Scheduling memory operations in OOO machines
• Inter-operation communication through memory
• Communication in multiprocessors
• Prefetching linked data structures
Problem: Branch Prediction

• Predict the outcome of a branch instruction (taken or not taken)

• Smith 2-bit predictor: learns about previous branch outcomes
  - observes outcomes of single branch

• Yeh and Patt-type adaptive predictors: branch outcomes correlated with other branch outcomes
  - learn branch correlations
  - Still exploiting an observation, rather than a cause

• Can we exploit primary (causal) information?
  - Yes, but not in this talk (e.g., Farcy (MICRO ’98), Roth (ICS’99))
Problem: Scheduling Memory Operations

- OOO instruction scheduler has collections of instructions to schedule, including loads/stores
- When to schedule a load instruction?
  - when it is not dependent on a pending store
    - may be too late
  - speculate no dependence, and recover if incorrect
    - need performance refiner to improve speculation accuracy
Dependence Speculation

- Selective may perform worse than blind
- Can also perform as well as the ideal
- In practice:
  performance behavior varies
Learning Dependence Relationships

- Dependence: (Load PC, Store PC)
- Temporal locality - Small Working Set.
- Use a small table to:
  - (1). track recent mis-speculations
  - (2). Predict dependences

**Memory Dependence Prediction Table**

<table>
<thead>
<tr>
<th>LDPC</th>
<th>STPC</th>
<th>PRED</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

① Misspeculation
② Allocate entry
① Execute?
② No! Synchronize
① Synchronize?
② Synchronize
**Solution Summary**

**Solution 1:** use prediction to stall offending loads
  - no program structure information required
  - not very effective

More Hesson, et. al., 1997

**Solution 2:** determine store-load dependences and use to synchronize speculation
  - use program structure
  - very effective

More Moshovos, et. al., 1997, Chrysos and Emer 1998
Problem: Communicating Values Through Memory

<table>
<thead>
<tr>
<th>Implicit</th>
<th>Explicit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Delays:</strong> 1. Calculate address 2. Establish Dependence</td>
<td>Store - Load: Direct Link No Delays</td>
</tr>
</tbody>
</table>

Timeline
- store
- value
- store addr
- load
- load addr
- store2 addr

New Methods for Exploiting Program Structure and Behavior in Computer Architecture
Speculative Memory Cloaking

Dynamically & Transparently convert implicit into explicit

- Dependence prediction $\rightarrow$ direct load-store link: synonym
- Speculative and has to be eventually verified

Dependence Prediction

Timeline

Synonym File

Traditional Memory Hierarchy
Memory Cloaking Summary

**Program structure:** Memory is a communication device for passing values from stores to loads. Not random: only certain stores to certain loads

**Speculative Memory Cloaking**
Link stores to loads explicitly, pass value along link

![Diagram of Memory Cloaking](image)
Speculative Memory Bypassing Summary

**Program structure:** Loads and stores are used for passing values from one instruction (DEF) to another (USE). Via memory? (maybe not, can do it directly)

**Speculative Memory Bypassing**
Collapse DEF-store, store-load, load-USE links into a direct DEF-USE link

More: Moshovos & Sohi, Tyson & Austin, MICRO-30
Problem: Fast Communication in Shared memory MP's

**Problem:** Optimize CC protocols for sharing patterns

**So far:** Detect patterns using address attributes

- Track state proportional in size to data (big)
- Little predictive power

**Program structure:** Sharing pattern property of program, not data

Detect using instruction relationships

- Track state proportional in size to program (small)
- Great predictive power, works much better

Problem: Prefetching Linked Data Structures

Linked Data Structures (LDS): pointer-based
- Lists, trees, graphs, etc.
- Prevalent: simulators, compilers, databases, OO-progs

```c
for (l = list; l; l = l->next)
    if (l->key == key)
        process(l);
```

As if memory latency wasn’t a problem already...

**Pointer Chasing Problem**

- Pointer loads serialized
- Latencies add

**Solution: Make sure latencies are short → Prefetch**
Potential Solution

\(v / \text{Prefetch} / := \text{issue loads as early as possible (as soon as address is ready)}\)

**First reaction:** Try to predict addresses
- LDS: See A, B, C → predict ?
Mechanics I - Overview

**Observation:** some piece of code must be producing addresses/traversing structure

**Step 1.** Examine running program, learn dependences, isolate code thread

**Step 2.** Pre-execute code to launch prefetches

More: [Roth, Moshovos & Sohi, ASPLOS-8, ISCA99]
Establish (dynamic) dependence between (static) loads
Use values exchanged to do this

1. Buffer recent load outputs
   - B \( l = l->\text{next} \)
   - Output Load
   - B \( l = l->\text{next} \)

2. Compare current load inputs
   - B \( l = l->\text{next} \)

Done
Can go
- \( l = l->\text{next} \)
- \( l->\text{key} \)
Mechanics III - Prefetching

1. Access dependence table

2. Compute prefetch address

B
l->key

B
l = l->next

C
l->key

C
l = l->next

D
l = l->next

Done
Can go
l = l->next
l->key
l = l->next
l = l->next

C->next

D-Cache
Summary and More Questions

• Several applications for very limited program structure information

• How should this information be gathered, managed, and provided to the hardware performance refiners?
  - described techniques used hardware
  - software knows this information trivially

• How can software get involved?
  - 4 models
Model 1 for Software Involvement

- Hardware extracts program structure information. Represents internally in declarative manner
  - Likely to have limited abilities; much more possible with software
  + may be only practical option
Model 2 for Software Involvement

- Software has program structure information available trivially
- Use information to develop mandate for hardware
- Express mandate in imperative language
- Does not work with a collection of hardware platforms
  - balance keeps shifting with disparate rates of changes in technology
- Been there, done that
Model 3 for Software Involvement

• Express computation as a dataflow graph
  ◦ dependence relationships available to hardware

• Should this be done? NO
  ◦ No repeatable state for program execution

• Advantages of repeatable state
  ◦ Easy to write debug software (*debatable*).
  ◦ Easy to design, build, verify hardware (*not debatable*).
Model 4 for Software Involvement

• Express computation in imperative manner
  ◦ repeatable state

• Provide advisory program structure information in a declarative manner

• Overheads for doing so
  ◦ provide information judiciously
**Research Issues**

**For a particular optimization**
- What program structure information is required?
- How do we represent this information?
- How do we collect and manage this information?

**More broadly**
- Where can we apply program structure?
- Is there a larger framework?
- What is general purpose program structure?
- Implementations?
Research Issues II: Implementations

**Hardware only**

**Software to hardware**
- Compiler has all kinds of program information
- How to express it? Instruction-like things awkward
- Where and how much to express?

**Software/hardware hybrids**
Summary

• Architectural/microarchitectural innovations (performance refiners) will be critical to future processor performance growth
• Current performance refiners are based on observed events
• Future performance enablers likely to need program structure to reason about program’s future actions
• Several examples presented; many others
• Several research issues in gathering, conveying, maintaining, and managing such information