Using Program Structure Information

- We can learn program dependences using dependence predictors
- Can we use program structure information in innovative ways?

Program Structure Information: An Example

Example: branch prediction

- Early: Branches predicted in isolation
- Major Leap: Branch correlation
- Then: Golden age of branch prediction

Great insight? Different branches related

programs have structure!

Example II: memory hierarchy design

- Early: Program structure not taken into account
- Now: Still not. Why not?
- Major leap: Coming soon

Secondary Information: Not Really Program Structure

Branch correlation is a **secondary** method

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: weak properties may not hold all the time

Problem II: Hard to figure out what’s going on sometimes

Primary Information: Real Program Structure

“Programs have structure” is too obvious

Primary information: relationships amongst operations

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

We have program structure handy! Can we exploit it?

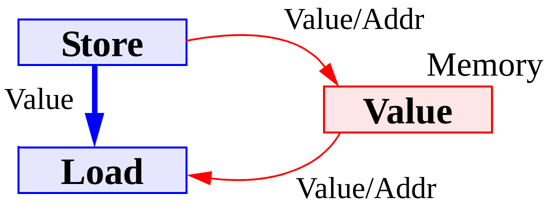
Application: Fast Communication Through Memory

Problem: Accessing memory is inherently slow, ambiguous

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

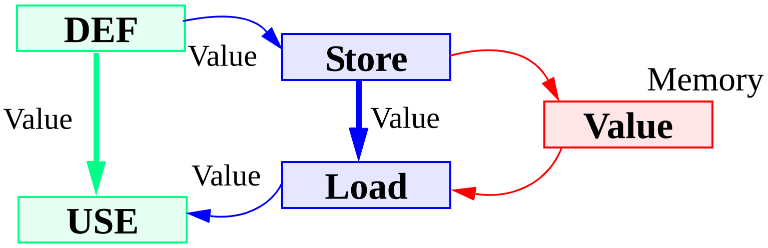


Fast Communication II

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 on Cloaking & Bypassing: [Moshovos & Sohi, MICRO-30]

Fast communication III: 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

More: Work by Kaxiras

Application: Prefetching Linked Data Structures

Problem: Linked data structures

- Chains of long-latency loads limit parallelism
- Hard to predict addresses for prefetching

Program structure: (l = list; l; l = l->next)
Traversal uses few static loads, few relationships

Learn structure and pre-execute speculatively:

- No explicit address prediction, predict loads and execute
- All we need to remember: **l = l->next**
- Compresses chains, removes aritificial issue delays

More: [Roth, Moshovos & Sohi, ASPLOS 1998]

Application: Branch Pre-execution

Program structure: Branches more closely related to instructions that feed them than to other branches

Learn dependences, use to pre-compute branches

- Early: avoid mis-speculation
- A little late: reduce penalty

Proof of concept: Virtual Function Calls

- Hard to predict: Multiple targets a problem
- Easy to pre-compute: Linear dependence chains
- Cuts misspeculation by ~80%

More: [Roth, Moshovos & Sohi, ICS 1999]

Dependence Based Prefetching for Linked Data Structures

1998 ASPLOS Conference

Amir Roth

Andreas Moshovos, Guri Sohi

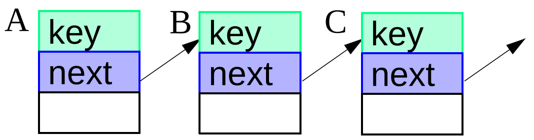
amir,moshovos,sohi@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison

Basics

Linked Data Structures (LDS): pointer-based

- Lists, trees, graphs, etc.
- Prevalent: simulators, compilers, databases, OO-progs



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

Pointer Load

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

Pointer Chasing Problem

→ **Pointer loads serialized**

→ **Latencies add**

Solution: Make sure latencies are short → Prefetch

Problem and Solution

v /**Pre.fetch**/ := Issue loads as early as possible
(as soon as address is ready)

First reaction: Try to predict addresses

- Array: See A[0], A[1], A[2] → predict A[3]
- LDS: See A, B, C → predict ?

Catch 22: Need to prefetch, but...
can’t predict addresses

Our work: what to do about this

1. **Schedule** pointer loads aggressively
2. **Isolate** pointer load thread and **pre-execute**
3. Use **dependence information** to do this transparently

...and this works!

Prefetching as Aggressive Scheduling

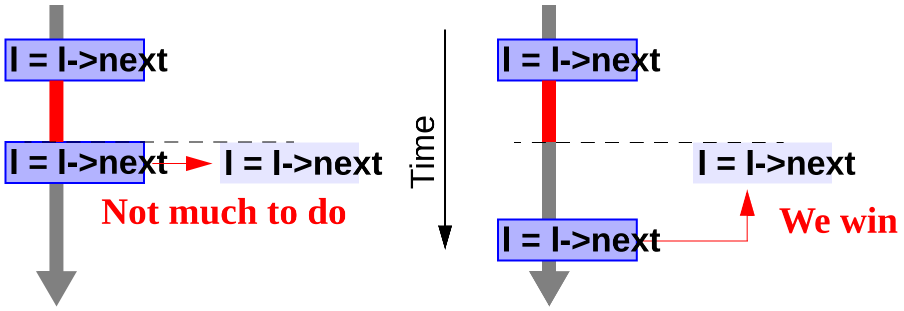
Strategy: Schedule loads when addresses ready

OOO issue tries, but constrained..

1. **window:** must see 2. **schedule:** consider all instructions

Key Issue: Distance between dependent pointer loads

Short: **“optimal”** **Long:** **constraints kick in**



I = I->next

I = I->next

I = I->next

Not much to do

Time

I = I->next

I = I->next

I = I->next

We win

Exploit long distances: Remove constraints

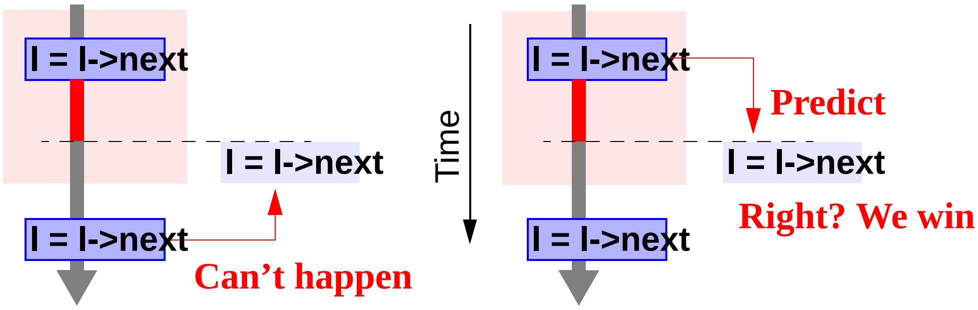
Prefetching as Aggressive Scheduling (cont.)

Build a prefetch engine

2. **schedule:** considers pointer loads only
1. **window:** issues pointer loads without seeing them

Q: Where do these come from? **A: Predict**

OOO issue **What we do**



I = I->next

I = I->next

I = I->next

Can’t happen

Time

I = I->next

Predict

I = I->next

I = I->next

Right? We win

Address Prediction → Load Prediction

Load Prediction for LDS Access

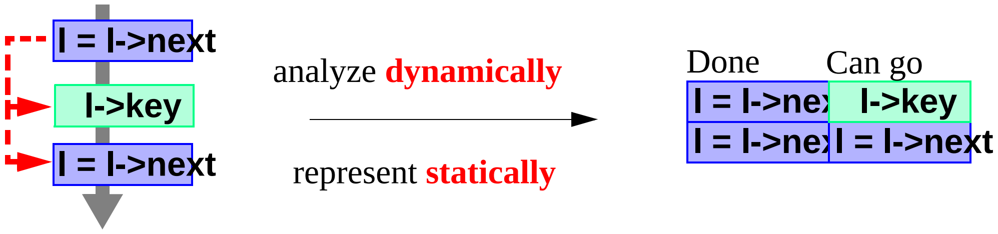
What do loads need to know:

"Is what I just loaded an address?"

"Who will use this address?" or "Who can issue now?"

Answer using Data Dependence Information

Why? Once a dependence, always a dependence (almost)



l = l->next

l->key

l = l->next

analyze **dynamically**

represent **statically**

Done Can go

l = l->nex	l->key
l = l->nex	l = l->next

Load Dependences

+ Tell us what we need: "who can go now?"

+ Ignore irrelevant info like sequencing

Address Prediction vs. Operation Prediction

Address: Addresses

Operation: Program operation that computes addresses

Arrays: Observe addresses, extract formula

Formula: Base + stride

But really: Captures program operation

LDS: Cannot extract formula from addresses

Observe program directly

Formula: Load dependences

Lesson on Pattern/History Based Prediction?

Simple/Expressed pattern? ⟶ Predict

Complex/Hidden? ⟶ Predict operation and pre-compute

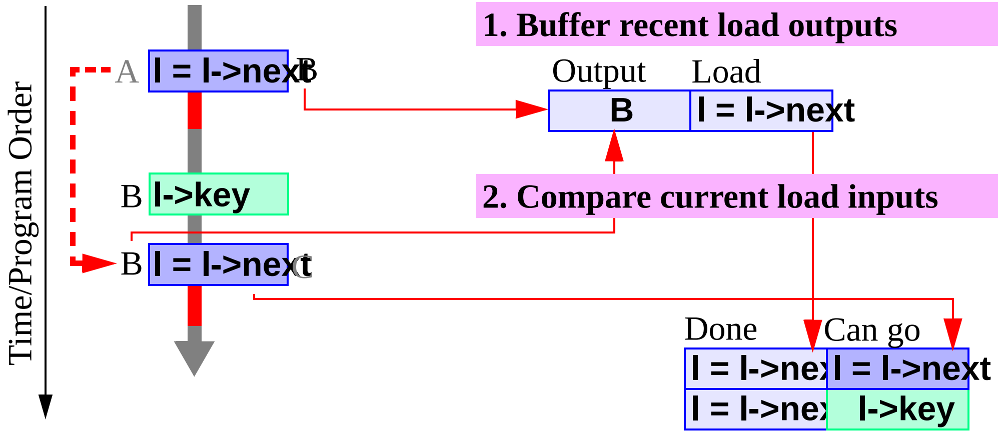
Mechanics I - Overview

Step 1. Examine running program, learn dependences
Step 2. Use dependences to launch prefetches

Mechanics II - Learning Dependences

Establish (dynamic) dependence between (static) loads

Use values exchanged to do this

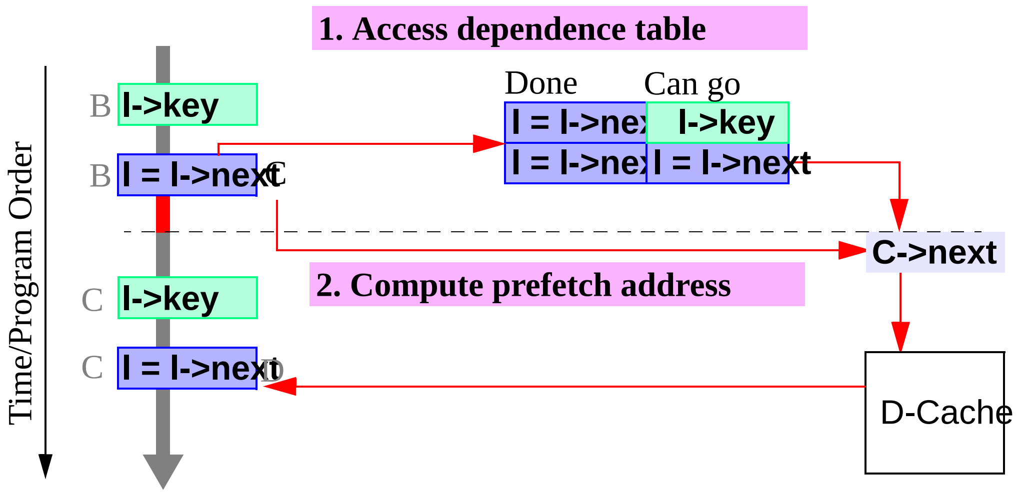


1. Buffer recent load outputs

2. Compare current load inputs

Parameters (see paper)
1. How many outputs can we buffer? (64 is enough)
2. How many deps can we remember? (256)

Mechanics III - Prefetching



1. Access dependence table

Done

Can go

B **l->key**

B **l = l->next** C

l = l->next **l->key**

l = l->next **l = l->next**

C->next

2. Compute prefetch address

C **l->key**

C **l = l->next** D

D-Cache

Time/Program Order

Issues/Lessons: (see paper)

1. Don’t contend for L1 ports, queue prefetches
2. Keep prefetch chains short
3. Prefetch into a small buffer?

Evaluation - Benchmark Programs

Olden Benchmarks:

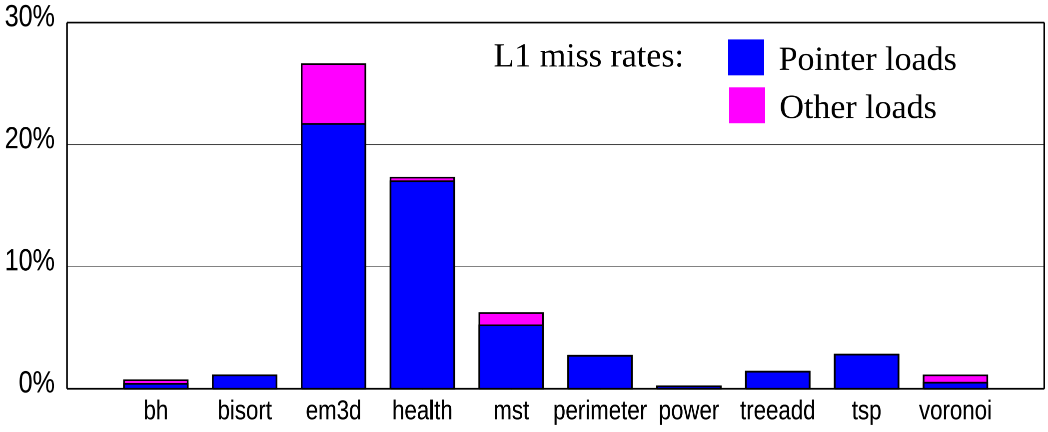
- Scientific simulations: Barnes-Hut, EM3D
- Models/solvers: Health, Power
- Graphics utilities: Perimeter, Voronoi
- Other: Bisort, MST, TSP, Treeadd (toy)

Data structures:

- Lists: EM3D, MST, Health, TSP
- Binary Trees: Bisort, Treeadd, TSP, Voronoi
- k-ary Trees: Barnes-Hut, Health, Power, Perimeter

Characterization I - Pointer Load Behavior

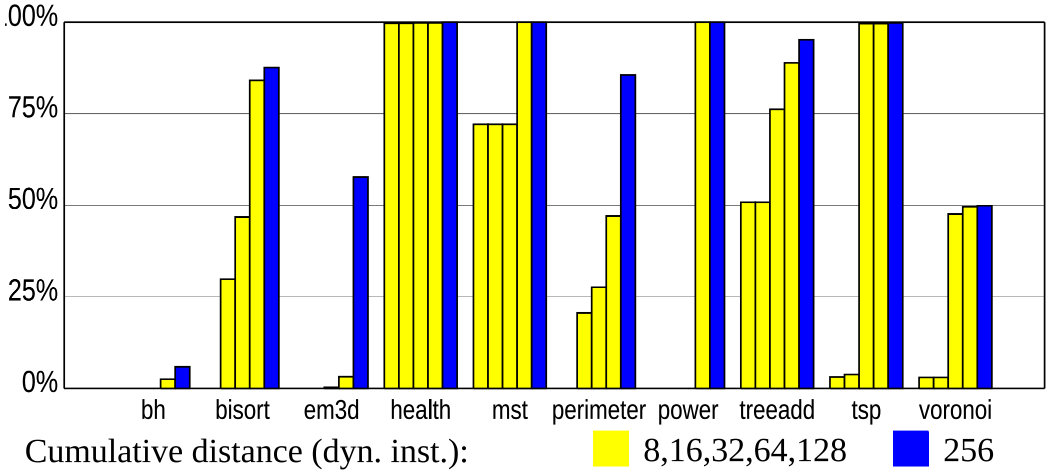
Are pointer loads a problem?



Yes: Account for the majority of data cache misses

Characterization II - Pointer Load Distances

Is there enough distance between pointer loads?



Overall: yes and no

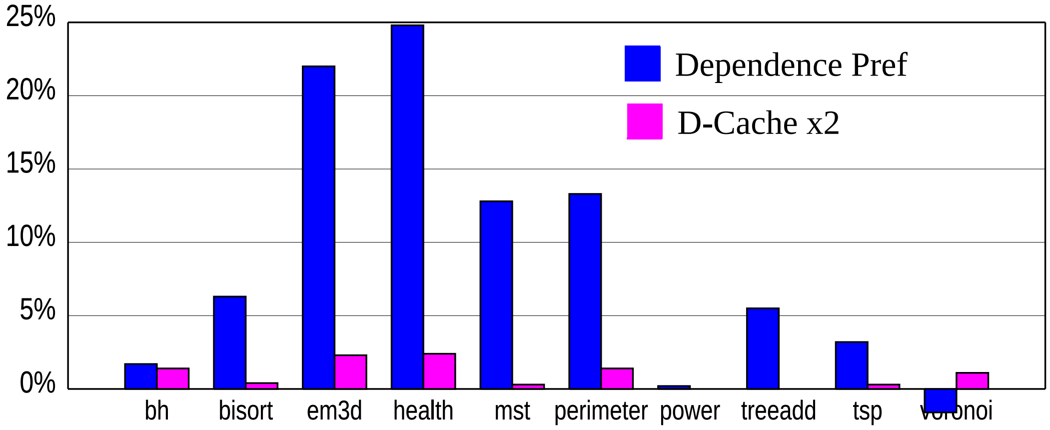
Future: prefetch LDS in short distance situations

Speedups

Base machine: 4 wide, 64 OOO, 32KB Data cache

Prefetch: 256 dependences/buffer 64 load outputs

D-Cache x2: 64KB cache, prefetch using spatial locality



Scaling conventional memory structures doesn’t work

Dependence Based Prefetching Works

Summary

LDS Double Trouble

Unpredictable addresses → use scheduling tricks

Serialized latencies → tricks better be good

Dependence Based Prefetching

Don’t predict addresses

Use dependences to predict loads, compute addresses

Aggressive scheduling without window restrictions

Better use of resources than larger cache

Other Results - Diagnostics

- **Miss coverage**:
  Would-be misses hidden: Fully: ~20%, Partially: ~60%
  → Not enough work between pointer loads (future)

- **Prefetch utilization**:
  Prefetched blocks used: ~80%

- **Bandwidth overhead**:
  L1: ~15%, L2: ~5%
  → Fetches converted to prefetches

Dependences give accurate address predictions

Can We Do This in Software?

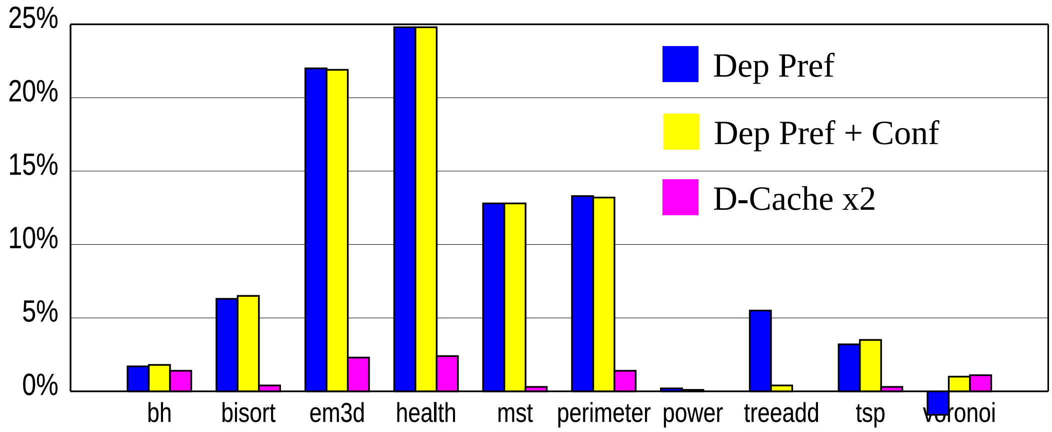
Yes [e.g., Mowry & Luk, ASPLOS ‘96]

+ Don’t have to build anything

Our solution

+ No explicit overhead
+ Can potentially prefetch sooner → hide more latency
+ Adapt to dynamic behavior
+ Easy path for existing software

Adding Confidence

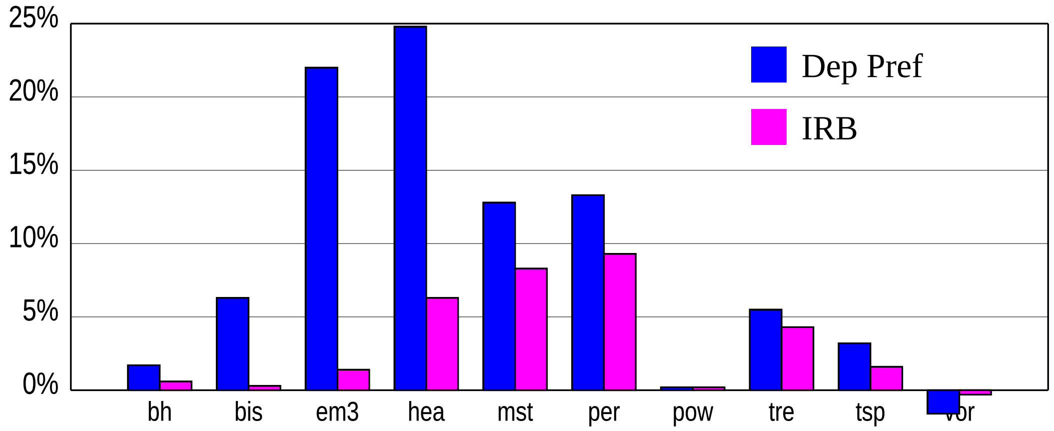


- Learn whether blocks prefetched by load used or not
- Simple mechanism (2-bit counters, stop at 0)
- Eliminates pathologies/unlearns bad prefetching

Prefetching Recurrent Loads Only

Simpler ways for prefetching self recurrent loads only

[e.g., IRB: Mehrotra & Harrison]



- Effective on simple structures: treeadd, perimeter, mst
- Potentially simpler to implement

Improving Virtual-Function-Call Target Prediction via Dependence-Based Pre-Computation

ICS 1999

Amir Roth, Andreas Moshovos and Guri Sohi

amir,sohi@cs.wisc.edu
moshovos@ece.nwu.edu

Computer Sciences Department
University of Wisconsin-Madison

Introduction

Goal: Reduce branch/target mispredictions

Idea: Dependence-Based Pre-Computation

- Supplement conventional prediction
- Pre-compute selected targets/branch outcomes
  - o Identify instructions that compute targets/branches
  - o Speculatively pre-execute these instruction sequences
  - o Use results as predictions
- This work: Virtual-Function-Call (V-Call) targets
  - o Proof of concept
  - \+ Simple implementation

Overview: Problem and Technique

Why do conventional predictors mispredict?

```
for (i = 0; i < ASIZE; i++)
    if (a[i]->valid == TRUE)
        print(a[i]);
```

They rely on expressed correlation (which may not exist)

- Local: **a[i]->valid == TRUE** using **a[i-1]->valid == TRUE**?
- Global: **a[i]->valid == TRUE** using i < ASIZE?

No Correlation? Use Pre-Computation

- Identify branch computation: **a[i]->valid == TRUE**
- Using **a**,**i** as inputs, pre-compute and store the result
- Use stored result as a prediction

+ No correlation necessary!

Virtual Function Calls (V-Calls)

Use: Polymorphism (C++/Java)

- Multiple dynamic function targets from single static call site
- Object type selects target at runtime

C++ types

```
class Base
    virtual int Valid();
    virtual void Print();
```

```
class Derived : Base
    int Valid();
    void Print();
```

Statically: one call site

```
for (i = 0; i < ASIZE; i++)
    if (a[i]->Valid())
        a[i]->Print();
```

a[0]->Base::Valid()
a[0]->Base::Print()
a[1]->Derived::Valid()
a[1]->Derived::Print()

Dynamically: multiple targets

Conventional V-Call Target Prediction

BTB’s (Branch Target Buffers) don’t work

- Single target per static call (need multiple)

Correlated (path-based) BTB’s are better

- Target history index [Driesen&Hoelzle ISCA97,98]

```
for (i = 0; i < ASIZE; i++)
    if (a[i]->Valid())
        a[i]->Print();
```

- Local: **a[i]->Valid()** using **a[i-1]->Valid()**? No (different object)

\+ Global 1: **a[i]->Print()** using **a[i]->Valid()**? Yes (same object)

- Global 2: **a[i]->Valid()** using **a[i-1]->Print()**? No (different object)

There is room for improvement!

Dependence-Based Pre-Computation

Idea: Watch the program and imitate

Three step process:

1. Identify and cache relevant instruction sequences
2. Speculatively instantiate with appropriate inputs
3. Match pre-computed results with predictions (challenge)

Why V-Calls?

\+ Simple dependence chain makes steps 1+2 easy

Conventional V-Call Target Prediction

BTB’s (Branch Target Buffers) don’t work

- Single target per static call (need multiple)

Correlated (path-based) BTB’s are better

- Target history index [Driesen&Hoelzle ISCA97,98]

```
for (i = 0; i < ASIZE; i++)
    if (a[i]->Valid())
        a[i]->Print();
```

- Local: **a[i]->Valid()** using **a[i-1]->Valid()**? No (different object)
+ Global 1: **a[i]->Print()** using **a[i]->Valid()**? Yes (same object)
- Global 2: **a[i]->Valid()** using **a[i-1]->Print()**? No (different object)

There is room for improvement!

Conventional V-Call Target Prediction

BTB’s (Branch Target Buffers) don’t work

- Single target per static call (need multiple)

Correlated (path-based) BTB’s are better

- Target history index [Driesen&Hoelzle ISCA97,98]

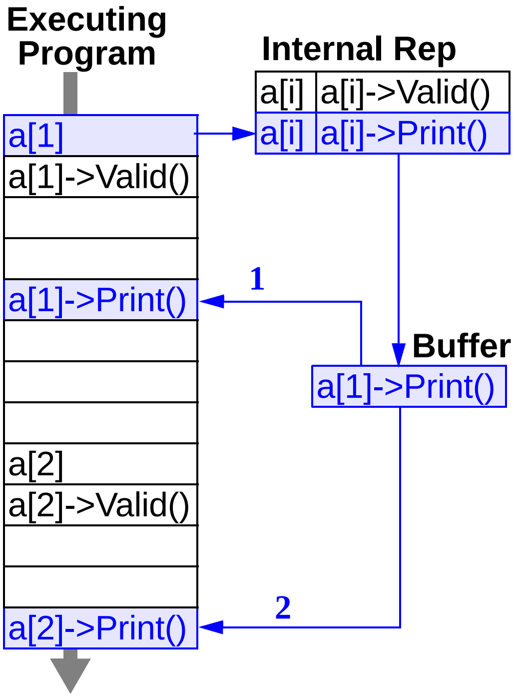
```
for (i = 0; i < ASIZE; i++)
    if (a[i]->Valid())
        a[i]->Print();
```

- Local: **a[i]->Valid()** using **a[i-1]->Valid()**? No (different object)
+ Global 1: **a[i]->Print()** using **a[i]->Valid()**? Yes (same object)
- Global 2: **a[i]->Valid()** using **a[i-1]->Print()**? No (different object)

There is room for improvement!

One Problem

Pre-computation and fetch/prediction are in a race



Pre-computation wins? Great

Prediction wins? Problems

1. Ineffectiveness/Waste

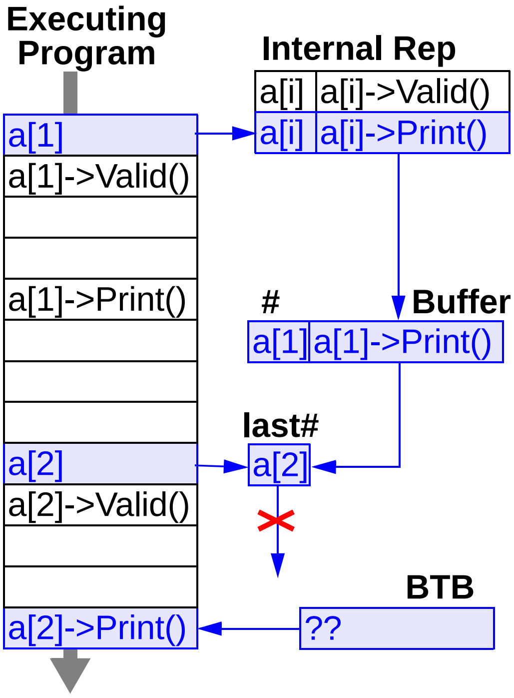
- Late pre-comps don’t help
- Pre-computed for nothing

2. Introduced Mispredictions

- **a[1]->Print()** may mess up **a[2]->Print()** prediction

Preventing Introduced Mispredictions

Idea: Invalidate a[1] pre-comps when a[2] is fetched



Mechanism

- Tag pre-comp with **a[i]** seq#
- Pre-comp good if seq# is most recent for **a[i]**

How it works

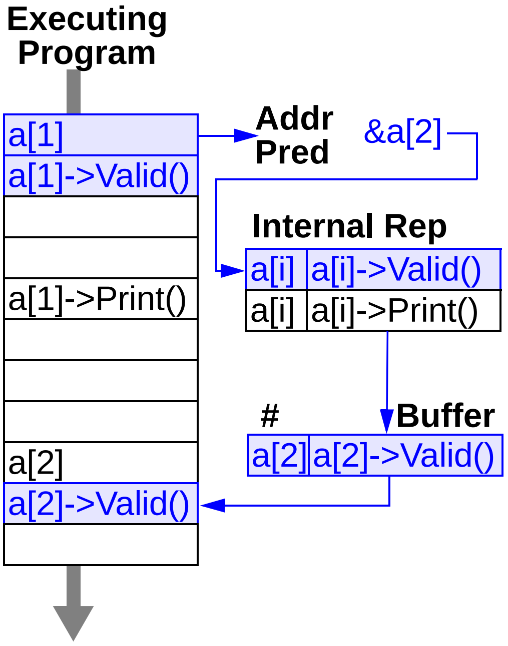
- **a[1]->Print()** pre-comp seq# is **a[1]**
- At **a[2]->Print()** prediction time, most recent seq# is **a[2]**
- Pre-comp with seq# **a[1]** stale (use BTB)

(see paper for more details)

Ineffectiveness: Lookahead Pre-Computations

Problem: Not enough distance from a[i] to a[i]->Valid()

Idea: Exploit distance from a[i-1] to a[i]->Valid()



Executing Program

a[1]

a[1]->Valid()

a[1]->Print()

a[2]

a[2]->Valid()

Addr Pred

&a[2]

Internal Rep

a[i]	a[i]->Valid()
a[i]	a[i]->Print()

**Buffer**

a[2]	a[2]->Valid()

Mechanism

- **a[i]** usually address predictable
- Using **&a[i-1]**, predict **&a[i]**
- Launch **a[i]->Valid()**
- Incorporate into seq# scheme
  (see paper)

Two schemes

- **Lookahead**: address prediction
- **Simple**: no address prediction

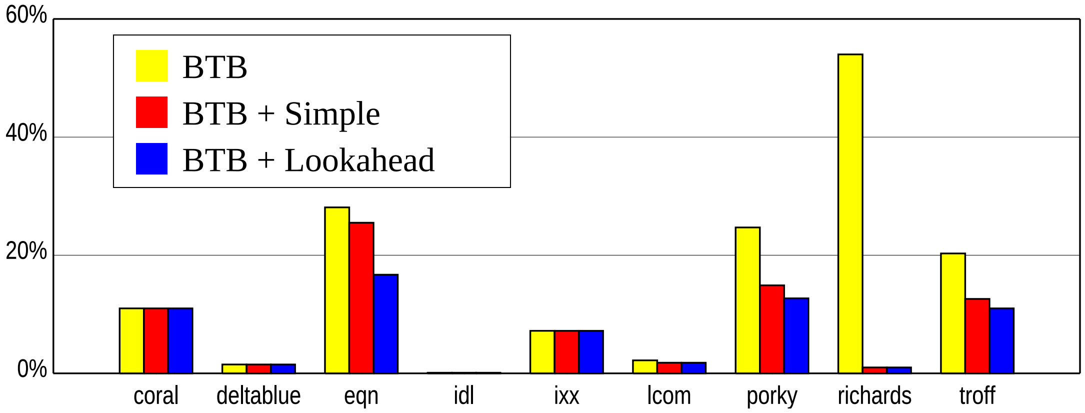
Experiments

Benchmarks: OOCSB (C++)

Simulations: SimpleScalar [MIPS, GCC]

- 4-wide superscalar, 5-stage pipe
- Speculative OOO-issue, 64 instructions in-flight
- 64 KB L1 D-Cache, 512KB L2 U-Cache
- Branches: 8K-entry combined 10-bit GSHARE + 2-bit counters
- Target prediction:
  - BTB: 2K-entry, 4-way associative
  - PATH: BTB + 2K-entry, DM, 2-level BTB, 3 target history

Numbers: BTB base predictor


Misprediction Rates

richards, eqn, lcom, porky, troff:

\+ **Simple** handles long distance cases (**a[i]->Print()**)

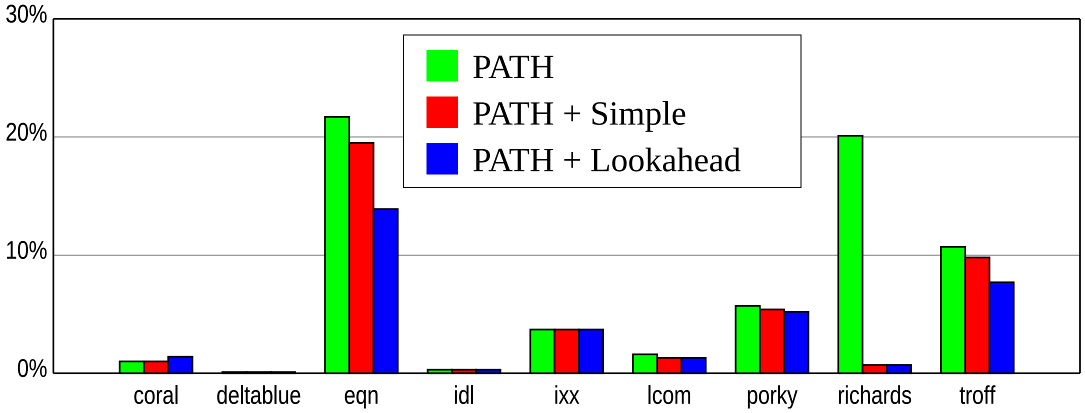
\+ **Lookahead** handles short distance cases (**a[i]->Valid()**)

others:

\- **Simple**: short distances, **lookahead**: unpredictable addresses

---

Numbers: PATH base predictor


Misprediction Rates (**NOTE: change in scale**)

overall:

- **PATH** handles correlated cases (**a[i]->Print()**)

richards, eqn, troff:

\+ **Lookahead** helps uncorrelated (**a[i]->Valid()**)

Numbers: Explanations

What about overall performance?

- V-Call rate low in absolute terms (1 per 200-1000 instructions)
- Performance improves by 0-2%

Sometimes (coral) more harm than good

- Lookahead pre-computation relies on address prediction
- Wrong address prediction? Wrong pre-computation

\+ Not common

Summary

Dependence-Based Pre-Computation

\+ Can be used to augment branch/target prediction

\+ Succeeds where statistical correlated prediction fails

- Similar technique prefetches linked structures [ASPLOS98]
  (where statistical address prediction also fails)

Closely related

- Branch Flow Window [Farcy et.al., MICRO98]

Can be generalized to handle all branches