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

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
  - Identify instructions that compute targets/branches
  - Speculatively pre-execute these instruction sequences
  - Use results as predictions
- This work: Virtual-Function-Call (V-Call) targets
  - Proof of concept
  - Simple implementation
Overview: Problem and Technique

Why do conventional predictors mispredict?

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!
Talk Outline

- Introduction
- Virtual Function Calls (V-Calls)
- Dependence-Based Pre-Computation
- Numbers
- Summary
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

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

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

Statically: one call site

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

Dynamically: multiple targets

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

```plaintext
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
Pre-Computation Mechanics

1. Isolate relevant instructions
   - Build internal representation
   - Work backwards from call
   - Track dependences (names)

2. Pre-Compute
   - Start from $a[i]$
   - Unroll representation
   - More? [ASPLOS 98]

3. Use Pre-Computation
   - Buffer pre-comp result
   - Pick up stored result
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
   - $a[1] \rightarrow \text{Print()}$ may mess up
     $a[2] \rightarrow \text{Print()}$ prediction
Preventing Introduced Mispredictions

Idea: Invalidate \texttt{a[1]} pre-comps when \texttt{a[2]} is fetched

**Mechanism**
- Tag pre-comp with \texttt{a[i]} seq#
- Pre-comp good if seq# is most recent for \texttt{a[i]}

**How it works**
- \texttt{a[1] -> Print()} pre-comp seq# is \texttt{a[1]}
- At \texttt{a[2] -> Print()} prediction time, most recent seq# is \texttt{a[2]}
- Pre-comp with seq# \texttt{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 \( a[i]->Valid() \)

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 super scalar, 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

<table>
<thead>
<tr>
<th></th>
<th>BTB</th>
<th>BTB + Simple</th>
<th>BTB + Lookahead</th>
</tr>
</thead>
<tbody>
<tr>
<td>coral</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>deltablue</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eqn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>idl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ixx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lcom</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>porky</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>richards</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>troff</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 ($a[i] \rightarrow \text{Print()}$)

richards, eqn, troff:
- Lookahead helps uncorrelated ($a[i] \rightarrow \text{Valid()}$)
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