Understanding the Backward Slices of Performance Degrading Instructions

Craig Zilles and Guri Sohi

University of Wisconsin - Madison

International Symposium on Computer Architecture

June, 2000

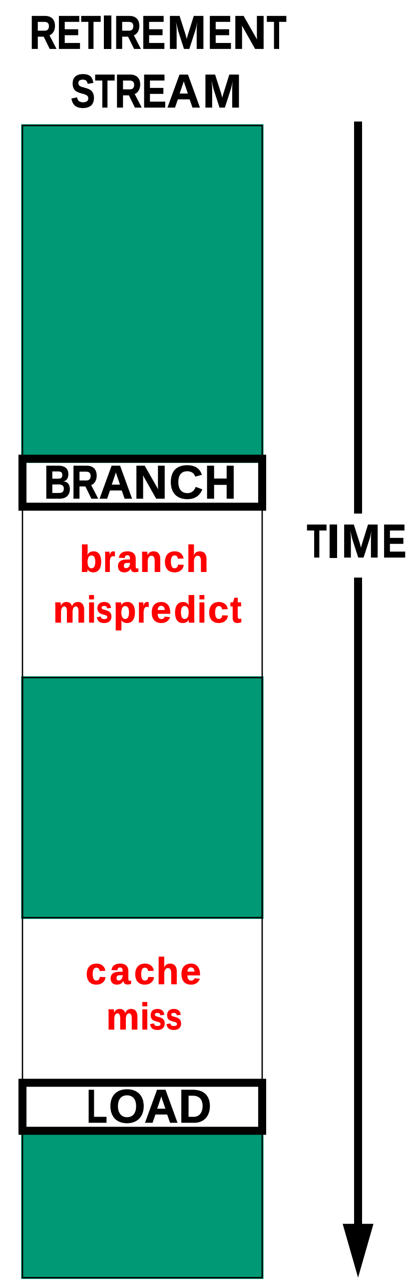
Motivation

PROCESSORS ACHIEVE ONLY A FRACTION OF PEAK PERFORMANCE ON MANY PROGRAMS

- *Performance Degrading Events (**PDE**)*
  - **branch mispredictions**
  - **cache misses**

LARGER CACHES AND PREDICTORS

- *handle easy cases*
- *concentrates **PDE**'s to a fraction of "problem" static instructions*
  - **UNCORRELATED (DATA-DEPENDENT) BRANCHES**
  - **HASH TABLE LOOKUPS AND POINTER CHASING**

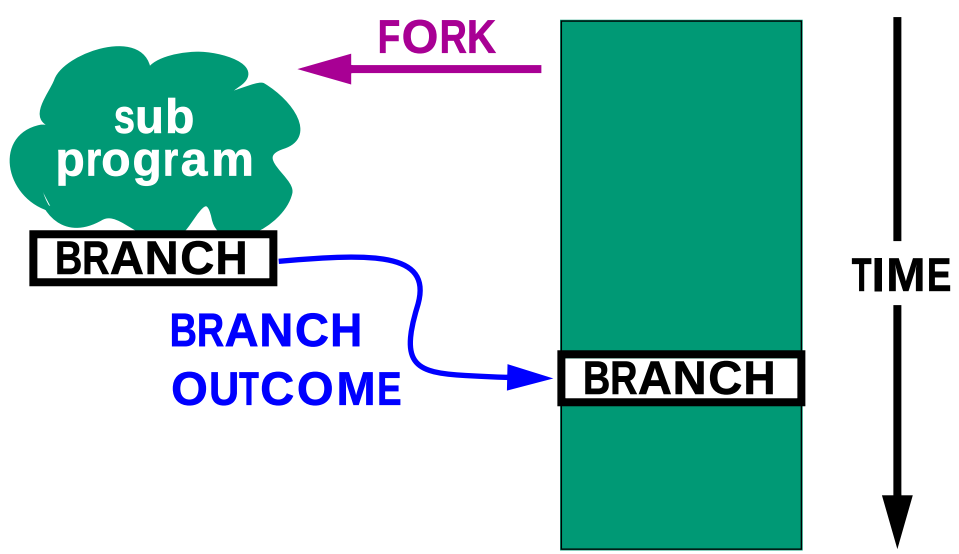


Motivation, cont.

PROGRAM BEHAVIOR IS DETERMINISTIC →

build predictors which use the program!

Pre-execution



AVOID MISPREDICTION

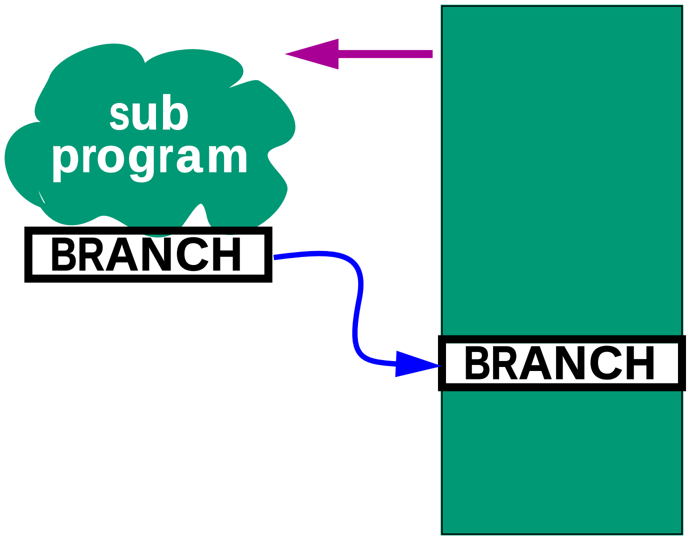
Pre-executed sub-program feeds prediction for branch fetched by the main thread

Pre-fetch memory similarly

Only pre-execute instructions which defy normal predictors

Motivation, cont.

THE EFFECTIVENESS OF PRE-EXECUTION IS DETERMINED BY THE SUB-PROGRAM



- *Sub-program must* **enable faster execution** *of the problem instruction*
- *Sub-program* **size determines "overhead"**

How can we build sub-programs to minimize their size while maintaining accuracy

Overview

- *MOTIVATION*
- PROGRAM SLICING
- EXPERIMENT OVERVIEW
- METHODOLOGY
- CONSERVATIVE SLICES
- EXAMPLE SPECULATIVE OPTIMIZATIONS
- ADDITIONAL OBSERVATIONS
- CONCLUSIONS

Program Slicing

THE SUB-PROGRAM INCLUDES ONLY THE SUBSET OF INSTRUCTIONS WHICH CAN INFLUENCE THE PROBLEM INSTRUCTION.

```
lda       r8,-8432(r29)
cmoveq    r18,r1,r0
ldl       r1,-19952(r29)
s4addq    r16,r8,r8
stl       r31,0(r8)
and       r4,r5,r5
sll       r5,4,r5
ldq       r23,-19408(r29)
ldl       r27,-19944(r29)
addl      r1,1,r1
addq      r0,r5,r5
bis       r31,r31,r0
stl       r1,-19952(r29)
ldq       r7,8(r5)
```

BACKWARD SLICE

```
cmoveq    r18,r1,r0
and       r4,r5,r5
sll       r5,4,r5
addq      r0,r5,r5
ldq       r7,8(r5)
```

follow dependences backward from criterion instruction

both data and control

Criterion Instruction

INITIAL CHARACTERIZATION OF SLICES

Categorization of instructions in the slice

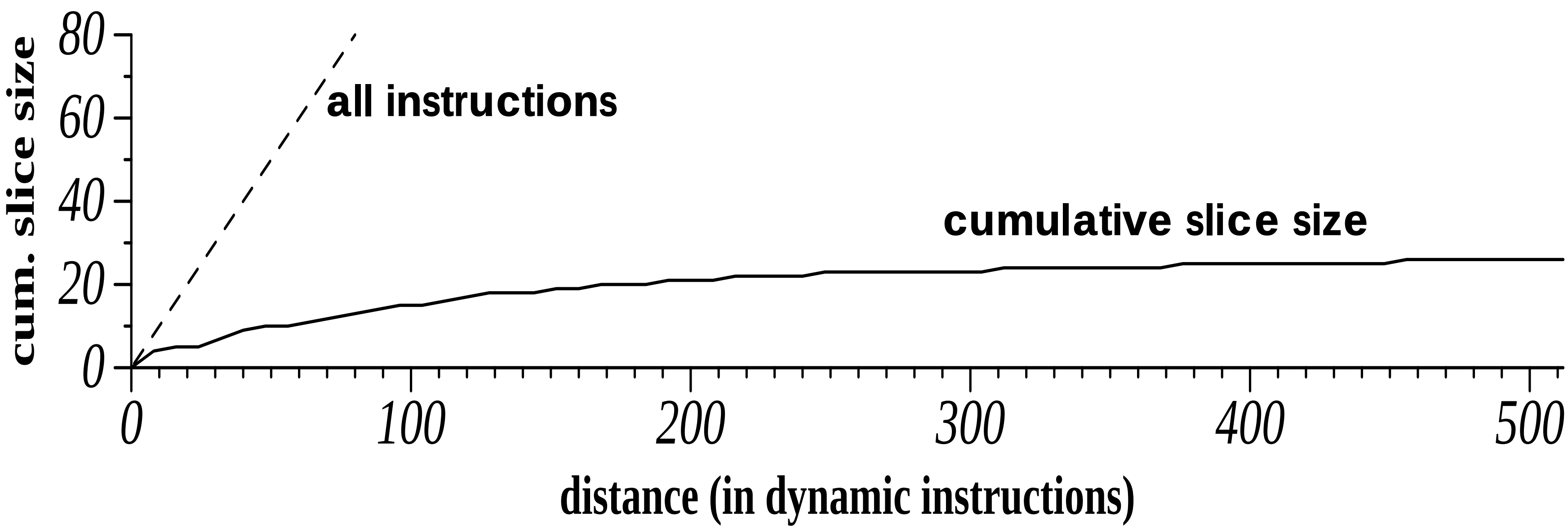
- *largest contributors:*
  - **control and memory dependence resolution**
  - NOT *dataflow*

Exploiting well-known phenomena to reduce slice size

- *highly-biased branches*
- *stability of memory dependences*

Methodology

- SPEC95 *integer benchmarks*
  - **Alpha architecture, optimized -O4**
- *Profiled to identified "problem" static instructions*
  - **Frequently caused mispredictions or cache misses**
- *Generated* **ASSEMBLY-LEVEL** *slices*
  - **Looked at the 512 dynamic instructions leading to the criterion**
  - **Removed NOPS, and SP/GP computation dependences**



Conservative Slices

PROGRAMS HAVE AMBIGUOUS MEMORY DEPENDENCES AND COMPLEX CONTROL FLOW → CONSERVATIVELY CONSTRUCTED SLICES CAN BE LARGE

- *50% of program necessary to compute all store addresses*
- *80% of program necessary to resolve all branches*

SOLUTION: EXPLOIT THE FACT THAT SLICES ONLY PROVIDE HINTS

- **construct speculative slices**
- **assume common-case behavior**
  - **profiling is required to detect the common-case**

TWO EXAMPLE OPTIMIZATIONS:

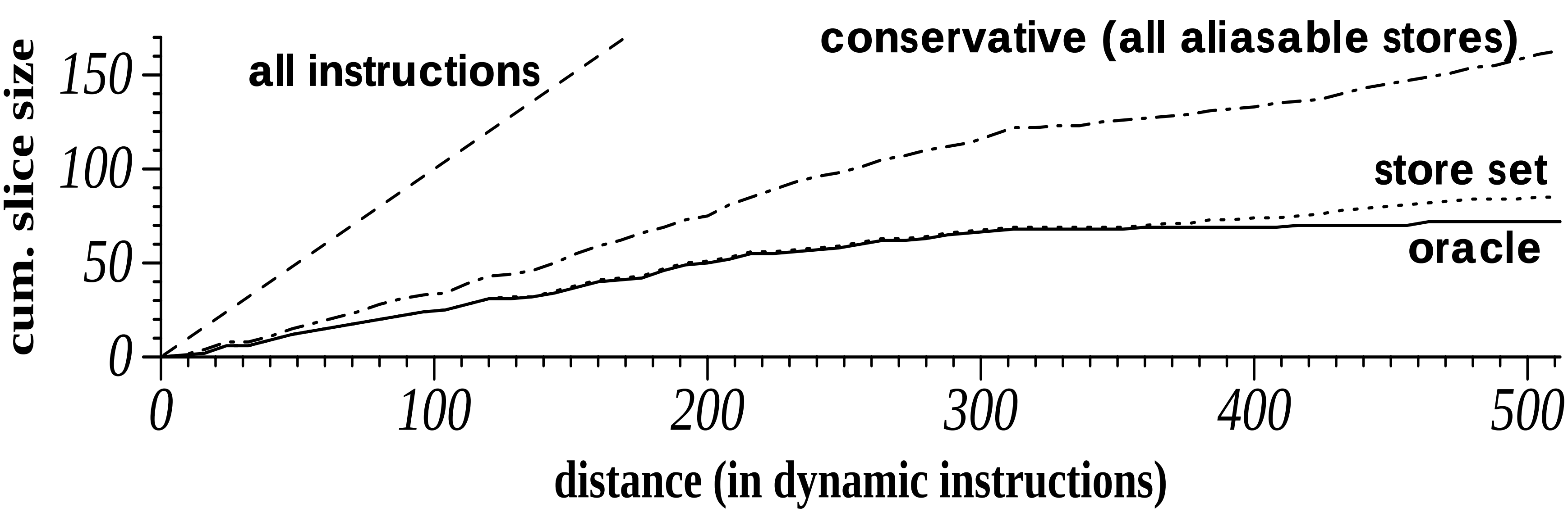
- **both targeting ambiguous memory dependences**

Speculative Optimizations

A CONSERVATIVE SLICE MUST COMPUTE THE ADDRESS FOR EVERY STORE WHICH COULD ALIAS WITH A LOAD IN THE SLICE

THE SET OF MEMORY DEPENDENCES ACTUALLY REALIZED IS A SUB-SET OF THOSE WHICH ARE POSSIBLE

- Profile to identify the store sets
- Only compute store addresses for these stores



Slices built using store sets approximate size of oracle slices

Speculative Optimizations, cont.

When memory dependences exist, often the load communicates with the most recent store from its store set.

- **much like communication through registers**

Exploit to reduce slice size:

1. Assume a communication pattern (imprecise transformation)

2. Remove load and store from the slice

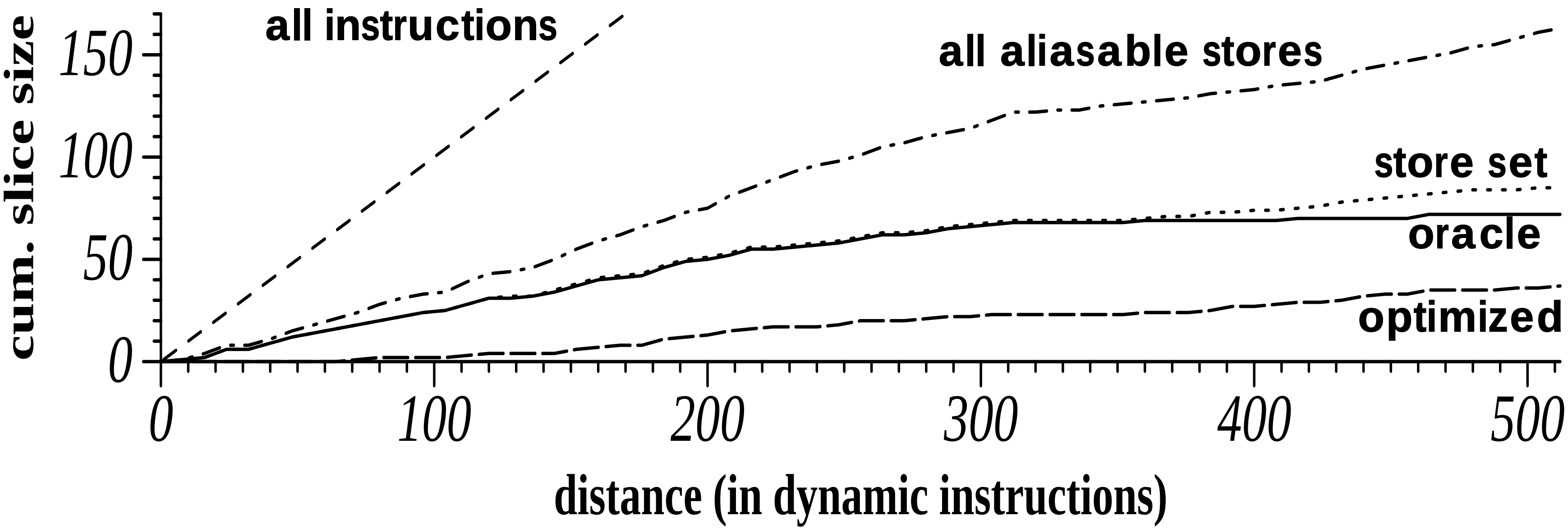
STORE R9 -> 0(R17)
ADD R10, R11 -> R12
STORE R12 -> 0(R18)
LOAD 0(R19) -> R13
ADD R13, R14 -> R15
....

~~STORE R9 -> 0(R17)~~
ADD R10, R11 -> R12
~~STORE R12 -> 0(R18)~~
~~LOAD 0(R19) -> R13~~
ADD R12, R14 -> R15
....

Speculative Optimizations, cont.

STATIC LOADS ARE HIGHLY BIASED WITH RESPECT TO THIS BEHAVIOR

- **Mis-speculation can be avoided**
- **Easily profiled to classify the dependences**



Removing such load-store pairs can further reduce slice size with little affect on accuracy

Reduced address sub-slice to 1/4 of conservative size

Additional Observations

Often data dependences are clustered near criterion

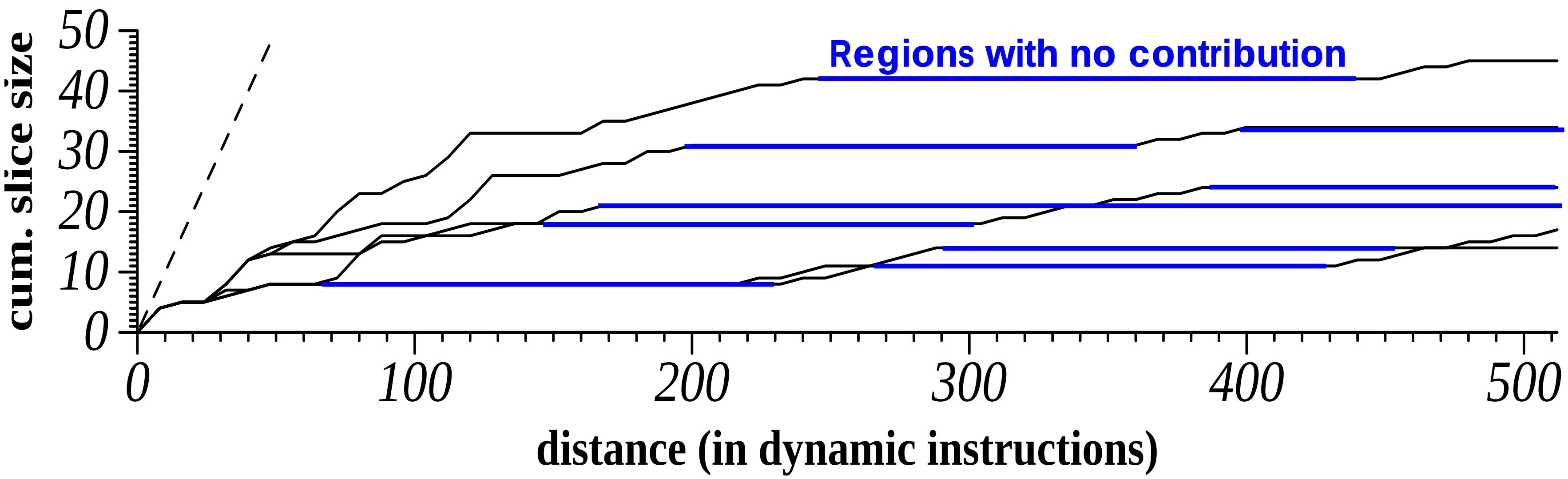
- possibly influences pre-execution mechanisms

Not uncommon for slices to overlap

- create a single “multi-slice”

Slices are bursty

- due to program structure



Additional Observations

False control dependences:

control dependent regions are part of slice, but all paths from the branch contribute to the slice equivalently.

Common case: Conditional function call

```
int A = ....;                 Function() {
if (B) {                          save A;
    Function();                   ....
}                                 restore A;
CRITERION → if (A)            }
```

Other case: Code replication

Currently refining infrastructure to handle these cases

Summary

PRE-EXECUTION: GENERAL TECHNIQUE FOR HANDLING "PROBLEM" INSTRUCTIONS

- Use the program to predict the program
- Requires small, accurate slices

SPECULATIVE SLICES:

- Exploit common-case behavior to reduce slice size while maintaining accuracy
- Some slices can be reduced to less than 10% of the 512 instructions preceding the criterion while maintaining greater than 95% accuracy
  - Much future work to be done
- Requires sophisticated profile information