SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization

James Tuck, Wonsun Ahn, Luis Ceze, Josep Torrellas

CESR, NC State University
jtuck@ncsu.edu

University of Washington
luisceze@cs.washington.edu

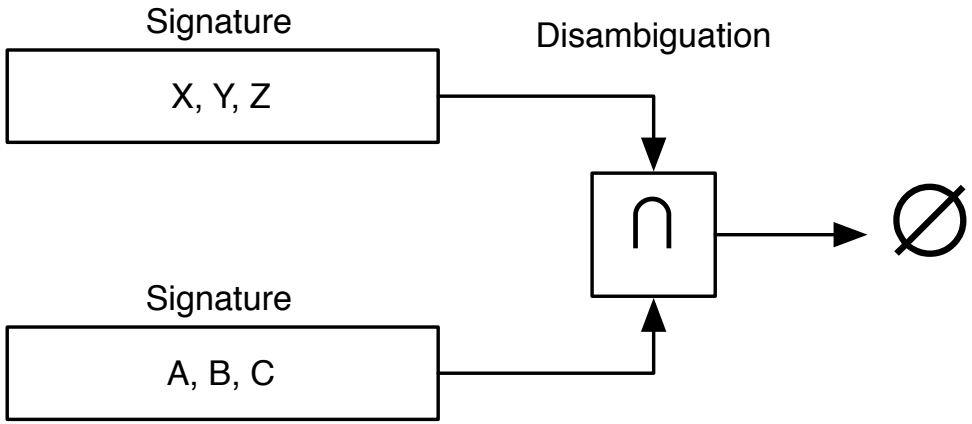
University of Illinois
{torrellas, dahn2}@cs.uiuc.edu

Introduction

- Many code analysis techniques need to determine if two variables have the same address --- a.k.a. **disambiguation**
  - Various code optimizations
  - Speculative parallelization and synchronization
  - Debugging
- The cost and frequency of disambiguation has led to hardware support

Disambiguation using signatures

- A **signature** is a hardware register that can represent a set of addresses
- Simple logical operations on signatures can disambiguate two sets of addresses but false positives are possible



- **Signatures are typically managed by hardware with, at most, a minimal software interface**

Proposal: SoftSig

- Expose a Signature Register File to software through a general, rich ISA
- Allow software to decide how best to use signature registers
  - What memory accesses should be collected in a signature
  - What addresses a signature should be disambiguated against
  - How to best manage each signature
- Case study: MemoiSE --- signature enhanced function memoization

Outline

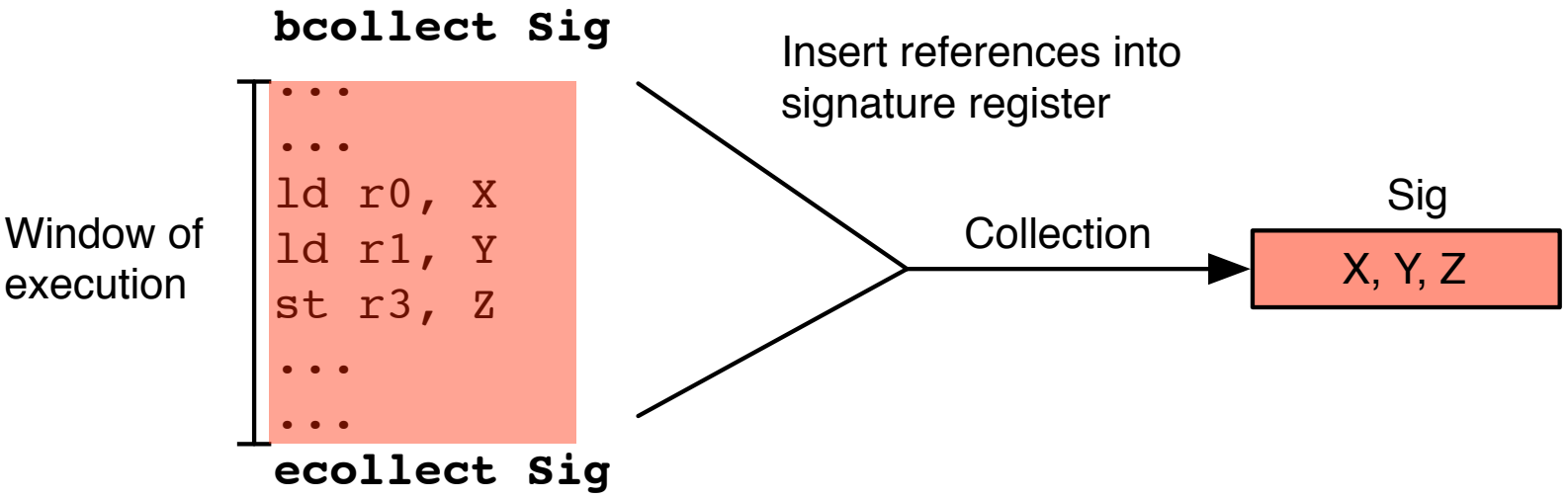
- **SoftSig Interface**
- SoftSig Architecture
- MemoiSE Algorithm
- Evaluation

SoftSig Interface

- Add a Signature Register File (SRF) and ISA extensions to a single core
- ISA extensions to support operations on Signature Registers (SRs)
  - Collection
  - Disambiguation
  - Conflict detection
  - Other unary and binary operations on signatures

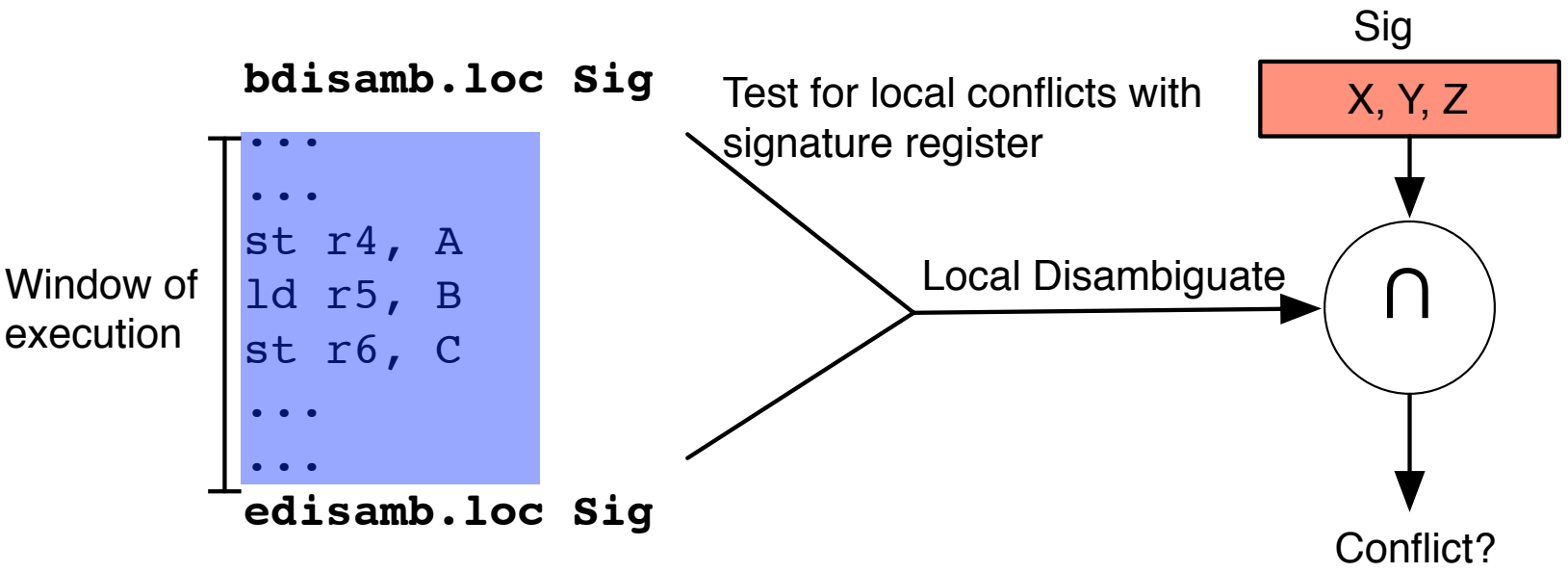
Collection

- **Software** specifies a window of program execution whose memory accesses must be recorded in a signature



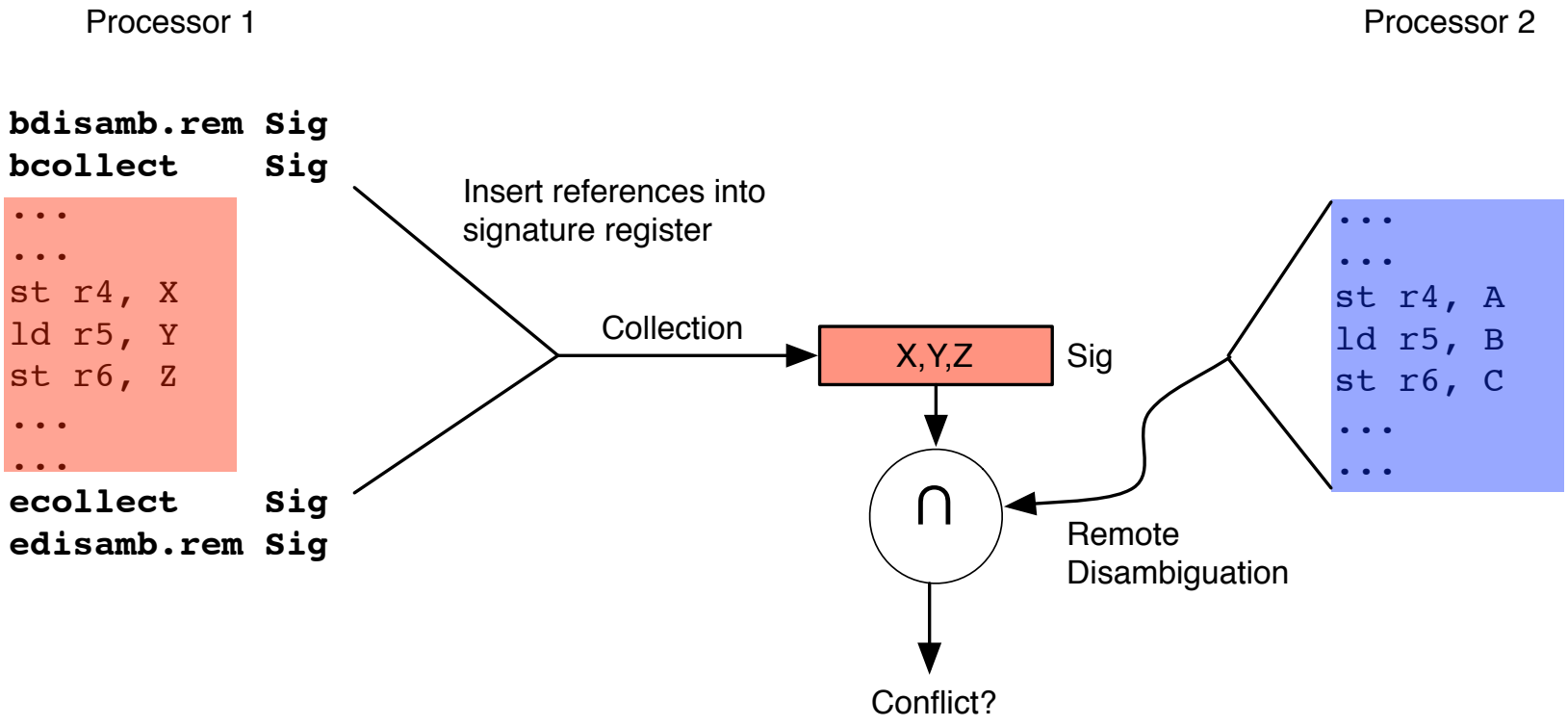
Disambiguation

- **Software** specifies that a given signature should be compared to the dynamic memory stream: either local, remote, or both



Remote disambiguation

- Especially useful while collecting a signature --- provides transaction-like conflict detection for a region of execution



Conflict detection

- **Software** tests for conflicts, and specifies what action to take on a conflict
  - The action is not necessarily a rollback, like in TLS or TM
- **Software** must cope with signature imprecision
  - False positives **must not** lead to an incorrect analysis or invalid optimization

➡ **Collection, Disambiguation, and Conflict detection on signatures can be composed in a variety ways for different uses**

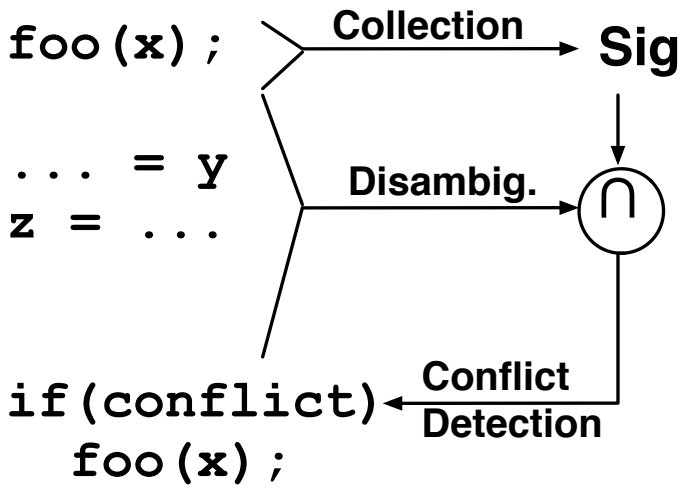
Other instructions for manipulating SRs

- Storing, loading, moving, clearing
- Explicit membership tests
- Intersection, Union, Member insertion
- Registering exceptions on conflicts

Example: Signature Enhanced Memoization

- Use signatures to record implicit inputs and side effects of a function call

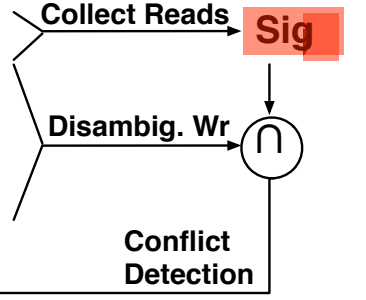
```
foo(x);

... = y
z = ...


foo(x);
```



Example: Checkpoint-based Optimization

- Move an expression out of a loop that is likely invariant
- The checkpoint before the loop preserves thread state
- Collect the expression into a signature
- Look for modification of the inputs/ outputs during execution
- Rollback and execute non-optimized code on a conflict

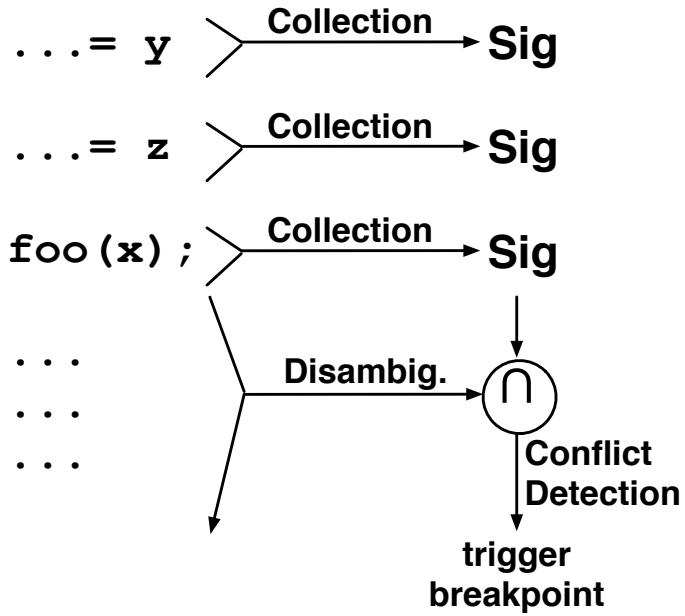
```
checkpoint()
reg = <expr>
while(...) {
    ...
    ... = reg
    ...
}
if (conflict)
  rollback(no_opt:)
else
  goto done:

no_opt:
  <original loop>
done:
```



Example: Debugging

- Detecting updates to a given memory address is a common debugging task

- Use SoftSig to support efficient data watchpoints:



SoftSig operations are widely applicable

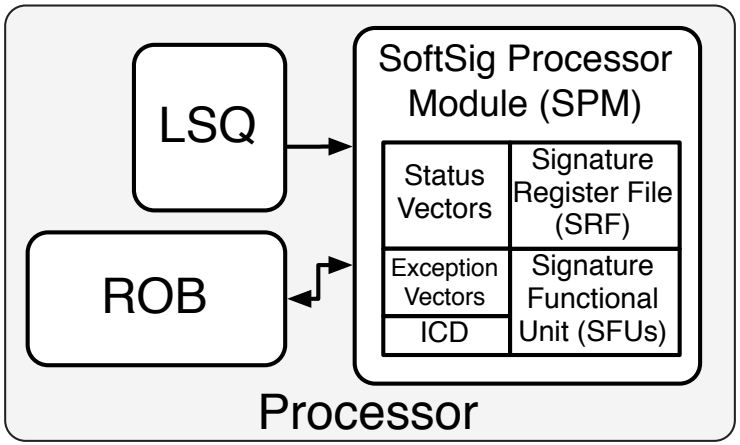
- Signatures provide efficiency by operating on sets of addresses
- Collection, disambiguation, and other operations **compose and nest** in a variety of ways
  - Unlike TM or TLS, collection and disambiguation are flexibly decoupled
- Software decides the best way to use the signatures
- These operations are useful for **both single-threaded and multi-threaded** analysis and optimization

Outline

- SoftSig Interface
- **SoftSig Architecture**
- MemoiSE Algorithm
- Evaluation

SoftSig Architecture

- The SoftSig Processor Module (SPM) provides necessary architectural support
  - Executes SoftSig instructions when they reach the head of the ROB
  - Contains the SRF
  - Functional Units for signature operations
  - In-flight Conflict Detector (ICD) --- supports nested collection and remote disambiguation



SoftSig Architecture Details

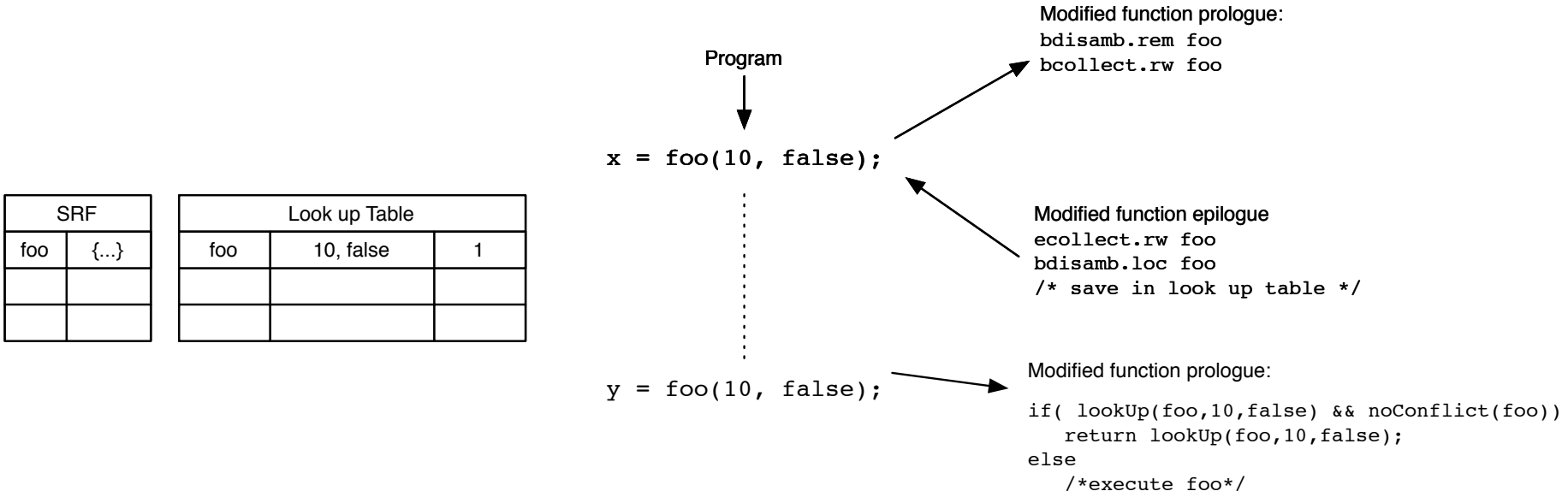
- Interaction with the rest of the processor is minimized by performing operations when they reach the head of the ROB
  - Adds some complexity to remote disambiguation
- Collection and disambiguation occur as each load/store retires
  - No speculative updates of signatures --- important since signatures are costly to update and imprecise
- Signatures are allocated/deallocated on demand by software

Outline

- SoftSig Interface
- SoftSig Architecture
- **MemoiSE Algorithm**
- Evaluation

MemoiSE Algorithm

- Extend prologue and epilogue with SoftSig operations to catch implicit inputs and side-effects



Conflict = modification to an implicit input or output

Outline

- SoftSig Interface
- SoftSig Architecture
- MemoiSE Algorithm
- **Evaluation**

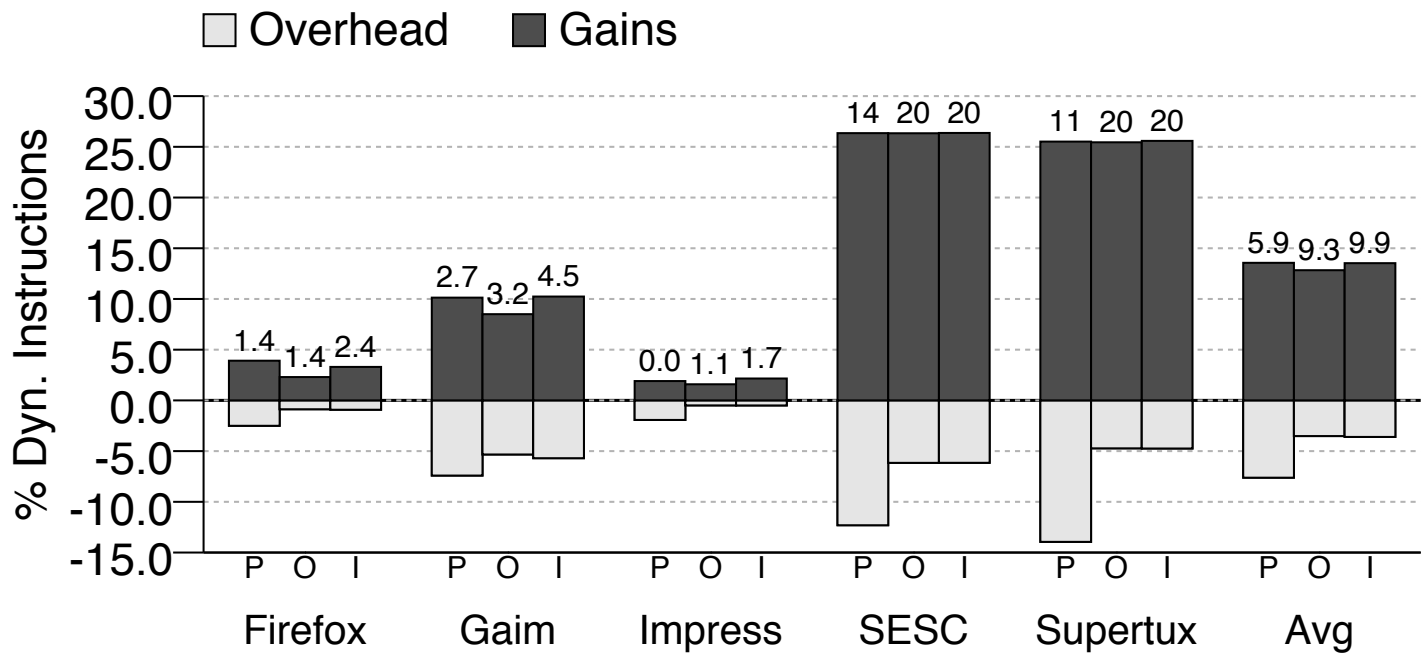
Experimental Setup

- Implemented MemoiSE in Pin
- Use a SESC-based simulator to model the multiprocessor memory subsystem
  - 6 processors, each with 16, 1 kilobit signature registers, 1 SRF access/cycle
  - 64 KB L1 Cache, 1 cycle access
  - 2 MB Shared L2 Cache, 10 cycle access
  - 500 cycle memory latency
- Applications: Firefox, Gaim, Impress, SESC, Supertux

MemoiSE Environments

- Baseline: no MemoiSE
- Plain (P): MemoiSE applied selectively to most beneficial functions
- Optimized (O): P optimized with smaller lookup tables to reduce overhead
- Ideal (I): Unlimited SRs with no false positives

Reduction in dynamic instruction count



- Reducing lookup tables in O significantly reduces overheads
- Optimized (O) is close to Ideal hardware scenario
- Results for O translate to a 9% improvement in performance

Tuck et al, “SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Otimization,” ASPLOS’08

SoftSig Summary

- Added a Signature Register File and generic operations on signatures to enable a variety of code analyses and optimizations
  - Signatures can be used flexibly by giving software full control
  - Hardware efficiently supports composing and nesting of signature operations
- Used SoftSig to implement MemoiSE, an aggressive function memoization algorithm
- Evaluated SoftSig and MemoiSE on 5 popular applications with an average 9% improvement in performance

SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization

James Tuck, Wonsun Ahn, Luis Ceze, Josep Torrellas

CESR, NC State University
jtuck@ncsu.edu

University of Washington
luisceze@cs.washington.edu

University of Illinois
{torrellas, dahn2}@cs.uiuc.edu

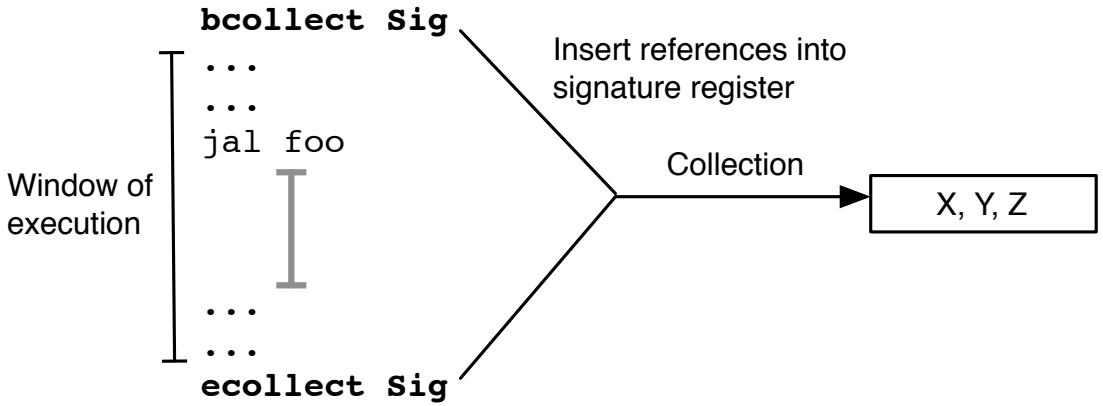
Backup Slides

Signature Registers are unlike GPRs

- SRs are 1 kilo-bit each in SoftSig

  ➡ Costly to read, move, or copy

- SRs **persist** in the SR File as long as needed, even through a function call



Managing persistence of signatures

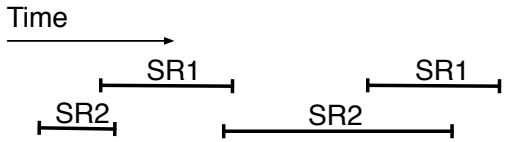
- Static allocation must conservatively assign SRs to potentially overlapping windows of execution

- Dynamic allocation can use SRs that are available at execution time

  ➡ Hardware may preemptively deallocate signatures to make room

  ➡ Need a way of referencing registers dynamically

(a) Static allocation



(b) Dynamic allocation

System must cope with imprecision

- Signatures have an imprecise encoding
  - Conflicts may be reported even when they weren't there (**false positives**)
- Signatures may be silently displaced
  - Optimizations or analysis may fail because of how the SRF is managed

Design Guidelines

1. Minimize SR accesses and copies

2. Manage the SRF through dynamic allocation

3. Imprecision should never compromise correctness

4. Manage signature uses to provide the most efficiency

5. Minimize imprecision and unnecessary conflicts

Related Work

- Other systems using signatures for disambiguation
  - Bulk [Ceze: ISCA'06], Log-TM SE [Yen: HPCA'07], SigTM [ISCA'07]
- Many other proposals have used Bloom filters
  - Scalable LSQ [Sethumadhavan: MICRO'03], Scalable Miss Handling [Tuck, MICRO'06], Jetty [Moshovos: HPCA'01]
- Memoization
  - Fine grained instruction-level re-use [Sodani: ISCA'97,ASPLOS'98], [Lipasti: MICRO'96, ASPLOS'96], [Sastry: FDDO'00]
  - Coarse-grained re-use [Connors: ASPLOS'99, MICRO'00] [Huang: HPCA'99]
  - Software-only function level reuse [Ding: CGO'04]