AXCIS: Accelerating Architectural Exploration using Canonical Instruction Segments

Rose Liu & Krste Asanović

Computer Architecture Group

MIT CSAIL
Simulation for Large Design Space Exploration

- Large design space studies explore thousands of processor designs
  - Identify those that minimize costs and maximize performance

- Speed vs. Accuracy tradeoff
  - Maximize simulation speed up while maintaining sufficient accuracy to identify interesting design points for later detailed simulation
Reduce Simulated Instructions: Sampling

- Perform detailed microarchitectural simulation during sample points & functional warming between sample points
  - SimPoints [ASPLOS, 2002], SMARTS [ISCA, 2003]
- Use efficient checkpoint techniques to reduce simulation time to minutes
  - TurboSMARTS [SIGMETRICS, 2005], Biesbrouck [HiPEAC, 2005]
Reduce Simulated Instructions: Statistical Simulation

- Generate a short synthetic trace (with statistical properties similar to original workload) for simulation
  - Eeckhout [ISCA, 2004], Oskin [ISCA, 2000]
  - Nussbaum [PACT, 2001]
AXCIS Framework

Stage 1 (performed once)

Program & Inputs -> Dynamic Trace Compressor -> CIST (Canonical Instruction Segment Table)

- Machine independent except for branch predictor and cache organizations
- Stores all information needed for performance analysis

Configs

- In-order superscalars:
  - Issue width
  - # of functional units
  - # of cache primary-miss tags
  - Latencies
  - Branch penalty

Stage 2

AXCIS Performance Model -> IPC1, IPC2, IPC3
In-Order Superscalar Machine Model

Parameters

- Branch Pred.
- Fetch
- Issue

Issue width

- ALU
- FPU
- LSU

Latency

Number of units

Completion

Blocking Lcache

Memory

Non-blocking Dcache

Latency

(organization & latency)

Organization & latency

(# primary miss tags)

Size & penalty

Org. & latency
Stage 1: Dynamic Trace Compression

Stage 1 (performed once)

Program & Inputs → Dynamic Trace Compressor → CIST (Canonical Instruction Segment Table)

Config

AXCIS Performance Model

IPC1

IPC2

IPC3

Stage 2
Instruction Segments

Events: (dcache, icache, bpred)

- `addq` (--> hit, correct)
- `ldq` (miss, hit, correct)
- `subq` (--> hit, correct)
- `stq` (miss, hit, correct)

- An instruction segment captures all performance-critical information associated with a dynamic instruction
Instruction Segments

Events: (dcache, icache, bpred)

- addq (--, hit, correct)
- ldq (miss, hit, correct)
- subq (--, hit, correct)
- stq (miss, hit, correct)

- An **instruction segment** captures all performance-critical information associated with a dynamic instruction
Dynamic Trace Compression

- Program behavior repeats due to loops, and repeated function calls
- Multiple different dynamic instruction segments can have the same behavior *(canonically equivalent)* regardless of the machine configuration

- Compress the dynamic trace by storing in a table:
  - 1 copy of each type of segment
  - How often we see it in the dynamic trace
Canonical Instruction Segment Table

<table>
<thead>
<tr>
<th>Int_ALU</th>
<th>CIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq</td>
<td>Segment</td>
</tr>
<tr>
<td>1</td>
<td>Int_ALU</td>
</tr>
</tbody>
</table>

- **addq** (\(-->\), hit, correct)
- **ldq** (miss, hit, correct)
- **addq** (\(-->\), hit, correct)
- **ldq** (miss, hit, correct)
- **subq** (\(-->\), hit, correct)
- **stq** (miss, hit, correct)
## Canonical Instruction Segment Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq (--, hit, correct)</td>
<td>Int_ALU</td>
</tr>
<tr>
<td>ldq (miss, hit, correct)</td>
<td>Load_Miss</td>
</tr>
<tr>
<td>addq (--, hit, correct)</td>
<td>Int_ALU</td>
</tr>
<tr>
<td>ldq (miss, hit, correct)</td>
<td>Load_Miss</td>
</tr>
<tr>
<td>subq (--, hit, correct)</td>
<td>Int_ALU</td>
</tr>
<tr>
<td>stq (miss, hit, correct)</td>
<td>Load_Miss</td>
</tr>
</tbody>
</table>

### CIST

<table>
<thead>
<tr>
<th>Freq</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Int_ALU</td>
</tr>
<tr>
<td>1</td>
<td>Load_Miss</td>
</tr>
</tbody>
</table>
### Canonical Instruction Segment Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Segment</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq</code> (hit, correct)</td>
<td>Int_ALU</td>
<td>1</td>
</tr>
<tr>
<td><code>ldq</code> (miss, hit, correct)</td>
<td>Load_Miss</td>
<td>1</td>
</tr>
<tr>
<td><code>addq</code> (hit, correct)</td>
<td>Int_ALU</td>
<td>1</td>
</tr>
<tr>
<td><code>ldq</code> (miss, hit, correct)</td>
<td>Load_Miss</td>
<td>1</td>
</tr>
<tr>
<td><code>stq</code> (miss, hit, correct)</td>
<td>Int_ALU</td>
<td>1</td>
</tr>
</tbody>
</table>
**Canonical Instruction Segment Table**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Frequency</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq</code></td>
<td>1</td>
<td>Int_ALU</td>
</tr>
<tr>
<td><code>ldq</code></td>
<td>2</td>
<td>Int_ALU, Load_Miss</td>
</tr>
<tr>
<td><code>addq</code></td>
<td>2</td>
<td>Int_ALU, Load_Miss</td>
</tr>
<tr>
<td><code>ldq</code></td>
<td>1</td>
<td>Load_Miss</td>
</tr>
<tr>
<td><code>subq</code></td>
<td>1</td>
<td>Int_ALU</td>
</tr>
<tr>
<td><code>stq</code></td>
<td>1</td>
<td>Int_ALU</td>
</tr>
</tbody>
</table>
### Canonical Instruction Segment Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CIST</th>
<th>Freq</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq (--, hit, correct)</td>
<td>Int_ALU</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ldq (miss, hit, correct)</td>
<td>Int_ALU</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>addq (--, hit, correct)</td>
<td>Load_Miss</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ldq (miss, hit, correct)</td>
<td>Load_Miss</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>subq (--, hit, correct)</td>
<td>Int_ALU</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>stq (miss, hit, correct)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Canonical Instruction Segment Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Segment</th>
<th>Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>Int_ALU</td>
<td>1</td>
</tr>
<tr>
<td>addq</td>
<td>Int_ALU</td>
<td>1</td>
</tr>
<tr>
<td>ldq</td>
<td>Load_Miss</td>
<td>2</td>
</tr>
<tr>
<td>ldq</td>
<td>Load_Miss</td>
<td>2</td>
</tr>
<tr>
<td>subq</td>
<td>Int_ALU</td>
<td>1</td>
</tr>
<tr>
<td>stq</td>
<td>Store_Miss</td>
<td>1</td>
</tr>
</tbody>
</table>

Total ins: 6
Stage 2: AXCIS Performance Model

Stage 1 (performed once)

Program & Inputs → Dynamic Trace Compressor → CIST (Canonical Instruction Segment Table) → Config → AXCIS Performance Model → IPC
AXCIS Performance Model

- Calculates IPC using a single linear dynamic programming pass over the CIST entries
  - Total work is proportional to the # of CIST entries

\[
\text{IPC} = \frac{\text{Total Ins}}{\text{Total Cycles}} = \frac{\text{Total Ins}}{\text{Total Ins} + \text{Total Effective Stalls}}
\]

\[
\text{Total Effective Stalls} = \sum_{i=1}^{\text{CIST Size}} \text{Freq}(i) \times \text{EffectiveStalls(DefiningIns}(i)\text{)}
\]

\[
\text{EffectiveStalls} = \text{MAX} (\text{stalls(DataHazards)}, \text{stalls(StructuralHazards)}, \text{stalls(ControlFlowHazards)})
\]
Performance Model Calculations

For each defining instruction:
- Calculate its effective stalls & its corresponding microarchitecture state snapshot
- Follow dependencies to look up the effective stalls & state of other instructions in previous entries

<table>
<thead>
<tr>
<th>Freq</th>
<th>Segment</th>
<th>Stalls</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Int_ALU</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Int_ALU</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Load_Miss</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Load_Miss</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Int_ALU</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Store_Miss</td>
<td>???</td>
<td>???</td>
</tr>
</tbody>
</table>

Total ins: 6

- Look up in previous segment
- Calculate
**Stall Cycles From Data Hazards**

- Use data dependencies (e.g. RAW) to detect data hazards

**Stalls(DataHazards)**

\[
\text{Stalls(DataHazards)} = \max (-1, \\
\text{Latency}( \text{producer} = \text{Load}\_\text{Miss} ) \\
- \text{DepDist} \\
- \text{EffectiveStalls}( \text{IntermediateIns} = \text{Int}\_\text{ALU} ) )
\]

\[
= \max (-1, \\
(100 - 2 - 99) )
\]

\[-1 \text{ stalls} (\text{can issue with previous instruction})\]
CISTs record special dependencies to capture all possible structural hazards across all configurations.

The AXCIS performance model follows these special dependencies to find the necessary microarchitectural states to:

1. Determine if a structural hazard exists & the number of stall cycles until it is resolved
2. Derive the microarchitectural state after issuing the current defining instruction

<table>
<thead>
<tr>
<th>Freq</th>
<th>Microarchitectural State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load_Miss</th>
<th>Int_ALU</th>
<th>Store_Miss</th>
<th>Stalls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>99</td>
<td>???</td>
<td></td>
</tr>
</tbody>
</table>

21 of 32
### Stall Cycles From Control Flow Hazards

- Control flow events directly map to stall cycles

<table>
<thead>
<tr>
<th>Freq</th>
<th>Icache</th>
<th>Branch Pred.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load Miss</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Int ALU</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Store Miss</td>
<td>hit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Icache</th>
<th>Bpred</th>
<th>Stalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit</td>
<td>Incorrect &amp; taken/not taken</td>
<td>Mispred penalty</td>
</tr>
<tr>
<td></td>
<td>Correct  &amp; taken</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Correct  &amp; not taken</td>
<td>-1</td>
</tr>
<tr>
<td>Miss</td>
<td>Incorrect &amp; taken/not taken</td>
<td>Memory latency + mispred penalty</td>
</tr>
<tr>
<td></td>
<td>Correct  &amp; taken</td>
<td>Memory latency</td>
</tr>
<tr>
<td></td>
<td>Correct  &amp; not taken</td>
<td>Memory latency - 1</td>
</tr>
</tbody>
</table>
Lossless Compression Scheme

Lossless Compression Scheme: *(perfect accuracy)*

- Compress two segments if they always experience the same stall cycles regardless of the machine configuration
- Impractical to implement within the Dynamic Trace Compressor

```
addq (--, hit, correct)  ldq always Issues with addq
ldiq (--, hit, correct)  addq (--, hit, correct)
stq (miss, hit, correct)  stq (miss, hit, correct)
```
Three Compression Schemes

- **Instruction Characteristics Based Compression:**
  - Compress segments that “look” alike (i.e. have the same length, instruction types, dependence distances, branch and cache behaviors)

- **Limit Configurations Based Compression:**
  - Compress segments whose defining instructions have the same instruction types, stalls and microarchitectural state under the 2 configurations simulated during trace compression

- **Relaxed Limit Configurations Based Compression:**
  - Relaxed version of the limit-based scheme – does not compare microarchitectural state
  - Improves compression at the cost of accuracy
Experimental Setup

- Evaluated AXCIS against a baseline cycle accurate simulator on 24 SPEC2K benchmarks
- Evaluated AXCIS for:
  - Accuracy: \[
  \text{Absolute IPC Error} = \left| \frac{\text{AXCIS} - \text{Baseline}}{\text{Baseline}} \right| \times 100
  \]
  - Speed: \# of CIST entries, time in seconds
- For each benchmark, simulated a wide range of designs:
  - Issue width: \{1, 4, 8\}, # of functional units: \{1, 2, 4, 8\},
  - Memory latency: \{10, 200\} cycles,
  - # of primary miss tags in non-blocking data cache: \{1, 8\}
- For each benchmark, selected the compression scheme that provides the best compression given a set accuracy range
Results: Accuracy

Distribution of IPC Error in quartiles

- **High Absolute Accuracy:**
  - Average Absolute IPC Error = 2.6%

- **Small Error Range:**
  - Average Error Range = 4.4%

Limit-based Scheme

Relaxed Limit-based Scheme

Characteristics-based Scheme
Results: Relative Accuracy

Average IPC of Baseline and AXCIS

High Relative Accuracy: AXCIS and Baseline provide the same ranking of configurations
Results: Speed

# of CIST entries & modeling time

Modeling time ranged from 0.02 – 18 seconds for billions of dynamic instructions

- AXCIS is over 4 orders of magnitude faster than detailed simulation
- CISTs are 5 orders of magnitude smaller than the original dynamic trace, on average
Discussion

- Trade the generality of CISTs for higher accuracy and/or speed
  - E.g. fix the issue width to 4 and explore near this design point

- Tailor the tradeoff made between speed/compression and accuracy for different workloads
  - **Floating point benchmarks** (*repetitive & compress well*)
    - More sensitive to any error made during compression
    - Require compression schemes with a stricter segment equality definition
  - **Integer benchmarks**: (*less repetitive & harder to compress*)
    - Require compression schemes that have a more relaxed equality definition
Future Work

- **Compression Schemes:**
  - How to quickly identify the best compression scheme for a benchmark?
  - Is there a general compression scheme that works well for all benchmarks?

- **Extensions to support Out-of-Order Machines:**
  - Main ideas still apply (instruction segments, CIST, compression schemes)
  - Modify performance model to represent dispatch, issue, and commit stages within the microarchitectural state so that given some initial state & an instruction, it can calculate the next state
Conclusion

- AXCIS is a promising technique for exploring large design spaces
  - **High absolute and relative accuracy** across a broad range of designs
  - **Fast:**
    - 4 orders of magnitude faster than detailed simulation
    - Simulates billions of dynamic instructions within seconds
  - **Flexible:**
    - Performance modeling is independent of the compression scheme used for CIST generation
    - Vary the compression scheme to select a different tradeoff between speed/compression and accuracy
    - Trade the generality of the CIST for increased speed and/or accuracy
Acknowledgements

This work was partly funded by the DARPA HPCS/IBM PERCS project, an NSF Graduate Research Fellowship, and NSF CAREER Award CCR-0093354.