
AXCIS: Accelerating Architectural Exploration using Canonical Instruction Segments

Rose Liu & Krste Asanović
Computer Architecture Group
MIT CSAIL

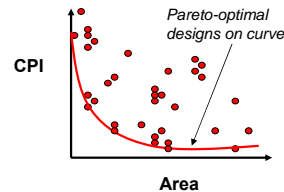


AXCIS is a new simulation technique for fast and accurate design space exploration.

Simulation for Large Design Space Exploration



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



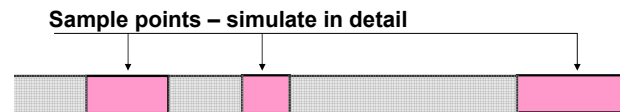
- **Speed vs. Accuracy tradeoff**
 - Maximize simulation speedup while maintaining sufficient accuracy to identify interesting design points for later detailed simulation

Large design space studies can explore thousands of processor configurations in order to identify the key designs that minimize costs such as power and area, while maximizing performance. AXCIS is a tool designed to quickly identify these interesting design points for later detailed simulation. Given the inherent tradeoff between simulation speed and accuracy, AXCIS aims to maximize simulation speed while maintaining sufficient accuracy to compare different designs.

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]



3 of 32

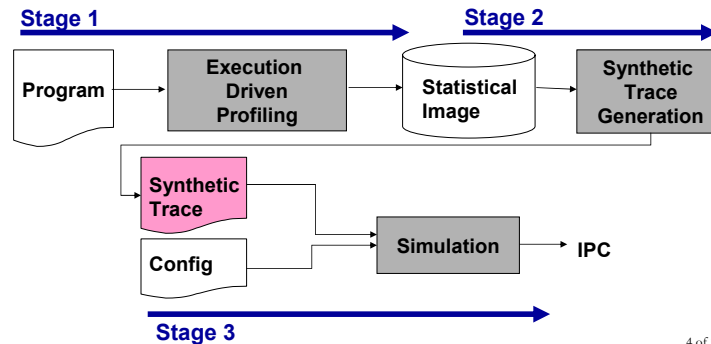
One of the main reasons why detailed cycle-accurate simulators are slow is because they simulate every instruction in a program's dynamic trace, which can contain hundreds of billions of instructions. Therefore researchers have proposed various techniques to speed up simulation based on the idea of reducing the number of simulated instructions.

One such technique is sampling, which only performs detailed microarchitectural simulation during select portions of the dynamic trace, called sample points, while functionally simulating the sections between sample points to warm microarchitectural state. However, in this form, sampling still analyzes every instruction in the dynamic trace, making it still too slow for large design space studies. But with the use of efficient checkpoint techniques, such as Livepoints from CMU (ISPASS 2006), you can eliminate the functional warming performed between sample points and reduce simulation time to minutes.

Reduce Simulated Instructions: Statistical Simulation

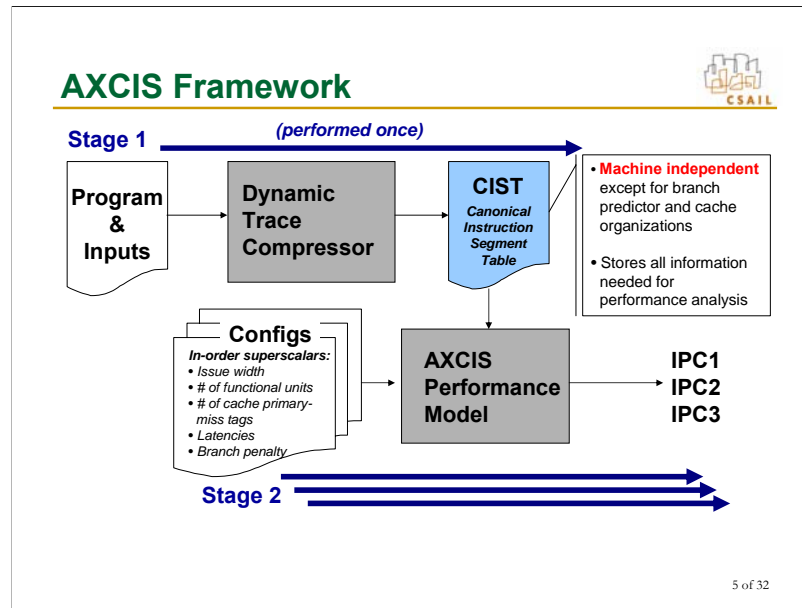


- Generate a short **synthetic trace** (with statistical properties similar to original workload) for simulation
 - Eeckhout [ISCA, 2004], Oskin [ISCA, 2000]
Nussbaum [PACT, 2001]



4 of 32

Statistical simulation also reduces the number of simulated instructions. It does so by simulating off of a short synthetic trace with the same statistical properties as the original. In this technique, the entire dynamic program trace is first profiled to gather statistics on important program characteristics, and cache and branch predictor events. These profiles are then used to generate a synthetic trace, that can be used to simulate different designs. Since you only need to simulate a synthetic trace until IPC converges, synthetic traces are effectively orders of magnitude smaller than the original dynamic trace, making this technique very fast.

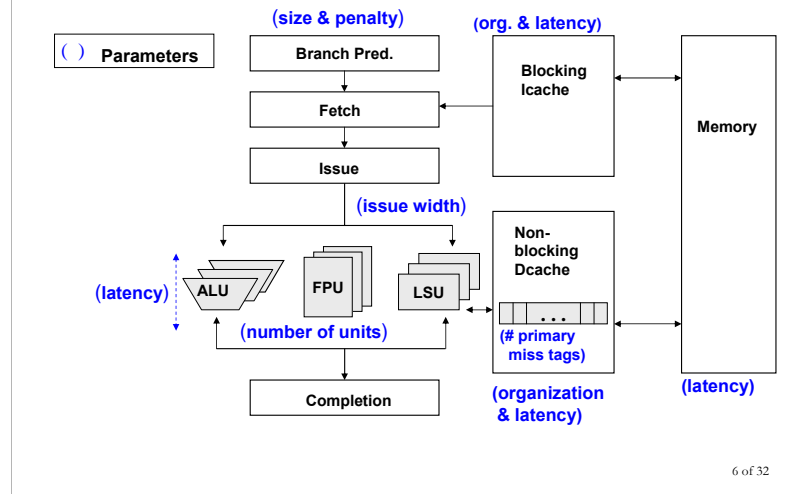


AXCIS also speeds up simulation by reducing the number of simulated instructions. But what's new about AXCIS is that it compresses the program's dynamic trace into a representation that can model many different machines. This compressed representation contains only the minimum subset of information required for accurate performance modeling. Therefore by eliminating redundant information, AXCIS significantly speeds up modeling time. For example, AXCIS can simulate the equivalent of billions of dynamic instruction within seconds.

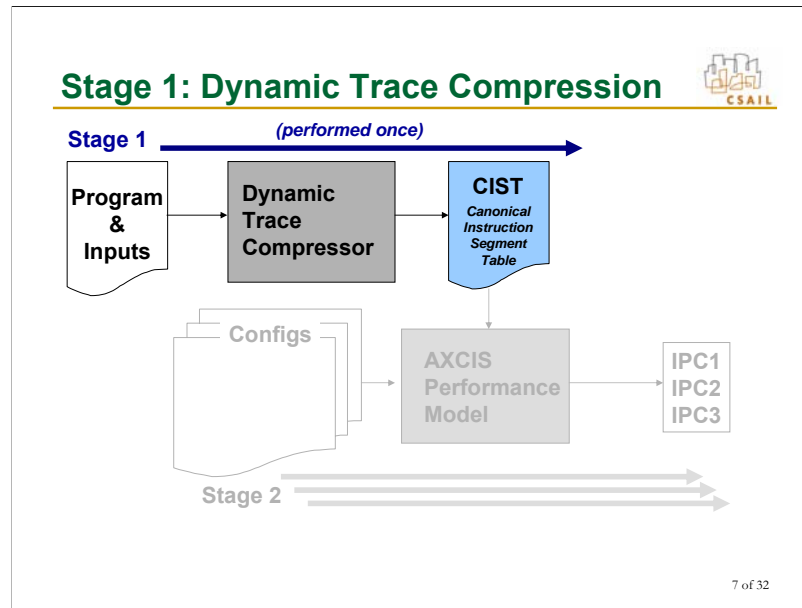
AXCIS is divided into two stages. In the first stage, AXCIS compresses the entire dynamic trace into a Canonical Instruction Segment Table (CIST), which contains all the information needed for accurate performance modeling. CISTs are mostly machine independent, and can be re-used to model many different designs. However, CISTs are dependent on branch predictors and cache organizations, but you can make CISTs more general, by simultaneously simulating multiple caches and branch predictors during compression. Because the trace compressor analyzes each dynamic instruction, this stage is about as fast as functional simulation plus warming of the caches and predictors. But, this stage only needs to be performed once per workload.

In the second stage, AXCIS estimates performance, given a CIST and a machine configuration. The performance model makes a linear dynamic programming pass over the CIST entries to calculate IPC. Analyzing the CIST entries is effectively the same as analyzing the program's entire dynamic trace, but the number of CIST entries is on average 5 orders of magnitude less than the number of dynamic instructions.

In-Order Superscalar Machine Model



We applied AXCIS to model in order superscalar processors which represent a large number designs for embedded applications and are becoming more popular in chip multiprocessors. AXCIS also models blocking instruction caches, non-blocking data caches supporting a varying number of outstanding misses, and bimodal branch predictors. As you can see, we have parameterized many machine features. The parameters are labeled in blue. Therefore AXCIS can simulate machines that vary in issue width, number of functional units, latencies, number of outstanding data cache misses, etc.

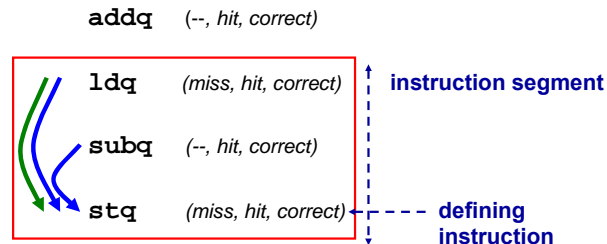


Now that I've given a high level picture AXCIS, I am going to go into the details of how trace compression works, and what exactly is a CIST.

Instruction Segments



Events: (*dcache, icache, bpred*)



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

8 of 32

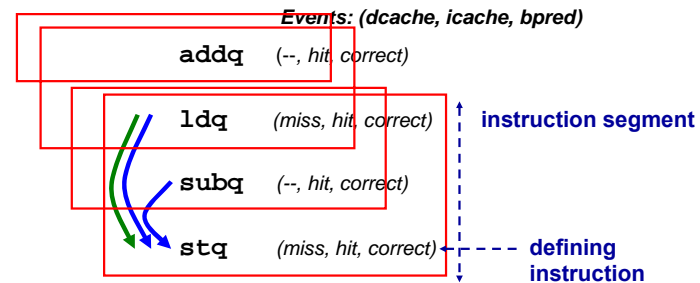
In order to calculate IPC, the performance model needs enough information to figure out the number of stall cycles experienced by each dynamic instruction. Since, stalls are caused by data hazards, structural hazards, and control flow hazards, the trace compressor needs to detect and record, in a machine independent way, all potential hazards that can occur between any two instructions.

Here we show a sequence of dynamic instructions, and all the cache and branch predictor information that the trace compressor knows about. What we will do now is identify all potential hazards that can affect the stall cycles of the `stq` instruction so that the performance model can later calculate its stalls in a particular machine configuration.

Each data dependency represents a potential data hazard. Therefore we identify all data dependencies of the `stq`. Structural hazards are a bit trickier because they depend on the specific machine configuration. Therefore we create special dependencies between instructions to record any potential structural hazards that may occur in any configuration. For example, the `stq` depends on the `subq` because there could be a structural hazard due to issue width limitations. The `stq` also depends on the `ldq` because they both missed in the data cache and there could be a structural hazard on the primary miss tags. And finally, we identify the control flow hazards by looking at the cache and branch predictor events.

Now that we have identified all the information needed to calculate the stalls of the `stq`, we package this up into an instruction segment, which consists of all instructions starting from the producer of the longest dependency, down to the instruction itself. Which we term the defining instruction, because this is the instruction that the segment was defined for. Instruction segments are the key behind our compression and performance modeling. It captures all the performance-critical information associated with a dynamic instruction. Instruction segments can span many basic blocks or consist of only one instruction, depending on the length of the dependencies of the defining instruction. As an optimization, we ignore dependencies that span more than 512 instructions, because they are unlikely to affect the stalls of the defining instruction.

Instruction Segments



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

Because every dynamic instruction has a corresponding instruction segment, the entire dynamic trace can be viewed as a chain of overlapping instruction segments.

Dynamic Trace Compression

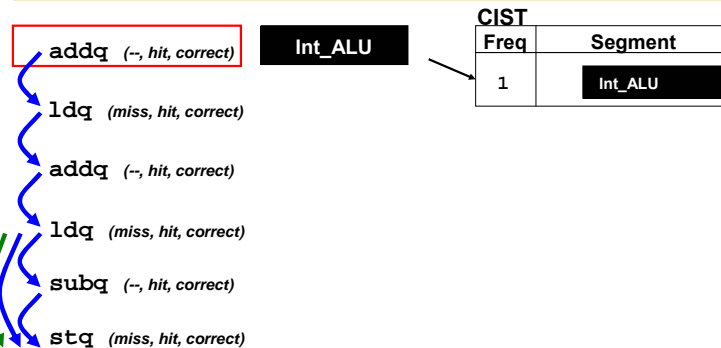


- Program behavior repeats due to loops, and repeated function calls
- Multiple different dynamic instruction segments can have the same behavior (**canonically equivalent**) regardless of the machine configuration
- Compress the dynamic trace by storing in a table:
 - 1 copy of each type of segment
 - How often we see it in the dynamic trace

10 of 32

Program behavior repeats due to loops and repeated function calls. So, what you will find is that many segments have the same behavior – meaning they are canonically equivalent. Therefore we can compress the dynamic trace by storing in a table, 1 copy of each type of segment, and how often we saw it in the dynamic trace. We refer to this table as the canonical instruction segment table or CIST.

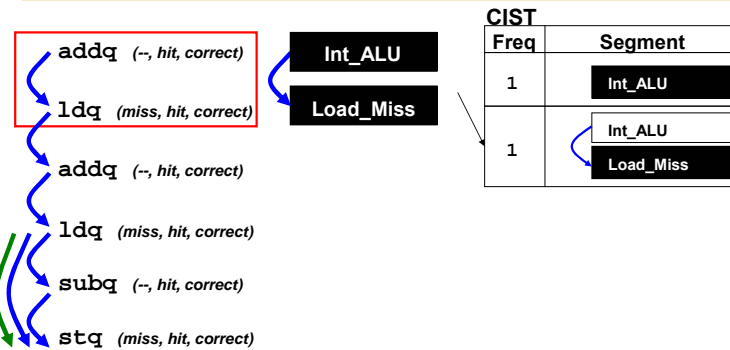
Canonical Instruction Segment Table



11 of 32

So now let's go through an example on how the trace compressor dynamically builds the CIST. On your left is a sequence of dynamic instructions. The trace compressor identifies the segment for each instruction, in this case the segment for the `addq` is just the instruction itself. Then the compressor abstracts the instruction into its instruction type. An instruction type is a group of opcodes that share the same functional unit so they have the same latency regardless of the configuration. This abstraction improves compression without losing any accuracy. Then the compressor checks to see if the new segment exists in the CIST, in this case it does not, so it adds the new segment to the CIST.

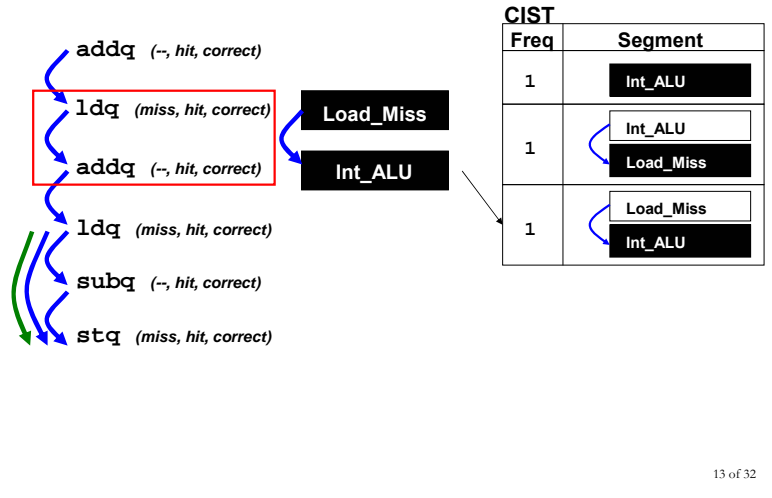
Canonical Instruction Segment Table



12 of 32

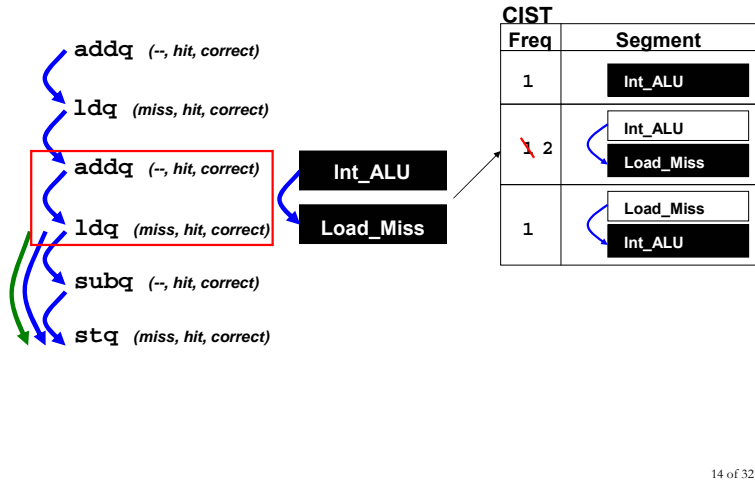
Similarly for the `ldq`, the compressor identifies its segment, abstracts the instructions into instruction types, and then checks to see if the segment already exists in the CIST. For now, we will compress segments that look the same – meaning they have the same instruction types, dependence distances, etc. Since we have not seen this segment before, we add it to the CIST.

Canonical Instruction Segment Table



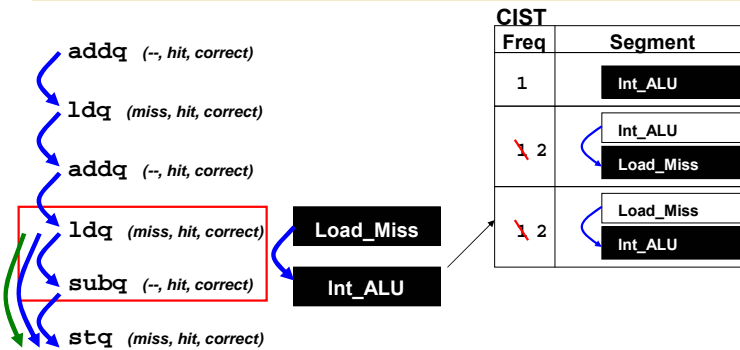
Same thing occurs for the `addq` instruction.

Canonical Instruction Segment Table



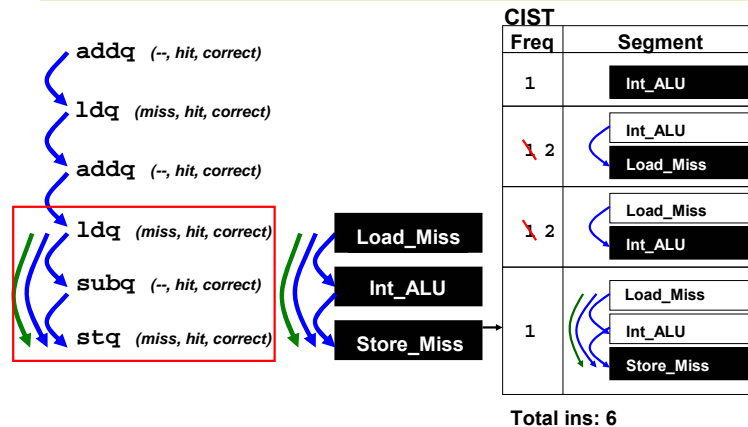
Now, the segment for `ldq` has been encountered before, therefore we can just increment the frequency count of the existing CIST entry.

Canonical Instruction Segment Table



Similarly, the segment for subq has also been seen before in entry 3, so we can just increment the frequency count there.

Canonical Instruction Segment Table



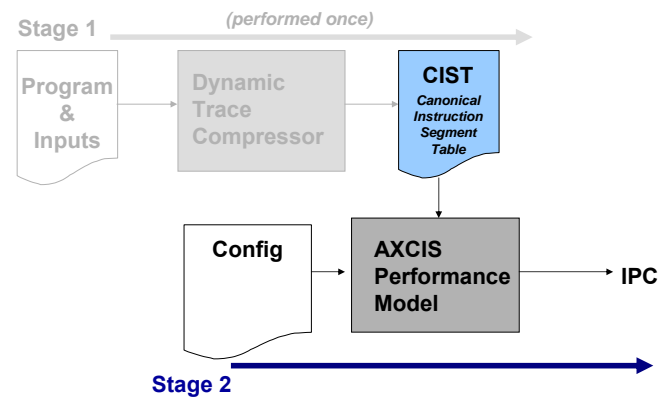
16 of 32

Now this segment for the `stq` is new, so we add it to the CIST. Finally, we also record the total number of dynamic instructions analyzed in the CIST.

As you can see, even in this very simple example, we have experienced compression.

We were able to represent 6 dynamic instructions in 4 CIST entries. Since, the work done in performance modeling is proportional to the number of CIST entries, we have already reduced simulation time. From our experiments, we found that on average, the number of CIST entries is 5 orders of magnitude less than the total dynamic instructions they represent.

Stage 2: AXCIS Performance Model



17 of 32

Now that I have gone over how CISTs are created, I will explain how to calculate IPC given a CIST and a particular machine configuration.

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

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

$$\text{Total Effective Stalls} = \sum_{i=1}^{\text{CIST Size}} \text{Freq}(i) * \text{EffectiveStalls}(\text{DefiningIns}(i))$$

$$\text{EffectiveStalls} = \text{MAX} (\text{stalls}(\text{DataHazards}), \\ \text{stalls}(\text{StructuralHazards}), \\ \text{stalls}(\text{ControlFlowHazards}))$$

18 of 32

The AXCIS performance model calculates IPC by performing a linear dynamic programming pass over the CIST entries. Therefore the total work is proportional to the number of entries.

IPC is expressed as the total dynamic instructions analyzed during trace compression, over the total instructions plus the total effective stall cycles experienced by each instruction. Since the total instructions is already recorded in the CIST, the performance model only needs to calculate the total effective stalls, which it does by taking the weighted sum of the effective stalls of each segment in the CIST weighted by its frequency count. The effective stalls of an instruction is the max of the stalls caused by each type of hazard.

In order to obtain an IPC greater than 1 for superscalar machines, the total effective stall cycles is negative. Therefore, if an instruction were to issue with a previous instruction, we would set its effective stalls to -1.

Performance Model Calculations



Freq	Segment	Stalls	State
1	Int_ALU	0	
2	Int_ALU Load_Miss	2	
2	Load_Miss Int_ALU	99	
1	Load_Miss Int_ALU Store_Miss	99 ???	???

Total ins: 6

 Look up in previous segment
 Calculate

- 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

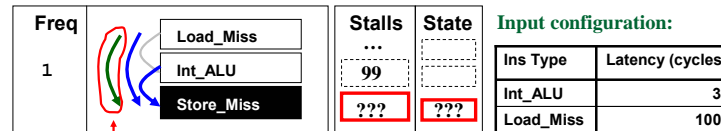
Because segments were added to the CIST in program order, each entry can only depend on previous entries, allowing the performance model to do one linear dynamic programming pass over the table.

For each CIST entry, the performance model calculates the effective stalls and the microarchitectural state corresponding to the defining instruction. The defining instruction of each entry is highlighted in black. In order to calculate the effective stalls and state of a particular entry, the performance model may need to follow dependencies to look up the stalls and state of previous entries.

For example, AXCIS first calculates the stalls and state of entry one. Then to calculate those of entry two, it follows the dependency up to the int_ALU instruction of entry 1 to look up its state. To calculate the state of the int_ALU in entry three, it follows the dependency to look up the state in entry 2. The store_MISS in entry 4 has dependencies to two instructions, and therefore it does two lookups. The performance model checks both state entries for any potential hazards, and then calculates the next state and the stalls experienced by the stored miss instruction.

The effective stalls of an instruction is the max of the stalls from data, structural, and control flow hazards. In the following slides I will explain how to calculate the stalls from each type of hazard.

Stall Cycles From Data Hazards



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

- Stalls(DataHazards)

$$\begin{aligned}
 &= \text{MAX} (-1, \\
 &\quad \text{Latency}(\text{producer} = \text{Load_Miss}) \\
 &\quad - \text{DepDist} \\
 &\quad - \text{EffectiveStalls}(\text{IntermediateIns} = \text{Int_ALU})) \\
 &= \text{MAX} (-1, \\
 &\quad (100 - 2 - 99)) \\
 &= \text{-1 stalls (can issue with previous instruction)}
 \end{aligned}$$


20 of 32

To calculate the stalls from data hazards, we need to refer to the data dependencies recorded in the segment. Data hazard stalls can be calculated by taking the difference between the latency of the producer associated with the dependency, the dependence distance, and the effective stalls of any intermediate instructions between the producer and defining instruction.

In this example the dependency is indicated by the green arrow, with a dependence distance of 2. The producer is the Load_Miss instruction with a latency of 100. And the intermediate Int_ALU instruction experienced 99 stalls. Since the difference between these values is negative, there are no data hazards that prevent the Store_Miss from issuing with a previous instruction. Therefore we set its stalls to -1.

Stall Cycles from Structural Hazards



Freq		Stalls	Microarchitectural State
1	
		99	...
		???	???

- CISTs record **special dependencies** to capture all **possible** structural hazards across **all** configurations
- The AXGIS 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

To calculate the stalls due to structural hazards, the performance model relies on the special dependencies recorded in instruction segments that capture all possible structural hazards across all configurations. By following these dependencies, the performance model can look up the necessary microarchitectural state snapshots to determine the structural hazard stalls and derive the state snapshot corresponding to the defining instruction. In this example, AXGIS follows the special dependencies indicated by blue arrows to look up the state of the Load_Miss and Int_ALU instructions, to calculate the stalls and current state.

Stall Cycles From Control Flow Hazards



Freq 1		Icache	Branch Pred.
	
	
		hit	correct & not taken

- Control flow events directly map to stall cycles

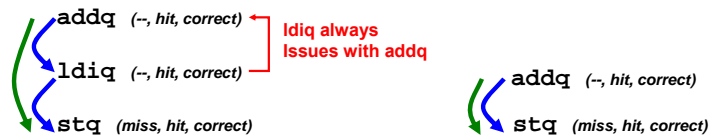
Icache	Bpred	Stalls
Hit	Incorrect & taken/not taken	Mispred penalty
	Correct & taken	0
	Correct & not taken	-1
Miss	Incorrect & taken/not taken	Memory latency + mispred penalty
	Correct & taken	Memory latency
	Correct & not taken	Memory latency - 1

Control flow events recorded in the instruction segments directly map to their stall cycles. Therefore the performance model only needs to do a table lookup. In this example, the Store_Miss hit in the instruction cache, was correctly predicted, and does not follow a taken branch. Therefore it experiences -1 stalls, meaning no control flow events stop it from issuing with a previous instruction.

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



23 of 32

Now that I've gone over how AXCIS works, I'd like to go back and talk about instruction segment compression schemes. If the goal were to perform lossless compression to maintain perfect accuracy, you can only compress 2 segments if they always experience the same stalls regardless of the machine configuration.

Let me illustrate this idea with an example. In the segment on your left, if the `ld` immediate instruction was always able to issue with the `addq` instruction in every machine configuration we wanted to explore, then the two segments shown are effectively equivalent – meaning the `stq` instructions in both segments would always experience the same number of stalls regardless of the input configuration. Therefore although these segments look different, they behave the same and can be compressed without losing any information.

In practice, such a scheme is hard to implement, because to do so, the trace compressor would have to simulate all possible configurations in order to determine if 2 segments had the same stalls. Also, a lossless scheme may not be what you want because you may want to have even more compression at the cost of some accuracy.

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

24 of 32

Therefore we propose 3 compression schemes that approximate the lossless scheme, but each selects a different tradeoff between speed and accuracy. The instruction characteristics based compression scheme compresses segments if they look alike. This is the scheme that we have used in our examples so far. The Limit configurations based compression scheme simulates two configurations (min & max) during trace compression, and compresses segments if their defining instructions have the same effective stalls, state snapshot, and instruction types. The relaxed version of this scheme does not compare state snapshots. By relaxing the segment equality definition, this scheme 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:
$$\text{Absolute IPC Error} = \frac{|\text{AXCIS} - \text{Baseline}|}{\text{Baseline}} * 100$$
 - 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

25 of 32

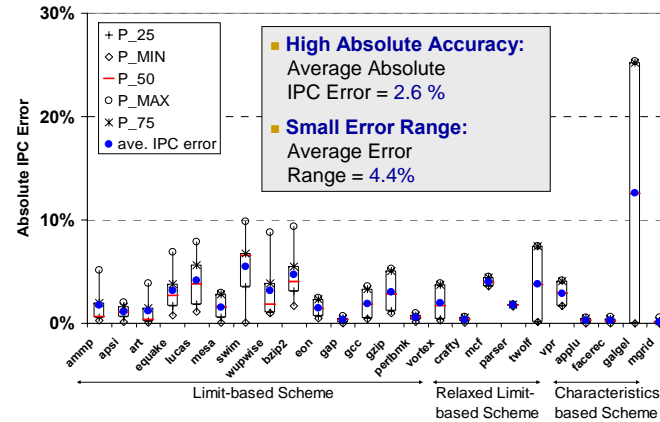
We evaluated AXCIS against our baseline cycle accurate simulator on 24 SPEC2K benchmarks. We evaluated AXCIS for accuracy and speed. Accuracy is measured in terms of absolute IPC error between AXCIS and the baseline, while speed is measured in terms of the number of CIST entries and time in seconds. For each benchmark, we simulated many configurations across a wide range of designs. These configurations differ in issue width and # functional units ranging from 1-8, with memory latencies of 10 and 200 cycles, and non-blocking data caches supporting 1 and 8 outstanding misses.

For each benchmark, we selected the compression scheme that provides the best compression given a set accuracy range.

Results: Accuracy



Distribution of IPC Error in quartiles



26 of 32

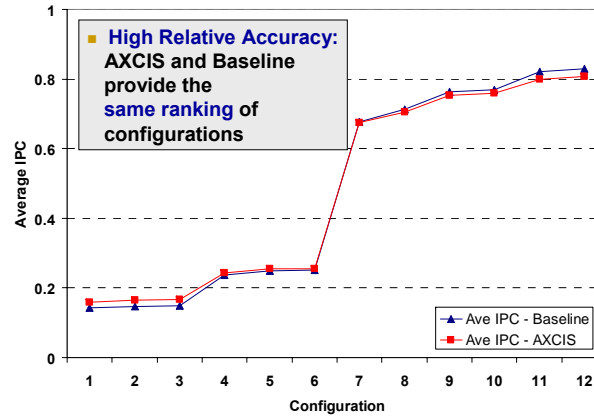
This graph shows the distribution of ipc errors in quartiles for each benchmark across all configurations. The average ipc error is 2.6% making AXCIS highly accurate. The maximum errors of all but galgel are less than 10%, making the error range very tight, only 4.4%, suggesting that AXCIS has high relative accuracy.

The cause for galgel's high error seems to be due to the fact that it is difficult for the performance model to accurately model microarchitectural structures with a long history - such as the primary miss tags.

Results: Relative Accuracy



Average IPC of Baseline and AXCIS



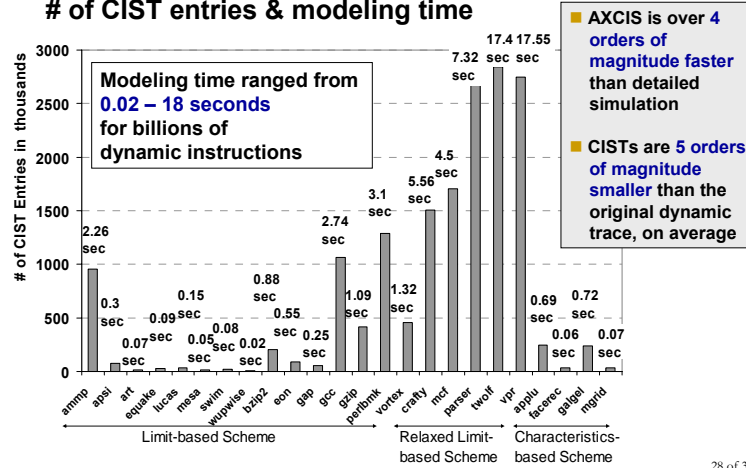
27 of 32

Here we plot the average IPC over all benchmarks for each configuration. The red line shows the average IPC obtained from AXCIS and the blue shows the average IPC obtained from the baseline. As you can see, both AXCIS and the baseline provide the same ranking of configurations, indicating that AXCIS has high relative accuracy.

Results: Speed



of CIST entries & modeling time



This graph shows the # of CIST entries (in thousands) which is proportional to the amount of work done by the performance model. We also show the average number of seconds to model a configuration for each benchmark. Each of these benchmarks were run for billions of dynamic instructions, but the maximum # of entries is less than 3 million. Therefore the work done by the performance model is significantly less than detailed simulators. As you can see, AXCIS finishes each run within seconds. While detailed simulation takes hours to simulate these billions of instructions, AXCIS takes seconds, making it over 4 orders of magnitude faster than detailed simulation.

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

29 of 32

AXCIS is very flexible. You can trade the generality of CISTs for higher accuracy and/or speed by fixing more microarchitecture features during trace compression and storing more machine dependent information in the CIST. This can both improve the accuracy of the performance model as well as decrease the amount of work it needs to do. For example, you can fix the issue width to 4 during compression and explore around this design point during modeling.

Because the performance modeling methodology is independent of the compression scheme, you can also vary the compression scheme to trade speed/compression for accuracy, and tailor the compression scheme for each type of benchmark. Floating point benchmarks are generally more repetitive and easier to compress. But this makes them more sensitive to any error made during compression since the error can be magnified at each repetition. Therefore we found that the floating point benchmarks required compression schemes with a stricter segment equality definition.

On the other hand, integer benchmarks are less repetitive and harder to compress, so they generally require compression schemes with a more relaxed equality definition to improve compression. We found that if we used a stricter compression scheme on some of these benchmarks, the CISTs would become very large very quickly, and we would run out of memory.

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

30 of 32

This leads us to the following questions. How do we quickly identify the optimal compression scheme for a benchmark without doing extensive studies? And Is there a general compression scheme that works well for all benchmarks? We leave this for future work.

We applied AXCIS to in-order machines, but AXCIS can be extended to model out-of-order machines. All the main ideas such as instruction segments, CISTs, and the compression schemes still apply. We believe that only the performance model requires significant changes. To model out-of-order machines, it would have to represent dispatch, issue, and commit stages into the microarchitecture state snapshots so that given some initial state and an instruction, it can calculate the next state. This dynamic programming algorithm should work for machines with age ordered scheduling.

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

31 of 32

We have introduced a new way for exploring large design spaces. Our initial results show that AXCIS is a promising technique that has high absolute and relative accuracy across a wide range of designs. AXCIS is very fast. Being 4 orders of magnitude faster than detailed simulation, it can simulate billions of dynamic instructions within seconds. AXCIS is also very flexible. Because performance modeling is independent of the compression scheme used, the user can define the tradeoff made between compression and accuracy by selecting an appropriate compression scheme. AXCIS also allows the user to trade the generality of the CIST for additional speed and/or accuracy during performance modeling.



Acknowledgements

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