In *IEEE Int'l Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, TX, March 2006.

# Accelerating Architectural Exploration Using Canonical Instruction Segments

Rose F. Liu and Krste Asanović

MIT Computer Science and Artificial Intelligence Laboratory
The Stata Center, 32 Vassar Street, Cambridge, MA 02139
{rliu, krste}@csail.mit.edu

## Abstract

*Detailed microarchitectural simulators are not well suited for exploring large design spaces due to their excessive simulation times. We introduce AXCIS, a framework for fast and accurate design space exploration. AXCIS achieves fast simulation times by exploiting repetitions in program behavior to reduce the number of instructions simulated. For each dynamic instruction encountered during an initial full run of a benchmark, AXCIS builds an* **instruction segment**, *which concisely represent performance-critical information. AXCIS then compresses the string of dynamic segments into a table of canonical instruction segments (CIST) to give a compact representation of the entire benchmark trace. Given a precomputed CIST and a target microarchitecture configuration, AXCIS can quickly and accurately estimate performance metrics such as instructions per cycle (IPC). For the SPEC CPU2000 benchmarks and all simulated configurations, AXCIS achieves an average IPC error of 2.6%. While cycle-accurate simulators can take many hours to simulate billions of dynamic instructions, AXCIS can complete the same simulation on the corresponding CIST within seconds.*

## 1. Introduction

In the early stages of processor design, computer architects are often faced with a very large design space. Although cycle-accurate simulation is effective and widely used for evaluating different processor configurations, long simulation times along with short time-to-market severely constrains the number of design points explored. Current detailed simulators are thousands of times slower than native hardware. For example, the popular simulator `sim-outorder` from the SimpleScalar tool set [3] simulates at around 0.35 MIPS on a 1.7 GHz Pentium-4 [1]. Benchmarks are also growing in size and complexity to better approximate real-world applications, further extending long simulation times.

Detailed simulators are slow because they simulate every instruction in a program's dynamic trace. To improve simulation time, researchers have proposed techniques to decrease the number of simulated instructions. One approach reduces a program's input set to execute a smaller dynamic trace [6], but studies have shown that program behavior can vary significantly between the original and reduced input sets [6, 17]. Another technique is sampling [12, 16], where only selected portions of the entire program execution are simulated in detail and instructions between detailed sampling points are simulated at a functional level to warm microarchitectural state [16]. Sampling, in this form, is relatively accurate but still too slow for initial design space exploration where thousands of configurations are explored. However, more recent work [2, 15] has shown that checkpointing can significantly reduce the simulation time of sampling, down to a few minutes per benchmark. Statistical simulation techniques [4, 9, 11] simulate on synthetic traces generated after profiling full program executions. These synthetic traces are several orders of magnitude smaller than the original dynamic traces, enabling rapid large design space studies. Statistical simulation techniques apply various heuristics to reduce the number of statistical parameters recorded during trace profiling, while attempting to capture enough program characteristics to generate representative traces for subsequent dynamic simulation. In particular, many combinations of events may be simply modeled as independent distributions because tracking joint probabilities is too expensive. In addition, these techniques require an empirical determination of convergence for different benchmarks and machine configurations.

In this paper, we present AXCIS (Architectural eXploration using Canonical Instruction Segments), a framework for fast and accurate early-stage design space exploration. AXCIS reduces simulation time by compressing a program's dynamic trace into a form that can model many different machines. AXCIS performs compression and performance modeling using a new primitive called the *instruction segment* to represent the context of each dynamic instruction, including all important dependencies. As shown in Figure 1, AXCIS is divided into two stages: dynamic trace compression and performance modeling.

The Dynamic Trace Compressor (DTC) identifies all instruction segments within the dynamic trace and compresses these into a Canonical Instruction Segment Table (CIST). In order to capture locality events within instruction segments, partially-specified branch predictors and caches are simulated. However, only the organizations of these structures are spec-

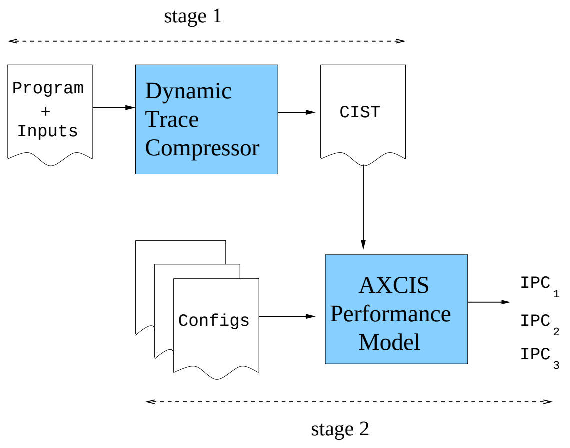


**Figure 1. AXCIS Simulation Framework.**

ified at this stage and not their latencies, allowing the generated CISTs to model multiple designs. To perform compression, the DTC compares each new dynamic segment against existing CIST entries, and either increments the count of an existing entry if a match is found or adds the new segment to the CIST if not. The CIST representation effectively captures a large set of joint probabilities in the trace, including occurrences of dynamically-established dependencies that cross many basic blocks. CIST generation can also be adjusted to trade simulation speed for accuracy by varying the DTC compression scheme. We explore three compression schemes in this paper.

The AXCIS Performance Model (APM) quickly estimates performance in terms of instructions per cycle (IPC) for each processor design, given a CIST and a set of microarchitecture configurations. The CIST format is designed such that a single linear dynamic programming pass over the CIST entries calculates performance for a given design point. Unlike statistical simulation techniques, AXCIS does not require dynamic simulation until convergence. Because CISTs are small and very representative of full dynamic traces, AXCIS is over four orders of magnitude faster than conventional detailed simulation. AXCIS reduces the simulation time of billions of dynamic instructions to a few seconds, while achieving an average IPC error of 2.6% on SPEC CPU2000 benchmarks [14]. We propose AXCIS as a complement to cycle-accurate simulation, where AXCIS can quickly identify regions of interest to be further explored in detail.

## 2. Machine Model

We apply AXCIS to characterize in-order superscalar processors, which represent the vast majority of new processor designs, especially for embedded applications. In-order cores are also becoming more popular in general-purpose applications with the emergence of chip multiprocessors, which emphasize thread-level throughput over single threaded performance and have stricter per-core area and power constraints. For example, Sun's UltraSPARC IV, Intel's Montecito, and Broadcom's BCM1480 all contain multiple in-order cores. This section describes the class of machines modeled by AXCIS and highlights the machine characteristics that affect performance through data, structural, and control flow hazards. As shown in Figure 2, we model in-order superscalars with blocking L1 instruction caches, non-blocking L1 data caches, and bimodal branch predictors. The parameterizable machine characteristics are drawn with dashed lines and labeled in parentheses.

Functional-unit, cache, and memory latencies as well as the cache organization determine when data hazards occur. We only model read-after-write (RAW) dependencies because write-after-read (WAR) and write-after-write (WAW) hazards are generally eliminated in well-designed in-order pipelines with synchronized writebacks and bypass networks. Cache organization parameters include the number and size of cache lines, associativity, and replacement policy. The number of specific units and issue width determine when structural hazards occur, while branch predictor size and misprediction penalty affect performance through control hazards.

Blocking caches stall the processor until a cache miss is serviced by the memory system, while non-blocking caches allow later instructions (even cache misses) to issue before earlier ones are back from memory. Nonblocking caches are difficult to model because they induce long-range dependencies between memory instructions to different addresses. In non-blocking caches, the maximum number of outstanding memory requests is determined by the number of *primary* and *secondary miss tags*. Primary miss tags track the unique cache lines being serviced by the memory system, while secondary miss tags track additional memory requests to cache lines recorded by the primary miss tags. Correspondingly, a *primary miss* is the first miss to a cache line, and a *secondary miss* is a miss to the line of an earlier primary miss. Structural hazards occur when none of the associated miss tags are available when a cache miss is evaluated for issue. Since primary miss tags have a larger effect on performance than secondary miss tags, AXCIS models non-blocking data caches with a parameterizable number of primary miss tags, while assuming an infinite number of secondary miss tags for simplicity.

## 3. Instruction Segments and CISTs

During dynamic trace compression, AXCIS builds an instruction segment for every dynamic instruction to represent its performance-critical program characteristics. The instruction segment of a dynamic instruction contains (1) all preceding instructions affecting its performance, (2) its instruction dependencies, and (3) its locality characteristics (i.e. cache and branch behaviors). A segment begins with the producer associated with the instruction's longest dependency and ends with the instruction itself, termed the *defining instruction* of the segment. Instructions within segments are abstracted into *instruction types*: integer ALU, integer and floating-point multiplies, integer and floating-point divides, floating-point add, floating-point compare, floating-point to integer conversion, floating-point square root, load cache hit, load cache miss, store cache

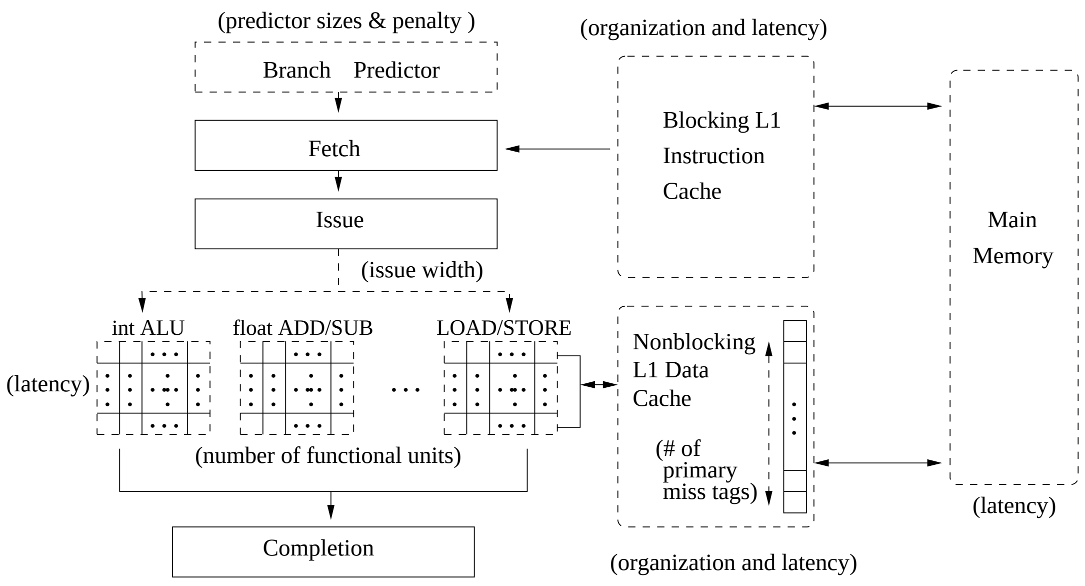

(predictor sizes & penalty )
Branch Predictor
Fetch
Issue
(issue width)
int ALU
float ADD/SUB
LOAD/STORE
(latency)
(number of functional units)
Completion
(organization and latency)
Blocking L1 Instruction Cache
Nonblocking L1 Data Cache
(# of primary miss tags)
(organization and latency)
Main Memory
(latency)

**Figure 2. Class of machines supported by AXCIS.**

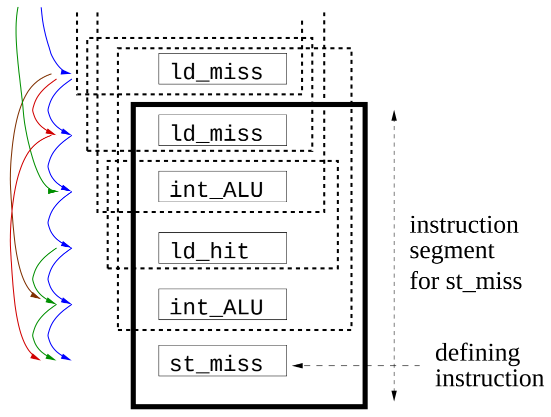


**Figure 3. Dynamic program trace partitioned into instruction segments.**

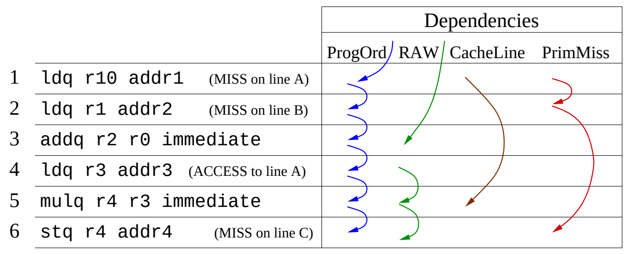
hit, store cache miss, and nop. Figure 3 shows a sequence of dynamic instructions and their instruction segments, which may overlap. Dependencies between instructions are represented by arrows, and the instruction segment for the `st_miss` instruction is highlighted.

Dependencies exist between two instructions if the earlier instruction directly affects the performance, measured in stall cycles, of the later instruction. Stall cycles are defined as the number of elapsed cycles starting from when an instruction is first evaluated for issue up to when it actually can issue. Instruction segments record the following types of dependencies: (1) RAW data dependencies, (2) *program-order dependencies*, (3) *cache-line dependencies*, and (4) *primary-miss dependencies*. The dependencies of each instruction are represented as a set of *dependence distances*, where each dependence distance is the number of dynamic instructions starting from the producer down to, but not including, the consumer associated with the dependency.

RAW and program-order dependencies are inherent within the program. Program-order dependencies form between consecutive dynamic instructions. They allow AXCIS to maintain the original dynamic instruction sequences within CISTs and identify structural hazards during performance modeling.

Locality events for the data cache are captured using cache-line and primary-miss dependencies that are identified during trace compression when a partially-specified nonblocking data cache is functionally simulated. By not specifying cache latencies or the number of primary miss tags during simulation, each CIST can model many cache configurations. Cache-line dependencies are data dependencies that form between consumer instructions, using the value produced by non primary-miss cache accesses, and the most recent primary miss to the requested cache line. Given a machine design, these dependencies allow the AXCIS performance model to calculate the latency of the non primary miss cache access by determining whether it is a secondary miss or a cache hit. Primary-miss dependencies form between consecutive primary misses and allow AXCIS to detect structural hazards on the primary miss tags.

Using the Alpha instruction set, Figure 4 shows all four types of dependencies in the trace sample from Figure 3. A cache-line dependency exists between instructions 1 (*Ins_1*) and 5 (*Ins_5*) because *Ins_1* affects the stall cycles of *Ins_5*. *Ins_5* cannot issue until instruction 4 (*Ins_4*) produces its value, which depends on when line A is fetched into the cache. This is determined by when *Ins_1* (the primary miss to line A) is issued. Notice that *Ins_4* is not dependent on *Ins_1* because *Ins_1* does not affect when *Ins_4* issues. A primary miss dependency exists between instructions 2 (*Ins_2*) and 6 (*Ins_6*) because the stalls of *Ins_6* depend on whether a primary miss tag is available when it is evaluated for issue, which can be derived from



```
1  ldq r10 addr1      (MISS on line A)
2  ldq r1 addr2       (MISS on line B)
3  addq r2 r0 immediate
4  ldq r3 addr3   (ACCESS to line A)
5  mulq r4 r3 immediate
6  stq r4 addr4       (MISS on line C)
```

**Figure 4. Dynamic instruction sequence with dependencies.**

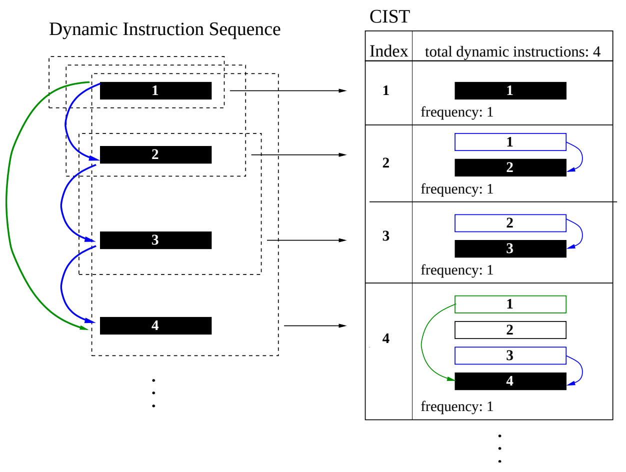


**Figure 5. Example of a CIST.**

the state of the miss tags after *Ins_2* issues.

To maintain reasonable segment lengths, AXCIS only records the dependencies of *primary consumers*, which are the first instructions to experience stalls due to the producer. Because we model in-order machines, we do not need to record dependencies on non-primary consumers as they will not introduce any additional stalls. We also discard dependencies that span more than 512 dynamic instructions, as these rarely cause stalls and removing them increases compressibility.

The CIST is maintained as an ordered array of CIST entries, where each entry consists of an instruction segment and a frequency count recording how often the segment occurred in the dynamic program trace. Instruction segments within a CIST are unique and are ordered based on their first occurrence in the dynamic trace. Each instruction segment in the CIST contributes one new instruction, the defining instruction of the segment. Because instruction segments overlap, a particular instruction may be in multiple CIST entries. Figure 5 shows the CIST corresponding to a sequence of dynamic instructions (numbered in program order) and their instruction segments. In this simple example none of the segments shown are canonically equivalent to each other, so no compression occurs. Note that the CIST also records the total number of instructions analyzed during trace compression.

Instruction cache and branch predictor locality events are captured by functionally simulating their corresponding structures during trace compression. For each dynamic instruction, AXCIS determines if it hit in the instruction cache, if it was correctly predicted, and if it follows a taken branch. Like the data cache, only the organizations of these structures are specified during simulation, allowing the CIST to support a variety of instruction caches and branch predictors that differ in access latencies and misprediction penalties. CISTs can be made more general by simultaneously simulating multiple caches and predictors during trace compression to create different segments for the same dynamic instruction. Then all segments can be compressed into one multi-configuration CIST, where each CIST entry has a separate frequency count for each cache and branch predictor configuration.

## 4. Dynamic Trace Compressor

The Dynamic Trace Compressor (DTC) functionally simulates the benchmark and creates an instruction segment for each dynamic instruction. Compression occurs when a new segment is determined to be equivalent to an existing one in the CIST. In this case, the DTC increments the existing segment's frequency count. Otherwise, the new segment is added to the CIST.

In an ideal compression scheme, two instruction segments are only compressed together if their defining instructions will experience the same set of stalls for all machine configurations. In this case, there will be no loss of accuracy due to compression. Since it is impractical for the DTC to determine the set of all possible stalls for every instruction under all configurations, we propose three heuristic compression schemes that approximate the ideal scheme with different tradeoffs between compression and accuracy:

**Limit Configurations-Based Compression Scheme.** The intuition behind this *limit-based scheme* is that if instruction segments have the same stall cycles in two very different configurations, then they are more likely to have the same stall cycles in all configurations. In this scheme, the DTC simulates the minimum and maximum microarchitecture configurations that will be simulated with the resulting CIST to calculate, for each dynamic instruction, a pair of stall cycles and *structural occupancies* to approximate its set of all possible stalls. Structural occupancies are snapshots of microarchitectural state (e.g., issue group size) at the time an instruction is evaluated for issue. Two segments are compressed if their defining instructions have the same (1) instruction types, (2) min/max stall cycles, and (3) min/max structural occupancies.

**Relaxed Limit Configurations-Based Compression Scheme.** In this *relaxed limit-based scheme*, only the defining instruction types and min/max stalls are compared for equality. By not comparing structural occupancies, this relaxed scheme can result in higher compression but lower accuracy compared to the limit-based scheme.

**Instruction Segment Characteristics-Based Compression Scheme.** The intuition behind this *characteristics-based scheme* is that if instruction segments look the same (i.e. have the same instruction segment characteristics), then they are more likely to have the same number of stall cycles under all configurations. In this scheme, two segments are equal if they have (1) equivalent segment lengths, (2) identical instruction types for each instruction in the segment, (3) identical instruction cache and branch prediction locality characteristics for each instruction in the segment, and (4) equivalent dependence distances for all but the first instruction in the segment.

By varying the compression scheme, the DTC can adjust CIST size and accuracy without affecting the operation of the performance model. Because different types of benchmark differ in their characteristics and instruction segment profiles, we found that each compression scheme performed best (in terms of speed and accuracy) for some different subset of the 24 SPEC CPU2000 benchmarks studied.

## 5. AXCIS Performance Model

The APM performs a single linear dynamic programming pass over the instruction segments in a CIST to compute the performance of a processor configuration. Performance can be expressed in instructions per cycle (IPC) as:

$$IPC = \frac{Total\_Ins}{Total\_Ins + Total\_Effective\_Stall\_Cycles} \tag{1}$$

where the denominator equals total cycles. Because the *total number of instructions* is recorded in the CIST, the main job of the APM is to calculate the *total effective stall cycles*.

Each instruction in the CIST experiences some number of *effective stall cycles* due to its dependencies. The APM calculates the total effective stall cycles by taking the sum of the effective stall cycles of each defining instruction in the CIST weighted by the frequency counts of the corresponding instruction segments, as follows:

$$Total\_Effective\_Stall\_Cycles = \sum_{i=1}^{CIST\_Size} Freq(i) * Effective\_Stall\_Cycles(Def\_Ins(i)) \tag{2}$$

where $i$ indexes over instruction segments.

Using dynamic programming, the APM traverses the CIST and analyzes each instruction segment in order. For each segment, the APM need only calculate the stall cycles of the defining instruction because those of earlier instructions in the segment have already been calculated. Using this efficient method, the amount of work required to calculate the *total effective stall cycles* is directly proportional to the number of segments in the CIST, which can be many thousands of times smaller than the total dynamic instructions in the original trace.

The effective stall cycles of an instruction are determined by its (1) data dependencies, (2) functional-unit structural hazards (3) primary-miss tag structural hazards, and (4) control flow events from the instruction cache and branch predictor. Each of these factors causes some number of stall cycles (positive, 0, or $-1$). If an instruction is influenced by more than one factor, its effective stall cycles are the maximum of all its stall cycles. When an instruction issues in parallel with one or more preceding instructions, its effective stall cycles are set to $-1$. This allows the total effective stall cycles to be negative to achieve an IPC greater than 1. In the following, we describe how to compute the stalls caused by each factor.

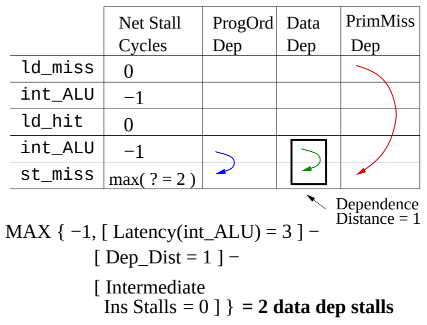


**Figure 6. Calculation of stall cycles from data dependencies.**

**Data Dependencies.** Data dependencies include RAW and cache-line dependencies, whose stall cycles are expressed by the following equation:

$$Data\_Dep\_Stalls(consumer) = MAX\,[-1,\ Latency(producer) - Dep\_Dist - \sum_{i=producer+1}^{consumer-1} Effective\_Stall\_Cycles(ins_i)\ ] \tag{3}$$

The latency of the producer is provided by the input configuration, and the dependence distance is recorded with the instruction. The effective stalls of the *intermediate instructions* between the producer and consumer ($Effective\_Stall\_Cyc(ins_i)$) have already been calculated and can be looked up in earlier segments in the CIST.

Figure 6 calculates the data dependency stalls for the `st_miss` defining instruction. The producer associated with the data dependency is the latest `int_ALU` instruction, which is a distance of 1 away from the `st_miss`. Assuming that ALU instructions take 3 cycles to complete, the `st_miss` experiences 2 data dependency stalls.

**Functional-Unit Structural Hazards.** To detect structural hazards on issue width and functional units, the APM models the relevant microarchitectural state associated with each instruction using *issue group* and *functional-unit structural occupancies*. An issue group occupancy is an integer representing the number of instructions in an issue group. A functional-unit

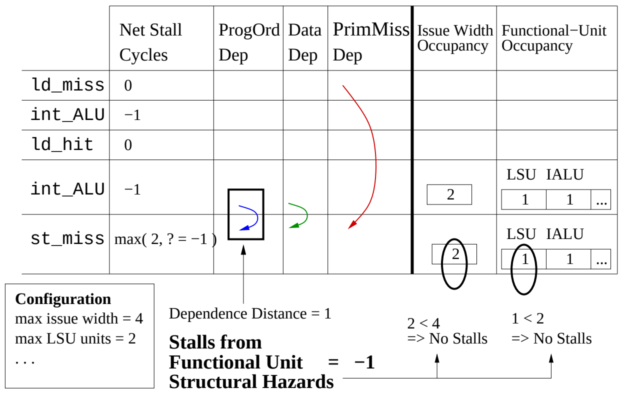


**Figure 7. Calculation of stalls from functional-unit structural hazards.**

occupancy is an array of integers, where each element represents the number of units allocated for a particular instruction type in the issue group.

Using the same segment as before, Figure 7 calculates the stalls of the `st_miss` instruction arising from functional-unit structural hazards. First, the occupancies of the `st_miss` are initialized to match those of the producer of the program-order dependency (the latest `int_ALU` instruction). Then the APM identifies any structural hazards by comparing the modeled occupancies with the machine parameters in the input configuration. In this example, there is an available LSU and enough issue bandwidth to allow the `st_miss` to issue in the current issue group. Therefore the functional-unit structural hazard stalls are set to $-1$. However, if structural hazards had been detected, these stalls would have been set to 0, indicating that the instruction issues in a new group. Currently, we only model fully pipelined functional units, but we can extend the APM to model partially pipelined units by applying the technique used to model primary-miss tags described below.

**Primary-Miss Tag Structural Hazards.** To detect structural hazards on the primary-miss tags, the APM models *primary-miss tag occupancies* which are arrays that capture the state of the primary miss tags when an instruction issues. The size of each array corresponds to the number of primary miss tags in the configuration, and each element represents the number of cycles until the tag becomes available.

Continuing with our example, Figure 8 calculates the stalls of the the `st_miss` arising from primary miss tag structural hazards. First, the miss tag occupancy is initialized to that of the producer associated with the primary-miss dependency, which in this case is the `ld_miss`. After initialization, the APM updates the miss tag occupancy to correspond to the current cycle, instead of the cycle the `ld_miss` was issued. To do this, the APM computes the number of elapsed cycles since the `ld_miss` was issued by summing the dependence distance to the `ld_miss` (4) with the net stalls experienced by all intermediate instructions ($-2$). Then the APM subtracts the elapsed cycles (2) from each entry of the miss tag occupancy. Next, the APM calculates the stalls by finding the minimum value in the miss tag occupancy, which in this case is 28 cycles.

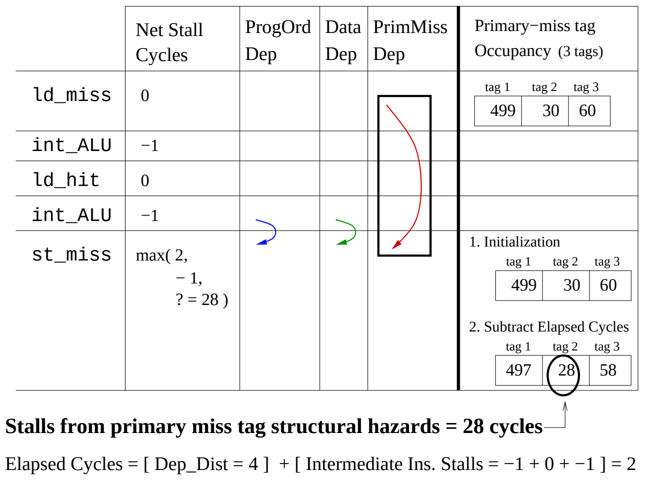


**Figure 8. Calculation of stalls from primary-miss tag structural hazards.**

| Icache | Branch Pred. | Stall Cycles |
|---|---|---|
| hit | incorrect & taken/not taken | mispred. penalty |
| hit | correct & taken | 0 |
| hit | correct & not taken | -1 |
| miss | incorrect& taken/not taken | memory latency + mispred. penalty |
| miss | correct& taken | memory latency |
| miss | correct& not taken | memory latency - 1 |

**Table 1. Mapping of icache and branch prediction status flags to control flow stalls.**

**Control Flow Events.** Stalls from control flow events are caused by instruction cache misses, branch mispredictions, and correctly predicted taken branches. These locality events directly map to an instruction's control flow stall cycles, as shown in Table 1. In our sample instruction segment, if the `st_miss` hit in the instruction cache, is correctly predicted by the branch predictor, and does not follow a taken branch, it will experience $-1$ control flow stalls.

**Calculating Net Stall Cycles.** After calculating the stalls from each dependency, the APM computes the effective stall cycles of an instruction by taking the maximum of all its stalls. In our example, the `st_miss` instruction experiences $MAX(2,-1,28,-1) = 28$ effective stall cycles. Next, the APM uses the effective stall cycles to update the instruction's structural occupancies to correspond to the microarchitectural state after the instruction issues.

| Parameter | Configurations | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| issue width | 1 | 1 | 1 | 1 | 4 | 4 | 4 | 4 | 8 | 8 | 8 | 8 |
| # primary-miss tags | 1 | 1 | 8 | 8 | 1 | 1 | 8 | 8 | 1 | 1 | 8 | 8 |
| memory latency | 10 | 200 | 10 | 200 | 10 | 200 | 10 | 200 | 10 | 200 | 10 | 200 |
| # units: | | | | | | | | | | | | |
| int alu/nop | 1 | 1 | 1 | 1 | 4 | 4 | 4 | 4 | 8 | 8 | 8 | 8 |
| int mult/div | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 4 |
| fbat add/cmp/cvt | 1 | 1 | 1 | 1 | 4 | 4 | 4 | 4 | 8 | 8 | 8 | 8 |
| fbat mult/div/sqrt | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 4 |
| load/store | 1 | 1 | 1 | 1 | 4 | 4 | 4 | 4 | 8 | 8 | 8 | 8 |

**Table 2. Twelve simulated configurations that span a large design space.**

| Parameter | Value |
|---|---|
| int alu/nop latency | 1 |
| int mult latency | 8 |
| int div latency | 16 |
| fbat add/cmp/cvt latency | 4 |
| fbat mult latency | 4 |
| fbat div latency | 16 |
| fbat sqrt latency | 24 |
| L1 latency | 3 |
| branch misprediction penalty | 3 |
| L1 instruction cache | 16KB, direct mapped, 32 byte blocks |
| L1 data cache | 16KB, 4-way associative, 32 byte blocks |
| Memory access bus width | 32 bytes |
| Branch predictor | bimodal |
| Branch target buffer | 512 sets, 4-way associative |
| Return address stack | 8 entries |

**Table 3. Shared configuration parameters.**

| Parameter | Min Config | Max Config |
|---|---|---|
| issue width | 1 | 10 |
| # primary miss tags | 1 | 20 |
| # units for each instruction type | 1 | 10 |
| branch misprediction penalty | 9 | 1 |
| latencies: | | |
| int alu/nop | 9 | 1 |
| int mult | 72 | 4 |
| int div | 144 | 8 |
| fbat add/cmp/cvt | 36 | 2 |
| fbat mult | 36 | 3 |
| fbat div | 144 | 8 |
| fbat sqrt | 216 | 10 |
| L1 cache access | 27 | 2 |
| memory access | 450 | 8 |

**Table 4. Min and max configurations used by limit-based and relaxed limit-based compression schemes.**

## 6. Experimental Setup

We evaluated AXCIS against our baseline cycle-accurate simulator, SimInOrder, which models the same processor characteristics as AXCIS. Both AXCIS and SimInOrder are implemented on top of `sim-safe`, an instruction-level execution-driven simulator, and the `cache` and `bpred` frameworks from the SimpleScalar 3.0 tool set [3]. In our experiments, we used the Alpha binaries [13] of 24 SPEC CPU2000 benchmarks and their corresponding `reference` input sets (one input was selected for benchmarks with multiple input sets) [14]. We could not run the other two SPEC benchmarks, `fma3d` and `sixtrack`, due to problems with our simulation framework.

In order to examine the behavior of AXCIS across a wide range of designs, we simulated 12 machine configurations, shown in Table 2 that differ in issue width, number of primary miss tags, memory latency, and number of functional units for each instruction type. Table 3 shows the functional unit latencies, cache, and branch predictor parameters that are shared by all configurations.

Table 4 shows the limiting configurations used in the limit-based and relaxed limit-based compression schemes. The minimum configuration is composed of parameters describing minimum bandwidth and maximum latency, while the maximum configuration consists of parameters describing maximum bandwidth and minimum latency.

## 7. Evaluation

We evaluated AXCIS for both accuracy and speed. Accuracy is measured in terms of the absolute IPC error between results obtained from AXCIS and SimInOrder.

$$\text{Absolute IPC Error} = 100 \times \frac{\mid \mathit{AXCIS\_IPC} - \mathit{SimInOrder\_IPC} \mid}{\mathit{SimInOrder\_IPC}}$$

Speed is measured in terms of the number of CIST entries, an implementation-independent metric that has a linear relationship with the APM's simulation time. For reference, we also show the APM's execution times for our implementation.

Before presenting results, we first provide some intuition behind AXCIS's behavior. A dynamic trace can be represented by a long chain of unique overlapping instruction segments, one per dynamic instruction. If this *entire* chain were stored in a CIST and simulated by the APM, the IPC error would be zero. But, without compressing the instruction segments, the APM would (1) achieve no speedup over detailed simulation because it would still be calculating stall cycles for every dynamic instruction, and (2) the CIST would be much larger than the program's dynamic trace, since the CIST stores one instruction segment per dynamic instruction.

Using an ideal lossless compression scheme, the DTC would compress together all instruction segments that have the same stall cycles across all configurations, allowing the APM to achieve perfect accuracy with reduced simulation time. Ideal lossless compression is impractical to implement and, in any case, lossless compression is not ideal if we are prepared to trade some accuracy for reduced simulation time. Each of our proposed compression schemes approximates the ideal compression scheme while selecting a different tradeoff between simulation speed and accuracy. In the following subsections, we evaluate AXCIS using the limit-based, relaxed limit-based, and characteristics-based compression schemes, referred to as AXCIS_LB, AXCIS_RLB, and AXCIS_CB respectively. More detailed results on these schemes can be found in [7].

## 7.1 Limit-Based Scheme

We evaluated AXCIS_LB on the 24 benchmarks for 10 billion dynamic instructions. However, since our implementation of the AXCIS DTC and APM keep the entire CIST in memory, we had to terminate five integer benchmarks (`crafty`, `mcf`, `parser`, `twolf`, and `vpr`) at around 1.4, 6.7, 3.6, 3.5, and 3.5 billion instructions when their CISTs grew too large to fit in the 3GB virtual memory limit on our 32-bit Linux machines. This memory limitation can be addressed by (1) running in a larger virtual address space (e.g. 64 bit machine), (2) augmenting the implementation so that only portions of the CIST need to be in memory, and (3) improving the compression scheme. We chose to focus on improving the compression scheme since it also decreases modeling time. The results of our improved scheme (AXCIS_RLB) for these five benchmarks are presented in Section 7.2. In this section, we present and analyze the results of the 19 benchmarks that compress well using AXCIS_LB.

Figure 9 (a) shows the distribution of absolute IPC errors of AXCIS_LB, specified in quartiles. Each benchmark was run for 10 billion instructions to generate a single CIST used to simulate all 12 configurations. For 15 of the 19 benchmarks, AXCIS_LB is highly accurate and configuration independent. The average IPC error over all configurations and benchmarks is 4.8%. AXCIS_LB performs better on the integer benchmarks, with an average error of 2%, while the average error for the floating-point benchmarks is 6.5%. The median error for each benchmark is within 10.8%, and 15 of the 19 benchmarks have median IPC errors less than 5%. Except for four floating-point benchmarks (`applu`, `facerec`, `galgel`, and `mgrid`), the maximum error for each benchmark is within 10%.

The limit-based compression scheme performs poorly for `applu`, `facerec`, `galgel`, and `mgrid` because many instruction segments that it considers equivalent (i.e., have the same stall cycles) in the min and max configurations have very different stall cycles in some intermediate configurations. Since the resulting CIST does not distinguish between these segments, the stall cycles calculated in these outlying configurations have high errors. Another potential cause for error is that CISTs do not capture dependencies that span more than 512 instructions, which affects the modeling of structural occupancies, particularly the primary miss tags. Section 7.3 presents results for AXCIS_CB which achieves high accuracy for these four floating point benchmarks.

Figure 9 (b) shows the number of CIST entries and the average APM execution time for each benchmark. Without CISTs, a conventional simulator would have to simulate all 10 billion dynamic instructions, which takes around 5 hours on SimInOrder. Using AXCIS_LB the APM analyzes around 260,000 instructions on average for each benchmark, taking about 0.72 seconds. The minimum and maximum number of instructions analyzed by the APM are 5,721 and 1.29 million instructions, taking on average 0.02 seconds and 3.1 seconds, for `wupwise` and `perlbmk` respectively. While the number of CIST entries for a benchmark is constant over all configurations, the APM execution time varies because the amount of work done at each CIST entry varies slightly depending on the configuration.

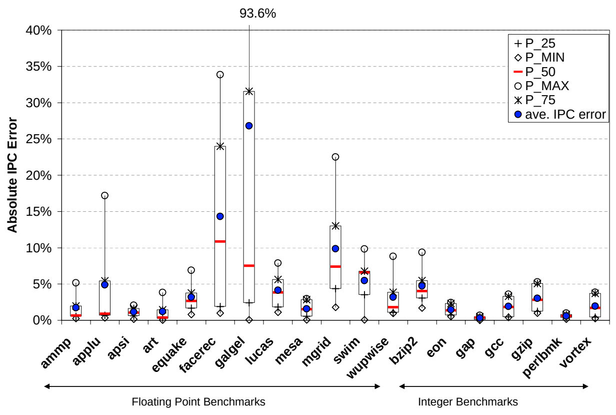
## 7.2 Relaxed Limit-Based Scheme

Using a more relaxed definition of instruction segment equality, AXCIS_RLB decreases the number of CIST entries of the five integer benchmarks (`crafty`, `mcf`, `parser`, `twolf`, and `vpr`) by 24% on average, compared to AXCIS_LB. The DTC was able to finish all 10 billion dynamic instructions for the two benchmarks (`crafty`, and `mcf`). However, the DTC ran out of memory after 4.8, 5.7, and 9 billion instructions for `vpr`, `twolf`, and `parser`, respectively. We present the results of AXCIS_RLB on these 5 benchmarks for 4 billion dynamic instructions. Due to limited machine time and the long simulation times of SimInOrder, we decreased the number of simulated configurations from 12 to the 6 shown in the even columns of Table 2, while still covering a large design space.

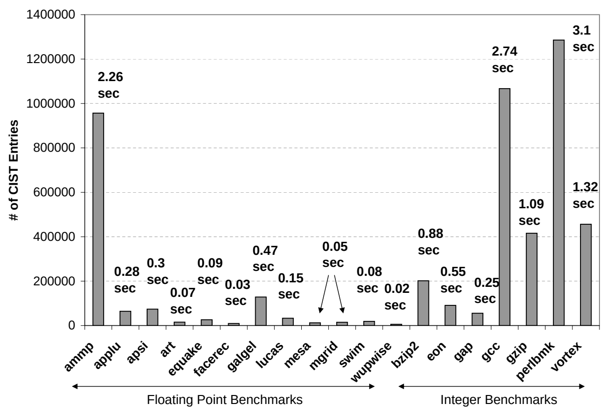
Figure 10 (a) shows that AXCIS_RLB is highly accurate and configuration independent for these benchmarks, achieving an average IPC error of 2.6% and tight error distributions. As shown in Figure 10 (b), the number of CIST entries is significantly less than the 4 billion dynamic instructions simulated, allowing the APM to achieve average execution times on the order of seconds for each benchmark. AXCIS_RLB performs well even with a loose definition of segment equality because these five integer benchmarks have limited ILP, and therefore structural occupancies are less important in determining stall cycles across configurations. Note that AXCIS_LB would be even more accurate than AXCIS_RLB, as it has a strictly more discriminative equality check.

## 7.3 Characteristics-Based Scheme

AXCIS_CB, is much more accurate than AXCIS_LB for `applu`, `facerec`, `galgel`, and `mgrid`. However, this accuracy came at the cost of compressibility. Figure 11 (a) compares their absolute IPC error distributions using the

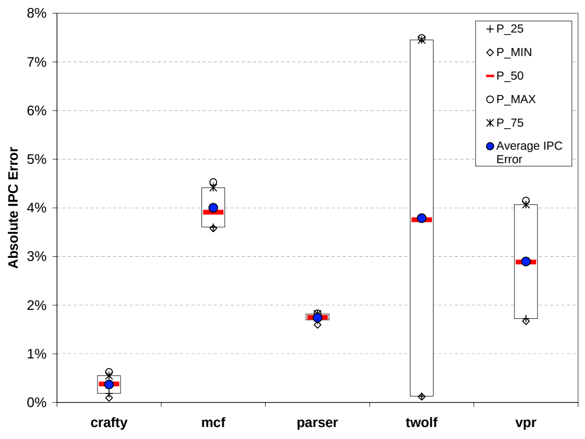


(a) Absolute IPC errors for 19 benchmarks and 12 configurations.

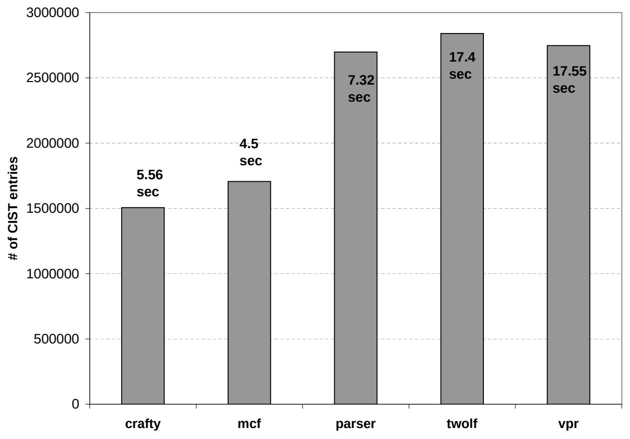


(b) Number of CIST entries and average APM execution times.

**Figure 9. Evaluation of AXCIS using the limit-based compression scheme in terms of (a) accuracy and (b) speed.**

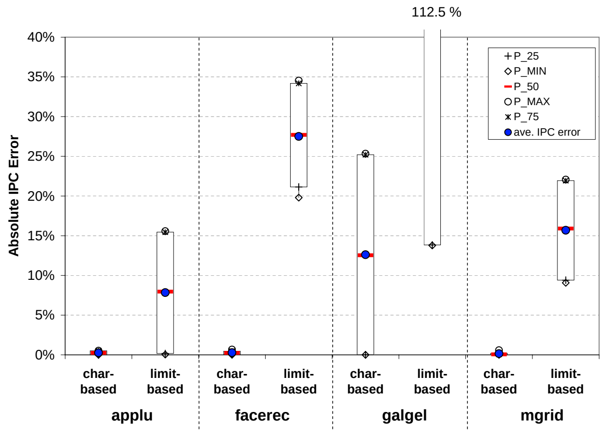


(a) Absolute IPC error.

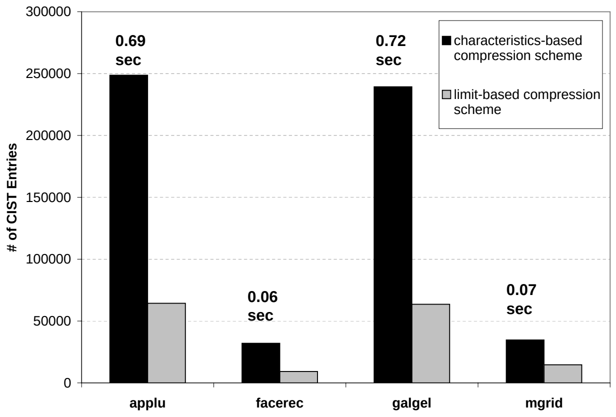


(b) Number of CIST entries and average APM execution times.

**Figure 10. Evaluation of AXCIS using the relaxed limit-based compression scheme in terms of (a) accuracy and (b) speed.**



(a) Comparison of the IPC errors obtained using the characteristics-based and limit-based compression schemes.



(b) Comparison of the number of CIST entries obtained using the characteristics-based and limit-based compression schemes.

**Figure 11. Comparison of AXCIS using the characteristics-based and limit-based compression schemes in terms of (a) accuracy and (b) speed.**

characteristics-based and limit-based compression schemes. Both sets of data correspond to CISTs generated from 3 billion dynamic instructions and the 6 configurations described in the even columns of Table 2. Using AXCIS_CB, the average IPC error of these four benchmarks was reduced from 13.97% to 3.33%, and their maximum errors decreased by 96.6% (applu), 98.1% (facerec), 77.5% (galgel), and 97.3% (mgrid). With tight error distributions, AXCIS_CB is both highly accurate and configuration independent.

AXCIS_CB uses a stricter definition of instruction segment equality than AXCIS_LB. Being more conservative, AXCIS_CB compresses fewer segments together, resulting in increased accuracy but more CIST entries. Figure 11 (b) compares the number of CIST entries and average APM execution times of AXCIS_CB to AXCIS_LB for the four floating point benchmarks. As expected, the number of CIST entries increases using the characteristics-based scheme, although AXCIS_CB is still able to complete the simulations within seconds.

## 7.4 Dynamic Trace Compression Speed

CIST generation should take about as long as functional simulation because the dynamic trace compressor analyzes each dynamic instruction. However, our unoptimized trace compressor implementation was about four times slower than SimInOrder because it encountered long collision chains in the simple hash table used to perform the matching of new instruction segments to existing ones. An optimized implementation should perform much better. Nevertheless, CIST generation is performed only once per benchmark so even with our current implementation, AXCIS is much faster than detailed simulation when exploring many design points.

## 7.5 Discussion

Because the compression scheme does not affect the calculations performed by the APM, the DTC can use different compression schemes to create CISTs for different benchmarks. We define the best scheme for a benchmark to be the one resulting in the highest compression, while still maintaining sufficient accuracy. More relaxed compression schemes such as AXCIS_RLB and AXCIS_LB highly value compression. In general, these schemes are best for codes that are less repetitive and require the added emphasis on compression in order to generate compact CISTs. Integer codes, which tend to have large instruction working sets and low instruction-level parallelism (ILP), fall into this category. Due to low ILP, more stalls arise from data hazards than structural hazards. Therefore matching structural occupancies is not as important for the integer codes as the floating point codes, which have high ILP. Stricter compression schemes such as AXCIS_CB value accuracy over compression. These schemes generally work best for codes with high ILP and lots of repetition (e.g., floating point benchmarks) that compensate for the lower emphasis on compression to generate compact CISTs. Also, instruction segments in highly repetitive codes have more similarities and may be harder to distinguish, requiring the stricter equality check.

From our three compression schemes, we found that AXCIS_LB was best for 15 of the 24 benchmarks, AXCIS_RLB was best for crafty, mcf, parser, twolf, and vpr, while AXCIS_CB was best for applu, facerec, galgel, and mgrid. Using the best scheme for each benchmark, AXCIS is highly accurate and configuration independent, achieving an average IPC error of 2.6% with an average error range of 4.4%. Except for `galgel` with a maximum error of 25.3%, the maximum error of all benchmarks is less than 10%. AXCIS is also very fast, completing simulations corresponding to billions of dynamic instructions within seconds. Using pre-generated CISTs, AXCIS is over 10,000 times faster than detailed simulation.

In addition to estimating performance, AXCIS can be used to predict other metrics such as buffer occupancies and functional unit utilization because all necessary information is already stored in the CIST. For example, we already estimate structural/buffer occupancies in order to calculate stall cycles. Also, since the absolute accuracy of AXCIS is mostly consistent across a broad range of configurations for each benchmark, AXCIS should track design changes and identify interesting portions of a performance curve.

Most of the ideas addressed in this paper can apply to both out-of-order and in-order processors (e.g., instruction segments, CISTs, the Dynamic Trace Compressor methodology, and instruction segment compression schemes). Only the stall calculation algorithm performed by the APM requires considerable changes to model out-of-order processors. Note that we already deal with buffered state and long overlapped stall latencies, e.g., in our handling of primary miss stalls. To model more complex out-of-order processors, the stall calculation algorithm performed on each defining instruction in the CIST would need to also take into account the stalls of a limited number of defining instructions that occur *later* in the CIST, in addition to the stalls of instructions that are within the defining instruction's segment. We leave this for further work. Also, AXCIS can be easily extended to model other ISAs. For example, to model a CISC ISA, we would first translate complex instructions into RISC-like ops (micro-ops) and then generate the CIST. AXCIS could also be modified to model VLIW ISAs, since we already model multiple-issue machines.

## 8. Related Work

The time-stamping algorithm used by Loh [8] efficiently approximates detailed simulation but does not reduce the number of simulated instructions. This technique is similar to the way we model structural occupancies. However, since the AXCIS Performance Model analyzes only the defining instructions in CISTs, the runtime of AXCIS should be much faster.

Iyengar et. al. [5] generated reduced traces based on fully qualified instructions, which are similar to our instruction segments except they contain only the previous *n* instructions as context. Our instruction segments more accurately capture dynamic instruction characteristics by adjusting segment lengths to include all producing instructions associated with relevant dependencies. Also, their generated traces are tailored towards a specific system, making their scheme less appropriate for large design studies.

In statistical simulation, statistical profiles on program and locality characteristics are gathered from a program's dynamic execution to generate a synthetic trace for simulation. Simpler models generated traces based on global distributions of program and locality characteristics, while more detailed models gathered distributions as a function of some parameter such as basic block size. In general, more detailed statistical models lead to higher accuracy. However, as the complexity of the models increases, the difficulty in generating synthetic traces to match those models also increases. AXCIS bypasses this problem by directly calculating IPC from the instruction segments in CISTs, instead of simulating from a generated synthetic trace. Statistical simulation techniques such as [9, 11] created synthetic traces by randomly generating instructions from statistical profiles. These techniques experienced high errors for certain benchmarks and designs because the original instruction sequences and relationships were not maintained in the synthetic trace. Eeckhout et. al. [4] improved these statistical techniques by introducing control flow modeling using statistical flow graphs (SFG) on basic blocks. Although SFGs enabled Eeckhout to capture the original instruction sequences and dependencies within basic blocks included in each node of the graph, these original sequences and dependencies were not maintained across the basic blocks of different nodes. AXCIS generates CISTs by compressing away redundant sequences of instructions. Unlike synthetic traces, CISTs naturally capture original dynamic instruction sequences to faithfully maintain all necessary relationships between dynamic instructions, as well as their program and locality characteristics. CISTs can even accurately represent applications with multiple phases such as `gcc` and highly repetitive code patterns such as the floating point benchmarks, allowing AXCIS to achieve high accuracy across a wide range of benchmarks and configurations.

After profiling dynamic program execution, Ofelt et. al. [10] computed machine performance by aggregating the estimated performance of individual paths. Path performance was approximated using a combination of simulation and analytical models that accounted for complicated relationships between base and successor paths. Because instruction segments capture all necessary relationships between individual instructions, the AXCIS performance model does not need to model intricate relationships between disjoint instruction sequences. Another difference is that AXCIS uses dynamic programming to directly calculate performance, which can be faster than simulation.

AXCIS differs from these above works in that AXCIS models in-order processors, which is easier in some ways but harder in others. For example, because the reorder buffer in out-of-order cores averages the stalls from dependencies, out-of-order cores may be easier to model using statistical simulation than in-order cores.

## 9. Conclusion

This paper presented AXCIS, a framework for accelerating architectural simulation in large design space studies. Using *instruction segments* that elegantly encapsulate all important program characteristics surrounding a dynamic instruction, AXCIS compresses a program's dynamic instruction stream into a Canonical Instruction Segment Table (CIST) that is used to quickly and accurately simulate a large number of designs. We proposed and evaluated three compression schemes, each with a distinct accuracy vs. speed tradeoff. Using the best compression scheme for each workload, AXCIS is highly accurate and configuration independent, achieving an average IPC error of 2.6%. Using pre-computed CISTs, AXCIS is over 10,000 times faster than detailed simulation.

## 10. Acknowledgments

We thank the anonymous reviewers and members of the MIT SCALE group, especially Heidi Pan, Ken Barr, and Chris Batten for their comments on the paper. This work was partly funded by the DARPA HPCS/IBM PERCS project, an NSF Graduate Fellowship, and NSF CAREER Award CCR-0093354.

## References

[1] T. Austin, D. Ernst, E. Larson, C. Weaver, R. Desikan, R. Nagarajan, J. Huh, B. Yoder, D. Burger, and S. Keckler. SimpleScalar tutorial (for release 4.0). In *International Symposium on Microarchitecture (MICRO-34)*, Dec. 2001.

[2] M. V. Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. In *International Conference on High Performance Embedded Architectures and Compilers*, Nov. 2005.

[3] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, pages 13–25, June 1997.

[4] L. Eeckhout, R. B. Jr., B. Stougie, K. Bosschere, and L. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *International Symposium on Computer Architecture (ISCA-31)*, June 2004.

[5] V. S. Iyengar and L. H. Trevillyan. Evaluation and generation of reduced traces for benchmarks. Technical Report RC 20610, IBM Research Division, T. J. Watson Research Center, Oct. 1996.

[6] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(2):10–13, June 2002.

[7] R. F. Liu. AXCIS: Rapid architectural exploration using canonical instruction segments. Master's thesis, Massachusetts Institute of Technology, 2005.

[8] G. Loh. A time-stamping algorithm of efficient performance estimation of superscalar processors. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2001.

[9] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, pages 15–24, Sept. 2001.

[10] C. Ofelt and J. L. Hennessy. Efficient performance prediction for modern microprocessors. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2000.

[11] M. Oskin, F. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *International Symposium on Computer Architecture (ISCA-27)*, pages 71–82, June 2000.

[12] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-10)*, Oct. 2002.

[13] SPEC2000 alpha binaries. from SimpleScalar LLC. http://www.simplescalar.com.

[14] SPEC CPU2000 benchmark suite. Standard Performance Evaluation Corporation. http://www.spec.org/cpu2000/.

[15] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2005.

[16] R. E. Wunderlich, T. F. Wenish, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *International Symposium on Computer Architecture (ISCA-30)*, June 2003.

[17] J. Yi, S. Kodakara, R. Sendag, D. Lilja, and D. Hawkins. Characterizing and comparing prevailing simulation techniques. In *International Symposium on High-Performance Computer Architecture (HPCA-11)*, Feb. 2005.