AXCIS: Rapid Processor Architectural Exploration using Canonical Instruction Segments

by

Rose F. Liu

S.B., Massachusetts Institute of Technology (2004)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2005

© Massachusetts Institute of Technology 2005. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
August 16, 2005

Certified by . . . .
Krste Asanović
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

AXCIS: Rapid Processor Architectural Exploration using Canonical Instruction Segments

by

Rose F. Liu

Submitted to the Department of Electrical Engineering and Computer Science
on August 16, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In the early stages of processor design, computer architects rely heavily on simulation to explore a very large design space. Although detailed microarchitectural simulation is effective and widely used for evaluating different processor configurations, long simulation times and a limited time-to-market severely constrain the number of design points explored. This thesis presents AXCIS, a framework for fast and accurate early-stage design space exploration. Using instruction segments, a new primitive for extracting and representing simulation-critical data from full dynamic traces, AXCIS compresses the full dynamic trace into a table of canonical instruction segments (CIST). CISTs are not only small, but also very representative of the dynamic trace. Therefore, given a CIST and a processor configuration, AXCIS can quickly and accurately estimate performance metrics such as instructions per cycle (IPC). This thesis applies AXCIS to in-order superscalar processors, which are becoming more popular with the emergence of chip multiprocessors (CMP). For 24 SPEC CPU2000 benchmarks and all simulated configurations, AXCIS achieves an average IPC error of 2.6% and is over four orders of magnitude faster than conventional detailed simulation. 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.

Thesis Supervisor: Krste Asanović
Title: Associate Professor

Acknowledgments

First, I would like to thank Krste Asanović for being such a dedicated and supportive advisor. I am immensely grateful for his invaluable guidance and encouragement throughout this work.

I would like to thank Pradip Bose, at IBM Research, for introducing me to the field of processor simulation. During my summers at IBM, he has been a wonderful mentor, providing me with the skills and knowledge to pursue this work.

I also would like to thank the members of the SCALE Group for all their help and support. I would like to thank my parents, Francis and Chia, for their love and encouragement. And finally, I would like to thank Vladimir for being the best husband I could ask for.

This work was supported by an NSF Graduate Fellowship and the DARPA HPCS/IBM PERCS project number W0133890.

Contents

1 Introduction **13**

2 Related Work **15**

2.1 Reduced Input Sets 16

2.2 Sampling 16

2.3 Synthetic Trace Simulation 17

2.4 Analytical Modeling 18

2.5 AXCIS 18

3 AXCIS Framework **19**

3.1 Overview 19

3.2 Dynamic Trace Compression 22

3.2.1 Identifying Instruction Segments 22

3.2.2 Instruction Segment Anatomy 25

3.2.3 Creating the Canonical Instruction Segment Table 29

3.2.4 CIST Data Structure 32

3.2.5 Dynamic Trace Compression: An Example 33

3.3 AXCIS Performance Model 34

3.3.1 Data Dependency Stalls 35

3.3.2 Primary-Miss Dependency Stalls 36

3.3.3 Control Flow Event Stalls 37

3.3.4 Program-Order Dependency Stalls 38

3.3.5 Calculating Net Stall Cycles 40

3.3.6 Calculating IPC 40
3.4 Stall Calculation during Dynamic Trace Compression 41

4 Evaluation **43**
4.1 Experimental Setup 43
4.2 Results 46
4.2.1 AXCIS Accuracy 46
4.2.2 AXCIS Performance Model Simulation Speed 51
4.2.3 CIST Size 52

5 Alternative Compression Schemes **55**
5.1 Compression Scheme based on Instruction Segment Characteristics 56
5.2 Relaxed Compression Scheme 58
5.3 Optimal Compression Scheme for each Benchmark 61

6 Conclusion **69**
6.1 Summary of Contributions 69
6.2 Future Work 70
6.3 Additional Applications 70

List of Figures

3-1 Top level block diagram of AXCIS. 20
3-2 Example of an instruction segment. 21
3-3 Example of two overlapping instruction segments. 21
3-4 Class of machines supported by AXCIS 22
3-5 Instruction entry format. 26
3-6 Anatomy of an instruction segment. 27
3-7 Sample dependencies recorded for different instruction types. 28
3-8 Example of a CIST. 33
3-9 CIST building example. 34
3-10 Structural occupancies for each CIST entry. 38

4-1 Absolute IPC errors for 19 benchmarks and 12 configurations. 47
4-2 IPC errors of (a) `applu`, (b) `facerec`, (c) `galgel`, and (d) `mgrid` for each configuration. 48
4-3 Absolute IPC error for 3 configurations with various latencies. 50
4-4 Normalized APM execution time vs. normalized number of CIST entries. . 52
4-5 Number of CIST entries and average APM execution times. 53
4-6 Number of instruction entry references in each CIST. 54

5-1 Comparison of the IPC errors obtained under the characteristics-based and limit-based compression schemes. 57
5-2 Comparison of the number of CIST entries obtained under the characteristics-based and limit-based compression schemes. 58

5-3 Comparison of the number of instruction entry references within a CIST, obtained under the characteristics-based and limit-based compression schemes. 59
5-4 Number of CIST entries obtained under the relaxed and limit-based compression schemes. 60
5-5 Number of instruction entry references in each CIST obtained under the relaxed and limit-based compression schemes. 61
5-6 Absolute IPC error obtained under the relaxed compression scheme. 62
5-7 Number of CIST entries and average APM execution times obtained under the relaxed compression scheme. 63
5-8 Number of instruction entry references in each CIST obtained under the relaxed compression scheme. 64
5-9 Absolute IPC error for each benchmark obtained under its optimal compression scheme. 65
5-10 Number of CIST entries and average APM execution times obtained under the corresponding optimal compression scheme. 66
5-11 Number of instruction entry references in each CIST obtained under the corresponding optimal compression scheme. 67

List of Tables

3.1 Processor confi guration parameters supported by AXCIS. 23
3.2 Mapping of icache and branch prediction status flags to control flow event stalls. 39

4.1 Inputs used for benchmarks with more than one reference input. 44
4.2 Minimum and Maximum processor parameters used by the DTC to generate the limiting confi gurations. 44
4.3 Twelve simulated confi gurations that span a large design space. 45
4.4 Functional unit latency parameters. 45
4.5 Cache, memory, and branch predictor confi gurations. 46
4.6 Three simulated confi gurations with various functional unit and memory latencies. 49

Chapter 1

Introduction

In the early stages of processor design, computer architects are faced with exploring a very large design space, which may include thousands of microarchitectural configurations. Cycle-accurate simulation is effective and widely used for evaluating different processor configurations. However, the long simulation times of these detailed simulators, along with a limited time-to-market, severely constrain the number of design points explored.

Current detailed simulators are over thousands of times slower than native hardware. For example, the popular simulator `sim-outorder`, of the SimpleScalar tool set [2], simulates at around 0.35 MIPS on a 1.7 GHz Pentium 4 [1]. Depending on the benchmark size and level of simulated detail, simulation time for one run varies from hours to weeks. Not only are the processors and memory systems modeled becoming more complex, additional design constraints are also being introduced for next generation processors such as power, temperature, and reliability, making simulators even more detailed. Also, benchmarks are growing in size and complexity to match those of real-world applications. For example, some benchmarks in the SPEC CPU2000 [12] suite have more than 300 billion dynamic instructions. The additional complexity in both benchmarks and simulators exacerbates long simulation time, further limiting design space exploration.

Because simulation time is a function of the dynamic program size, researchers have proposed various techniques to decrease the number of simulated dynamic instructions. These techniques include reduced input sets [6], sampling [13, 10], reduced traces [4], and statistical simulation [3, 9, 8]. The reduced input set technique modifies the input data,

while sampling selectively simulates important sections of the dynamic instruction stream. Statistical and reduced-trace simulation use short synthetic traces that are generated after profiling the original dynamic trace. However, many of these techniques experience high errors, in corner cases, because these reduced programs are missing simulation-critical data. Therefore the challenge is to extract all data that affect simulation accuracy from the full dynamic trace. A simulation technique that processes only this critical subset will minimize simulation time without sacrificing accuracy.

We introduce *instruction segments* as a new primitive for extracting and representing simulation-critical data from full dynamic traces. Simulation-critical data contained in instruction segments include original dynamic instruction sequences as well as microarchitecture independent and microarchitecture dependent characteristics. We also present AXCIS (Architectural eXploration using Canonical Instruction Segments), a new framework for fast and accurate early-stage design space exploration. AXCIS abstracts each dynamic instruction and its microarchitecture independent/dependent contexts into an instruction segment. AXCIS then uses a compression scheme to compress the instruction segments of the full dynamic trace into a table containing only canonical instruction segments (CIST - Canonical Instruction Segment Table). CISTs are not only small, but also very representative of the full dynamic trace. Therefore, given a CIST and a processor configuration, AXCIS quickly and accurately estimates performance metrics such as instructions per cycle (IPC). We propose AXCIS as a complement to detailed simulation. Because CISTs can be reused to simulate many processor configurations, AXCIS can quickly identify regions of interest to be further analyzed using detailed simulation. In this work, we apply AXCIS to in-order superscalar processors. In-order processors are becoming more popular with the emergence of chip multiprocessors (CMP), which have stricter area and power constraints and emphasize thread-level throughput over single threaded performance.

This thesis is structured as follows. Chapter 2 provides an overview of related works on efficient simulation techniques. Chapter 3 describes the AXCIS framework and instruction segments in detail. Chapter 4 evaluates AXCIS for accuracy and speedup, in comparison with a cycle-accurate simulator. Chapter 5 proposes alternative compression schemes for AXCIS and evaluates their speed and accuracy trade-offs. Chapter 6 concludes this thesis.

Chapter 2

Related Work

In large design space studies, architects may need to simulate and compare thousands of processor configurations. Since detailed simulation is too slow to complete these studies in a timely manner, much work has been done on reducing processor simulation time. Many previously proposed techniques decrease simulation time by reducing the number of dynamically simulated instructions. These techniques include reduced input sets, sampling, and synthetic trace simulation. Another approach to improving simulation time is analytical modeling, which does not involve any simulation to evaluate different configurations once the analytical equations have been specified.

All these efficient simulation techniques produce results that approximate those obtained using detailed simulation. These techniques are usually evaluated based on the absolute and relative accuracies of their approximations. Absolute accuracy, which is harder to obtain than relative accuracy, refers to the technique's ability to closely follow the values measured by the detailed simulator. Relative accuracy refers to the technique's ability to produce results that reflect the relative changes across a variety of processor configurations. While absolute accuracy requires the absolute errors of the approximations to be small, relative accuracy can be obtained when the error is consistently positive or negative over a broad range of configurations. Configuration dependence also plays a role in evaluating the accuracy of these techniques. Configuration independent techniques produce results with similar error regardless of the simulated configuration, while the error of configuration dependent techniques vary depending on the simulated microarchitecture, making it difficult

to compare configurations.

2.1 Reduced Input Sets

Reduced input sets such as MinneSPEC [6] modify the `reference` input set to reduce simulation time. Because the dynamic instruction sequence generated using reduced input sets can be very different from that generated using `reference` input sets, the reduced input set technique cannot provide absolute accuracy but relative accuracy may be achieved. Ideally, since the entire program is simulated, the dynamic execution characteristics obtained from reduced and `reference` input sets should track. Therefore simulation results from a reduced input set should correlate to those obtained from the corresponding `reference` input set. However, as shown by Yi et al., the relative accuracy of the reduced input set technique is poor [14]. Yi et al. also shows that the accuracy of reduced input sets varies widely, depending on the simulated configuration. Therefore, low accuracy as well as configuration dependence make reduced input sets less appropriate for design space exploration.

2.2 Sampling

Sampling performs detailed simulations on selected sections of a benchmark's full dynamic instruction stream, while functionally simulating the instructions and warming the microarchitectural structures before and between these selected sections. Functional simulation and warming is needed to eliminate cold-start effects, to improve the accuracy of data gathered during detailed simulation. Two popular sampling techniques are SimPoint [10] and SMARTS [13]. SimPoint is based on representative sampling, which attempts to extract a subset of the benchmark's dynamic instructions to match its overall behavior. SimPoint uses profiling and statistically-based clustering to select representative simulation points. After simulation, SimPoint weighs the results from each simulation point to calculate the final results. SMARTS is based on periodic sampling, where portions of the dynamic instruction stream are selected at fixed intervals (sampling units) for detailed simulation.

SMARTS optimizes periodic sampling by using statistical sampling theory to estimate error between sampled and `reference` simulations to give recommendations on sampling frequency. This provides a constructive procedure for selecting sampling units at a desired confidence level. To speed up functional simulation between sampling units, SMARTS only performs detailed warming of microarchitectural structures in periods before the sampling units. As shown by SimPoint, SMARTS and Yi et al. [14], these sampling techniques have high absolute and relative accuracy and reduce simulation time. Although sampling is an efficient way of performing detailed simulation, it is still not fast enough to quickly explore large design spaces. For example, sampling techniques still have to simulate all dynamic instructions, at various levels of detail, in order to avoid cold start effects. Also, sampling techniques redundantly simulate branch predictors and caches when multiple processor configurations share the same cache and branch predictor settings.

2.3 Synthetic Trace Simulation

Both statistical simulation [3, 9, 8] and reduced-trace simulation [4] use profiling to create smaller synthetic traces. In statistical simulation, profiling gathers program execution characteristics to create distributions, histograms, or graphs on basic blocks, instruction contexts, dependence distances, instruction-type frequencies, cache miss rates, and branch misprediction rates. Synthetic traces are then created by statistically generating a stream of instruction types and then assigning dependencies based on the execution characteristics. These generated traces are simulated until performance converges to a value. The main drawback of statistical simulation is that the instruction sequences in these synthetic traces are not equivalent to the ones in the original dynamic instruction streams. This discrepancy can cause large errors in performance simulation. In reduced-trace simulation, profiling gathers information about each instruction, including the previous n instructions as context. Instructions are then categorized, and these categories are used with the R-metric and a graph of the program's basic blocks to generate a synthetic trace tailored towards a target system. The configuration dependent nature of these synthetic traces make reduced-trace simulation less appropriate for large design space studies. Also, for some programs such

as `gcc`, reduced-trace simulation was not able to generate a representative synthetic trace.

2.4 Analytical Modeling

Analytical models abstract away detail and focus only on key program and microarchitecture characteristics. These characteristics are then used to compute model parameters to estimate performance. Once the analytical equations are specified, results for a particular configuration can be obtained very quickly since no simulation is involved. Noonburg and Shen [7] use probability distributions, of program and machine characteristics, in simple functions to model parallelism in control flow, data dependencies, and processor performance constraints on branches, fetch, and issue. Karkhanis and Smith [5] base their performance model on ideal IPC, using only data dependencies, and later adjust for performance degradation from cache and branch miss events. Analytical models are fast, generally accurate, and provide valuable insight into processor performance. However, these models also rely on many assumptions about the simulated microarchitecture, making them difficult to modify and adapt to new designs. Therefore they are not suitable for detailed design space exploration.

2.5 AXCIS

Like reduced input sets, sampling, and synthetic trace simulation, AXCIS also decreases simulation time by reducing the dynamically simulated instructions. First AXCIS compresses the dynamic instruction stream, of a particular benchmark, into a CIST. During compression, AXCIS also simulates a branch predictor and caches. Once created, the CIST can be used to simulate a large set of configurations. Therefore, unlike sampling, AXCIS does not need to simulate all dynamic instructions or re-simulate branch predictors and caches for configurations sharing equivalent settings for these structures. Also, because CISTs retain the original instruction sequences, CISTs can be more representative of dynamic traces than synthetic traces. Because AXCIS is highly efficient, we propose AXCIS as a complement to detailed simulation techniques such as sampling.

Chapter 3

AXCIS Framework

In this chapter, we describe the AXCIS framework and define instruction segments. In particular, we describe how AXCIS compresses the dynamic traces into CISTs and how CISTs are used to estimate processor performance.

3.1 Overview

AXCIS is divided into two stages: dynamic trace compression and performance modeling. In the first stage, the *Dynamic Trace Compressor* (DTC) identifies and compresses all dynamic instruction segments into a Canonical Instruction Segment Table (CIST), as shown in Figure 3-1 (a). In the second stage, the *AXCIS Performance Model* (APM) calculates the performance (IPC) of a particular microarchitecture given a CIST and a processor configuration, as shown in Figure 3-1 (b).

An instruction segment is defined for each each instruction in the dynamic trace. The instruction segment, of a particular dynamic instruction, consists of the sequence of instructions, starting from the instruction producing the oldest dependency and ending with the dynamic instruction itself. This last instruction is termed the *defining instruction* of the segment because the instruction segment is defined for this particular instruction. All instructions in the instruction segment are abstracted into their instruction types (i.e. integer ALU, floating point multiply, etc.) because the specifics of each instruction are not needed for performance simulation. Figure 3-2 shows a sample instruction segment, whose defining

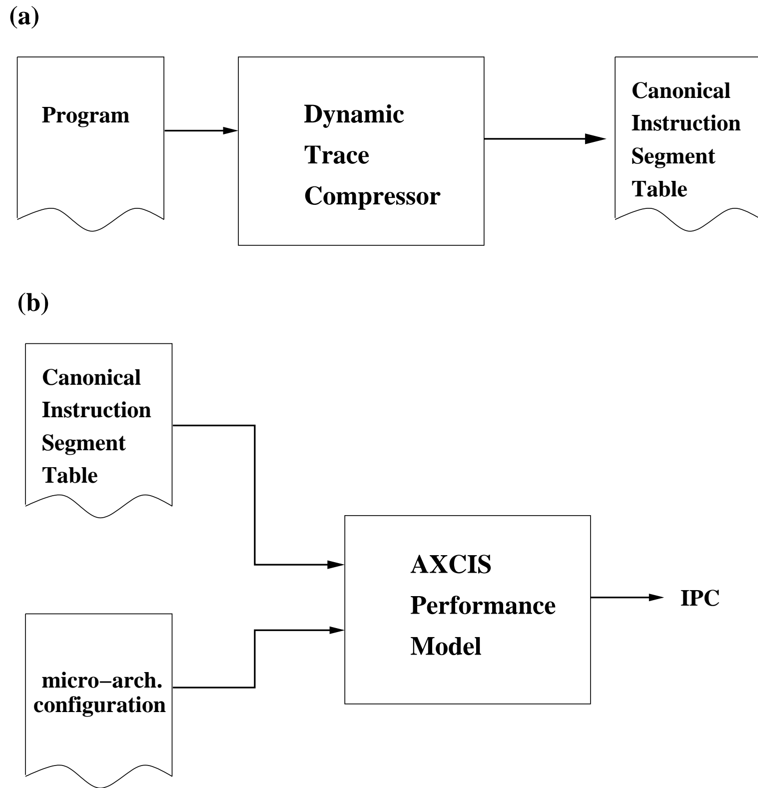

Figure 3-1: Top level block diagram of AXCIS. (a) Dynamic trace compression. (b) Performance modeling.

instruction is the last instruction in the portion of the dynamic trace shown. Overlapping dependencies cause the instruction segments to overlap as well, as shown in Figure 3-3. Note that we use the Alpha instruction set in all examples in this work.

A CIST contains one instance of all instruction segments defined in the dynamic trace. All CIST entries are unique, and each entry in the CIST contains an instruction segment and a frequency count. The frequency count represents the number of segments in the dynamic trace that are canonically identical to the segment in the CIST entry. As the DTC identifies instruction segments, it compares them to existing segments in the CIST. Compression occurs when a newly identified segment is equal to an existing segment in the CIST. In this case, the DTC increments the existing segment's frequency count. If an equivalent segment is not found, the DTC adds the new segment to the CIST. The compression scheme, used by the DTC, defines instruction segment equality. Therefore by varying the compression scheme, AXCIS can adjust the size of the CISTs as well as their representativeness to the

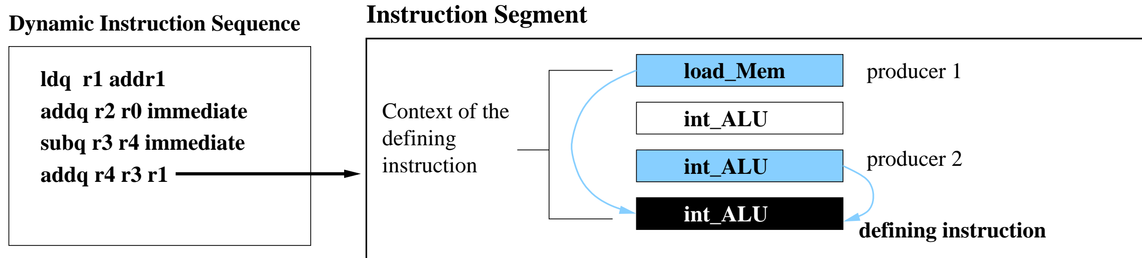

Figure 3-2: Example of an instruction segment. A portion of the dynamic instruction sequence is shown on the left. The instruction segment, shown on the right, is defined for the last instruction in this sequence, termed the defining instruction. The defining instruction has two dependencies, represented by arrows.

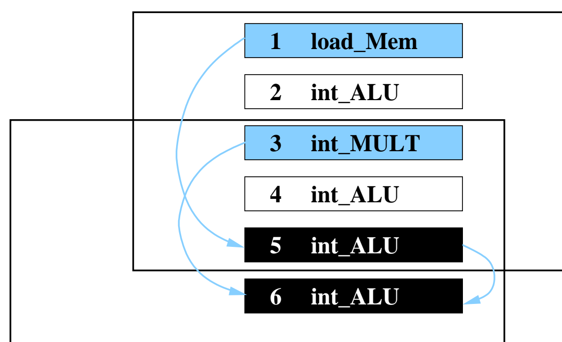

Figure 3-3: Example of two overlapping instruction segments. Instruction entry 5 is both a defining instruction of the first segment, as well as a producer of a value consumed by instruction entry 6.

dynamic trace.

The APM uses the structure of the CIST to perform dynamic programming to quickly estimate instructions per cycle (IPC), for a given configuration. For each instruction segment in the CIST, the APM calculates the stall cycles of the defining instruction of the segment. Once the APM has calculated the stall cycles of all defining instructions in the CIST, the APM estimates IPC using the net stall cycles of the entire CIST. Note that the use of different compression schemes in the DTC does not change how the APM calculates performance.

In this work, we apply AXCIS to model the class of machines shown in Figure 3-4, which include in-order superscalar processors, blocking L1 instruction caches, nonblocking L1 data caches, and bimodal branch predictors. More specifically, these machines include

all configurations that can be described by instantiating the parameters listed in Table 3.1.

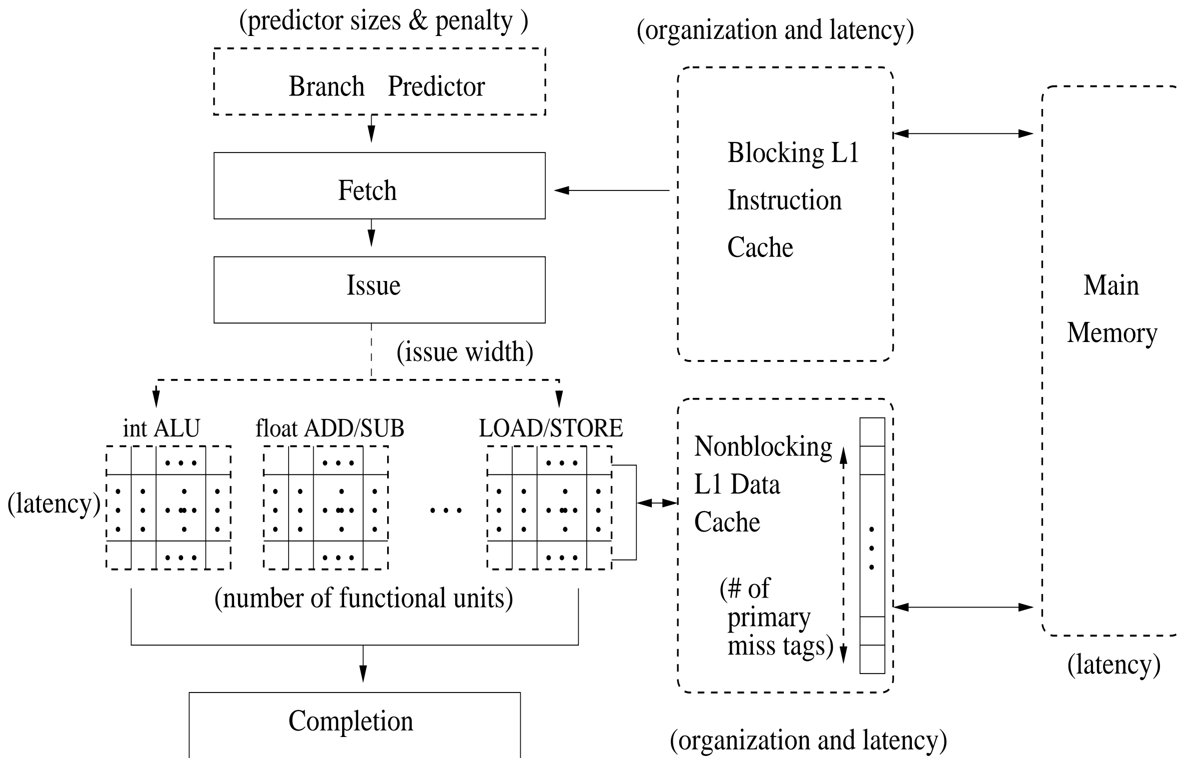

Figure 3-4: Class of machines supported by AXCIS. All parameterizable machine characteristics are drawn with dashed lines and labeled in parentheses.

3.2 Dynamic Trace Compression

Dynamic trace compression is divided into two main tasks. The first task identifies the instruction segment defined for each instruction in the dynamic trace. The second task compresses the instruction segments into a CIST.

3.2.1 Identifying Instruction Segments

All simulation-critical data relating to one dynamic instruction can be compactly represented by an instruction segment. An instruction segment contains both microarchitecture independent and dependent characteristics. Microarchitecture independent characteristics

Parameter
# of functional units for each instruction type (14 types)
Latency of each instruction type
Branch misprediction penalty
Issue bandwidth
# of primary-miss tags supported by data cache
Instruction and data cache access latencies
Memory latency
Instruction and data cache organizations: (# of blocks, block size, associativity, replacement policy)
Bimodal Branch Predictor table size
Return address stack size

Table 3.1: Processor configuration parameters supported by AXCIS.

are inherent within the program and do not depend on machine configuration, while microarchitecture dependent characteristics refer to locality characteristics that depend on cache and branch prediction architectures.

Microarchitecture Independent Data

Each dynamic instruction has associated microarchitecture independent data such as instruction type, context, and set of data and program-order dependencies.

AXCIS categorizes the instructions into 14 types: integer ALU, integer and floating point multiplies, integer and floating point divides, floating point add, floating point compare, floating point to integer converter, floating point square root, load cache access, load memory access, store cache access, store memory access, and nop.

We only consider read-after-write (RAW) data dependencies, since write-after-read (WAR) and write-after-write (WAW) hazards either do not occur or are generally eliminated in in-order architectures. We ignore memory address dependencies between store and load instructions because the effects of these dependencies are already modeled by the memory instruction types. In in-order machines, if a producing store instruction has not completed due to a cache miss, the consuming load instruction will also miss in the data cache. Therefore the effects of this memory dependency will be captured by the cache miss event and represented by the memory access instruction type.

In order for the APM to model structural hazards that limit issue width in multiple-issue machines, the DTC must capture the program order of dynamic instructions. This is done using program-order dependencies which form between consecutive dynamic instructions. Except for the first instruction, every instruction is dependent on its preceding instruction in the dynamic trace.

Each dynamic instruction belongs to a particular program context, which is defined by the dependencies of the instruction. The context of an instruction refers to the instruction sequence, starting from the producer of the instruction's oldest dependency and ending with the instruction itself.

Microarchitecture Dependent Data

Each dynamic instruction has associated microarchitecture dependent data such as instruction cache hit/miss, data cache hit/miss, and branch prediction/misprediction results.

The DTC simulates a branch predictor and instruction and data caches. During dynamic trace compression, only the organizations of these structures need to be specified. Latency assignments are not needed. For caches, the DTC needs to know their sizes, associativities, and line sizes. For branch predictors, the DTC needs to know the type of branch predictor and the sizes of their associated buffers. By simulating these structures, the DTC determines whether each instruction hit or missed in the instruction cache. If the instruction follows a branch, the DTC determines the type of the branch (taken/not taken) and whether or not it was correctly predicted. If the instruction is a load or store, the DTC determines if it hit or missed in the data cache. Hits in the data cache refer to both true hits as well as secondary misses. On true hits, data is found in the cache. On secondary misses, data is not yet in the cache, but a request has already been sent to fetch the line from memory. All hits in the instruction cache are true hits because we model in-order processors that block until the required instruction is fetched from memory.

In order for CISTs to be general enough to support non-blocking data caches with a varying number of outstanding misses, the DTC also records *cache line dependencies* and *primary-miss dependencies*. Cache line dependencies form between consumers of cache accesses (that are not primary misses) and the most recent primary miss of the requested

cache line. Cache line dependencies allow the APM to simulate a wide range of latencies by distinguishing between true cache hits and secondary misses, in a non-blocking data cache. Primary-miss dependencies form between two adjacent primary misses, and are used by the APM to simulate a varying number of outstanding misses, by modeling structural hazards on primary miss tags.

Because the DTC simulates caches as well as a branch predictor, the resulting CISTs can only be reused to simulate configurations sharing the same branch predictor and cache organizations. Although this constraint still allows CISTs to support a large number of configurations, CISTs can be made more general to support an even wider range of machines by having the DTC simultaneously simulate multiple caches and branch predictors to create different segments for the same dynamic instruction. Then all the segments can be compressed into one multi-configuration CIST, where each CIST entry has a separate frequency count for each cache and branch predictor organization.

3.2.2 Instruction Segment Anatomy

For each dynamic instruction, the DTC identifies its corresponding instruction segment. Each instruction segment contains some data and a sequence of instruction entries to represent the context of the defining instruction, which is the last instruction entry in this sequence. The specifics of the data, stored in the segments and instruction entries, depend on the particular compression scheme used by the DTC. The following descriptions of the instruction segment anatomy are based on the limit-based compression scheme presented in Section 3.2.3.

A sample instruction entry is shown in Figure 3-5. Each instruction entry contains the following fields:

Instruction type - One of 14 types.

Sorted set of dependence distances - Captures all dependencies of an instruction entry.
A *dependence distance* refers to the number of dynamically executed instructions in the sequence starting from the producer down to, but not including, the consumer associated with the dependency.

CIST index - CIST index of the instruction segment defined for this instruction entry.

Icache status - Hit or miss in the instruction cache.

Branch prediction status - Predicted or mispredicted during fetch. In blocking instruction caches, instructions that do not immediately follow branches are automatically correctly predicted because the instructions immediately following branches experience all associated stall cycles. If the instruction immediately follows a branch, the type of branch (taken or not taken) is also recorded.

Min/Max stall cycles - Used by the DTC to find canonically equivalent segments. Section 3.2.3 describes this field in detail.

Type	Dep_Dist_Set	CIST Index	Icache_stat	bpred_stat	stalls<min, max>
LD_L1	{ 5, 9, na, na, na }	5	miss	correct taken	< 2, 10 >

Figure 3-5: Instruction entry format.

Instruction segments also contain pairs of minimum and maximum *structural occupancies* pertaining to the defining instruction. Structural occupancies are snapshots of microarchitectural state (e.g. issue group size) at the time an instruction is evaluated for issue. The pairs of min/max structural occupancies of the defining instruction are used with the min/max stall cycles of instruction entries to identify canonically equivalent instruction segments. Figure 3-6 shows the anatomy of an instruction segment and its corresponding dynamic code sequence.

The first five instruction entry fields are required for all compression schemes explored in this thesis. Only the last field (min/max stall cycles) and the min/max structural occupancies are specific to the limit-based compression scheme described in Section 3.2.3.

Sorted set of dependence distances

The following four types of dependencies are recorded in the dependence distance set:

1. Data dependency

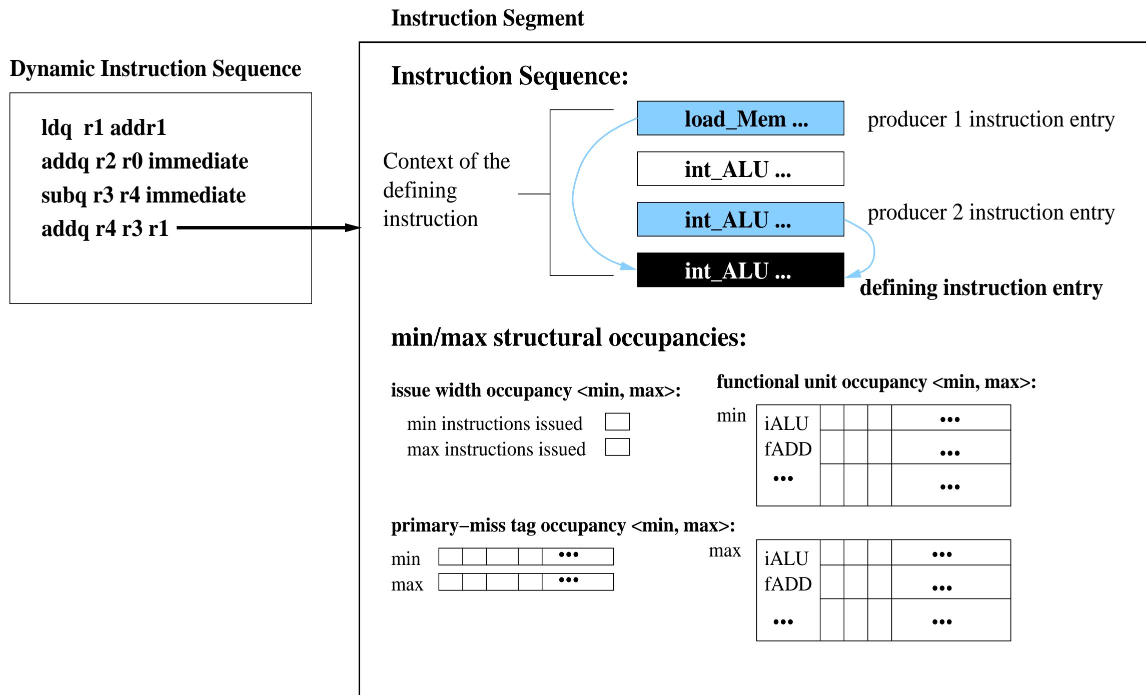

Figure 3-6: Anatomy of an instruction segment.

2. Cache line dependency

3. Primary-miss dependency

4. Program-order dependency

All instructions referenced by the dependence distance set are included in the instruction segment sequence. These instructions as well as any intermediate instructions form the context of the defining instruction of the segment. If any of the first three types of dependencies exist in a dependence distance set, the previous instruction would already be included in the instruction segment. Therefore the dependence distance set does not need to explicitly record program-order dependencies for instructions that have other dependencies. The dependence distance set explicitly records program-order dependencies only for instructions without other types of dependencies.

The number of elements in the dependence distance set varies between 1 and 5. Integer, floating point, and cache-hit instructions have up to 4 entries in their dependence distance sets. Up to 2 entries correspond to the producers of the operands, and up to 2 entries cor-

respond to the producers of cache line dependencies. Cache line dependencies only occur if the immediate producers are loads that hit in the cache. Cache-miss instructions have up to 5 entries in their dependence distance sets. Four of these entries are identical to the ones described above. The last entry corresponds to the producer of the primary-miss dependency, which is the previous primary cache miss. Except for the first dynamic instruction, instructions without any other dependency have at least the program order dependency. Figure 3-7 shows the dependencies recorded for each type of instruction.

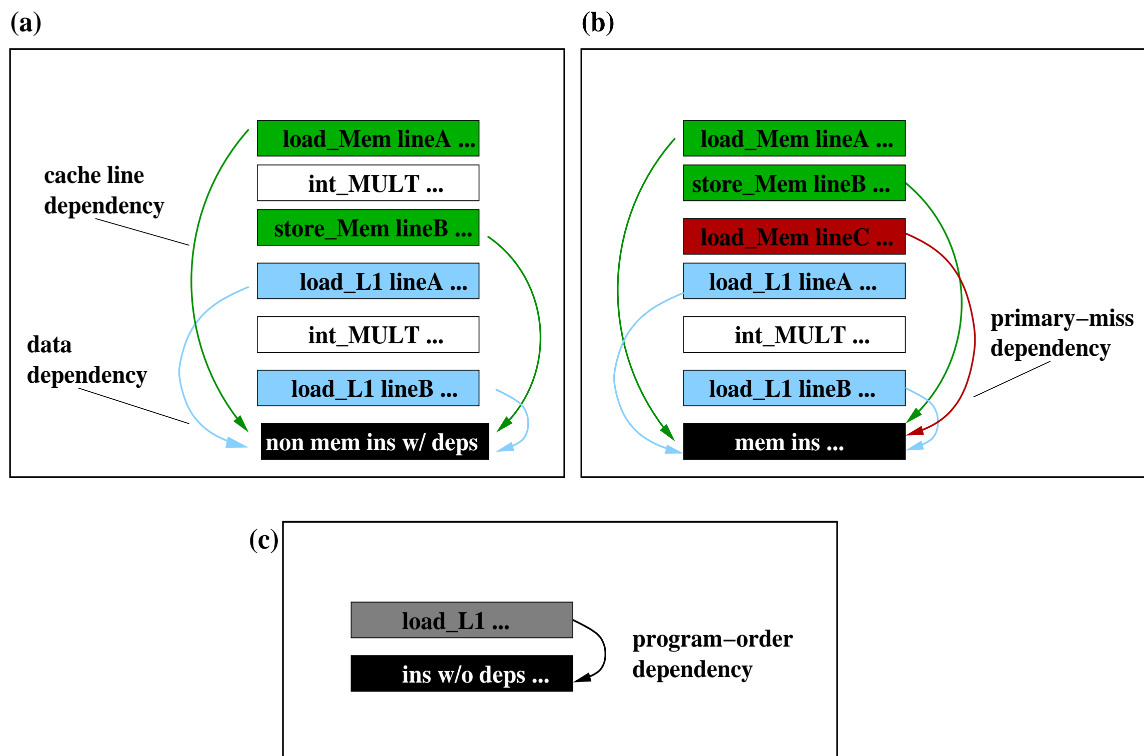

Figure 3-7: Sample dependencies recorded for different instruction types. The dependencies shown correspond to the defining instruction of each segment. (a) Dependencies recorded for non-memory access instructions. (b) Dependencies recorded for memory access instructions. (c) Dependency recorded for instructions with only one dependency.

In order to maintain a reasonable instruction segment length, we limit the maximum instruction segment size by recording only dependence distances less than MAX_DEP_DIST. By varying MAX_DEP_DIST, we can play with the inherent trade-off between accuracy and CIST size. In general, large MAX_DEP_DISTs produce good accuracy but larger CISTs. Small MAX_DEP_DISTs leave out information from the instruction segment, and

therefore produce poorer accuracy but smaller CISTs. We set MAX_DEP_DIST to 512. We further minimize the length of each instruction segment by pruning away non-crucial dependencies that do not cause stalls in any configuration. *Primary consumers* are the first instructions to experience all stalls corresponding to the producer. *Secondary consumers* follow primary consumers in program order, and never experience any stalls from the producer. Therefore we do not need to record the dependency of a secondary consumer.

3.2.3 Creating the Canonical Instruction Segment Table

The DTC profiles the dynamic trace one instruction at a time. For each instruction, the DTC first identifies the corresponding instruction segment, by gathering the simulation-critical data described above. The DTC then determines the uniqueness of the instruction segment by comparing the segment to the entries in the Canonical Instruction Segment Table (CIST). If the instruction segment is canonically equivalent to an entry in the CIST, then the frequency count of the CIST entry is incremented. If the instruction segment does not match any entry in the CIST, then it is added to the CIST. In this manner, the CIST only contains unique instruction segments. The DTC also records the total number of dynamic instructions into the CIST.

Compression Scheme and Definition of Segment Equality

Because the definition of segment equality determines when instruction segments are compressed, it has a large impact on the number of entries in the CIST, which affects the accuracy of AXCIS. A relaxed equality definition results in high compression but poor accuracy, while a strict definition results in high accuracy but poor compression. Therefore the goal is to find a canonical equality definition with the best accuracy and compression trade-off.

An ideal equality definition should compare only instruction segment characteristics that affect performance. Comparisons of other characteristics overly constrain the definition and produce larger CISTs, without improving accuracy. The number of stall cycles experienced by each instruction directly affects performance. Therefore, the DTC should

compress two segments, A and B, if the stall cycles of their defining instruction are equal in all configurations to be simulated using the CIST. We define canonical equality as follows. For all configurations Z, two segments A and B are equal if

$$\begin{aligned} \forall z \in Z,\ Stall_Cycles(A, z) &= Stall_Cycles(B, z)\ and \\ Ins_Type(Defining_Ins(A)) &= Ins_Type(Defining_Ins(B)), \end{aligned} \tag{3.1}$$

where $Stall_Cycles(A, z)$ are the stall cycles experienced by the defining instruction of segment A in configuration z, and $Ins_Type(Defining_Ins(A))$ is the instruction type of the defining instruction of segment A. The instruction type is used by the APM to calculate stall cycles, given a particular configuration.

However, because the DTC does not have full knowledge of the simulated microarchitecture and it is not practical to simulate all possible microarchitectures, exact stall cycles cannot be determined during trace compression. Therefore, the DTC matches instruction segments based on heuristics to approximate canonical equality. We explore several compression schemes in this thesis. One is described in the rest of this chapter, and the others are described in Chapter 5.

Compression Scheme based on Limit Configurations

In order to get some idea of the stall cycles experienced by an instruction, the DTC simulates two microarchitecture configurations. We use these two configurations to approximate the set of all configurations to be simulated using the CIST. The basic intuition is that if two segments have the same stall cycles under two very different configurations, they are more likely to have the same stall cycles under all configurations. We chose these two configurations to be the limiting (minimum and maximum) microarchitecture configurations to be simulated using the CIST.

Using these limiting configurations and instruction segments, the DTC calculates the minimum and maximum stall cycles for each instruction. Sections 3.3 and 3.4 describe the stall calculation procedure in detail. This pair of limiting stall cycles is recorded with each instruction entry, and is the first characteristic that is compared when determining segment

equality.

The minimum and maximum stall cycles provide a range of possible stall cycles. Depending on the configuration, the exact number of stalls experienced by an instruction can be anywhere in this range. Therefore, even if the defining instructions of two segments have identical stall pairs, there is no guarantee that their exact stall cycles are equal. Therefore, we also compare minimum and maximum structural occupancies to more accurately determine canonically equivalent segments.

Structural occupancies play a large role in determining the exact stalls seen by an instruction. Because the DTC simulates two limiting configurations, there are two sets of structural occupancies pertaining to a defining instruction. These occupancies include:

Issue group size: an integer representing the number of instructions in the current issue group.

Functional unit allocation: an array of integers, where each element represents the number of units allocated for a particular functional unit type, in the current issue group.

Primary-miss tag usage: an array of integers. The array size is determined by the number of primary-miss tags in the specified data cache configuration. Each element in the array corresponds to the number of cycles before the miss tag can be re-allocated.

The DTC also compares the types of the defining instructions in the segments. During performance modeling, the APM matches latencies with instruction types, to calculate exact stalls. Therefore instruction type information must not be lost.

To summarize, two instruction segments are equal if:

1. The pairs of limiting stall cycles, corresponding to the defining instruction, are equal.
2. The pairs of issue group sizes are equal.
3. All elements in the pairs of functional-unit allocation arrays are equal.
4. All elements in the pairs of primary-miss tag usage arrays are equal.
5. The instruction types of the defining instruction in the segments are equal.

Efficient Lookup in CISTs

In order to check for canonical equivalence, each new instruction segment needs to be compared to existing CIST entries until either a match is found or all entries have been searched. Since CISTs can grow to tens of thousands of entries, the time required using a linear search algorithm is unacceptable. Therefore we hash the CIST entries into a hash table (CIST Hash Table) to speed up the lookup process. For each new instruction segment, the DTC computes its hash and only compares the segment with entries hashed to the corresponding index of the CIST Hash Table. The DTC calculates the hash of an instruction segment by computing the XOR of all characteristics that determine segment equality.

3.2.4 CIST Data Structure

CISTs compactly record simulation-critical data by exploiting the repetition of instruction segments from loops, function calls, and code re-use. Note that CISTs contain only the information needed for accurate performance simulation. CISTs cannot be used to recreate the original dynamic trace.

A CIST is essentially an ordered array of instruction segments, as shown in Figure 3-8. CISTs have the following properties:

- Each CIST entry contains an instruction segment and its corresponding frequency count. The frequency count indicates the number of times a canonically identical segment has been encountered in the dynamic trace.
- CIST entries are ordered based on their first occurrence in the dynamic program trace.
- Each CIST entry introduces a new instruction to the CIST. This new instruction is the defining instruction of the instruction segment contained in the CIST entry.
- CIST entries may refer to defining instructions of previous CIST entries.
- Because instruction segments overlap, a particular instruction may be referenced by multiple CIST entries.

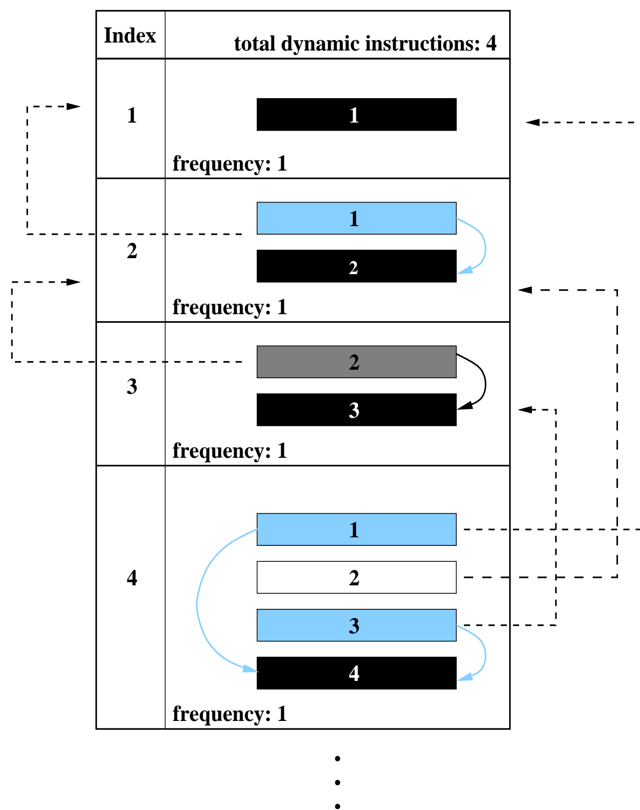

Figure 3-8: Example of a CIST. The instruction entries in the CIST are numbered according to their order of appearance in the dynamic trace. Using these numbers for reference, one can see that CIST entries follow program order, each CIST entry introduces one new instruction, and CIST entries overlap and point to previous entries.

3.2.5 Dynamic Trace Compression: An Example

The left side of Figure 3-9 shows a sequence of dynamic instructions, numbered in program order. The dashed lines represent dependencies between the instructions. The boxes group the instructions according to their instruction segments. For example, instruction 1 does not depend on any previous instructions and therefore is the sole instruction in the segment. Instructions 1 through 4 belong in instruction 4's segment because instruction 1 is the earliest producer of a value consumed by instruction 4. The right side of Figure 3-9 shows the CIST corresponding to this dynamic sequence of instructions. Because none of the segments shown are canonically equivalent to each other, each CIST entry has a frequency count of 1 and no compression occurs.

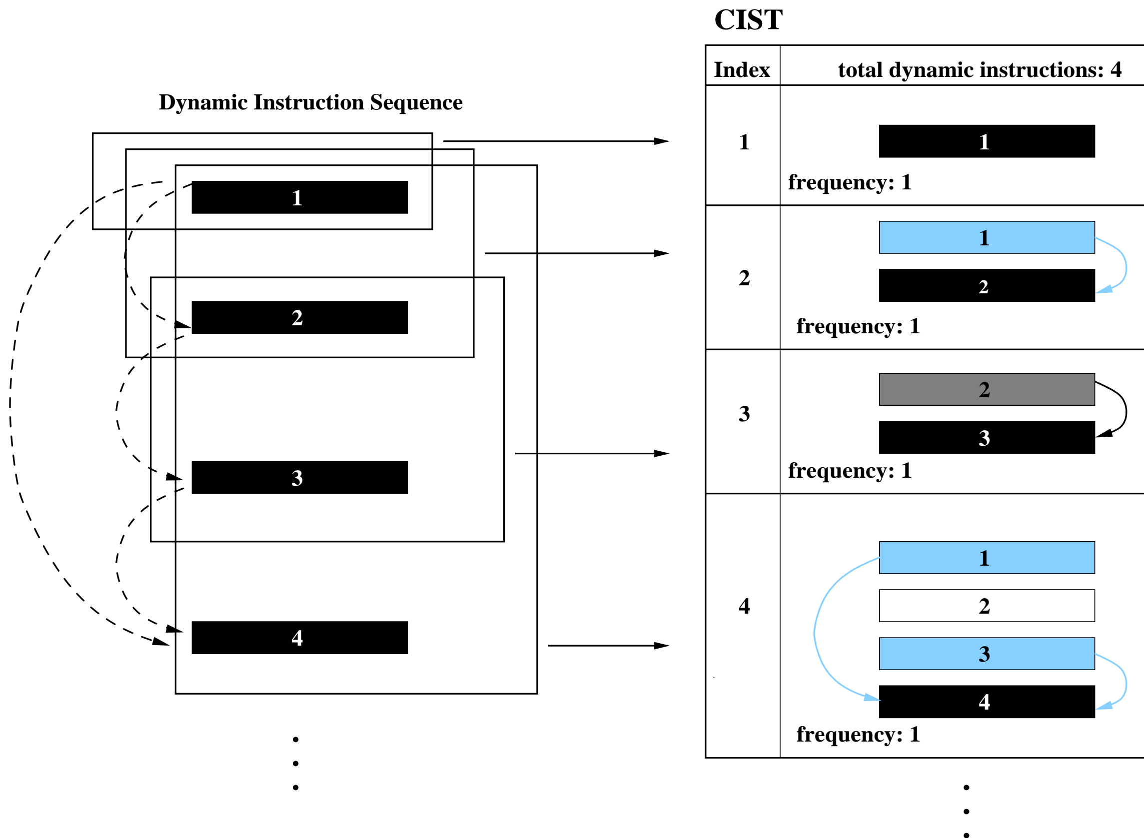

Figure 3-9: CIST building example.

3.3 AXCIS Performance Model

Given a CIST and a processor confi guration, the APM computes performance in terms of instructions per cycle (IPC). IPC is expressed as:

$$IPC = \frac{Total_Instructions}{Total_Instructions + CIST_Net_Stall_Cycles} \quad (3.2)$$

The total number of instructions is recorded in the CIST and refers to the number of instructions profi led by the DTC. The job of the APM, is to calculate the net stall cycles experienced by the entire CIST. Note that net stall cycles may be negative for multiple-issue machines.

As mentioned earlier, stall cycles experienced by different instructions may overlap. Therefore a naive method that sums the stall cycles of individual instructions, without

modeling overlap, overestimates the total number of stall cycles and produces a pessimistic IPC. Stall overlap is confined within the instruction segment primitive. Therefore stalls experienced by an instruction, within some segment, cannot overlap with the stalls of an instruction outside the segment. Based on this principle, the APM accurately calculates the stall cycles of an instruction by taking into account stall cycles of preceding instructions in its segment.

The APM exploits the order-dependent nature of this algorithm by using *dynamic programming* to quickly calculate the net stall cycles for an entire CIST. Because each CIST entry introduces one new instruction, only the stall cycles of this new instruction must be calculated. The stall cycles of the other instructions in the CIST entry can be obtained from the defining instruction entries of previous CIST entries. Also, since CIST entries are created in program order, the APM can calculate the stall cycles of each new instruction sequentially, starting from the first CIST entry. Using dynamic programming, the amount of work required to calculate the net stall cycles of an entire CIST is directly proportional to the number of CIST entries. Because the number of CIST entries can be thousands of times smaller than the total dynamic instructions, the APM can simulate much faster than conventional cycle-accurate simulators.

Stall cycles are caused by the following factors: data, primary-miss, and program-order dependencies as well as control flow events. Each factor associated with an instruction results in some number of stall cycles. If an instruction is affected by more than one factor, its net stall cycles is the maximum of all its stall cycles. The APM calculates the stalls from each type of factor separately, and then takes the maximum to compute the net stalls for an instruction entry. The net stall cycles, of the entire CIST, is the sum of all defining instruction entry stalls weighted by the corresponding frequency counts of their CIST entries. The following sections describe the APM's stall calculation methodology in detail.

3.3.1 Data Dependency Stalls

Data dependency stalls are caused by read-after-write and cache-line dependencies. These dependencies cause stalls when a consumer is ready to issue but its operands have not been

produced. Data dependency stalls depend on the latency of the producer, the dependence distance between the producer and consumer, and the stall cycles of all intermediate instructions between the producer and consumer.

The stall cycles caused by one data dependency is expressed by the following equation.

$$DataDep_Stalls(consumer) = \begin{array}{c} Latency(producer) - Dep_Dist - \\ \sum_{i=producer+1}^{consumer-1} Net_Stall_Cycles(ins_i) \end{array} \quad (3.3)$$

The latency of the producer is provided by the input configuration. The dependence distance is recorded in the instruction entry of the consumer. The net stalls of the other instructions have already been calculated by the APM and can be looked up in their corresponding instruction entries in the CIST.

For each defining instruction entry in the CIST, the APM computes all its corresponding data dependency stalls. Then the APM calculates its net data dependency stalls by taking the maximum of the stalls.

$$Net_Datadep_Stalls(consumer) = MAX(datadep_stalls_1,\ datadep_stalls_2,\ ...) \quad (3.4)$$

3.3.2 Primary-Miss Dependency Stalls

Primary-miss dependency stalls occur in memory access instructions that cannot issue because all primary-miss tags are in use.

In the nonblocking data cache modeled by AXCIS, a primary-miss tag is allocated for each outstanding memory access. These miss tags are de-allocated when the memory access completes. The APM uses primary-miss tag arrays (one type of structural occupancy), shown in Figure 3-10 (a), to maintain the status of these tags. The size of the array corresponds to the number of miss tags in the configuration, and each element represents the number of cycles until the tag becomes available. The APM creates a primary-miss tag array for each CIST entry with a memory access defining instruction. If the defining instruction of the CIST entry has a primary-miss dependency, the values of the array are

copied from the array of the producer. If the defining instruction does not have a primary-miss dependency, all the entries in the array are initialized to -1. This indicates that all miss tags are available this cycle. Memory access instructions that do not have a primary-miss dependency, are either the first memory access or the dependence distance (to the previous primary-miss) is greater than MAX_DEP_DIST.

After initializing the primary miss tag array, the APM updates the array to correspond to the current cycle, instead of the cycle the producer was issued. To do this, the APM computes the number of elapsed cycles since the producer was issued. The number of elapsed cycles is calculated by summing the dependence distance to the producer with the net stalls experienced by all intermediate instructions (between the producer and consumer).

$$Cycles_Elapsed = Dep_Dist + \sum_{i=producer+1}^{consumer-1} Net_Stall_Cycles(ins_i) \quad (3.5)$$

The APM then subtracts the elapsed cycles from each entry of the primary-miss tag array. If no producer exists, the array remains unmodified.

Next, the APM calculates the primary-miss dependency stalls by finding the minimum value in the array. This value is the minimum number of cycles before a primary-miss tag is available.

$$PM_Dep_Stalls = MIN_ENTRY(primary_miss_tag_array) \quad (3.6)$$

3.3.3 Control Flow Event Stalls

Control flow event stalls are caused by instruction cache misses, branch mispredictions, and correctly predicted taken branches.

The icache and branch prediction status flags of an instruction, recorded by the DTC, directly map to the instruction's control flow event stall cycles. Table 3.2 shows the control flow stalls of an instruction based on its icache and branch prediction status flags.

Instructions that hit in the instruction cache will not experience any stalls, unless they follow mispredicted or taken branches. Mispredicted branches break the current issue

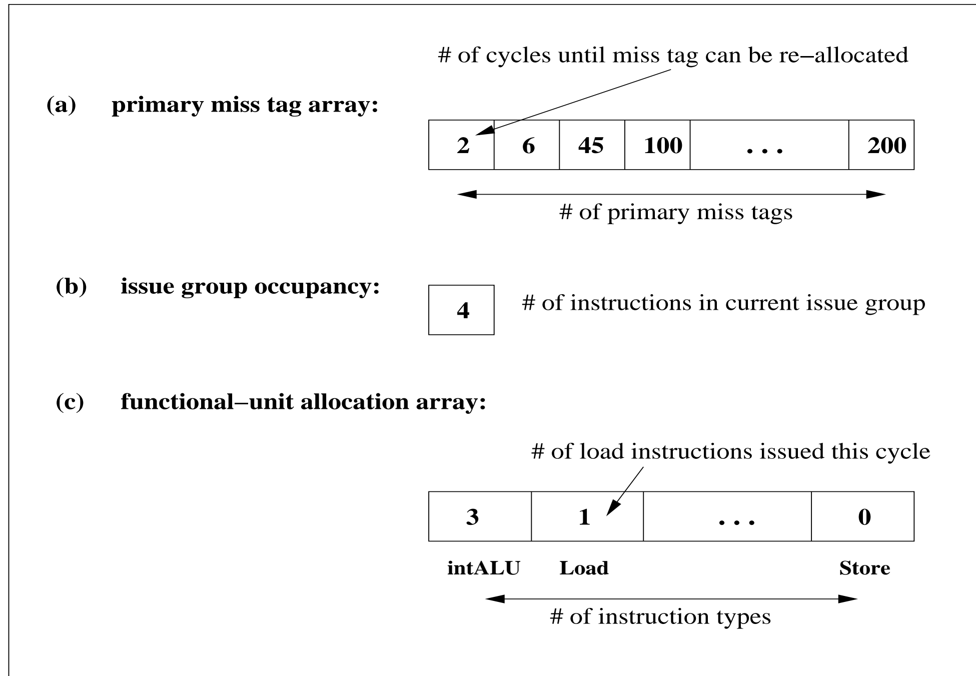

Figure 3-10: Structural occupancies for each CIST entry.

group and cause the corresponding number of stall cycles before another useful instruction can be issued. Correctly-predicted taken branches also break the current issue group, resulting in at least one stall cycle.

3.3.4 Program-Order Dependency Stalls

Program-order dependency stalls are caused by structural hazards on issue bandwidth and functional units.

The APM models issue width limitations using issue group occupancies. An issue group occupancy, shown in Figure 3-10 (b), is an integer representing the number of instructions in the current issue group. The APM creates an issue group occupancy for each CIST entry. The issue group occupancy of the first CIST entry is initialized to zero, because no instructions have been issued this cycle. The defining instructions of all other CIST entries have a program-order dependency on their preceding instruction. Therefore the issue group occupancies of these other CIST entries are copied from the entries of their producers.

Icache Status	Branch Status	CF_Dep_Stall Cycles
hit	mispredicted & taken	misprediction penalty
hit	mispredicted & not taken	misprediction penalty
hit	correctly predicted & taken	0
hit	correctly predicted & not taken	-1
miss	mispredicted & taken	memory latency + misprediction penalty
miss	mispredicted & not taken	memory latency + misprediction penalty
miss	correctly predicted & taken	memory latency
miss	correctly predicted & not taken	memory latency - 1

Table 3.2: Mapping of icache and branch prediction status flags to control flow event stalls.

Structural hazards occur when too many instructions of one type are ready to issue in one cycle. AXCIS assumes that all functional units are fully pipelined. Therefore at the beginning of each cycle, all functional units are available. The APM models functional-unit structural hazards using functional-unit allocation arrays, shown in Figure 3-10 (c). Each element of the array corresponds to the functional units of one instruction type. The elements contain the number of instructions, of that type, that are being issued this cycle. The APM creates a functional-unit allocation array for each CIST entry. Except for the first entry, the arrays of all other entries are copied from the producing CIST entry of the program-order dependency. The array, of the first CIST entry, is initialized to all zeros. AXCIS may be extended to model partially pipelined functional units by applying the technique used to model primary-miss tags.

The APM calculates the program-order dependency stalls, of each CIST entry, by comparing the issue group occupancy and functional-unit array with constraints specified in the input configuration. For example, if the issue group occupancy is less than the maximum issue width, the instruction will not experience any hazards from limited issue bandwidth. Also, if the corresponding functional unit array entry is less than the number of available units of that instruction type, the instruction will not experience any functional-unit structural hazards. When no structural hazards are detected, the program-order dependency stalls are set to -1. This indicates that the instruction issues in the current cycle. When structural hazards are detected, these stall cycles are set to 0, indicating that the instruction

issues in the next cycle.

$$PO_Dep_Stalls = \begin{cases} -1, & \text{no structural hazards} \\ 0, & \text{structural hazards} \end{cases} \tag{3.7}$$

3.3.5 Calculating Net Stall Cycles

After calculating the stalls from each type of dependency, the APM computes the net stall cycles, of a defining instruction entry, by taking the maximum of the stalls.

$$\begin{aligned} Net_Stall_Cycles = MAX \;\; (&Net_DataDep_Stalls, \\ &PM_Dep_Stalls, \\ &CF_Event_Stalls, \\ &PO_Dep_Stalls) \end{aligned} \tag{3.8}$$

Then the APM issues the instruction by updating the occupancies of the corresponding CIST entry. If Net_Stall_Cycles is negative, the instruction issues in the current issue group. In this case, the APM increments the issue group occupancy and the corresponding functional-unit array entry. Otherwise, the instruction issues in a new group. In this second case, the APM sets the issue group occupancy and the corresponding functional-unit array entry to 1. All other entries in the functional-unit array are set to 0. If the instruction accesses memory, the APM makes two updates to the primary miss tag array. In the first update, the APM simulates the net stall cycles experienced by the instruction by subtracting these stalls from each array entry. In the second update, the APM resets the entry containing the minimum cycles with the memory latency subtracted by 1.

3.3.6 Calculating IPC

To calculate instructions per cycle (IPC), the APM needs to first compute the net stall cycles of the entire CIST. In the previous sections we described how to calculate the net stall cycles for each defining instruction entry. To calculate the net stall cycles of the entire CIST, the APM takes the weighted sum of the stall cycles and frequency count of each CIST entry.

The stall cycles of each CIST entry correspond to the stall cycles of the defining instruction of that entry.

$$CIST_Net_Stall_Cycles = \sum_{i=1}^{CIST_Size} Freq(i) * Net_Stall_Cycles(Defining_Ins(i)) \quad (3.9)$$

With this value and the total instructions, the APM computes IPC using Equation 3.2.

3.4 Stall Calculation during Dynamic Trace Compression

The DTC computes the minimum and maximum stall cycles of each instruction using the same methodology as the APM. However, there are two main differences. The first difference is that the DTC computes two stall cycle values for each dynamic instruction, using the two limit configurations. The APM computes only one stall cycle value corresponding to the input configuration. The other difference is that the DTC only computes the stall cycles for the current instruction. The APM computes the stall cycles for all defining instructions in the CIST. Also, the APM computes the net stall cycles of the entire CIST to calculate IPC.

Chapter 4

Evaluation

This chapter evaluates the accuracy and efficiency of AXCIS compared to detailed cycle-accurate simulation. We start by describing our experimental setup, and then we present and analyze the results.

4.1 Experimental Setup

We evaluated AXCIS against our baseline cycle-accurate simulator, SimInOrder. SimInOrder models the same processor characteristics as AXCIS. For example, SimInOrder models an in-order superscalar processor that takes into account RAW data dependencies, structural hazards, bimodal branch prediction, blocking L1 instruction cache, and non-blocking L1 data cache.

AXCIS and SimInOrder are implemented on top of `sim-safe`, an instruction-level execution-driven simulator from the SimpleScalar 3.0 tool set. We also used the `cache` and `bpred` frameworks, from SimpleScalar, to model branch predictors and caches in both AXCIS and SimInOrder.

In our experiments, we used Alpha binaries of SPEC CPU2000 benchmarks [12], obtained from the SimpleScalar website [11]. For each benchmark, we used the corresponding `reference` input sets. Table 4.1 shows the inputs we used for benchmarks with more than one reference input set.

We evaluated AXCIS, using the limit-based compression scheme, on 19 benchmarks

Benchmark	Input
art	reference input with -startx 110 -starty 200 -endx 160 -endy 240
bzip2	source
eon	rushmeier
gcc	166
gzip	graphic
perlbmk	splitmail 704 12 26 16 836
vortex	lendian2

Table 4.1: Inputs used for benchmarks with more than one reference input.

for 15 confi gurations. In total, we ran 285 experiments. Due to the long simulation times of SimInOrder, we limited each run to 10 billion instructions. The two limiting confi gurations, used by the DTC to compress instruction segments, are generated using the parameters specifi ed in Table 4.2. Parameters describing minimum bandwidth and maximum latency are used to generate the minimum confi guration. On the other hand, parameters describing maximum bandwidth and minimum latency are used to generate the maximum confi guration.

Parameter	Minimum	Maximum
issue width	1	10
# primary miss tags	1	20
# units for each instruction type	1	10
branch misprediction penalty	1	9
int alu/nop latency	1	9
int mult latency	4	72
int div latency	8	144
float add/cmp/cvt latency	2	36
float mult latency	3	36
float div latency	8	144
float sqrt latency	10	216
L1 latency	2	27
memory latency	8	450

Table 4.2: Minimum and Maximum processor parameters used by the DTC to generate the limiting confi gurations.

In order to examine the behavior of AXCIS across a wide range of designs, we sim-

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
# units: int mult/div	1	1	1	1	2	2	2	2	4	4	4	4
# units: float add/cmp/cvt	1	1	1	1	4	4	4	4	8	8	8	8
# units: float mult/div/sqrt	1	1	1	1	2	2	2	2	4	4	4	4
# units: load/store	1	1	1	1	4	4	4	4	8	8	8	8

Table 4.3: Twelve simulated configurations that span a large design space.

ulated 12 configurations for each of 7 integer and 12 floating-point benchmarks. These configurations were selected to span a large design space. We would have simulated more configurations, but we were constrained by the long simulation times of our detailed baseline simulator, SimInOrder. Table 4.3 shows the 12 configurations that were simulated. We distributed these configurations evenly throughout the design space, in order to obtain meaningful distributions on the results. In these configurations, we varied issue width, number of primary-miss tags, memory latency, and number of functional units, while fixing the functional unit latencies to the values shown in Table 4.4. We also fixed the cache and branch predictor configurations to those shown in Table 4.5.

Parameter	Latency
int alu/nop latency	1
int mult latency	8
int div latency	16
float add/cmp/cvt latency	4
float mult latency	4
float div latency	16
float sqrt latency	24
L1 latency	3
branch misprediction penalty	3

Table 4.4: Functional unit latency parameters.

Structure	Confi guration
L1 instruction cache	16KB, direct mapped, with 32 byte blocks
L1 data cache	16KB, 4-way associative, with 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 4.5: Cache, memory, and branch predictor confi gurations.

4.2 Results

The results presented in this chapter are produced by AXCIS using the limit-based compression scheme described in Section 3.2.3.

4.2.1 AXCIS Accuracy

Accuracy is measured in terms of absolute error between the IPC estimates obtained from AXCIS and SimInOrder.

$$\%\text{ Absolute IPC error } = 100 * \frac{\mid AXCIS_IPC - SimInOrder_IPC \mid}{SimInOrder_IPC}$$

Figure 4-1 summarizes the IPC errors for each benchmark. The average IPC error, over all confi gurations and benchmarks, is 4.8%. AXCIS performs better on the integer benchmarks, with an average error of 2%, while the average error for the floating point benchmarks is 6.5%. Excluding `facerec`, and `galgel`, the average errors of all benchmarks are within 10%, and 15 of the 19 benchmarks have average errors within 5%. The maximum average errors are observed for `galgel` at 26.8%, followed by `facerec` at 14.3%. All median errors are 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 errors for all benchmarks are within 10%.

Using the limit-based compression scheme, AXCIS is highly accurate for the majority (15) of benchmarks over all confi gurations simulated. Except for four floating point benchmarks (`applu`, `facerec`, `galgel`, and `mgrid`), the range of absolute IPC errors for

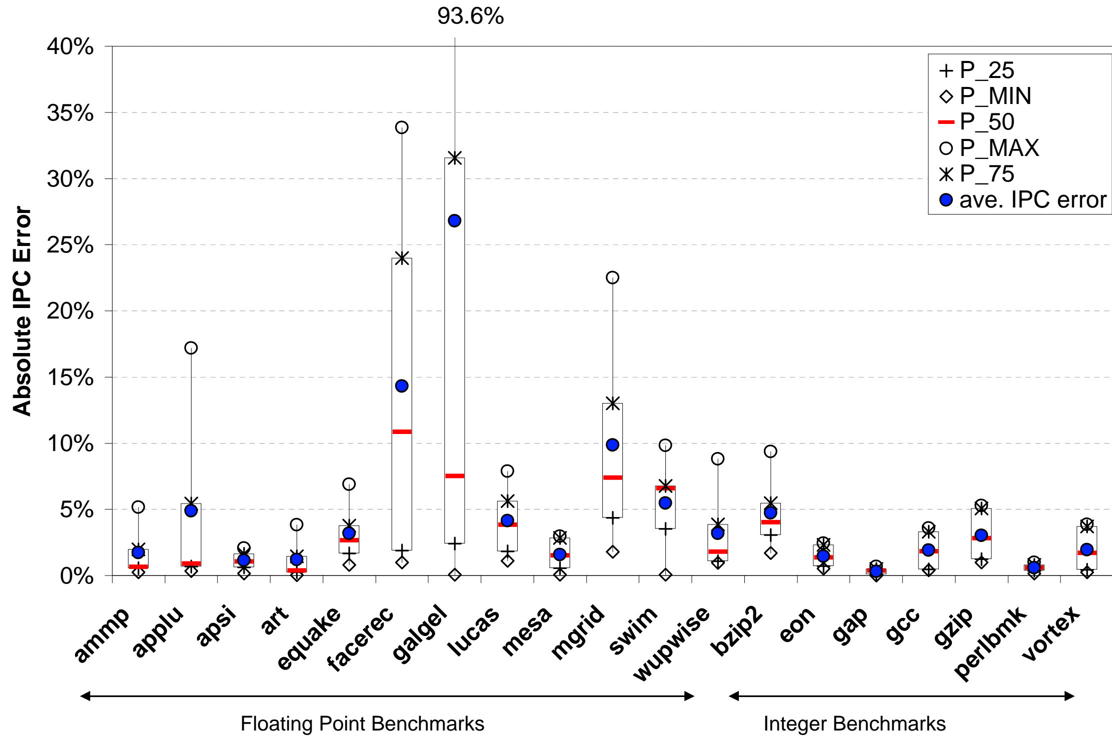

Figure 4-1: Absolute IPC errors for 19 benchmarks and 12 configurations.

each benchmark is less than 10% over all configurations. This small range of errors shows that the accuracy of AXCIS is generally configuration independent. However, for these four floating point benchmarks, the accuracy of AXCIS varies widely depending on the configuration. For example, `galgel` experiences errors ranging from 0.05% to 93.6%. Memory latency and the number of primary-miss tags are two configuration parameters that highly influence the accuracy of AXCIS, for these exceptional benchmarks. As can be seen in Figure 4-2 (a) - (d), configurations with the same memory latency and number of primary-miss tags have similar errors. This suggests that our model, for primary-miss structural hazards, performs poorly for these benchmarks. From these four exceptional cases, we also observed that larger memory latencies result in higher errors than smaller memory latencies, while holding the number of miss tags constant.

Although, Figure 4-2 suggests a correlation between longer memory latencies and higher IPC errors, further analysis using all benchmarks, showed that there is no direct correlation. To investigate further, we ran a second set of experiments to evaluate the accu-

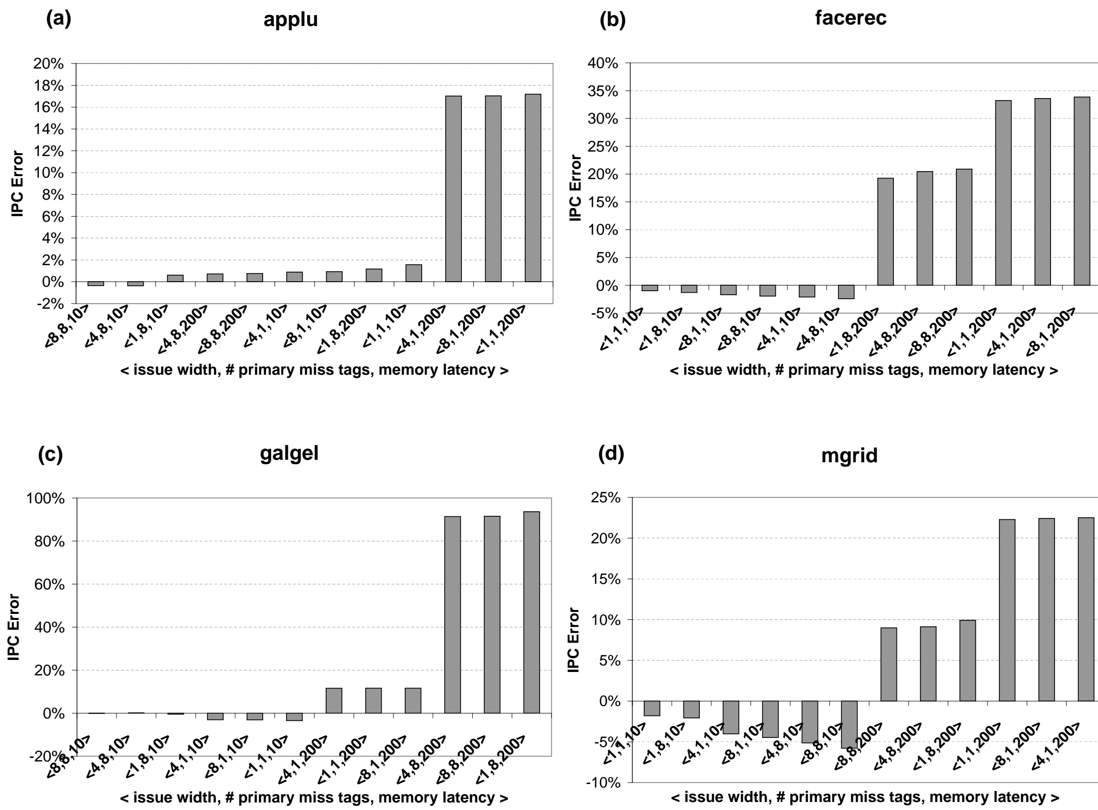

Figure 4-2: IPC errors of (a) `applu`, (b) `facerec`, (c) `galgel`, and (d) `mgrid` for each confi guration.

Parameter	Configurations 1	2	3
memory latency	100	200	10
int alu/nop latency	3	5	8
int mult latency	20	25	50
int div latency	30	35	60
float add/cmp/cvt/mult latency	10	18	25
float div latency	30	35	60
float sqrt latency	50	45	80
L1 latency	6	8	15
memory latency	100	250	350
branch misprediction penalty	7	8	9

Table 4.6: Three simulated configurations with various functional unit and memory latencies.

racy of AXCIS, while varying functional unit and memory latencies. In these experiments, we fixed the issue width to 4, number of primary-miss tags to 1, and functional units to those in Table 4.3 column 5 and Table 4.5. Table 4.6 shows the configurations simulated.

As seen in Figure 4-3, there is no clear trend indicating that higher latencies result in larger IPC error. Although configuration 3 has the highest latencies, the absolute IPC errors for configuration 3 are not the largest over all benchmarks. In these experiments, we again identified `applu`, `facerec`, `galgel`, and `mgrid` as exceptions. Excluding these four benchmarks, all benchmarks had errors within 5%. Also, for all but the four exceptional cases, the range of IPC errors is less than 4.6%, for each benchmark. Again, this narrow error range shows that the accuracy of AXCIS is configuration independent for most workloads. Of the exceptional benchmarks, `facerec` had the highest range, with errors varying from 14.2% to 25.0%.

A dynamic trace can be represented by a long chain of instruction segments. If this entire instruction segment chain were stored in a CIST and simulated by the APM, the IPC error would be zero. However, in order to compress the instruction segments into a concise CIST, AXCIS introduces the sources of error described below.

The accuracy of the APM depends on the representativeness of the CIST to the dynamic trace. The representativeness of the CIST directly depends on two factors:

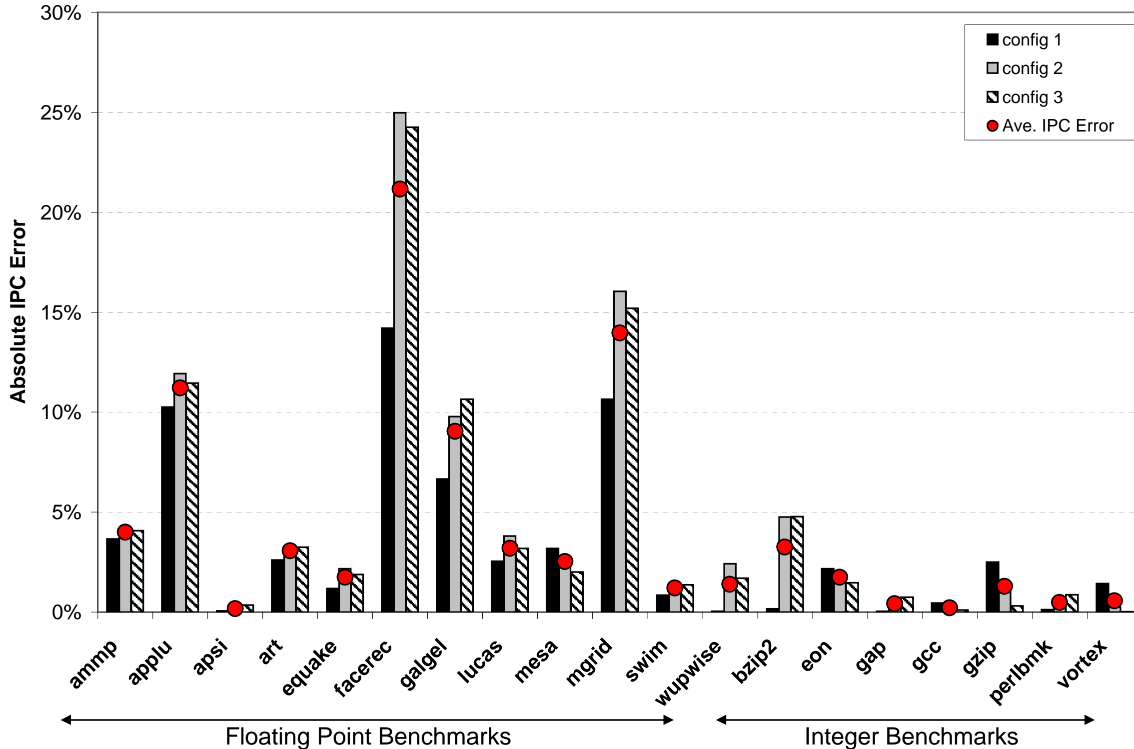

Figure 4-3: Absolute IPC error for 3 confi gurations with various latencies. The specifi cations for each confi guration are shown in Table 4.6.

- The compression scheme - used by the DTC to identify canonically equivalent instruction segments.
- Maximum dependence distance - MAX_DEP_DIST determines the dependencies that are recorded in the CIST.

An ideal compression scheme compresses two segments only if the stall cycles of their defi ning instructions are equal over all confi gurations. The limit-based compression scheme, used in AXCIS, only approximates this ideal. Our results show that this compression scheme does not identify canonically equivalent segments perfectly, causing errors in IPC estimation. As described in section 3.2.3, the DTC uses the stall cycles and structural occupancies, obtained by simulating two limiting confi gurations, to identify canonically equivalent segments. However, equivalent instruction segments under these confi gurations may not be equivalent under other confi gurations. Since the resulting CIST does not distinguish between these two segments, the stall cycles calculated for these other confi gurations

would be inexact.

The DTC only records dependencies up to a distance of MAX_DEP_DIST. Longer dependencies are not recorded in the CIST because they are unlikely to cause any stalls, and recording them would increase the memory required to store the CIST. However, by ignoring these long dependencies, we make it difficult to model primary-miss tag occupancies. If the primary-miss dependency for a memory access instruction is ignored, the APM will freshly initialize the primary-miss tag array instead of making a copy from its producer. By freshly initializing the primary-miss tag array, the APM loses valuable stall information. Therefore the stalls calculated, for that particular segment and any future memory accesses, may be different from the actual stalls.

4.2.2 AXCIS Performance Model Simulation Speed

The speed of the APM is determined by the number of analyzed instructions, which is determined by the number of CIST entries. Figure 4-4 confirms this intuition by showing a linear relationship between the number of CIST entries and APM simulation time. The number of CIST entries varies for each benchmark, and is determined by the compression scheme and inherent benchmark characteristics.

Figure 4-5 shows the number of CIST entries and the average APM execution time for each benchmark, under the limit-based compression scheme. The number of CIST entries, for a benchmark, is the same over all configurations. The APM execution time varies because the amount of work done at each CIST entry varies slightly, depending on the configuration. On average, the APM simulates around 260,000 instruction entries for each benchmark, which takes about 0.72 seconds. Without CISTs, a conventional simulator would have to simulate all 10 billion instructions of the dynamic trace. The minimum and maximum number of instructions simulated by the APM are 5,721 and 1.29 million instructions, for `wupwise` and `perlbmk` respectively. The corresponding minimum and maximum simulation times are 0.02 seconds and 3.1 seconds.

The detailed simulations, performed by SimInOrder, took around 5 hours for each benchmark. Using pre-generated CISTs, AXCIS is over 10,000 times faster than detailed

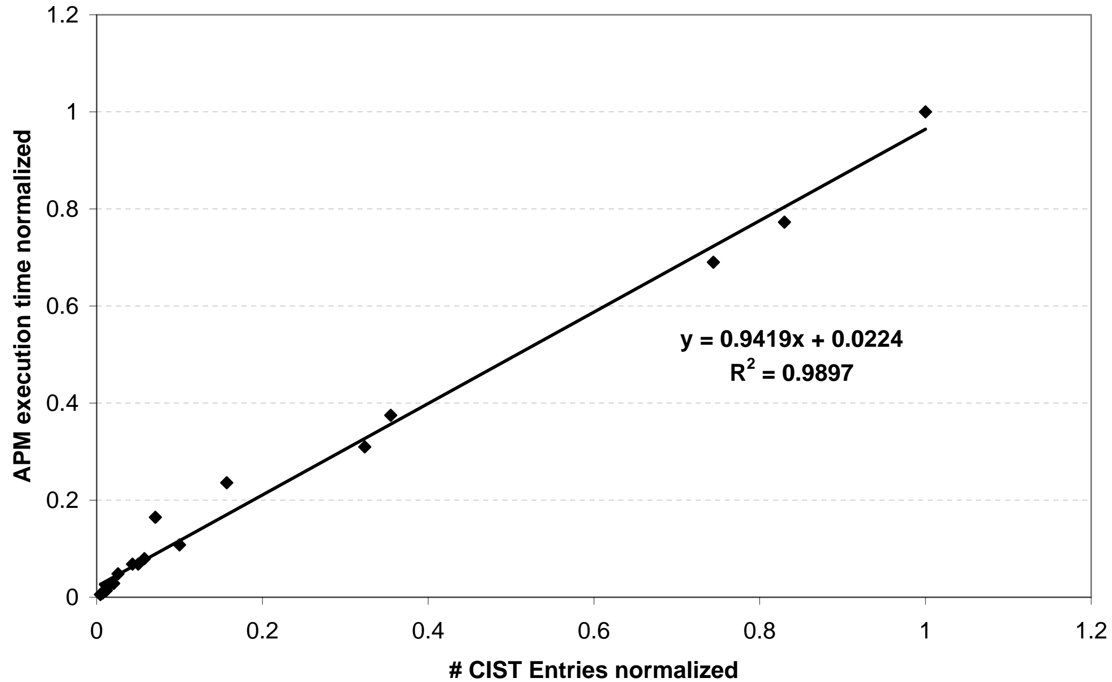

Figure 4-4: Normalized APM execution time vs. normalized number of CIST entries.

simulation. Although CIST generation by the DTC under our unoptimized implementation is about four times slower than detailed simulation, CIST generation was only performed once per benchmark. Therefore, AXCIS is much faster than detailed simulation for large design space studies.

4.2.3 CIST Size

Each CIST entry represents an instruction segment, which may refer to one or more instructions. The CIST size is proportional to the total number of instruction entry references. Figure 4-6 shows the number of instruction entry references in each CIST. The average number of instruction entry references is 15.4 million, and the minimum and maximum references are 220,000 and 101 million for `wupwise` and `perlbmk` respectively. Although CIST size does not directly affect APM simulation time, it does determine the amount of memory required to store the CIST.

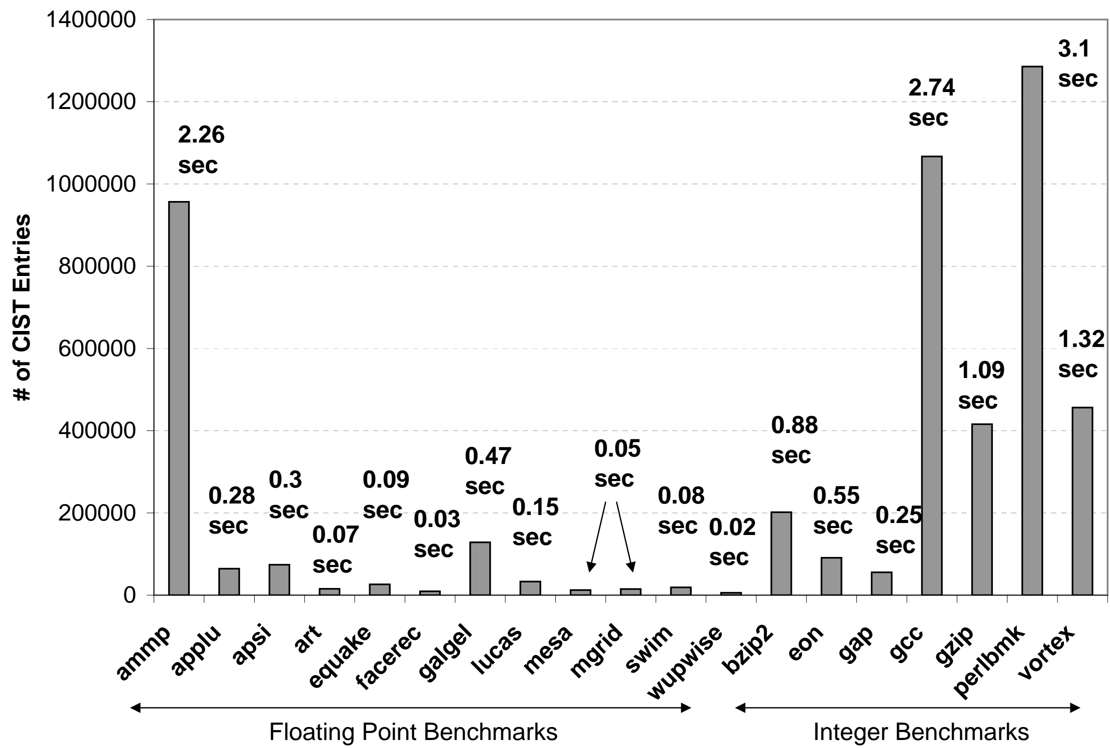

Figure 4-5: Number of CIST entries and average APM execution times.

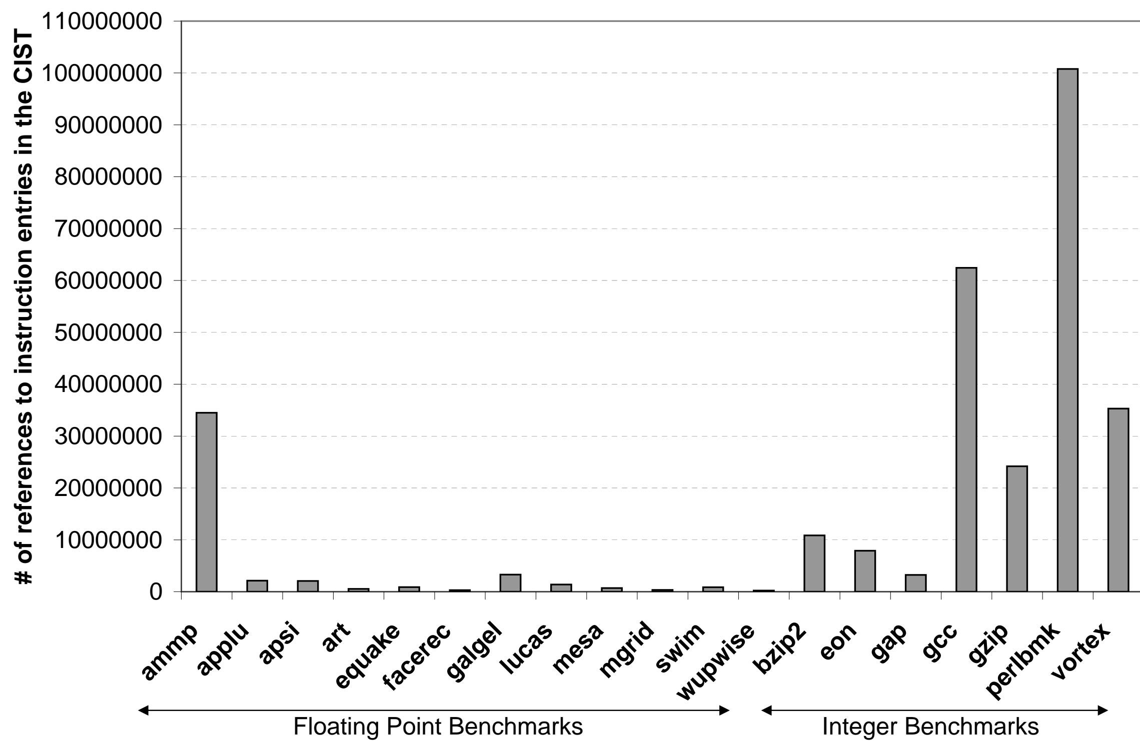

Figure 4-6: Number of instruction entry references in each CIST.

Chapter 5

Alternative Compression Schemes

In AXCIS, the inherent trade-off between speed, space, and accuracy can be expressed as a function of the compression scheme used by the DTC. An ideal *strict* compression scheme compresses two segments only if they have the same stall cycles for all simulated configurations. On the other hand, a more *relaxed* compression scheme compresses two segments if there is some probability that they have the same stall cycles for most configurations. Stricter compression schemes result in higher accuracy but longer simulation times and larger CISTs. More relaxed compression schemes result in lower accuracy but shorter simulation times and smaller CISTs. Because ideal strict compression schemes are very heavy-weight and require too much configuration dependent information in the DTC, our original compression scheme, described in Section 3.2.3, only approximates a strict scheme by using the stall cycles and structural occupancies of two limiting configurations. As shown in Section 4.2, this limit-based scheme worked well for all but four floating point benchmarks: `applu`, `facerec`, `galgel`, and `mgrid`. This chapter explores two alternative compression schemes. The first scheme approximates a strict compression scheme, by using instruction segment characteristics to identify equivalent segments. We show that this scheme significantly improves the accuracy of AXCIS for the four exceptional benchmarks. The second scheme is a relaxed version of the original limit-based scheme. We show that this relaxed scheme creates smaller CISTs for the integer benchmarks (`crafty`, `mcf`, `parser`, `twolf`, `vpr`), while maintaining high accuracy over a large range of configurations.

5.1 Compression Scheme based on Instruction Segment Characteristics

As shown in Section 3.2.3, one way to approximate a strict compression scheme is to have the DTC simulate two limit configurations and compare the stall cycles and structural occupancies of the instruction segments. Another way to approximate a strict compression scheme, is to compare instruction segment characteristics. For example, under this scheme, two segments are equal if they have the same segment characteristics described below.

1. Segment length.
2. Instruction types of all instruction entries.
3. Instruction cache and branch prediction status flags of all but the first instruction entry in the segment.
4. Dependence distance sets of all but the first instruction entry in the segment.

The dependence distance set of the first instruction entry does not need to be compared because stalls experienced by the first instruction do not affect the stalls of the defining instruction of the segment. Although the same logic applies for the instruction cache and branch prediction status flags of the first instruction entry, we realized this after performing the following simulations. Therefore in our simulations, we also compared the instruction cache and branch prediction flags of the first instruction segment. If we had not compared these two characteristics, we would have obtained smaller CISTs with little change in accuracy.

We evaluated this characteristics-based compression scheme with the four floating point benchmarks (`applu`, `facerec`, `galgel`, `mgrid`) that performed poorly under the limit-based scheme. For each of these benchmarks, we simulated six configurations that span a large design space. These six configurations are described in Table 4.4, Table 4.5, and the even columns of Table 4.3. Due to time constraints, we limited each simulation to 3 billion dynamic instructions.

Figure 5-1 summarizes the absolute IPC errors obtained using this characteristics-based compression scheme. The absolute IPC errors, obtained using the limit-based scheme, are also shown for comparison. Both sets of IPC errors correspond to 3 billion dynamic instructions. As a stricter compression scheme, the characteristics-based compression scheme dramatically improved the accuracy of AXCIS for these four benchmarks. Under this compression scheme, the average IPC error was reduced from 13.97% to 3.33%. The maximum errors of `applu`, `facerec`, `galgel`, and `mgrid` decreased by 96.6%, 98.1%, 77.5%, and 97.3% respectively. The range of these errors is also small, making AXCIS both highly accurate and confi guration independent.

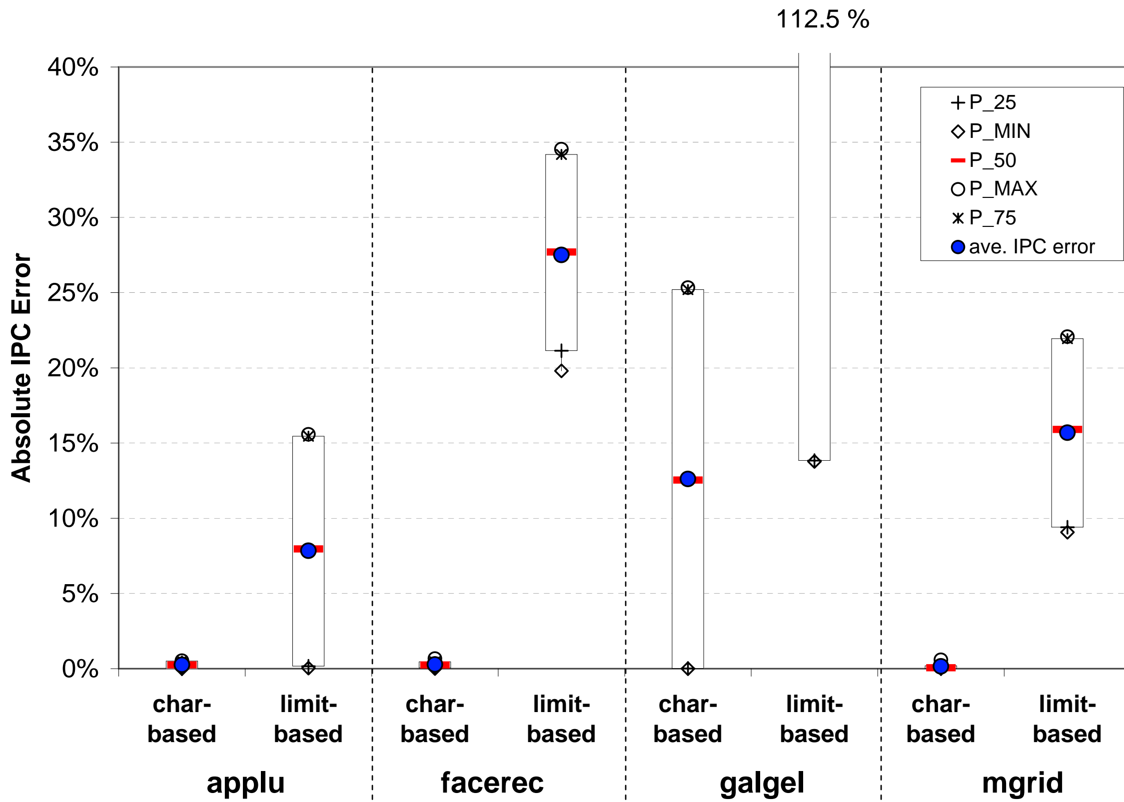

Figure 5-1: Comparison of the IPC errors obtained under the characteristics-based and limit-based compression schemes.

Figure 5-2 shows the number of CIST entries and average APM execution times for each benchmark. The number of CIST entries obtained using the limit-based scheme are also shown for comparison. As expected, the number of CIST entries increased because

the equality definition in the characteristics-based scheme is stricter than that of the limit-based scheme. Therefore less compression occurs under this new scheme, resulting in better accuracy but larger CISTs. As seen in Figure 5-3, the number of instruction entry references in the CISTs are also higher than before. However, despite the increase in CIST size, the APM remains very fast. All these simulations, corresponding to 3 billion dynamic instructions, completed within one second.

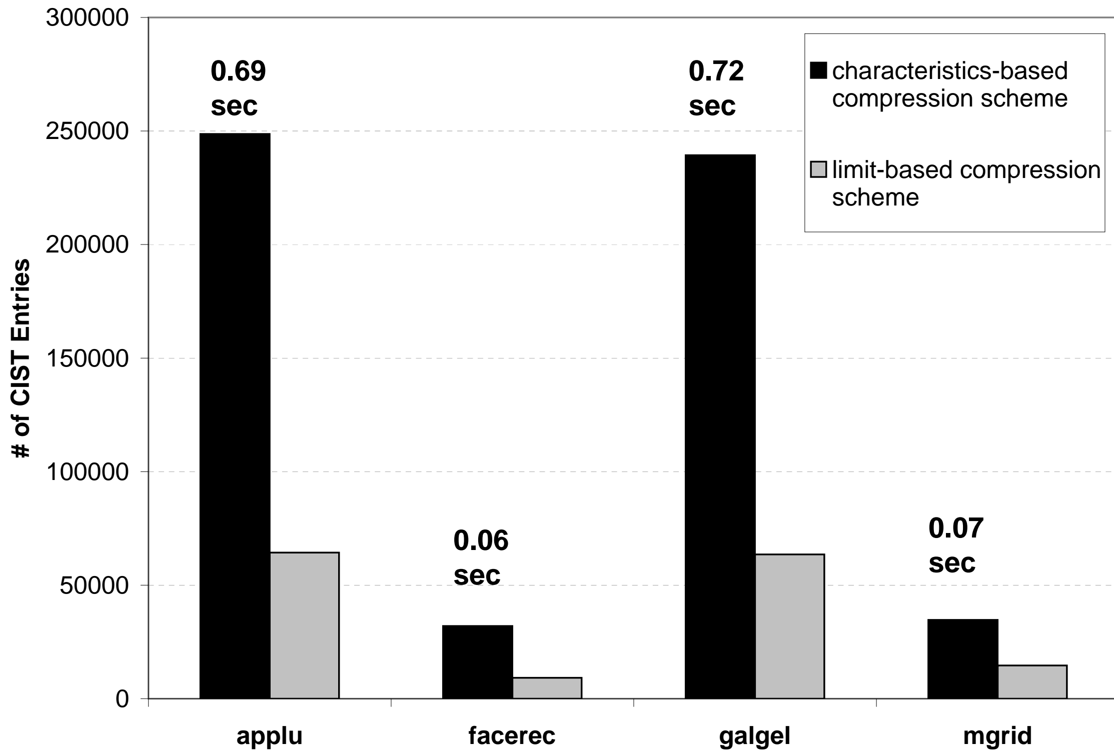

Figure 5-2: Comparison of the number of CIST entries obtained under the characteristics-based and limit-based compression schemes.

5.2 Relaxed Compression Scheme

The CISTs generated for five SPEC INT benchmarks (`crafty`, `mcf`, `parser`, `twolf`, and `vpr`) using the original limit-based compression scheme were extremely large. To represent 1.4 billion dynamic instructions, these CISTs contained an average of 947,902 entries and 44,434,598 instruction entry references. Therefore, we propose a relaxed ver-

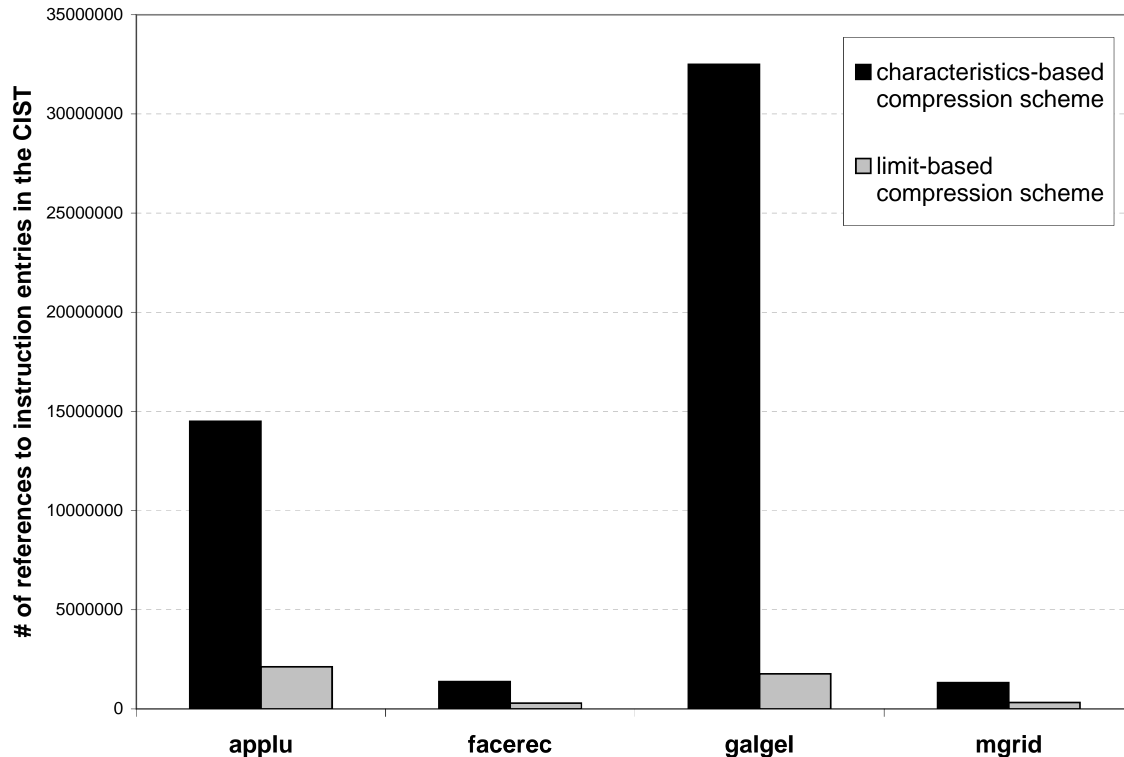

Figure 5-3: Comparison of the number of instruction entry references within a CIST, obtained under the characteristics-based and limit-based compression schemes.

sion of the limit-based scheme to improve compression. This relaxed scheme is identical to the limit-based scheme except only the minimum and maximum stalls are compared for equality. None of the structural occupancies are compared.

Using this relaxed compression scheme, we were able to significantly decrease the sizes of the CISTS for `crafty`, `mcf`, `parser`, `twolf`, and `vpr`. Figure 5-4 and Figure 5-5 compare the relaxed and limit-based compression schemes in terms of CIST entries and instruction entry references. The results for these integer benchmarks correspond to 1.4 billion dynamic instructions, except for `twolf` which corresponds to 3.5 billion dynamic instructions. Using the relaxed compression scheme, the average number of CIST entries decreased by 23.9% and the average number of instruction entry references decreased by 31.4%.

In order to evaluate the accuracy of AXCIS using this relaxed compression scheme, we simulated these five benchmarks for six configurations, described in Table 4.4, Table 4.5,

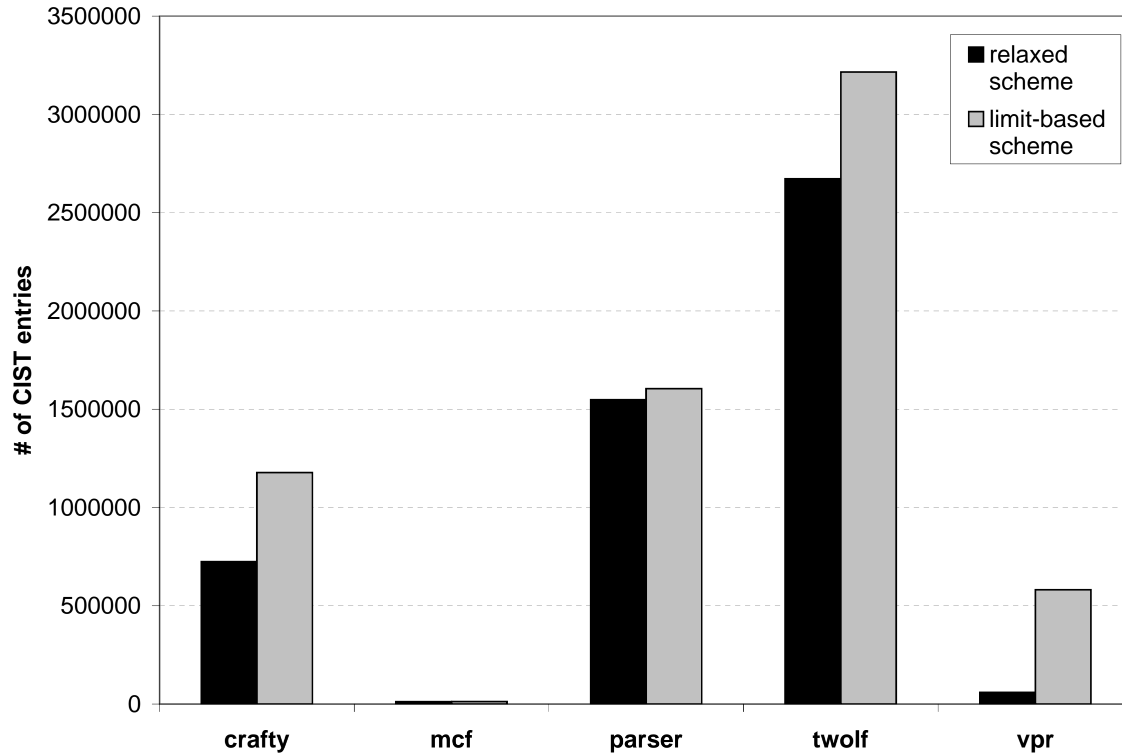

Figure 5-4: Number of CIST entries obtained under the relaxed and limit-based compression schemes.

and the even columns of Table 4.3. We ran each of these simulations for 4 billion dynamic instructions.

Figure 5-6 summarizes the absolute IPC errors obtained using this relaxed compression scheme. The average error is only 2.6%, and the maximum error is observed for `twolf` at 7.5%. The maximum range of errors is also observed for `twolf` and is only 7.4% across all configurations. AXCIS remains highly accurate and configuration independent while using this relaxed compression scheme because these integer benchmarks have a large variety of instruction segments. Even under a very broad definition of segment equality, enough unique segments are identified and recorded into the CISTs to maintain high accuracy. This relaxed compression scheme should not be used for benchmarks with a lot of repetition and few instruction segment varieties (i.e. floating point benchmarks) because the generated CISTs will be very small and accuracy will be poor.

Figure 5-7 shows the number of CIST entries and average APM execution times, under

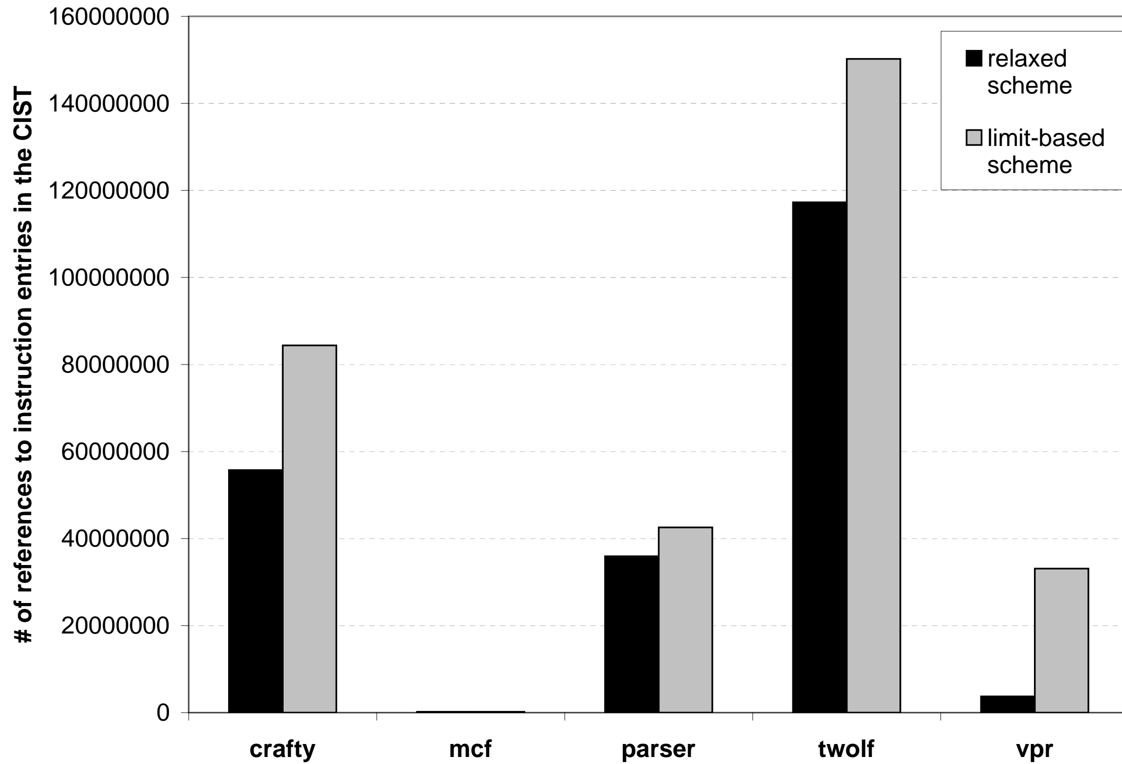

Figure 5-5: Number of instruction entry references in each CIST obtained under the relaxed and limit-based compression schemes.

the relaxed compression scheme. Figure 5-8 shows the number of instruction entry references in each CIST. Although these CIST sizes are quite large, for representing 4 billion dynamic instructions, AXCIS is still much faster than detailed simulation.

5.3 Optimal Compression Scheme for each Benchmark

Because the compression scheme does not affect the methodology of the APM, the DTC does not need to use one compression scheme to create the CISTs of all benchmarks. Therefore, the DTC can optimize the accuracy, speed, and/or space of AXCIS by using the most suitable compression scheme for each benchmark.

Figure 5-9 summarizes the IPC error of AXCIS using the optimal compression scheme for each benchmark, selected from the three schemes explored in this thesis. Figure 5-10 shows the number of CIST entries and average APM execution times. Figure 5-11 shows

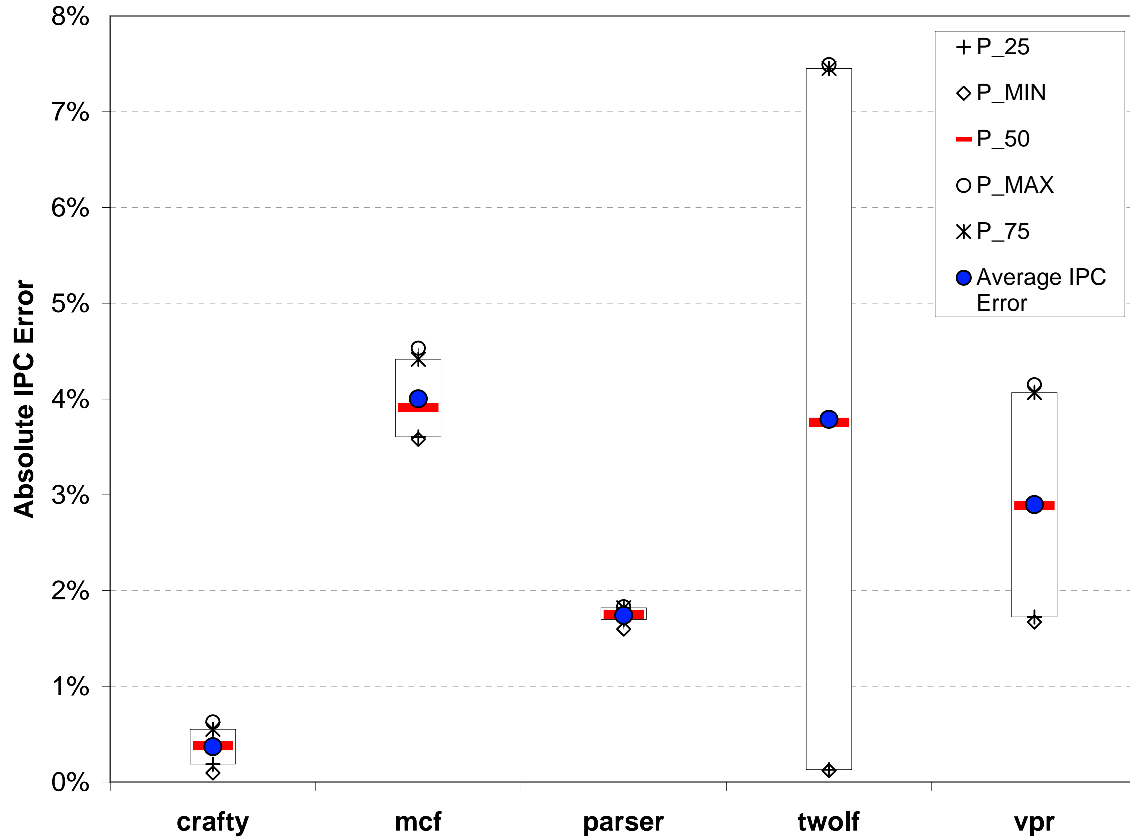

Figure 5-6: Absolute IPC error obtained under the relaxed compression scheme.

the number of instruction entry references in each CIST, under the corresponding optimal compression schemes. Benchmarks using the limit-based compression scheme were run for 10 billion instructions. Because of time constraints, benchmarks using the relaxed and characteristics-based compression schemes were run for 4 billion and 3 billion instructions, respectively.

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`, the maximum error of all benchmarks is less than 10%. The maximum error of `galgel` is 25.3%. AXCIS is also very fast, completing simulations corresponding to billions of dynamic instructions within seconds. High accuracy, configuration independence, and short simulation times make AXCIS an effective tool for large design space studies.

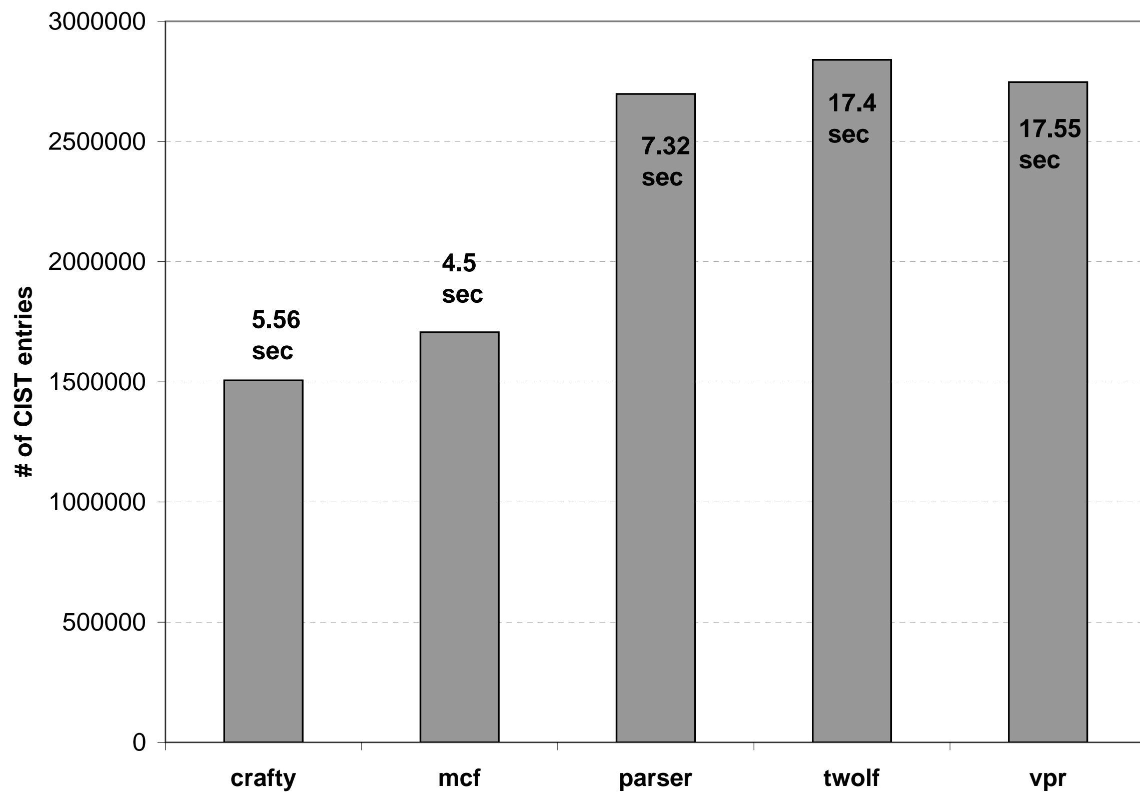

Figure 5-7: Number of CIST entries and average APM execution times obtained under the relaxed compression scheme.

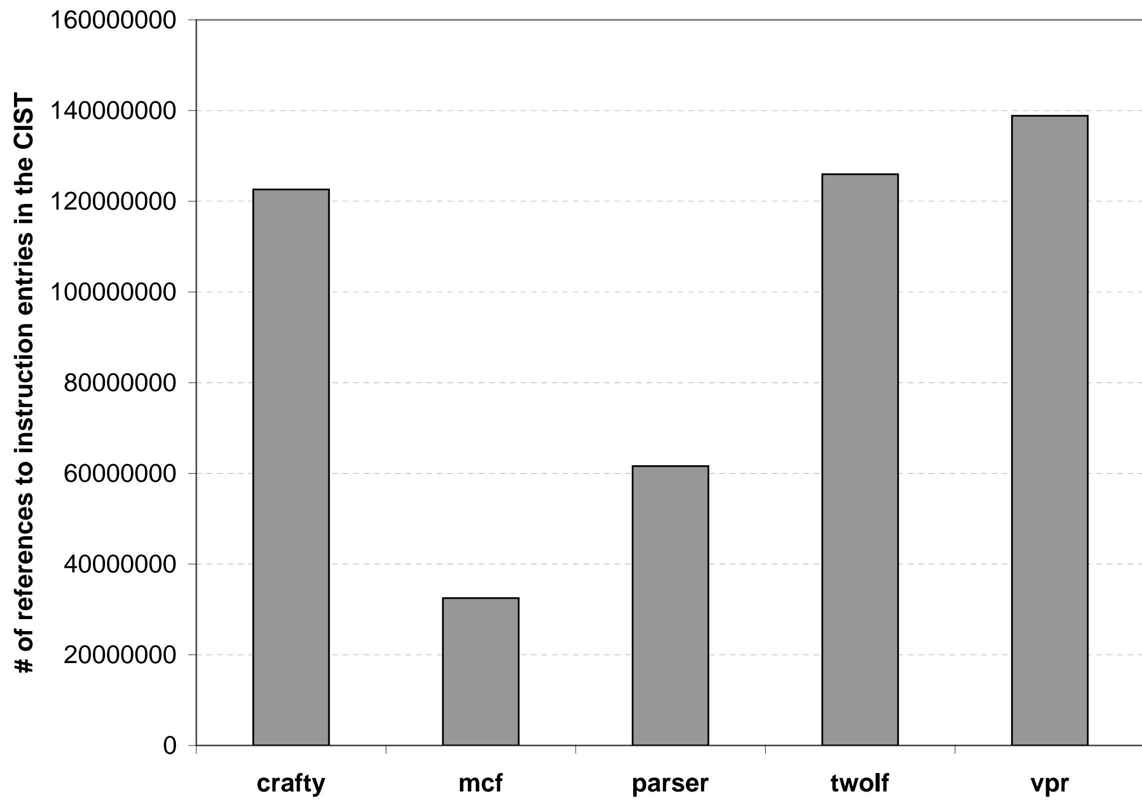

Figure 5-8: Number of instruction entry references in each CIST obtained under the relaxed compression scheme.

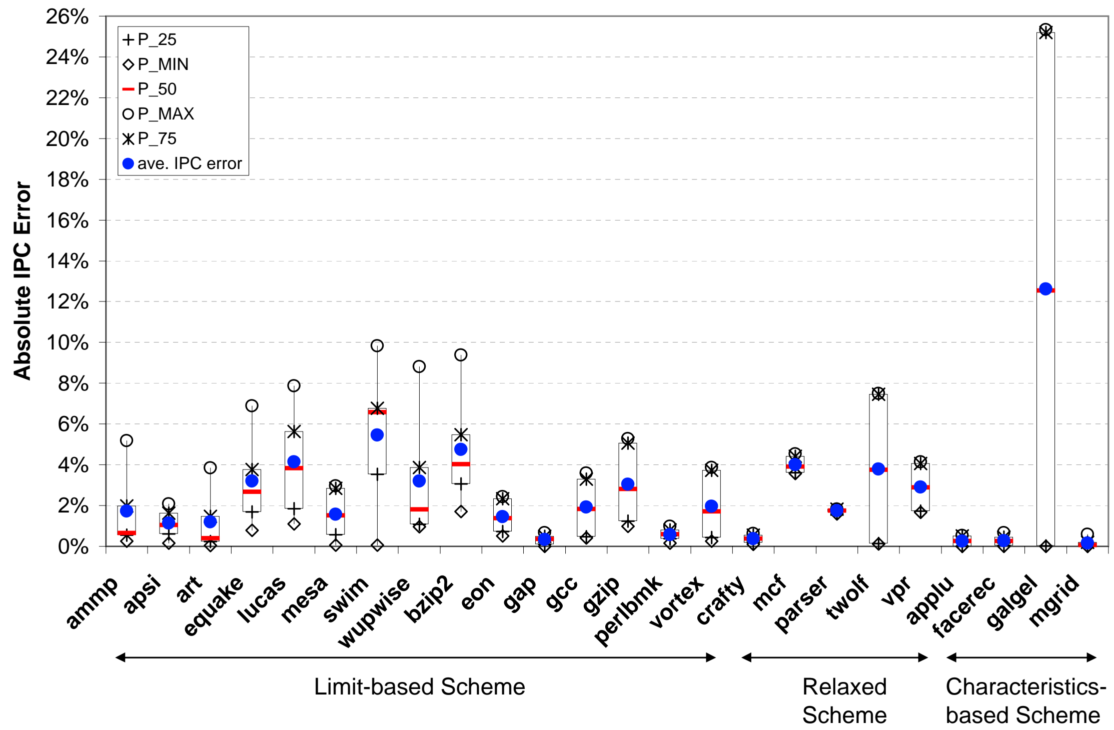

Figure 5-9: Absolute IPC error for each benchmark obtained under its optimal compression scheme.

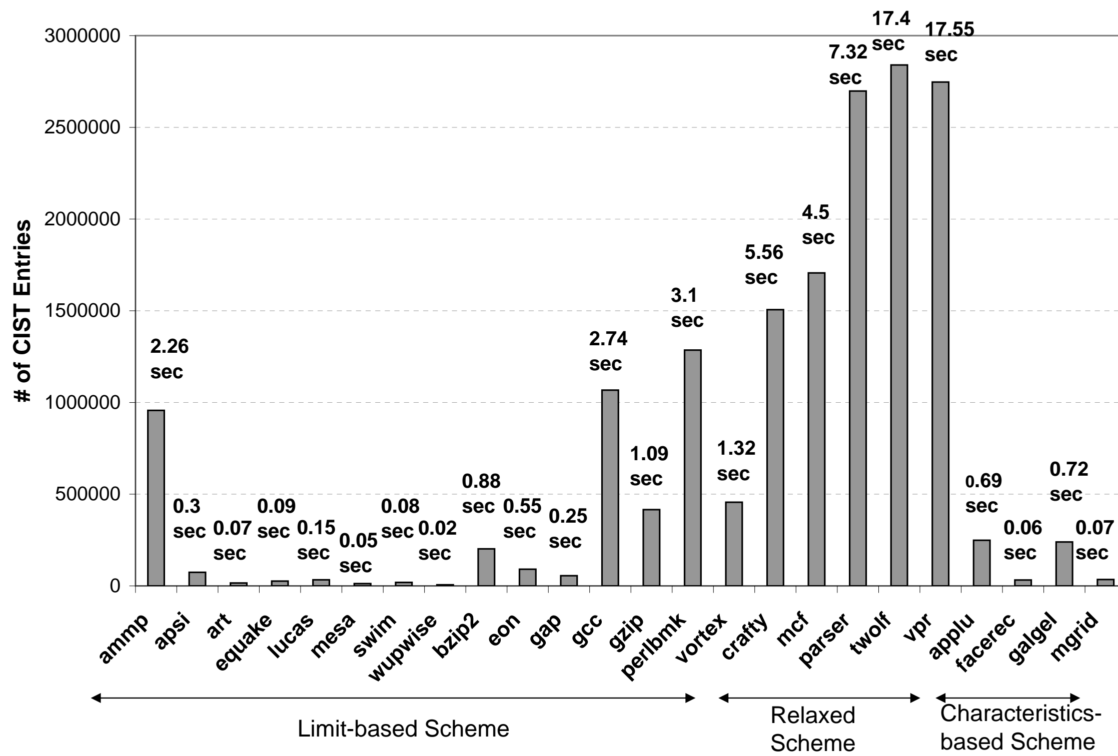

Figure 5-10: Number of CIST entries and average APM execution times obtained under the corresponding optimal compression scheme.

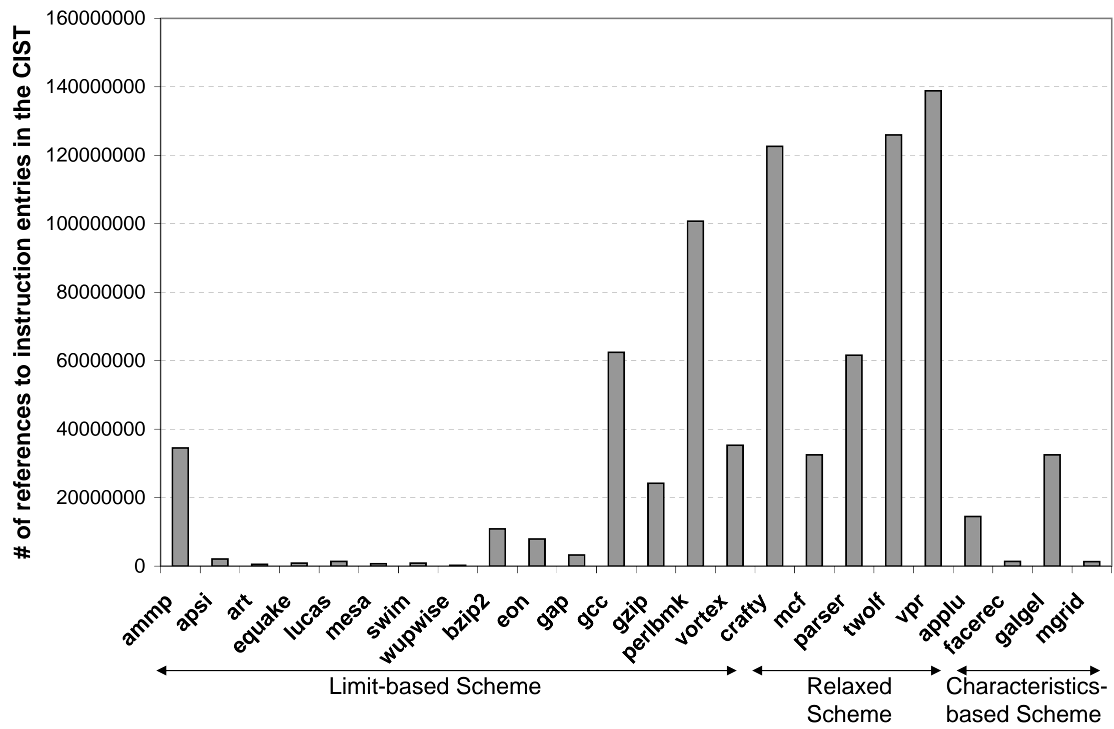

Figure 5-11: Number of instruction entry references in each CIST obtained under the corresponding optimal compression scheme.

Chapter 6

Conclusion

This chapter highlights the contributions of this thesis, describes some future work, and examines additional applications for instruction segments.

6.1 Summary of Contributions

This thesis presented AXCIS, a viable framework for accelerating architectural simulation in large design space studies. Based on *instruction segments*, a novel primitive for representing microarchitectural independent and dependent workload characteristics, AXCIS compresses the dynamic instruction stream of a program into a Canonical Instruction Segment Table (CIST). CISTs are small and highly representative of the original dynamic trace. Therefore they can be used to quickly and accurately simulate a large number of designs.

The inherent trade-offs between accuracy, simulation time, and space can be expressed as a function of the instruction segment compression scheme used to create CISTs. We defined two classes of compression schemes (*strict* and *relaxed*). Stricter compression schemes result in higher accuracy but longer simulation times and larger CISTs. More relaxed compression schemes result in less accuracy but shorter simulation times and smaller CISTs. We proposed and evaluated three instruction segment compression schemes, each with a distinct trade-off. Our results show that the optimal compression scheme, with respect to accuracy, simulation time, and space, depends on the target workload. Using the

optimal compression scheme for each workload, AXCIS is highly accurate and configuration independent, achieving an average IPC error of 2.6%. Although CIST generation in our unoptimized implementation is about four times slower than detailed simulation, CIST generation was only performed once per benchmark. Using pre-computed CISTs, AXCIS is over 10,000 times faster than detailed simulation. 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.

6.2 Future Work

Because workloads differ in their instruction segment varieties, we proposed three different compression schemes to accommodate the variations in the 24 SPEC CPU2000 benchmarks that were evaluated. More work needs to be done to identify one global compression scheme that performs well for all benchmarks. Ideally, this global compression scheme should create CISTs with the best accuracy, simulation time, and space trade-offs.

To increase the design space that can be explored, AXCIS should be extended to support out-of-order processors and simultaneous multi-threading (SMT). In order to support these types of machines, the APM would have to be modified. For example, in order to simulate out-of-order machines, the stall cycles of the defining instruction of each segment would not only depend on previous instruction entries within its segment, but also the instruction entries in other segments.

Currently a CIST can only be reused to simulate processors with the same branch predictor and cache configurations. The DTC should be extended to simultaneously simulate multiple cache and branch predictor configurations in order to create more general CISTs that can simulate a wider range of machines.

6.3 Additional Applications

As shown in this thesis, the instruction segment is a very useful primitive. Instruction segments elegantly encapsulate all important microarchitecture independent and dependent

characteristics of dynamic instructions. Apart from processor simulation, instruction segments can also be used in workload characterization. Because CISTs concisely summarize all important workload characteristics, they can be efficiently analyzed in workload characterization studies. Also, the instruction segment primitive can be used as a metric to identify different categories of workloads. For example, workloads can be categorized based on the variations of instruction segments that occur in their CISTs.

Bibliography

[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] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *Computer Architecture News*, pages 13–25, June 1997.

[3] L. Eeckhout, R. Bell Jr., B. Stougie, K. Bosschere, and L. John. Control Flow Modeling in Statistical Simulation for Accurate and Effi cient Processor Design Studies. In *International Symposium on Computer Architecture (ISCA-31)*, June 2004.

[4] 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.

[5] T. S. Karkhanis and J. E. Smith. A First-Order Superscalar Processor Model. In *International Symposium on Computer Architecture (ISCA)*, pages 338–349, June 2004.

[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] D. B. Noonburg and J. P. Shen. Theoretical Modeling of Superscalar Processor Performance. In *International Symposium on Microarchitecture (MICRO-27)*, pages 52–62, Nov. 1994.

[8] 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.

[9] 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.

[10] 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.

[11] SPEC2000 Alpha Binaries. SimpleScalar LLC. http://www.simplescalar.com.

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

[13] 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.

[14] 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.