Classification of Execution Intervals of Program/Input pair using cache miss and Hidden Markov Model

A Project Report

submitted by

HARDIK SONI

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.
June 2011

THESIS CERTIFICATE

This is to certify that the thesis entitled **Classification of Execution Intervals of Program/Input pair using cache miss and Hidden Markov Model**, submitted by **Hardik Soni**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. B. Ravindran
Research Guide
Assistant Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

This project report is the outcome of my one year of work during which I have been accompanied, guided and supported by many people. It is my great pleasure to thank Indian Institute of Technology, Madras for giving me this opportunity to work in such a beautiful and cooperative environment. I express my earnest gratitude to my guide **Dr. B. Ravindran** lightened me with great innovative ideas and constructive comments. I am very thankful to **Dr. B. Shankar** and **Dr. Madhu Mutyam** for their valuable inputs and feedback which drove me towards better understanding of project. Also, I would like to thank **Dr. C. Siva Ram Murthy** for providing departmental facilities Special thank goes to my **parents** and **friends** for their support and help.

ABSTRACT

KEYWORDS: Hidden Markov Model, Cache Miss, Simulation Points, Phase Based Behavior of Program execution

Clock rate of physical memory is not increasing with increase in microprocessor's clock rate, hence the number of clock cycles required for memory access is huge compare to any other operation of CPU. Performance of system is heavily penalized by this high memory access latency. To hide this memory access latency, Multi level cache with different data prefetching technique is used. Since these data prefetching techniques operate at instruction level, they cannot exploit the repetitive behavior of the program on a larger scale. For a given input, during the entire execution, programs do follow certain patterns and exhibit some repetitive behavior.

Over the last few years, the increase in clock frequency of single core processors has stopped because of the power considerations. Hence, multiple cores are now fabricated on a single chip to improve performance. Advances in silicon technology have increased the number of cores on chip, but rarely all the cores are utilized. Idle cores can be used to improve performance of other busy cores. An intelligent prefetcher running on the idle cores and having knowledge of repetitive behavior of program execution can exploit locality of reference to increase the performace of busy cores.

We used the phase based behavior of a program and built a Discreet Hidden Markov Model using cache miss traces of all the execution intervals of each phase. For that, L1 cache miss trace of entire execution of the program is collected. Execution intervals of the program having similar behavior are classified into the same phase using SimPoint tool. For each phase an HMM is built to learn the cache miss pattern of the phase using cache miss stream obtained during execution intervals of the phase. 50 percent of total execution intervals of each phase are used to build the corresponding HMM. An interval is classified in a phase if the HMM of the phase gives higher probability of observing cache miss stream of the interval compared to HMMs of other phases. The intervals classified in a same phase based on cache miss will have same pattern for cache misses. Achieving good classification accuracy will give higher reliability while predicting the cache misses using the HMM of executing phase. As our long term goal is to predict the future misses using HMM of executing phase, so it is necessary to achieve good classification accuracy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

HHMM	Hierarchical Hidden Markov Model
HMM	Hidden Markov Model
RLE	Run Length Encoding
IPC	Instructions per Clockcycle
BBV	Basic Block Vector

CHAPTER 1

INTRODUCTION

The rate of improvement in microprocessor speed exceeds the rate of improvement in DRAM memory speed. Even though both are growing exponentially, the exponent for microprocessor is larger than that for DRAM. Hence the gap between the improvements in both technology is also growing exponentially. The microprocessor performance followed Moore's law and improved by 50% annually 2004 and 20% since, memory performance has been improving by a mere 9% a year. This growing disparity of speed between microprocessor and memory off chip is known as the *Memory Wall*. The main reason of it is limited off chip communication bandwidth which results in higher memory access latency.

Data access from memory takes tens to hundreds of processor cycle, which creates bottleneck to processor performance and degrades the performance of overall system. To reduce this data access latency data is cached in fast and smaller on-chip memory. There are limitations of power and area on size of on-chip cache memory.

Whenever processor accesses the data, instead of single requested byte a block of bytes is brought to cache memory on the basis of locality of reference. To further reduce the cache miss and increase the performance of overall system, data can also be prefetched from main memory to cache before processor actually requests it.

1.1 Data Prefetching

The use of cache memory hierarchies has reduced the memory access latency. However many scientific and commercial applications spend more than half their runtimes stalled on memory requests.These applications' poor cache utilization is partially a result of the memory fetch policy of most caches. Under these policies cache controllers fetches the data from main memory only after processor requests the data and it is not found in cache. Any cache replacement policy will always result in a cache miss the first time application tries to access a particular datablock which is called a cold miss or a start miss.

In other scenario, when larger data array than the size of cache is accessed, cache will have to evict the data based on replacement policy to make room for new elements of array. If the same data array element is accessed by processor again it has to be fetched from the main memory, these misses are called capacity misses. These cold and capacity misses can be reduced by prefetching data. Data prefetching speculates the cache misses and prefetches the data before processor access it. The prefetching is done in parallel with processor computation and processor need not to stall for long time to access data as it is already availabe in cache.

Taxonomy of data prefetching is described in [4]. Necessary requirements like Accuracy, Timeliness, Overhead and Coverage for effective data prefetching techniques are described in [7].

1.2 Prefetching for Multicore Processors

There are several challenges in designing prefetching strategies for multicore processors. In multicore environment several cores contend to fetch regular data and prefetch data from main memory while sharing memory bandwidth. In single core main memory accepts prefetching requests only from single core, while in multicore environment main memory has to respond prefetch requests from all cores.

As proposed in [6], a dedicated core to prefetch the data and decoupling data access from computing cores can provide good strategy for prefetching data in multicore environment. As number of cores have increased on a single chip and idle core can be used to run intelligent prefetcher agent for other busy cores. So we have tried to explore prefetching technique assuming availability of dedicated core to run the algorithm. In thesis, we have explored the strategy to prefetch the data for single threaded program running on a single core and not sharing data with other cores.

1.3 Exploiting Program Behavior

Various data prefetching techniques are implemented in hardware and software. All these techniques do not make use of repetitive behavior of program execution over billions of instruction. Larger benefits can be achieved by tuning the effective cache size, replacement policy and prefetching data according to the execution behavior of the program. Programs do not behave randomly during their entire execution. Most programs do some repetitive work and exhibit similar behavior at different execution time. Timespan over which program repeat its behavior is so large that behavior can not be memorized by existing data prefetching techniques.

Here, we have tried to build HMMs for all the unique phases of program/input pair execution. All the HMMs are trained by cache miss streams collected during the execution of the corresponding phases. So during the repetitive execution of same phase over different timespan, if processor accesses memory in the same pattern than corresponding HMM can be invoked to predict future misses for the phase.

1.4 Motivation for Classifying Execution Intervals based on Cache Miss Trace

A program can have stable behavior over multiple execution intervals of fixed length. We fixed execution interval size and collected L1 cache miss trace for the all the intervals of the program/input execution. SimPoint tool was used to create baseline classification model for all the intervals of program/input execution. It classifies an execution interval in a phase based on basic block vector analysis. To achieve good prediction accuracy for future cache misses of a phase using the HMM, we tried to evaluate observation probabilities of cache miss trace of an interval with respect to HMMs of all the other phases. If the HMM of same phase as interval emits higher probability compare to others than good accuracy can be achieved while predicting the cache misses during execution of the phase.

1.5 Challenges involved in Classification of Execution Intervals

There were several challenges in modelling cache miss traces of all intervals of the phase. On every cache miss virtual address of the requested data was collected from system having 32 bit virtual address. So there can be 2^{32} possible unique symbols in observation space. The first challenge was to reduce the observation space of discrete symbols for each cache stream from 4 GB.

Execution intervals having similar behavior were classified in the same phase. It was necessary to train the HMM of each phase using cache miss trace of 50% of all the intervals belonging to the phase. The HHMs could be trained using cache miss traces of multiple intervals by applying multiple sequence training as proposed by Rabiner in [8]. The individual frequencies of occurrence for each sequence are weighted by the probability of observing the corresponding sequence measured on original scale.

The length of cache miss trace of an interval was ranging from 0.1 to 1 million. Representing the probability of such large observation sequence measured on real scale within the dynamic range of machine was not possible. Hence, to estimate the HMM parameters using cache miss traces of multiple intervals different approach was taken.

The other major issue was handling the symbols which were not encountered during training phase. As we were evaluating observing cache miss pattern of an execution interval with HMMs of all the phases, it may happen that the data region accessed during the execution interval of the phase is not accessed during execution interval of other phases. So there is a probability of encountering the unseen symbols while evaluating the probability of cache misses trace of an interval of the other phases given the parameters of the HMM of the phase.

The other problem was high deviation in number of distinct symbols observed during estimation of parameters among the HMMs of different phases. So HMMs having less number of distict symbols were emitting higher probability for cache miss trace of any interval compared to HMMs with higher number of distict symbols. The reason behind this is difference in probability distribution of obervation symbols.

We have tried to address all these issues with different approaches. The results with different approaches and techniques are explained in this thesis.

CHAPTER 2

Related Work and Literature Survey

2.1 Phase-Based Behavior

Modern processors can executes billions of instructions in a single second. It is difficult to learn the program's behavior at this speed and at the level of clock cycle. Architectural optimizations can provide large benefits but they are unable to see the program behavior in a larger context. Many researchers have examined the runtime behavior of programs over longer periods of time in [3, 10]. It has been shown that programs execute as a series of phases, where each phase may be very different from the others, while still having a fairly homogeneous behavior within a phase. This time varying behavior can help to improve power management, cache control etc. A phase is an interval of execution during which a measured program metric relatively stable. A program phase also includes all similar sections of execution regardless of temporal adjacency. When a phase change ocuurs during the execution of program, all the architecture metric changes at the same time.

Figure 2.1 show the behavior of benchmark programs gcc and gzip respectively, as measured by various different statistics over the course of their entire execution time. X-axis shows the number of instruction executed. All Y-axis measures the different architecture parameters. Each point on the graph is taken over 10 million instructions worth of execution. The arhcitecture paramateres from top to bottom shown are number of unified L2 cache misses(ul2), energy consumed by the execution of instructions, the number of data cache misses (dl1), the number of branch mispredictions (bpred) and the average IPC.

2.2 Discovering and Exploiting Program Phases

In the paper [10] by Timothy Sherwood offline and online phase classification algorithms are described. To learn the higher level behavior of the program/input

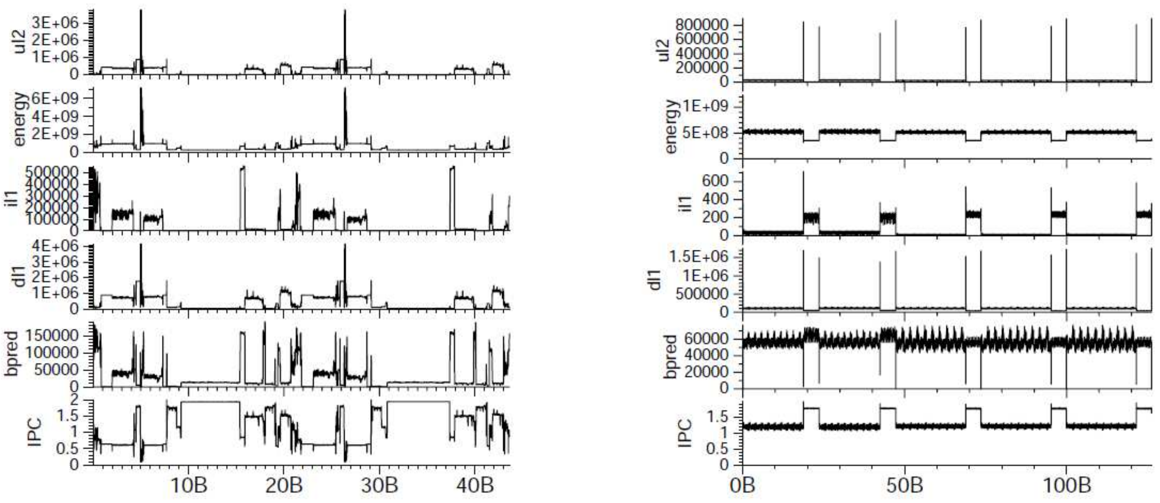


Figure 2.1: Behavior of gcc and gzip benchmark programs for entire execution [3]

pair execution, it is classified into number of phases. Any execution of any program with the specified input is not totally random. Program repeats the behavior during the execution.

2.2.1 Some Definition

Phase is set of intervals within programs execution that have the similar behavior. The program's behavior can be measured in terms of architecture metrics like Instructions per Clockcycle(IPC), Branch misprediction, data cache misses, instruction cache misses, energy used etc. The behavior of each metric changes in unison with the other matrics.

Interval is a time slice. It is a section of continuous execution within a program with specified input.

Phase classification algorithm categorizes all the intervals of program/input pair execution with similar behavior.

Similarity is how close the behavior of two intervals are as measured across the set of metrics. Well-formed phases should have intervals with similar behavior across architecture matrics.

Authors used the architecture independent metric to classify the interval of the program into different phases. The metric should be capable enough to capture changes in program's behavior over the entire execution. To overcome the architecture dependency, authors used Basic Block Vector as measure of similarity between two intevals.

2.2.2 Off-Line Phase Classification Algorithm

Basic Block Vector Analysis

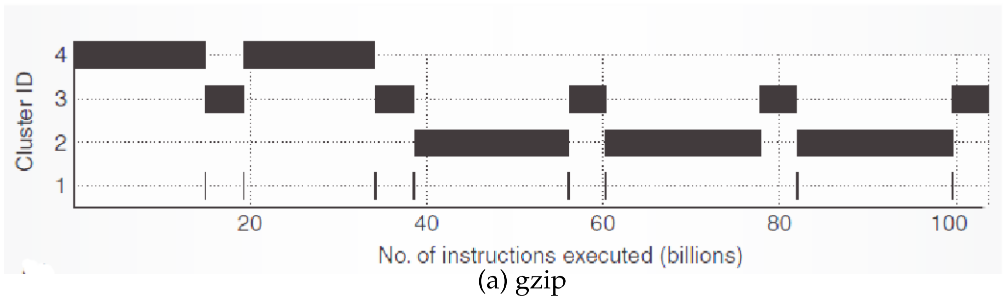
A basic block is a fragment of code executed from start to finish with one entry and one exit. BBV is created for all the intervals of the program/input execution. BBV is a one dimension array having as many number of dimensions as number of static basic blocks in the program. BBV is generated by counting number of times program execution enters all the basic block. Each element in BBV is weighted by the number of instructions in the block and noramlized by sum of all the elements. The similarity between two basic block vectors are measured using the Manhattan Distance. A small distance means vectors are similar and execute more common number of common basic blocks and instructions.

Random Projection

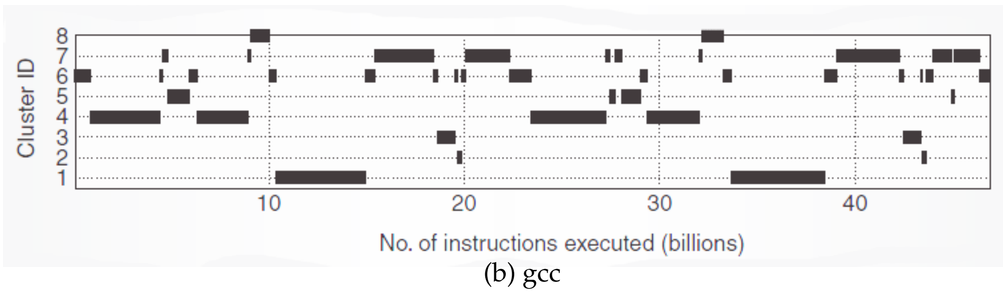
SPEC benchmarks have number of basic blocks ranging from few thousands to hundred of thousands. Any clustering algorithms suffers from high dimentional data. So before running clustering algorithm dimensionality is reduced. Author used random linear projection to reduce the dimensionality of data. This operation involves a matrix multiplication and preserves the structure within the vectors needed for clustering.

Using K-Means Clustering

Clustering algorithm is used to break the complete program/inpute pair execution in smaller groups, also known as phase. The projected BBVs are used as input to clustering algorithm. The K-means clustering was used to divide the execution into phases. It needs number of cluster as input. Algorithm is run for different values of K and *Bayesian Information Criterion* (BIC) was used as a measure of the goodness of fit to a clustering within dataset. BIC gives approximated probability of the clustering given the data that has been clustered. Hence higher the BIC score better the model. Clustering with smallest value of K and greater than predifined percentage of best BIC score has been chosen.



(a) gzip



(b) gcc

Figure 2.2: Graphs showing the phase clustering of the execution intervals [10]

2.3 On-Line Phase Tracking, Prediction and Classification

In [3] Sherwood et al have propsed a unified profiling architecture to capture, classify and predict the phase based program behavior on large time scales. Using this phase information they have built the Markov predictor to predict the next phase.

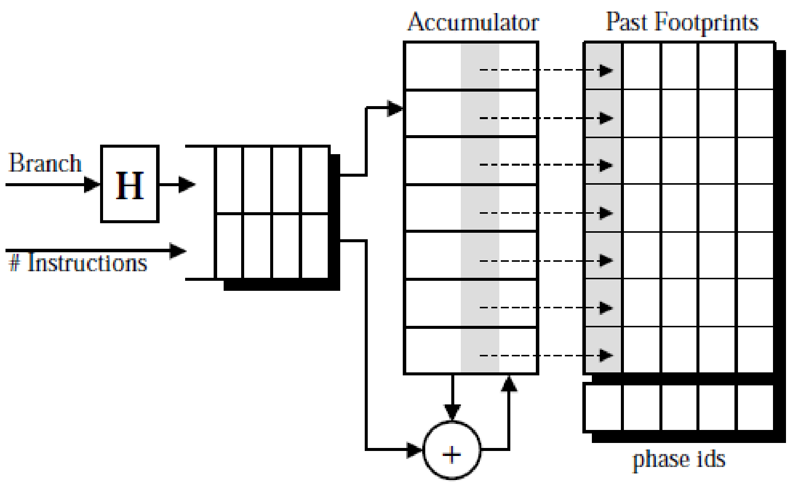


Figure 2.3: Phase Classification Architecture

[11]

2.3.1 Tracking Phases by Executed Code

Phase tracker architecture gathers profile information very quickly in order to keep up with processor speeds, while at the same time it compares any data it gathers with inf ormation collected over the long term. Each branch Program Counter(PC)s and number of instruction executed between branches are branches are captured to approximate BBV. The branch identifier PC and number of instruction executed between the current branch and previous branch is supplied to the architecture as show in figure 2.3. Phase tracker is independent of any architecture as it classifies phases by examining only the that is executed. Essentially, this technique tracks the frquency of the code executed.

2.3.2 Capturing the Code Profile

The branch PC is used to index into the accumulator. It is reduced to a number from 1 to Nbuckets using a hash function. A counter is maintained for each bucket and is incremented by the number of instrucions from the last branch to current branch being processed. The accumulator table is updated once for each branch is executed by processor.

Here hashing the branch PC approximates the random projection used in off-line phase classification akgorithm. Hashing scheme is a degenerate form of random projection that makes a hardware implementation feasible with low error. If all the elements of the random projection matrix consist of either a 0 or a 1, and they are placed such that no column of the matrix contains more than a single 1, then the random projection is identical to this simple hashing mechanism.

2.3.3 Forming a Footprint

History of past phase information is maintained to classify the phase. Phase classification done at the end of every interval. An interval is fixed number of instructions executed and it can be 10M or 100M instruction. Full counter value for each bucket is not stored in past history table. Instead few most significant bits of counter is stored in Past Footprint table. The number of counter value bits that is need to be observed is related to number of buckets. The accumulator counter is reduced by $(bucket[i] \times Nbuckets)/intervalsize$. This is a shift operation if *Nbuckets* and *intervalsize* are in power of 2.

2.3.4 Classifying a Footprint to a Phase ID

The current interval is classified by comparing the footprint to a set of representative past footprints in past history table. This is a vector comparision and it is not the straight forward comparision. The current vector need not to be the same as one of the vector in the past footprint table, as two sections of execution with very similar footprints can be considered a match, even if they do not compare exactly. Manhattan distance with threshold is used between the two vectors to determine if the current interval should be classified as same phase ID as one of the past footprint intervals.

If there is a match, the current interval is classified into the same phase as past foot print vector and current vector is not inserted into the past footprint table. If there is no match, then new phase has been detected and unique new phase ID is created to classify it.When allocating a new phase ID new past footprint entry is also allocated and current vector is stored in that entry. This allows th future similar phases to be classified with the same phase ID. In this way only single vector for each phase is kept to represent the phase.

Using the phase classification information, phase prediction technique is also proposed in [11]. Markov Predictor is implemented in hardware to predict the change in phase and length of the phase.

2.3.5 Markov Predictor

The Markov predictor uses Markov model to predict the phase change of program in execution. The basic idea of Markov model is the future state is independent of its past states conditionally on the present states, which is also called Markov property or *Memorylessness*. A Markov chain is simplest Markov model. A Markov chain is sequence of random variables $X_1, X_2, X_3, ...$ having the property that, given the present, the future is conditionally independent of the past. Formally,

$$P(X_t = j | X_0 = i_0, X_1 = i_1, X_1 = i_1, ..., X_{t-1} = i_{t-1}) = P(X_t = j | t - 1 = i_{t-1})$$

An example markov model is shown in Figure 2.4.

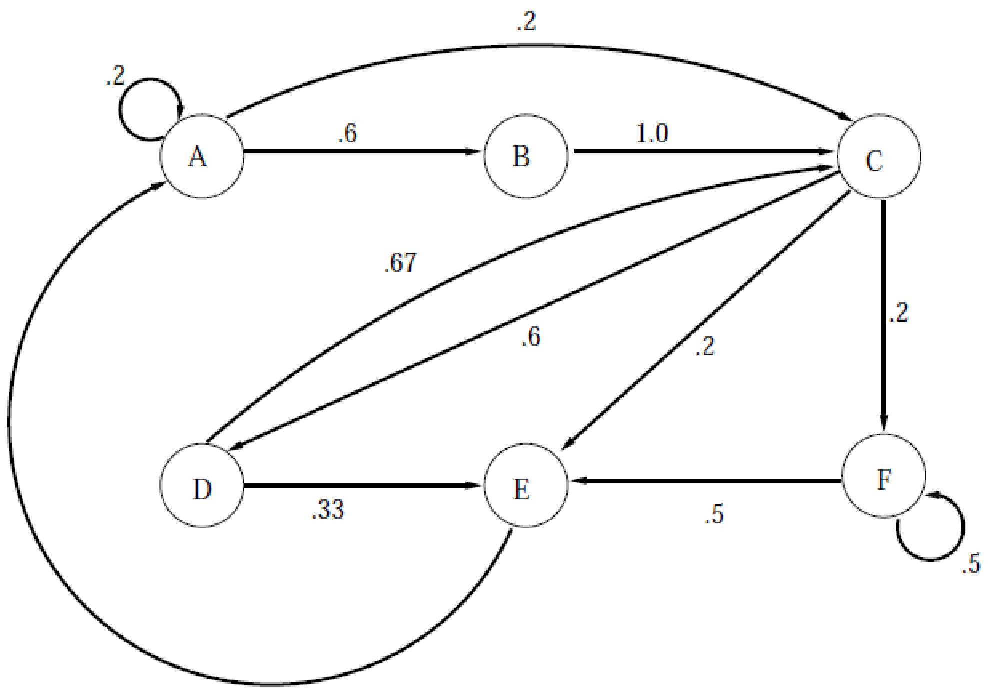


Figure 2.4: An Example of Markov Model

The phase information is characterized by many sections of stable behavior interspersed with abrupt phase changes. Phase change often preceded by stable conditions, so having information about recent intervals it may not be possible to discriminate between sections of stable behavior that precede a phase change and the sections that will continue to be stable. As it is not possible to store entire history, phase information is compressed into a piece of information that can be used as a state of a Markov model.

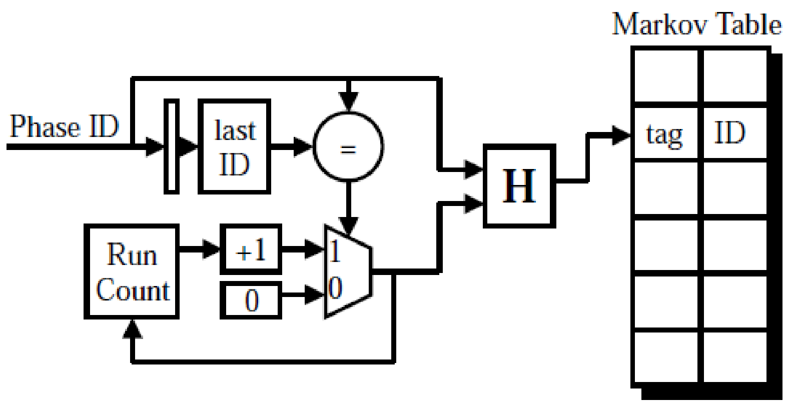


Figure 2.5: Run Length Encoding Markov predictor
[11]

To compress the stable state *Run Length Encoding* (RLE) Markov predictor is used as shown in figure 2.5. The predictor uses the run-length encoded version of history to index into a prediction table. The index into the prediction table is a hash of the phase identifier and the number of times the phase identifier has occured in a row. The lower order bits of the hash function provide an index into the prediction table and higher order bits are a tag. If there is tag match corresponding phase ID stored in the table provides prediction about the next phase to occur in execution. In case of tag miss prior phase ID is assumed to be the next phase ID to occur in the program's execution.

The predictor table is updated if there is a change in phase ID or in case of tag match. The entry is inserted if there is a phase ID change, as when phase change will occur needs to predicted. Successive execution intervals of same phase ID is not stored as they are correctly predicted as last phase ID when table miss occured. If tag match occurs new phase ID is predicted and the run length of current phase ID may have changed, hense predictor table needs to updated.

2.4 Hidden Markov Model and Hierarchical HMM

2.4.1 Hidden Markov Model

A Hidden Markov Model is a statistical Markov model in which system being modeled is assumed to be a Markov process with unobserved states. States are directly visible to the observer in a regular Markov model. In HMM every state has a probability distribution over the possible observable symbols. Also, every state has a transition probability distribution over all the states. An HMM is characterized by following.

1. N, the number of states in the model. The individual states are denoted as $S = \{S_1, S_2, ..., S_N\}$ and the state time t is q_t
2. M, the number of distinct observation symbols per state. The individual symbols are denoted as $V = \{v_1, v_2, ..., v_M\}$
3. The state transition probability distribution $A = \{a_{ij}\}$ where,

$$a_{ij} = P[q_{t+1} = S_j \mid q_t = S_i], \qquad 1 \leqslant i, j \leqslant N. \tag{2.1}$$

4. The observation symbol probability distribution in state j, $B = \{b_j(k)\}$ where,

$$\begin{aligned} b_j(k) = P[v_k \; at \; t \mid q_t = S_j], \qquad & 1 \leqslant j \leqslant N, \\ & 1 \leqslant k \leqslant M. \end{aligned} \tag{2.2}$$

5. The initial State distribution $\pi = \{\pi_i\}$ where,

$$\pi_i = P[q_1 = S_i], \qquad 1 \leqslant i \leqslant N. \tag{2.3}$$

2.4.2 Hierarchical Hidden Markov Model

In the hierarchical hidden Markov model each state is considered to be self-contained probabilistic model as described in [5]. Each state of an HHMM is iteslf an HHMM. The states of HHMM emit sequence of observation symbols rather

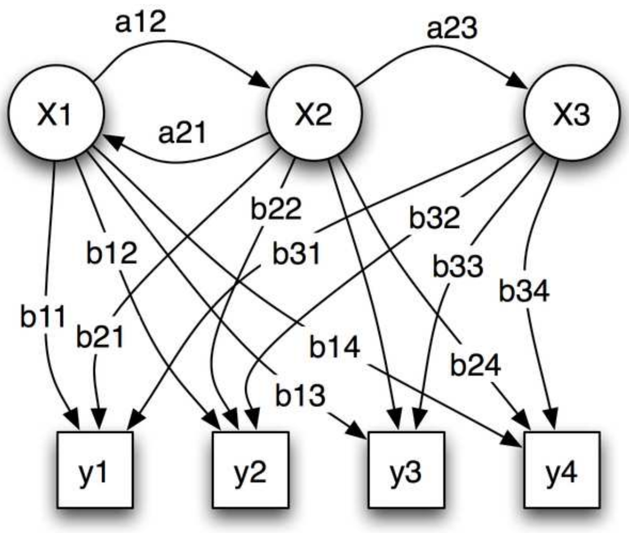


Figure 2.6: An example of Discrete Hidden Markov Model having 3 states and 4 symbols

than single observation symbol. An HHMM generates sequences by a recursive activation of one of the substates of a state. This recursive activation of substates stops when speacial states called production states are reached. Only prduction states emit output symbols through the usual HMM state output mechanism. Hidden states that do not emit obervable symbols directly are called internal states. Activation of substate by an internal state is called a vertical transition. State transition performed in same level is called horizontal transition. After completing a vertical transition control returns to the state which originated the recursive activation chain.

2.5 Predicting the Cache Misses using Hidden Markov Model

Earlier predicting the L1 cache misses for program in execution was explored by Rao in [9]. This work aimed at building an intelligent prefetcher which can run on a dedicated core in a multi core environment. In this section some techniques used in [9] are described. We followed the idea of reducing the observable symbol space.

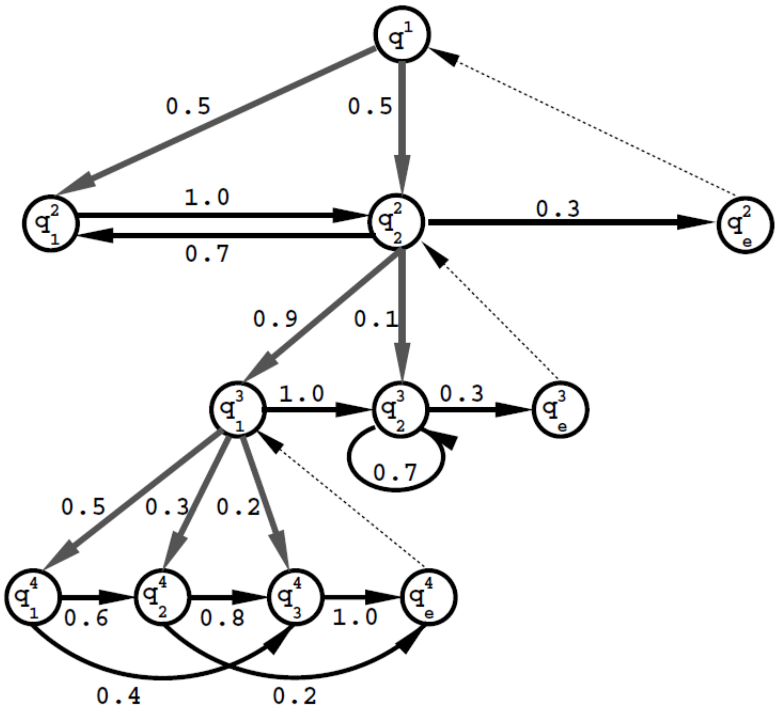


Figure 2.7: An example of HHMM of four levels

[5]

2.5.1 Reducing Observable Symbol Space

We used the technique described in [9] to reduce the observation symbol space. The cache miss traces were taken by simulating system having virtually tagged cache and 4 GB of virtual address space. Each cache block assumed to have 256 bytes. Each cache blocks is addressed by block addresses. On every cache miss, the cache block containing the requested bytes is fetched from main memory and conflicting block on cache is written back to memory. The same cache block is used when any of the 256 bytes is accessed. So 8 bits starting from the least significant bit can be ignored, since cache was simulated with *Write Back* and *Write Allocate* policies.

For example, A part of sequence of virtual addresses for a cache miss trace is shown below.

```
08055123
b7fbe658
c03b9446
c2e96005
bf8c8f08
08055d56
08055112
b7fbe6a5
c03b94c8
c2e960d9
```

Right 24 bits from the maximum significant bit forms the block address. There can not be two successive misses for the same cache block as write back and write allocate policies are enforced. So Last two hexadecimal digits can be discarded from the virtual address. Hence following sequence of addresses will look like,

```
080551
b7fbe6
c03b94
c2e960
bf8c8f
08055d
080551
b7fbe6
c03b94
c2e960.
```

Considering 6 hexadecimal digits has reduced the observation symbol sapce to 2^{24} from 2^{32}. Even 2^{24} is a large number for possible observable symbols from any state of an HMM. It is require to reduce further. Principle of locality of reference was used for futher reduction.

Majority of the accesses might be concentrated in certain address region due spatial locality of reference. To capture the region six each hexadigit was splited into a prefix symbols and an offset in the range. For each prefix count was used to calculate the number of occurence of prefix in trace. As most of the misses are addressed at particular prefix region observation symbol space can be reduced. An example of it is given below in Table 2.1.

Table 2.1: Various prefix length with count of occurence

Prefix (size 1)	count	Prefix (size 2)	count	Prefix (size 3)	count
0	3	08	3	080	3
b	3	b7	2	b7f	2
c	4	c0	2	c03	2
		c2	2	c23	2
		bf	1	bf8	1

From the above observation use of Hierarchy of HMMs was unavoidable. The two level HMMs were used, the top level HMM was trained using prefix symbols and lower level HMMs was trained with discontinuous traces. The top level HMM models different distributions over range of addresses. Each state has distribution of symbols over prefix symbols and each prefix symbols defines range of addresses. There is a corresponding lower level HMM for each prefix symbol in higher level HMM. This lower level HMM is used to model different distributions of the addressed contained in the prefix symbol range. All these addresses are observation symbol for lower level HMMs.

To further reduce the size of trace Coarsening was applied. Similar consecutive traces of prefix symbols were merged in a single prefix symbol. This has reduced the length of observation sequence for higher level HMM. So for prefix size 3 above example trace would look like

080(1)

```
b7f(1)
c03(1)
c2e(1)
bf8(1)
080(2)
b7f(1)
c03(1)
c2e(1)
```

Every symbol emitted at top level indicates short period of time where addresses prefixed with the symbols are emitted. Special terminal symbols and states were used to indicate end of every such time period. So terminal symbol *x* was used to indicate end of sequence having same prefix symbol. Describing the idea with the same illustrated trace as above,

```
080551
xx
b7fbe6
xx
c03b94
xx
c2e960
xx
bf8c8f
xx
08055d
080551
xx
b7fbe6
xx
c03b94
xx
c2e960.
```

The terminal state was modelled as extra state with non zero probabilities of incoming transitions, zero for outgoing transitions and one for self loop. The probability of emitting terminal symbols from non terminal state was set to zero while same from the terminal set was one. The trained HHMM was then used for prediction.

2.5.2 Prediction

The two level HHMM helped to predict the symbols on large number time steps into the future. The HHMM does not emit any symbol at intermediate level as given in [5], but internal states were allowed to emit the symbols. The symbols predicted at top level states defines the range over, which future cache misses may occur. After predicting the prefix symbol at higher level HMM, corresponding lower level HMM is activated to emit possible future misses.

Every transition at the top level of HMM corresponds traversal time steps greater than or equal to one. Each state in top level has expected waiting time which is defined as number of time steps transition process will wait before moving to th next state. This expected waiting time for particular state was calculated as give in [9]

$$E_{waiting_t ime}(i) = \sum_{j=1}^{number\,of\,symbols} b(O_j) * E_{LengthofHMM}(O_j) \tag{2.4}$$

Expected length of lower level HMM is calculated. It estimates the time period of transition in it before termination. That is given by

$$E_{length}(O) = \sum_{length=2} Pr(\;non\;terminal\;state\;transition\,for\;length - 1) \\ * Pr(terminal\;state\;transition\,for\;length) * \text{length} \tag{2.5}$$

Having expected waiting time for a state in top level, probability of being in a state j at time t is given as

$$Pr(j, t) = \sum_{j=1}^{number\,of\,states} Pr(i, t - E_{waiting\;time}(i)) * a_{ij} \tag{2.6}$$

In this work, prediction was done without considering phase based behavior of program. We built HHMM for each phase of the program and tried to classify all the execution intervals based on cache miss probabilities. The classification and different techniques are discussed in chater 4

CHAPTER 3

Experimental Setup

The execution intervals of a program, for a given input, have to be classified based on the cache miss occurances during execution. Various steps are involved in this process. First we classified the execution intervals of the program/input pair using SimPoint3.2[1] tool. This classification was used as baseline classification.

3.1 Generating Basic BLock Vector

SimPoint3.2 requires basic block vectors of the executed program with given input. Various tools were explored to generate Basic Block Vectors(BBVs).

Pin is a dynamic binary instrumentation tool. It provides APIs for dynamic instrumentation of prgram. Pin does not instrument an executable statically by rewriting it, but adds the code dynamically while the executable is running. This also makes it possible to attach Pin to an already running process. Pin supports linux binary executables for Intel (R) Xscale (R), IA-32, Intel64 (64 bit x86), and Itanium (R) processors, Windows executables for IA-32 and Intel64 and few others. Pin tool has APIs for identifying basic blocks and profiling. Program can be developed to generate BBV using those APIs, but it is difficult to test or verify the correctness of such program.

Simple Scalar can be used to generate the BBVs. *Sim-Fast* simulator has been modified by SimPoint3.2 authors to generate BBVs . The drawback of *Sim-fast* simulator is it requires *COFF* alpha binaries of program to be simulated. Simplescalar only understands Ultrix system calls. Cross compliler alpha-linux-gnu creates *ELF* binaries, the *ELF* binaries can be converted into *COFF* format using *objcopy* command. The major issue is *alpha-linux-gnu* uses the GNU C Library, which in turn uses Linux system calls. GCC needs to be cross compile separtely so that it can generate alpha binaries compatible to Simplescalar.

Valgrind3.5.0 was used to generate BBVs for program in execution. Valgrind is available for the variety of the platfirms like x86/Linux, AMD64/Linux, PPC32/Linux, PPC64/Linux, x86/MacOSX, AMD64/MacOSX. The Valgrind distribution includes a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache profiler, and a heap profiler. It also includes two experimental tools: a heap/stack/global array overrun detector, and a SimPoint BBV generator.

3.2 SimPoint Tool

SimPoint3.2 was modified to retrieve cluster numbers for all the intervals of the program in execution. BBVs of program-input pair generated by Valgrind3.5.0 was used as input to SimPoint3.2. Simics simulator was used to retrieve data cache miss trace of the program.

3.3 Generating Cache Miss Trace

Simics simulator was used to generate the cache miss trace for the entire execution of the program. Simics is an efficient, instrumented, system level instruction set simulator. Simics simulations run very fast, some as fast as real hardware or faster than that. Speed of simics simulation also depends on the modules attached with the simulator to gether information. It is capable to provide very detailed information during simulation of workloads. Simics can model target computer at the level of an operating system. Simics can model the binary interfaces to buses, interrupt controllers, disks, video memory, etc. Simics can boot unmodified operating system kernels from raw disk dumps.

To geneate the multiprocessor scenario a four core 32 bit 2.0GHz x86 processors runnig Fedora core 5 system was simulated.

3.3.1 G-Cache Module

Simics provides its own development platform to create new modules, modify existing ones and integrate them. G-Cache module can be modified to generate cache miss trace of execution of the program for the input. G-Cache module is written in C++. Defaut G-Cache module does not provide the sequnce of cache miss addresses. Modified G-Cache module was used to log virtual and physical addresses of cache misses and to discriminate between L1 miss and L2 miss. G-Cache module was modified as discribed in [9]

In Default configuration G-Cache modules are not attached with processors. Various cache oraganizations like multilevel cache, Instrunction cache and Data cahe can be modeled using g-cache module. Using G-Cache modules parameters like cache line size, number of lines, associativity, type of cache, replacement policy, coherence protocol, read/write time, read/write penalty can be configured Cache profilers can be attached to retieve the different statistics of different parameters like cache hits, cache misses etc.

Cache miss traces were collected for two different L1 cache organiation, direct mapped and 4-way Set Associative. 64 K L1 private cache for each of the cores and 1 MB 16 way set-associative L2 cache shared among all the processors as shown in Figure. More details about collecting cache miss traces are explained in the appendix A.2

3.3.2 Benchmarks

SPEC benchmarks are workload prgrams developed by The Standard Performance Evaluation Corporation. SPEC benchmarks contains workloads to create real world user application scenario to test system's processor, memory systems, compilers etc. SPEC benchmarks are provided with source code of workload, various tool sets and precompiled version of tool sets. These tool sets are expected to work with various architectute and operationg systems to compile the workload code and create executables for native architecture.

401.bzip2 and *403.gcc* Spec2006 benchmarks were used to generate the cache miss trace for all the intervals of program in execution. Memory access intensive benchamrks were used to generate cache miss traces. Only integer benchmarks were used so the cache miss traces can be generated rapidly compare to the floating benchmarks. Benchmarks and its execution scripts were modified to capture cache miss traces accurately. Necessary changes in benchmark code to include simics magic instructions is explained in appendix A.2.3.

CHAPTER 4

Implementation

4.1 Baseline Classification Using SimPoint Tool

SimPoint tool applies random projection on basic block vectors of all the intervals. As dimensions of the vectors are very high, to reduce the dimension and at the same time to maintain the structure of underlying data random projection was used. So we generated different classification models with different dimensions of BBVs. The Simpoint tool also uses K-Means clustering to classify the intervals. Essentially, the number of clusters defines the number of phases into which intervals are classified. Given maximum number of phases to classify and fixed number of dimensions SimPoint evaluates the BIC score on the clusterd data. It performs binary search based on BIC score to identify best fit for the data. We generated different baseline classification models for best clustered data as well as various number of clusters.

4.2 Classification of Execution Intervals Based on Memory Region Accesses

In this approach, the classification of execution intervals was done based on number of memory regions accessed and number of time each memory region accessed during all the phases. Fixed number of maximum significant bits were used as memory region. Hexadecimal digits derived from these bits can also be called as prefix symbol, while rest of the bits form the offset to block address from the region. We focused on the prefix symbols as it identifies the memory region. 50 percent of the cache miss traces of all the intervals from each phase were used to collect the information about different memory regions. Then we classified an interval into the phase which has minimum number of new memory regions accesssed with respect to phase's memory regions. We also classified based on number of times new memory regions were accessed during an interval. We generated the baseline classification model using SimPoint3.2 for program/input pair. The results were compared with generated classification model. The results are described in chapter 5.1.

4.3 Training HHMM

To learn the repetitive pattern of memory access during the phase execution, HHMM was built for each phase of the program execution. Two level of hierarchy was used to model traces. We allowed the top level states of HHMM to emit symbols. Each symbol emitted at top level has a corresponding lower level HMM. Each symbol at top level states was modelled with a prefix symbol. For lower level HMMs traces were generated by concatening the discontinuous sequences of offsets. These traces were used to train lower level HMMs.

4.3.1 Training of an HMM with Multiple Sequence

Each phase of program execution is a set of execution intervals having similar behavior. So the top level HMM needs to be trained by cache miss traces of the intervals of the phases. To estimate the parameters of an HMM with multiple sequence, following technique can be used as suggested in [8].

$$\bar{a}_{ij} = \frac{\sum_{k=1}^{K} \frac{1}{P_k} \sum_{t=1}^{T_k-1} \hat{\alpha}_t^k(j)\, a_{ij}\, b_j(O_{t+1}^k)\, \hat{\beta}_{t+1}^k(j)}{\sum_{k=1}^{K} \frac{1}{P_k} \sum_{t=1}^{T_k-1} \hat{\alpha}_t^k(j)\, \hat{\beta}_{t+1}^k(j)} \tag{4.1}$$

$$\bar{b}_j(l) = \frac{\sum_{k=1}^{K} \frac{1}{P_k} \sum_{t=1_{s.t. O_t = v_l}}^{T_k-1} \hat{\alpha}_t^k(j)\, \hat{\beta}_t^k(j)}{\sum_{k=1}^{K} \frac{1}{P_k} \sum_{t=1}^{T_k-1} \hat{\alpha}_t^k(j)\, \hat{\beta}_{t+1}^k(j)} \tag{4.2}$$

Where, superscript k denotes variables for kth sequence and,

$$\begin{aligned} a_{ij} &= \text{State Transition probability from state i to state j,} \\ b_j(O_t) &= \text{Probability of the observing symbol at time step t from state j,} \\ \hat{\alpha}_t(j) &= P(O_1\, O_2\, ...\, O_t,\ q_t \ = \ S_i \mid \lambda), \\ \hat{\beta}_t(j) &= P(O_{t+1}\, O_{t+2}\, ...\, O_T \mid q_t \ = \ S_i,\ \lambda). \end{aligned}$$

The state transitions and observation symbols distribution can be estimated using equations 4.1 and 4.2 respectively. P_k denotes the probability of observation sequence on real scale. For each sequence $O^{(k)}$, the same scale factors will appear in each term of the sum over t as appears in the P_k term and hence will cancel exactly. It was necessary to scale the forward and backward variables as the length of observation sequence was huge. Evaluating probability of any sequence $O^{(k)}$ on real scale was not possible. It was going out of the dynamic range of machine. We modified the calculation approach to estimate the parameters without evaluating the probability on real scale.

The state transition probabilites were estimated by

$$\bar{a}_{ij} = \frac{\sum_{k=1}^{K} \sum_{t=1}^{T_k-1} \bar{\xi}_t^k(i,j)}{\sum_{k=1}^{K} \sum_{t=1}^{T_k-1} \bar{\gamma}_t^k(i)} \tag{4.3}$$

The observation symbol probabilites were estimated using

$$\bar{b}_j(l) = \frac{\sum_{k=1}^{K} \sum_{t=1_{s.t. O_t = v_l}}^{T_k-1} \bar{\gamma}_t^k(j)}{\sum_{k=1}^{K} \sum_{t=1}^{T_k} \bar{\gamma}_t^k(j)} \tag{4.4}$$

The initial state distribution was estimated by

$$\bar{\pi}_i = \frac{\sum_{k=1}^{K} \hat{\alpha}_1^k(i)\, \hat{\beta}_1^k(i)}{\sum_{k=1}^{K} \sum_{j=1}^{N} \hat{\alpha}_1^k(i)\, \hat{\beta}_1^k(i)} \tag{4.5}$$

where,

$$\begin{aligned} \gamma_t(i) &= P(q_t \ = \ S_i \mid O,\ \lambda), \\ \xi_t(i,j) &= P(q_t \ = \ S_i,\ q_{t+1} \ = \ S_j \mid O,\ \lambda). \end{aligned}$$

In equation 4.3 numerator denotes expected number of transitions from state S_i to S_j and denominator denotes expected number of transitions from state S_i from all the training sequences. In equation 4.4 numerator denotes expected number of times state S_j emits symbol v_l and denominator denotes expected number of times in S_j from all the training sequences. To estimate initial state distribution likelihood of being in state i at time $t = 1$ was evaluated and denominator used to normalize the probabilities in 4.5

HMMs were trained using above method to re-estimate probability distributions without evluating probability of any sequence on real scale.

4.3.2 Handling New Symbols While Testing

The major challenge was handling the new symbols(called *unseen* symbols) while testing. The symbols not encountered during training of an HMM is called unseen symbol. The cache miss traces of intervals of other phases may have accessed memory regions not accessed by the phase.

We derived a symbol to represent all unseen symbols, which may be encountered during testing but were not encounter during training. The symbol is called *unseen* symbol for the HMM. It was necessary to have occurrence of this representative unseen symbol during estimation of parameters. Whenever a symbol is encountered first time during the training of an HMM, it was treated as *unseen* symbol. To model the occurrences of the representative unseen symbol every first occurrence of every symbol was treated as *unseen* symbol. It is necessary to have observation probability of unseen symbol lower than any other known symbol. As more the occurrence of unseen symbols lesser should be the probability of emitting the sequence from the HMM.

The first occurrence of unseen symbols are replaced after coarsening the trace. The offsets of first occurrence of all prefix symbols are concatenated to form the offset sequence for lower level HMM of the *unseen* symbol. The following example shows the use of *unseen* symbol for a part of cache miss trace of an interval.

The example block address sequence of a part of cache miss tarce is given below.

a3f0e0

a3f0e4
a3f0e8
a3f0ec
c2e960
a3f0f0
a3f0f4
a3f0f8
a3f0fc
c2e960

Let the entire address space for block addresses are partitioned into memory regions having 4k blocks each. The resulting trace would be

a3f
c2e
a3f
c2e

The discontinuous offset sequence for *a3f* and *c2e* are

0e0, 0e4, 0e8, 0ec
960
0f0, 0f4, 0f8, 0fc
960

The final trace for the top level HMM would be

usn
usn
a3f
c2e

and offset sequence for the each symbol at top level would be

$$\begin{array}{rcl} \text{usn} & \longrightarrow & \text{0e0, 0e4, 0e8, 0ec, 960} \\ \text{a3f} & \longrightarrow & \text{0f0, 0f4, 0f8, 0fc} \\ \text{c2e} & \longrightarrow & \text{960} \end{array}$$

The same procedure of replacing first occurence of every offset symbol in lower level trace with the *unseen* symbol is repeated.

An interval is classified into the phase if the HMM corresponding to that phase has maximum likelihood of emitting the cache miss trace. The probabilities of observing cache miss traces of all the intervals were evaluated with respect to HMMs of all the phases.

4.4 Evaluating Probability of Cache Miss Trace of an Interval

In this section, various approaches for evaluating probability of cache miss trace of an interval using trained HMM is discussed.

4.4.1 Evaluating Probability Using Memory Regions and Offsets

The HHMM was trained as described in section 4.3. The the block address space of cache miss trace was partitioned into same size memory regions as HHMM. If an HHMM has prefix symbol size of 12 bit, the block addresses of the trace were partitioned with 12 maximum significant bits as prefix and rest of the bits with offsets. The successive cache misses for a memory region are compressed to single cache miss for the region and an offset sequence was generated belonging to the memory region miss. The example given in section 4.3 concatenates all the offset sequences for the specific memory region. The following example gives better idea of partitioning the cache miss trace for evaluation of probability. The trace

```
8318db
c2ea91
c2eb11
c2e991
f7e011
c2e991
c2ea11
83193f
8318dc
```

831940

is partitioned as given below.

$$\begin{array}{rcl}
831 & \longrightarrow & \text{8db} \\
\text{c2e} & \longrightarrow & \text{a91 b11 991} \\
\text{f7e} & \longrightarrow & 011 \\
\text{c2e} & \longrightarrow & \text{991 a11} \\
831 & \longrightarrow & \text{93f 8dc 940}
\end{array}$$

The minor modification in forward variable calculation of top level HMM was done to evaluate the probability of entire trace. Two probabilities are associated with each prefix symbol. The probability of observing the prefix symbol and the probability of emitting the offset sequence from the corresponding HMM at lower level. The probability of emitting the offset sequence from the lower level HMM was multiplied with the probability of observing prefix symbol from a state of top level HMM. For top level HMM forward variable $\alpha_t(i)$ were calculated as follows.

$$\alpha_1(i) = \pi_i \; b_i(O_1) \; P(\bar{O}_1|\bar{\lambda}_{O_t}) \qquad 1 \leqslant i \leqslant N \qquad (4.6)$$

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) \; a_{ij} \right] b_j(O_{t+1}) \; P(\bar{O}_{t+1}|\bar{\lambda}_{O_t}) \qquad 1 \leqslant t \leqslant T-1$$

$$1 \leqslant j \leqslant N \qquad (4.7)$$

Where,

O_t = prefix symbol at time t

$\bar{O}_t$ = The offset sequence corresponding to the prefix symbol at time t,

$P(\bar{O}_t|\bar{\lambda}_O)$ = Probability of observing offset sequece $\bar{O}_t$ corresponding to prefix symbol O_t at time t with respect the HMM of O_t.

4.4.2 Evaluating Probability Using Memory Regions and Lengths of corresponding discontinuous traces

Only top level HMM was modelled using prefix symbols, which represents memory regions. We observed the repetitions in lengths of the offset sequences. Instead of modelling offset sequence using lowerlevel HMM, we assumed the exponential distribution on lengths of offset sequences for a specific prefix symbol. The parameter of exponential distribution was estimated using maximum likelihood method. Forward variables for top level HMM were calculated using following way.

$$\alpha_1(i) = \pi_i \, b_i(O_1) \, P(\ell(\bar{O}_1)|\lambda_{O_1}) \qquad 1 \leqslant i \leqslant N \tag{4.8}$$

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) \, a_{ij}\right] b_j(O_{t+1}) \, P(\ell(\bar{O}_{t+1})|\lambda_{O_t}) \qquad 1 \leqslant t \leqslant T-1, \quad 1 \leqslant j \leqslant N \tag{4.9}$$

Where,

O_t = prefix symbol at time t

$\bar{O}_t$ = The offset sequence corresponding to the prefix symbol at time t,

$\ell(\bar{O}_t)$ = Length of offset sequence $\bar{O}_t$,

$P(\ell(\bar{O}_t)|\lambda_{O_t})$ = Probability of observing length of offset sequece $\ell(\bar{O}_t)$ corresponding to prefix symbol O_t at time t with respect the HMM of O_t.

4.4.3 Evaluating Probability Using Memory Regions only

In this approach we tried to evaluate the probability of cache miss trace using memory regions of cache miss trace only. Single level HMM was trained using sequence of prefix symbols only. The offset part from each cache miss address was dropped while evaluating the probability of cache miss trace of an interval.

CHAPTER 5

Results and Observations

Experiments were performed on 401.bzip2 and 403.gcc SPEC 2006 benchmarks. 401.bzip2 was executed with *input.source* file as an input to the benchmark. 403.gcc was executed with input as *166.i*. The cache miss traces were collected for both program/input pairs execution.

5.1 Classification Execution Intervals Based on Memory Region Accesses

As the simpoint uses K-Means clustering, we generated the various classification models using different values of *k*. For each classification model we also varied the size of memory region. For the given classification model, memory regions of 64K block addresses, 4K block addresses and 256 block addresses were tried. So the size of prefix symbol representing memory access region was 2, 3 and 4 hexadecimal digits. The random projection was not used while classifying execution intervals using SimPoint for any classification model for 401.bzip2/input.source execution.

Table 5.1: 401.bzip2/input.source

Prefix Size	Number of Clusters	Unlabelled intervals	Correctly Classified Intervals	Total Intervals
2	10	433	12	508
	22	446	2	510
	25	435	6	512
	35	450	1	511
3	10	388	15	508
	22	409	6	510
	25	387	6	512
	35	394	1	511
4	10	395	12	508
	22	412	6	510
	25	394	4	512
	35	394	6	511

Table 5.2: 403.gcc/166.i

Prefix Size	Number of Clusters	Unlabelled intervals	Correctly Classified Intervals	Total Intervals
2	10	411	0	411
	15	401	1	407
	20	389	8	405
3	10	375	6	411
	15	389	15	407
	20	349	22	405
4	10	209	85	411
	15	210	63	407
	20	173	81	405

Table 5.3: 401.bzip2/input.source

Prefix Size	Number of Clusters	Unlabelled intervals	Correctly Classified Intervals	Total Intervals
2	10	433	12	508
	22	446	2	510
	25	435	6	512
	35	447	1	511
3	10	387	16	508
	22	406	6	510
	25	381	6	512
	35	384	3	511
4	10	396	13	508
	22	407	6	510
	25	390	4	512
	35	386	6	511

Table 5.4: 403.gcc/166.i

Prefix Size	Number of Clusters	Unlabelled intervals	Correctly Classified Intervals	Total Intervals
2	10	409	1	411
	15	395	3	407
	20	385	11	405
3	10	373	6	411
	15	360	16	407
	20	340	23	405
4	10	203	86	411
	15	197	69	407
	20	169	84	405

The tables 5.1 and 5.2 show results for execution intervals of 401.bzip2 amd 403.gcc benchmarks classified based on new memory region accesssed, where for table 5.3 and 5.4 classification was done based on number times new memory region accessed.

It is clear from above results that classification can not be done by merely checking common memory access regions accessed during the phase execution and an interval execution. So we must look at to the cache miss pattern of an execution intervals.

5.2 Classification Using HMM

The section 5.2.1 discusses the performance of of HMM with different classification models. Each model were classified using on two methods. In one method cache miss pattern of only accessed memory region was considered and in other length of offset sequence was modelled using exponential distribution.

Modelling offset sequences using lower level HMMs was not possible as increase in length of offset seqence heads the probability to zero exponentially. In turn, probability of cache miss traces of many intervals could not represented by machine even in double precision.

5.2.1 Results for 401.bzip2

The 401.bzip2 benchmark with *input.source* input has 1011 execution intervals. Each interval length was 100 million instruction. The number of basic blocks were around 3100. All the intervals for a program/input pair was clustered intp phases using SimPoint with different values of k (number of cluster) 10, 25 and 35. While clustering the intervals based on BBV, number of dimensions of BBV was also varied. So clustering was done for mentioned values of k with non projected data as well as data projected on 15 dimensions. SimPoint also chooses the best k value for clustering based on BIC score given higher limit of number of clusters. For non projected BBVs of 401.bzip2/input.source. it chose the 22 clusters and 15 clusters for projected BBVs to 15 dimension.

After applying SimPoint phase clustering algorithm, HMM was used to classify all the intervals based on cache miss traces. Various parameter can be changed while training HMMs of all the phases. Size of memory region was varied while training the HMMs. The experiments were carried out for prefix symbol size from 2 to 4 hexadecimal digits. Four hexadecimal prefix symbol creates memory regions of size 256 block addresses. Dividing the block address space at this granularity created too many memory region. As each memory region is represented as prefix symbol at top level HMM, number of distinct symbols for top level HMM increased to thousands. So for each baseline classification model generated by SimPoint clustering, we experimented with prefix symbol of size 2 and 3 hexadecimal digits.

The only classes which are dominating the entire test set are showed in tables. Precision, Recall and F-Measure for these class are measured using one against all method.

Classification with 10 Clusters

Table 5.5: 10 Clusters With Projected BBVs

Class	Prefix Size 2				Prefix Size 3			
	Top Level HMM		Length Distribution		Top Level HMM		Length Distribution	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
0	0.33	0.08	0	0	0.11	0.01	0.17	0.03
1	0	0	0	0	0.01	0.1	0	0
3	0.11	0.05	0	0	0.04	0.17	0.05	0.125
4	0.08	0.16	0.07	0.16	0.11	0.25	0.18	0.21
5	0	0	0	0	0.04	0.23	0.03	0.21
6	0.10	0.52	0.08	0.47	0.10	0.17	0.05	0.12
7	0.21	0.36	0.19	0.23	0.65	0.21	0.79	0.17

Table 5.6: 10 Clusters Without Projected BBVs

Class	Prefix Size 2				Prefix Size 3			
	Top Level HMM		Length Distribution		Top Level HMM		Length Distribution	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
0	0.25	0.01	0.33	0.01	0.16	0.41	0.17	0.46
1	0	0	0	0	0.01	0.09	0	0
2	0.2	0.036	0.09	0.01	0.33	0.03	0.5	0.025
4	-	0	-	0	0.09	0.09	0.13	0.11
5	-	0	-	0	0.01	0.25	0	0
6	0.09	0.67	0.09	0.67	0.09	0.15	0.02	0.03
7	0.26	0.56	0.30	0.58	0	0	0.5	0.01

In table 5.5 classes 4,6 and 7 are dominating the most of execution intervals. The HMMs of these classes could not converge. The difference between log likelihood of sequences in successive iterations was oscillating. After fixed number of iterations training was stopped. The clear observation can be made that irrespective of method of evaluating probability or size of prefix symbol same set of classes are dominating the test intervals. The overall accuracy of classification was about 10% for any these models.

In table 5.6, majority of data points were classifed in 0, 2, and 6th cluster. The HMMs of these clusters were able to converge but these classes shared higher number of common symbols with rest of the classes, while the HMMs of the other classes faced higher number of unseen symbols during testing.

Classification with 25 Clusters

Table 5.7: 25 Clusters With Projected BBVs

Class	Prefix Size 2				Prefix Size 3			
	Top Level HMM		Length Distribution		Top Level HMM		Length Distribution	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
0	-	0	-	0	0.08	0.2	0.04	0.28
2	0	0	0	0	0.1	0.02	0	0
3	0.03	0.46	0.03	0.30	0	0	0	0
6	-	0	-	0	0.05	0.14	0.03	0.11
9	0	0	0	0	0.15	0.16	0.2	0.06
11	0.05	0.15	0.04	0.15	0.05	0.15	0.05	0.09
16	0.03	0.03	0	0	0.2	0.03	0.2	0.03
17	0.02	0.03	0.04	0.06	0	0	0.05	0.04
19	-	0	-	0	-	0	1	0.04
20	0.03	0.12	0.039	0.12	0	0	0	0
22	0.35	0.14	0.11	0.03	0.27	0.23	0.5	0.19
23	0	0	0.03	0.11	0.02	0.03	0.05	0.125

The change in prefix symbol size significantly changes result of classification for the same model. It can be seen that evaluating the probability of the cache miss trace of an interval based on only prefix sequence gives almost same result as considering exponential distribution on offset sequence of the prefix symbol at top level HMM from table 5.7 and 5.8 .

Table 5.8: 25 Clusters Without Projected BBVs

Class	Prefix Size 2				Prefix Size 3			
	Top Level HMM		Length Distribution		Top Level HMM		Length Distribution	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
1	0.13	0.12	0.1	0.06	-	0	-	0
2	-	0	0	0	0	0	0.11	0.04
5	0.17	0.11	0.15	0.07	-	0	0	0
7	0.27	0.45	0.28	0.31	0.34	0.47	0.40	0.48
14	-	0	0	0	0.05	0.12	0.02	0.05
15	0	0	0	0	0.01	0.25	0.01	0.25
16	0.12	0.26	0.07	0.07	0	0	0	0
17	0.03	0.18	0.03	0.22	0.06	0.31	0.05	0.13
19	0	0	0	0	0.05	0.22	0.09	0.35
20	0.07	0.05	0.04	0.05	-	0	0	0
22	0.04	0.38	0.05	0.38	0	0	0.15	0.12
23	0.62	0.22	0.5	0.13	0	0	0.10	0.26

It is observed that reducing the dimensions of BBVs for 401.bzip2/input.source using Random Projection degrades accuracy compared to non projected BBV. When clustering is done without random projection on BBVs HMMs are able to classify execution intervals more accurately.

Classification with 35 Clusters

Table 5.9: 35 Clusters With Projected BBVs

Class	Prefix Size 2				Prefix Size 3			
	Top Level HMM		Length Distribution		Top Level HMM		Length Distribution	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
0	0.01	0.05	0.03	0.16	0	0	0	0
5	0.05	0.36	0.05	0.31	0	0	0	0
6	0.75	0.10	0.75	0.10	-	0	-	0
7	0.01	0.15	0.01	0.15	0.06	0.07	0.07	0.09
8	-	0	-	0	0.09	0.12	0.04	0.25
9	0	0	0	0	0	0	0.4	0.15
11	-	0	-	0	0.5	0.06	0.2	0.08
12	-	0	-	0	0.01	0.16	0	0
14	0	0	0	0	0.02	0.07	0.03	0.1
15	0	0	-	0	0.01	0.25	0	0
16	-	0	0	0	0.07	0.14	0.07	0.04
17	-	0	-	0	0.03	0.11	0.05	0.11
23	0.07	0.36	0.04	0.15	0.03	0.21	0.04	0.23
25	-	0	-	0	-	0	0.16	0.1
31	-	0	-	0	0.07	0.07	0	0

Tables 5.9 and 5.10 do not give any behavior pattern of classification.

Table 5.10: 35 Clusters Without Projected BBVs

Class	Prefix Size 2				Prefix Size 3			
	Top Level HMM		Length Distribution		Top Level HMM		Length Distribution	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
1	0	0	0	0	0.01	0.42	0	0
2	0.05	0.19	0.05	0.28	0	0	0	0
7	0	0	0	0	0	0	0.25	0.09
14	0	0	0	0	0.05	0.07	0.11	0.12
16	0.02	0.02	0.01	0.02	0	0	0	0
17	-	0	-	0	0	0	0.33	0.06
20	-	0	-	0	0.03	0.07	0.03	0.07
22	0.01	0.1	0.01	0.1	-	0	-	0
23	0.06	0.38	0.08	0.33	0.07	0.09	0.07	0.22
24	0.03	0.07	0	0	-	0	0	0
25	-	0	0	0	0.02	0.06	0.01	0.06
26	-	0	-	0	0.12	0.07	0	0
27	0	0	0.01	0.16	-	0	-	0
28	-	0	-	0	0.02	0.06	0	0
29	0.25	0.12	0.14	0.12	0.08	0.12	0.12	0.25
30	0	0	-	0	0.04	0.11	0.05	0.14
33	0.06	0.02	0.06	0.02	0	0	0	0

Classification with 19 Clusters

Table 5.11: 19 Clusters With Projected BBVs

Class	Prefix Size 2				Prefix Size 3			
	Top Level HMM		Length Distribution		Top Level HMM		Length Distribution	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
0	0.14	0.09	0.2	0.14	0.25	0.05	0.12	0.02
1	0.14	0.05	0.28	0.23	0.25	0.32	0.66	0.13
3	-	0	-	0	0.05	0.21	0.07	0.11
6	0.03	0.11	0.01	0.11	0.05	0.11	0	0
7	0.06	0.46	0.05	0.31	0.15	0.37	0.06	0.08
8	0	0	0	0	0.03	0.03	0.08	0.10
10	0	0	0.14	0.07	0.04	0.07	0.03	0.18
11	0.07	0.09	0.13	0.12	0	0	0	0
12	-	0	-	0	0.01	0.25	0	0
13	-	0	0	0	0.03	0.04	0	0
14	0.06	0.21	0.09	0.17	0	0	0	0
15	0	0	0	0	0.2	0.09	0.14	0.06
16	0.11	0.07	0.13	0.07	0.33	0.02	0.15	0.06
17	-	0	-	0	0	0	0.03	0.25

SimPoint tool gave the best clsutering model for 19 clusters for BBVs projected on 15 dimensions. The table 5.11 also shows that evaluating probability using only prefix sequence or length of offset sequence distribution gives almost same results. That also follows for classification of non projected BBVs into 22 clusters shown in table 5.12.

Classification with 22 clusters

Table 5.12: 22 Clusters Without Projected BBVs

Class	Prefix Size 2				Prefix Size 3			
	Top Level HMM		Length Distribution		Top Level HMM		Length Distribution	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
1	0.04	0.5	0.04	0.37	0.12	0.06	0.25	0.07
3	0.83	0.16	0.71	0.16	0	0	0.02	0.13
4	0.03	0.5	0.05	0.64	0.02	0.21	0	0
5	0.05	0.12	0.06	0.2	0.07	0.4	0.07	0.17
7	0.1	0.12	0.11	0.12	0	0	0	0
8	0	0	0	0	-	0	0.08	0.03
10	0	0	0	0	0.08	0.04	0	0
13	-	0	-	0	0.06	0.12	0.06	0.16
15	0	0	0	0	0.06	0.03	0	0
16	0.11	0.04	0.11	0.04	0.46	0.13	0.66	0.18
18	-	0	-	0	0.08	0.18	0.16	0.13

The major reason of some classes(phases) dominating the entire test intervals was the number of distinct symbols observed by the HMM of the class during training. The HMMs with higher number of distinct symbols could not classify the execution intervals of same class. The HMMs with very low number of distict symbols faced too many unseen symbols from cache miss trace of interval of its own phase and other phase also. The HMMs of phases having most shared number of symbols with other phases emitting more likelihood than the HMMs of rest of the phases.

5.2.2 Results for 403.gcc

The 403.gcc benchmark with 166.i input has 836 execution intervals. It has more than 60 thousands basic blocks, hence the dimension of the BBV is huge. The random projection was used to reduce the dimension while clustering the intervals into phases. The classfication models were generated having 10 and 15 clusters and each with 15 and 3100 dimensions of BBVs.

Classification with 10 Clusters

Table 5.13: 10 Clusters With 15 Dimension BBVs

Class	Prefix Size 2				Prefix Size 3			
	Top Level HMM		Length Distribution		Top Level HMM		Length Distribution	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
0	0.31	0.63	0.39	0.59	0.61	0.59	0.78	0.59
1	0.62	0.40	0.28	0.63	0.26	0.55	0.32	0.95
2	0.41	0.71	0.39	0.87	0.44	0.41	0.29	0.29
4	0.10	0.29	0.03	0.20	0.25	0.29	0.05	0.40
5	0.30	0.23	0.48	0.15	0.36	0.33	0.48	0.28
6	0.51	0.23	0.41	0.18	0.58	0.41	0.59	0.33
7	0.40	0.18	0.43	0.27	0.26	0.91	0.27	0.91
8	0.24	0.22	0.27	0.32	0.24	0.40	0.18	0.23
9	0.00	0.00	0.29	0.03	0.00	0.00	0.00	0.00

Table 5.14: 10 Clusters With 3100 Dimension BBVs

Class	Prefix Size 2				Prefix Size 3			
	Top Level HMM		Length Distribution		Top Level HMM		Length Distribution	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
0	0.45	0.52	0.45	0.57	0.47	0.45	0.83	0.47
2	0.11	0.16	0.07	0.06	0.20	0.03	0.13	0.10
5	0.84	0.10	0.69	0.15	0.50	0.34	0.58	0.28
6	0.53	0.63	0.53	0.32	0.52	0.38	0.63	0.60
7	0.43	0.82	0.26	0.82	0.24	0.73	0.24	0.64
8	0.25	0.43	0.31	0.50	0.22	0.50	0.17	0.38
9	0.54	0.37	0.40	0.33	0.47	0.37	0.46	0.68

403.gcc gives better results compared to 401.bzip2 benchmark. The spread of number of distinct symbols among HMMs of all the phases was less compared to the 401.bzip2 benchmark. Apart from the spread, the symbols were well shared by HMMs of the each phase, hence we do not see few class labels dominating all the test intervals in tables 5.13 and 5.14. Although, there are classes in which none of the execution intervals get classified. The HMMs of these classes had cache miss traces of very few intervals compared to other classes for training as very low proportion of total intervals classified into the corresponding phases by SimPoint.

The good classification accuracy was achieved with prefix symbol size of 3 hexadecimal digit compared to 2. Overall 35% classification accuracy was achieved with 3 digit prefix symbol size and 25 % with prefix symbol size 2. It captures more information at the cost of increased number of distinct prefix symbols for top level HMMs.

Classification with 15 Clusters

Table 5.15: 15 Clusters With 15 Dimension BBVs

Class	Prefix Size 2				Prefix Size 3			
	Top Level HMM		Length Distribution		Top Level HMM		Length Distribution	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
0	0.26	0.39	0.36	0.46	0.63	0.41	0.76	0.51
1	0.21	1.00	0.16	0.60	0.21	0.80	0.25	0.60
2	0.10	0.25	0.17	0.38	0.50	0.25	0.40	0.25
5	0.60	0.14	0.50	0.08	0.40	0.14	0.32	0.15
6	0.60	0.42	0.41	0.30	0.52	0.53	0.65	0.56
7	0.52	1.00	0.23	0.91	0.25	0.18	0.33	0.18
8	0.25	0.10	0.75	0.16	0.29	0.45	0.26	0.30
9	0.29	0.10	1.00	0.03	0.19	0.12	0.11	0.03
10	0.00	0.00	-	0.00	0.25	0.23	0.30	0.27
11	-	0.00	-	0.00	0.25	0.78	0.23	0.88
12	0.03	0.10	0.05	0.22	0.06	0.20	0.09	0.50
13	0.29	0.94	0.28	0.94	0.33	0.82	0.29	0.94
14	0.64	0.37	0.50	0.33	0.47	0.37	0.50	0.32

Increasing the number of dimensions for random projection and using prefix symbol of size 3 digit gives better precision and recall for 403.gcc benchmark in 10 and 15 cluster configuration. Classes 3 and 4 could not label any execution interval in any configuration of any classification model. They did not have cache miss traces of enough intervals. They suffered due very low proportion of intervals classified into them by SimPoint.

Table 5.16: 15 Clusters With 3100 Dimension BBVs

Class	Prefix Size 2				Prefix Size 3			
	Top Level HMM		Length Distribution		Top Level HMM		Length Distribution	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
0	0.36	0.33	0.62	0.41	0.70	0.55	0.79	0.56
1	0.50	1.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.22	0.75	0.26	0.86	0.60	0.38	0.25	0.38
5	0.12	0.09	0.12	0.10	0.18	0.16	0.13	0.14
6	0.49	0.55	0.37	0.32	0.52	0.70	0.59	0.63
7	0.38	0.91	0.23	0.91	0.50	0.64	0.54	0.64
8	0.12	0.10	0.45	0.13	0.32	0.35	0.29	0.27
9	0.47	0.40	0.37	0.58	0.42	0.40	0.38	0.55
10	0.15	0.50	0.16	0.50	0.09	0.36	0.12	0.45
11	0.43	0.67	0.46	0.67	0.00	0.00	0.00	0.00
12	0.25	0.67	0.08	0.50	0.25	0.17	0.20	0.25
13	0.65	0.19	0.41	0.11	0.41	0.32	0.42	0.33
14	0.30	0.10	-	0.00	0.38	0.10	0.33	0.04

CHAPTER 6

Conclusions And Future work

6.1 Conclusions

- Simics can be used to generate L1 cache miss traces of single threaded program executing on multi core system without operating system's interference.
- SimPoints gives some reasonable clusters having enough number of intervals, while others have very few intervals.
- The clusters having enough number of intervals and sharing high number of symbols give better precision recall compare to other clusters.
- It is difficult to build classification model for memory intensive program using cache miss traces of interval using HMM, as the 401.bzip2/input.source has more average memory usage than 403.gcc/166.i.
- Modelling cache miss traces of intervals with memory regions only gives more or less same result as modelling with exponential distribution of offset sequence length. So to classify execution intervals very detailed cache miss modelling is not necessary, but to predict the future misses, HHMM with detailed information is required.

6.2 Future work

- Model cache miss traces with relative difference between virtual addresses of two successive cache misses.
- In this work, cache miss modelling was done based on program/input execution pair. Customize the HMM to model cache miss traces of program execution over various inputs.

APPENDIX A

Simics

Simics is developed and maintained by Virtutech. It provides free floating license under Wind River University Program through registration. Simics provides raw disk dumps for wide range of architectures and operating system platforms. Simics simulates the specified target operating system using raw disk dumps.

A.1 Installation and Configuration

Simics needs a license server for floating licenses. *FLEXlm* license server was used to install floating license. Single floating license can be used by multiple users in local network to run simics. Details about installing floating license can be founded at [2]

Virtutec provides link to download simics packages. Simics packages are tar file and available for all 32 bit or 64 bit host architecture and platform specified in license application. Extracting the tar files for base simics package and other add on modules then invoking the install script will install the simics.

A single simics installation can be used by several users on the host machine, and therefore first workspace need to be created. Executing *[simics]/bin/workspace-setup myworkspace* will create workspace directory named myworkspace. *[simics]* is installation directory of simic.

A.2 Simics Usage

To launch the simulation architecture of the target machine must be specified. Simics configurations files having *.simics* extension can also be used to start simulation for the target machine. Other command line options can be provided with configuration file. *tango-common.simics* configuration was used with command line options *num_cpus*, *memory_megs* and *freq_mhz*. Options were set to 4 cores, 3 GB RAM and 2GHz frequency respectively. A disk *dump tango1-fedora5.craff* having unmodified fedora core operating system was used to simulate 32 bit x86 architecture.

Simics gives its own command prompt after launching the simulation with necessary configuration file and command line options. Simulation does not start automatically. The command *continue* or *c* needs to be entered. This will boot the operating system installed in *craff* file. The command *continue* can also be specified with number of instructions to be executed from current state. In multi-core simulation all cores will execute the specified number of instructions before stopping. The command stop can be used to stop the simulation.

A.2.1 Creating Checkpoint

Simics allows to create the checkpoints in order to save the current execution state of the simulation. Simics saves states of all the necessary devices like RAM, ROM, VGA, Disk into the checkpoint files. Simulations need to be stopped before creating the checkpoint. Stop the simulation with *stop* command. The command *write-configuration file = <<checkpointfilename>>* will create the checkpoint. Simulation from the checkpoint file can be resumed from one of the following ways.

1. *read-configuration file = <<checkpointfilename>>* will resume the simulation.
2. Checkpoint file can also be provide as command line argument with *-c* command line option to the simics executable.

A.2.2 Copying Benchmark on Simulated Machine

There are two ways to copying data in and out of simulated system.

If *SimicsFS* is installed on the simulated machine, it can access entire file tree of the host computer from the mount point *host*. But the operating system in the dump *tango1-fedora5.craff* has compatibility issues with file system used in the current host machines.

The other recommended and safe way is to create an iso image of benchmarks to be copied. Create a cdrom object for the simulated machine using command *new-file-cdrom <<isofile>>*. Using cd0.insert cdrom object name cdrom object can be inserted on to the system. Create a temporary directory on the simulated system, mount the /dev/cdrom to the temporary mount point. Now, benchmarks can be copied from the temporary directory to any directory of simulated system.

A.2.3 Magic Breakpoints and Benchmarks Modifications

Simics chooses a special no-operation instructions for each possible target architecture. Such instructions when executed by simulator triggers a *Core_Magic_Instruction* hap and calls the registered function for the hap. Macro *MAGIC(n)* is defined in the header file named *magic-instruction.h*. This file can be found in *[simics]/src/include/simics*. The macro can be used to place a magic instruction in program, where n is the immediate value. The architecture x86 provides only one magic instruction *MAGIC(0)* with immediate value 0. Magic breakpoints are implemented in simics using the single magic instruction. Macro named MAGIC_BREAKPOINT is defined in magic-instruction.h file. This macro places the magic instruction with correct parameter value for the target architecture.

To break the execution of the benchmark at first instruction in simulated machine, MAGIC_BREAKPOINT is inserted as the first line in the main program of the benchmark. The *magic-instruction.h* file needs to be included in the benchmark file having main function. On simics command prompt command *enable-magic-breakpoint* will attach the internal call back handler to the hap. If the command is not invoked on simics prompt prior to the execution of magic-instruction on the simulated, simulation will continue without any effect. Executing the command before will stop the simulation.

A.2.4 Process Trackers

Multi core target architecture was used to collect the cache miss trace of the benchmark running on the simulated system. At a time only one benchmark was executed on the simulated system. The continue command executes the specified number of instructions in all the cores before stopping. To avoid operating system's interference process tracker was used apart from miss trace of idle cores. Simics provides two types of trackers. *Cpu-mode-tracker* is used to distinguish between supervisor mode and user mode execution. *Linux-process-tracker* can track the processes on the x86, PowerPC and UltraSPARC target architecture.

The object of *linux-process-tracker* was created and configured to monitor the benchamrk process execution and collect the traces only when it is active. Simics provides the *Core_Trackee_Active* hap, which invokes the attached handler whenever process with specified pid changes the state from active to inactive or vice versa. The handler can be attached using SIM_hap_add_callback_obj_index. So the traces can be collected only when the process was active.

A.2.5 G-Cache and Cache Configuration

Simics does not model any cache system with default configuration. To speed up the execution simics uses its own memory system. By default, simics does not model incoherence for simplicity and performance. Simics always updates the memory with latest CPU and device transactions. Memory access are atomic and takes no time. Simics can observe and alter memory transaction, hence it is suitable for cache modelling.

G-Cache module can be loaded in the workspace by invoking *workspace-setup* executable with *–copy-device* command line argument. G-cache comes with two types of cache simulations. G-Cache module was changed as described in [9]'s thesis

Cache Profiling Cache profiling simulation collects the information about the cache behavior of a system or an application. If the accurate timing of the memory operations are not desired then no stalling is necessary. The timing-model interface is used to inform all transaction sent by the processor. Simulating the cache at the same time as the execution enables a number of optimizations that simics models make good use of.

Cache Timing The cache timing simulation can be used to study the timing behavior of an application or a system. When it is necessary to measure CPI or cache miss latency or memory access latency timing-model interface should be used. When an object attached to the timing-model interface receives a memory-transaction, it can modify the timing of the transaction by returning a stall time. Simics blocks the transaction for that number of processor cycles before executing it upon receiving non-zero stall time.

A.2.6 Collecting Cache miss trace

Simics provides different simulation(or execution) mode to control the availability features like instruction and data profiling, memory timing and the micro architecture. By default simics uses fastest possible mode. The other simulation modes are -stall, -ma, -fast etc. The simulated operating system was booted in fast mode. Benchmarks are copied and compiled in fast mode and checkpoint was saved. Simulation was resumed from the saved checkpoint file in stall mode. As it is necessary to start simulation in stall mode so the G-Cache can stall the memory transactions.

Modified G-Cache module writes the information of each cache miss trace on simics prompt. Hence, the trace was collected by using command output-file-start <<filename>> in *Core_Trackee_Active* handler. Reading the miss information from the simics command prompt can be stopped by output-file-stop <<filename>> depending on the status of the process

REFERENCES

[1] ”http://cseweb.ucsd.edu/ calder/simpoint/simpoint-3-0.htm” .

[2] ”http://www.macrovision.com/services/support/index.shtml” .

[3] (2003). *30th International Symposium on Computer Architecture (ISCA 2003), 9-11 June 2003, San Diego, California, USA*. IEEE Computer Society, 2003. ISBN 0-7695-1945-8.

[4] **Byna, S.**, **Y. Chen**, and **X.-H. Sun**, A taxonomy of data prefetching mechanisms. *In Proceedings of the The International Symposium on Parallel Architectures, Algorithms, and Networks*. IEEE Computer Society, Washington, DC, USA, 2008. ISBN 978-0-7695-3125-0. URL `http://portal.acm.org/citation.cfm?id=1396806.1397185`.

[5] **Fine, S.**, **Y. Singer**, and **N. Tishby** (1998). The hierarchical hidden markov model: Analysis and applications. *Mach. Learn.*, **32**, 41–62. ISSN 0885-6125. URL `http://portal.acm.org/citation.cfm?id=325865.325879`.

[6] **he Sun, X.**, **S. Byna**, and **Y. Chen**, Improving data access performance with server push architecture. *In International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium*. 2007.

[7] **Kim, J.**, **K. V. Palem**, and **W.-F. Wong**, A framework for data prefetching using off-line training of markovian predictors. *In ICCD*. 2002.

[8] **Rabiner, L.** (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, **77**, 257–286.

[9] **Rao, G.** (2010). *Reducing Cache Misses by Data Pre-fetching in a Multi-core Architecture using Hidden Markov Models*. Master’s thesis, Indian Institute of Technology, Madras.

[10] **Sherwood, T.**, **E. Perelman**, **G. Hamerly**, **S. Sair**, and **B. Calder** (2003). Discovering and exploiting program phases. *Micro, IEEE*, **23**(6), 84 – 93. ISSN 0272-1732.

[11] **Sherwood, T.**, **S. Sair**, and **B. Calder**, Phase tracking and prediction. *In ISCA*. 2003.