The Fuzzy Correlation between Code and Performance Predictability

Murali Annavaram°, Ryan Rakvic°, Marzia Polito[1], Jean-Yves Bouguet[1], Richard Hankins°, Bob Davies[1]

°*Microarchitecture Research Lab (MRL),* [1]*Systems Technology Labs (STL)*

Intel® Corporation

Abstract

Recent studies have shown that most SPEC CPU2K benchmarks exhibit strong phase behavior, and the Cycles per Instruction (CPI) performance metric can be accurately predicted based on program's control-flow behavior, by simply observing the sequencing of the program counters, or extended instruction pointers (EIPs). One motivation of this paper is to see if server workloads also exhibit such phase behavior. In particular, can EIPs effectively predict CPI in server workloads? We propose using regression trees to measure the theoretical upper bound on the accuracy of predicting the CPI using EIPs, where accuracy is measure by the explained variance of CPI with EIPs. Our results show that for most server workloads and, surprisingly, even for CPU2K benchmarks, the accuracy of predicting CPI from EIPs varies widely. We classify the benchmarks into four quadrants based on their CPI variance and predictability of CPI using EIPs. Our results indicate that no single sampling technique can be broadly applied to a large class of applications. We propose a new methodology that selects the best-suited sampling technique to accurately capture the program behavior.

1. Introduction

Several recent studies [11][25][27][28][32] have shown that many SPEC CPU2K benchmarks have a strong relationship between control flow behavior and observed performance. Control flow is usually characterized by the sequence of program counters or basic blocks, while performance is typically characterized by average Cycles per Instruction (CPI) metric. These studies also showed that there are only a few dominant phase behaviors in most SPEC CPU2K benchmarks and, hence, simulating only one representative sample from each phase can improve simulation speed without unduly sacrificing accuracy.

Server class applications, such as On-Line Transaction Processing (OLTP) and Decision Support Systems (DSS), are crucial benchmarks for the design and performance analysis of server processors. Previous studies on server workloads [8][9][14][20] showed that there are significant differences between SPEC and server workloads. For instance, server workloads are multithreaded, have much larger data and instruction foot prints, contain non-loopy code, incur significant OS activity, and suffer from higher cache miss rates, branch mispredictions and frequent context switches [14].

Our research focuses on analyzing server workloads and using the insights gained from the analysis to innovate microarchitecture techniques to improve server processor designs. Hence, the natural question arises: do server workloads also exhibit similar phase behavior in spite of their differences from SPEC? In this research our primary goal is to quantify how well the program counter predicts CPI. The program counter is called Extended Instruction Pointer (EIP) in Intel architectures; in this paper we use EIP to mean program counter. We employ regression trees to precisely quantify the relationship between EIPs and CPI; regression trees allow us to measure the theoretical upper bound on the accuracy of predicting CPI using EIPs.

Using regression trees this paper presents a thorough analysis of the predictability of CPI from EIPs for three commercial workloads: an OLTP workload called the Oracle Database Benchmark-C (ODB-C) [15][1], a DSS workload called the Oracle Database Benchmark-H (ODB-H) [2][1] and SPECjAppServer (SjAS) [4]. Our results show that for ODB-C and SjAS that suffer significant number of L3 misses, miss penalty overshadows stalls due to other microarchitectural bottlenecks. Hence, CPI is primarily determined by L3 misses and is independent of the EIPs. Other benchmarks, such as query Q13 in ODB-H, execute a small code segment repeatedly and predictably over a large data set and exhibit cyclic phase behavior where CPI can be determined by observing EIPs.

Using CPI variance and CPI predictability we classify the benchmarks into four quadrants. We show that the predictability across benchmarks varies widely and there is a fuzzy boundary between phase and no-phase behavior. Hence, no single sampling technique can be broadly applied to a large class of workloads. We propose using quadrant based classification to better understand the wide range of workload behaviors and select the best-suited sampling technique to accurately capture the program behavior for each workload.

The rest of this paper is organized as follows. Section 2 describes the setup and tuning of the three server workloads. Section 3 describes the data capture process. Section 4 introduces regression tree concept, which is the

[1] ODB-C and ODB-H are not compliant TPC-C and TPC-H Benchmarks™, even though there may be similarities in the database schema and the transactions in the workload. Any results presented here should not be interpreted as or compared to any published TPC Benchmark results. TPC-C and TPC-H Benchmarks are trademarks of Transaction Processing Performance Council (TPC).

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 **IEEE**

heart of our methodology that quantifies the CPI and EIP relationship. The results from the analysis are presented in Section 5 and Section 6. Section 7 describes the quadrant classification, and explains which sampling techniques are best suited for each quadrant. Section 8 briefly describes relevant prior work. We conclude in Section 9.

2. Workload Setup and Tuning

For this research, we use three commercial grade server workloads: an OLTP workload, a DSS workload (both based on the Oracle database server), and the SjAS application server workload. For the purpose of comparison, in Section 7 we also present interesting and contrasting results from the SPEC CPU2K benchmarks.

2.1 Server Workload Description

The OLTP workload used in this study is the Oracle® Database Benchmark-C (ODB-C), which is derived from the Oracle 10*g* RDBMS. ODB-C simulates an order-entry business where clients execute transactions against a database. A more detailed description of ODB-C can be found in [15].

The DSS workload used in this study is called Oracle Database Benchmark-H (ODB-H). ODB-H also uses the Oracle 10*g* RDBMS as the underlying database server. ODB-H consists of 22 business oriented read-only queries similar to those in TPC-H [2]. These queries examine large volumes of data, perform complex computations, and give answers to critical business questions. In our setup all the queries are run sequentially and individual query performance is separately measured.

SPECjAppServer (SjAS) [4] is designed to measure the performance of Java 2 Enterprise Edition (J2EE) [5] on application servers. This workload emulates supply chain management, manufacturing, and order/inventory systems of a large corporation. SjAS comprises of four components. (1) The driver models customer orders that induce transactions into the system. (2) The supplier emulator models the supplier domain of the system. (3) The database provides the repository capabilities for the corporate, orders, manufacturing and supplier transaction data. (4) The application server models the middle-tier that handles the presentation logic, which serves the driver and supplier emulator. It implements the business rules, prepares the information for the presentation logic, and queries the database. In this study we focus only on the application server built on top of BEA™ Weblogic Platform JRockit™ JVM version 8.1. The backend database server is built using Oracle 9i Release 2 RDBMS.

2.2 Hardware Configuration

All our workloads, except SjAS, are run entirely on an Intel® Itanium® 2 processor based system. The experimental system has four 900 MHz Itanium 2 processors running Red Hat Linux Advanced Server 2.1 using the kernel 2.4.9-e.10smp. The processor has three levels of caches. The first level has a 64 KB split instruction and data cache, while the second and third levels have unified caches of 256 KB and 3 MB, respectively. Our system is populated with 16 GB of PC200 DDR memory and has 34 Ultra320 SCSI drives, each with 73 GB of capacity. In our SjAS setup, the Intel Itanium 2 processor based system is used as the application server. An Intel Xeon based 4-way server is used as the database backend. The driver and supplier emulator are also run on the same backend database server, thereby emulating the three-tier system on a two-tier physical system. Since we focus only on the application server of SjAS, combining database and client layers does not impact our results.

2.3 Workload Tuning

Server workloads typically have numerous configuration parameters that can influence their execution behavior. For ODB-C, we use 14 GB as the System Global Area (SGA). SGA is the main memory buffer cache managed by the Oracle database server, which is intended to hold as much of the database working set as possible in memory. The results presented in this paper use an 800 warehouse ODB-C configuration with 56 clients. The data is striped across 32 disks, one disk is used as a log disk and one disk is used for OS. The CPU utilization for our ODB-C run is near 95%.

ODB-H uses a 30 GB (similar to a TPC-H scaling factor of 30) database that is striped across 32 disks and 1 disk is used as log disk. Oracle does not use as much SGA when running ODB-H and hence the SGA is set to 2 GB.

The SjAS workload tuning is done in several steps. In the first step, the backend database server is tuned using a process similar to the process used for ODB-C. In the second step, the application server is tuned by setting the heap size for the JVM appropriately so as to reduce frequent garbage collection invocation. In our setup we used 1.5 GB as the heap size with parallel garbage collection algorithm. We used an injection rate of 100 with 18 threads. Injection rate is the number of business transactions requests per second that are injected by the driver application into the application server.

3. Data Collection Methodology

This section describes the tools and framework used to capture data and analyze phase behavior of programs running on native hardware. Our analysis tool is built on top of VTune [1], a commercially available software performance analyzer for Intel architectures. It has the ability to non-intrusively analyze any program, along with the OS, running on native hardware, including multithreaded programs running on multiprocessors. These features are essential to analyze multithreaded

™ BEA, Weblogic Platform and jRockit are trademarks of BEA Systems Inc.

server programs, which are neither amenable for code instrumentation nor recompilation (due to the lack of source code). Our tool uses a multi-step process to analyze the relationship between CPI and EIPs. In this section we describe the first two steps that collect the raw data and format the raw data for regression tree analysis. Detailed description of our data collection methodology and our tool infrastructure is presented in our previous work [32].

3.1 Data Collection Using VTune

In the first step, the tool uses the underlying VTune driver to monitor a large number of performance/code execution attributes stored in the embedded event counters of the Intel processors while a program is being executed on a physical system. It collects information, such as EIPs and clock ticks, which are used in the later steps for regression tree analysis and validation. VTune interrupts execution at regular intervals (as measured by the number of retired instructions) and records the EIP at the point of interruption and event counter totals (e.g. clocktick count, instruction count). Sampling at high frequency can increase execution overhead. Conversely, too low a sampling frequency will lead to sparse data. Based on our previous study [32], we sampled all benchmarks once every one million retired instructions, and sampled SjAS once every 100,000 retired instructions. SjAS is sampled at higher rate to capture any short dynamic code changes due to JIT compilation. The worst case VTune overhead is 5% for SjAS, and at the one million sampling rate the typical overhead of using VTune is about 2%.

3.2 Creating EIP Vectors (EIPVs)

The VTune samples from the first step are combined to form EIP vectors. The execution of a program is divided into equal intervals each of length 100 million instructions. Each interval is represented by a vector that corresponds to the histogram of EIPs collected during that interval. Let N be the total number of unique EIPs recorded by VTune during a complete run of a benchmark. The j^{th} interval of 100 million instructions is then represented by the *one*-dimensional vector $vec_{x\_j} = [x_{1j}; x_{2j}; \ldots; x_{Nj}]^T$, where x_{ij} is the total number of times the i^{th} unique EIP has been sampled by VTune during the j^{th} interval. If VTune is set to sample code execution at its default rate of once every million instructions executed, then each histogram vector is computed on the basis of 100 consecutive samples. We call vec_{x_j} the j^{th} EIPV. For each sampling interval we subtract the time stamp counter value at the beginning of the sampling from the time stamp counter value at the end of the sampling period. The difference divided by the number of instruction retired in that sample period gives the instantaneous CPI. The average instantaneous CPI EIPV interval can then be computed by averaging the instantaneous values of all the samples in that interval.

3.3 EIPVs versus BBVs

While the main purpose of this work is to identify the relationship between EIPs and CPI using regression trees, our tool is capable of doing machine independent phase detection using k-means clustering of EIPVs, similar to the basic block vector (BBV) clustering approach used in [27]. Note that in [27] the authors collected BBVs by tracking the execution of every basic block using full code profiling. Due to the limitations in our data collection methodology, it is not feasible to collect EIPs at such a high frequency. Furthermore, VTune driver associates an EIP with every sample rather than a basic block. Hence we used EIPVs in this study. While we believe that VTune has adequately sampled code execution to capture all the necessary information for phase analysis [32], a direct comparison with BBVs is beyond the scope of this paper.

4. Regression Tree Analysis of EIPVs

One objective of this paper is to quantify the relationship between CPI and EIPs across a broad range of workloads. In other words, how accurately can one predict the CPI using only EIPs? To quantify this relationship we use *regression trees* [6][7]. This section introduces the concept of regression trees and describes how they can be used to quantify this relationship.

4.1 Building a Regression Tree

The input to the regression tree build algorithm is a collection of data points *(x,y)*, formed by a multidimensional input *x* (*EIPV* in our case), and an one-dimensional output *y* (in our case, *CPI*). Regression trees optimally and recursively subdivide the *x* space with "walls", portions of linear subspaces, so that the resulting "chambers" contain input points whose corresponding *y* values are as homogeneous as possible. Intuitively, in our setup the EIPV space is recursively subdivided into groups, such that the CPIs of all the EIPVs in each group have the theoretically smallest possible variance.

In this section we first describe the algorithm for building the root node of a regression tree, and then describe how the algorithm recursively builds the entire tree. The algorithm chooses the first unique EIP (say, EIP_0) as a tentative candidate for the root node and forms a corresponding collection of tuples, $tup_{EIP0} = [(n_1, CPI_1); (n_2, CPI_2); \ldots;(n_m, CPI_m)]$, where m is the total number of execution intervals (of 100 million instructions), n_i is the number of times EIP_0 is executed in the i^{th} interval, and CPI_i is the instantaneous CPI of that interval.

The tup_{EIP0} is first split into $tup\_l_{EIP0\ and}$ $tup\_r_{EIP0,}$ based on the EIP_0 execution count n_1. The execution count of EIP_0 in each of the tuples in tup_l_{EIP0} is less than or equal to n_1 and those in tup_r_{EIP0} is greater than n_1. The algorithm then computes CPI variances of the tup_l_{EIP0} and tup_r_{EIP0}, and computes the weighted sum (weighted by the number of tuples) of the two CPI variances. The algorithm then repeats this same split process for each unique n_i, i.e. $n_1, n_2 \ldots n_m$. It then chooses the split value n_i that minimizes CPI variance. In other words, of all the

possible two-way splits of tup_{EIP0}, the split that reduces the CPI variance most is selected as the optimal split for EIP_0.

The algorithm then repeats the above process for all the N unique EIPs, and for each EIP it selects the optimal split. Finally, (EIP_{root}, n_{root}) is picked as the root node, where the EIP_{root} minimizes the CPI variance over all the EIPs and n_{root} is the corresponding split value. Intuitively, the algorithm chooses an EIP that most closely tracks the CPI variance as the root node of the tree.

After the initial split, all the EIPVs are separated into two groups depending on the number of times EIP_{root} is executed in each vector. The algorithm then recursively splits each of the sub population of EIPVs to form a *binary regression tree*.

4.2 Regression Tree Example

In this section we use data from Table 1 as input to a hypothetical example to explain the regression tree building process. In this example the program has 3 unique EIPs (EIP_0, EIP_1, and EIP_2) and is divided into eight execution intervals. The EIPV for each interval shows the number of times each unique EIP occurs during that interval (in Millions); the CPI during that interval is also shown in Table 1.

	CPI	EIP_0	EIP_1	EIP_2
$EIPV_0$	**1.0**	100	0	0
$EIPV_1$	**1.1**	80	0	20
$EIPV_2$	**2.6**	0	20	80
$EIPV_3$	**0.6**	80	20	0
$EIPV_4$	**2.0**	20	20	60
$EIPV_5$	**2.1**	20	20	60
$EIPV_6$	**2.5**	20	0	80
$EIPV_7$	**0.7**	80	20	0

Table 1 Example EIPV Table

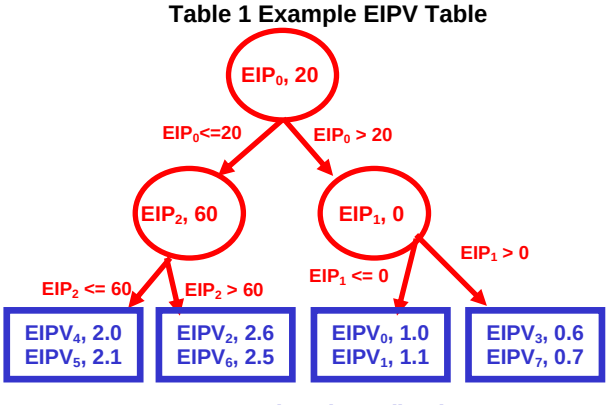


Figure 1 Example Regression Tree with 4 Chambers

The regression tree algorithm first selects EIP_0 as a tentative candidate for the root node and forms a corresponding tuple collection, tup_{EIP0}. Each tuple contains the number of times EIP_0 is executed in an interval along with the CPI in that interval. The tup_{EIP0} is then split into two, tup_l_{EIP0} and tup_r_{EIP0}, so as to minimize the CPI variance. In this example an execution count of 20 is used for the split and hence the tup_l_{EIP0} contains $EIPV_2$, $EIPV_4$, $EIPV_5$, and $EIPV_6$; and tup_r_{EIP0} contains $EIPV_0$, $EIPV_1$, $EIPV_3$, and $EIPV_7$. Similarly, EIP_1 and EIP_2 are next considered as potential choices for the root node. For each choice of root node the reduction in CPI variance is noted. After comparing the CPI variance reduction of the three EIPs, EIP_0 is selected as the root node, since that split reduces CPI variance more than splitting based on any other EIP.

The root node is marked as (EIP_0, 20). All EIPVs in which EIP_0 is executed for no more than 20 times are placed in left subtree, and those EIPVs in which EIP_0 is executed for more than 20 times are placed in the right subtree. Hence, $EIPV_2$, $EIPV_4$, $EIPV_5$, and $EIPV_6$ are placed in the left subtree; and $EIPV_0$, $EIPV_1$, $EIPV_3$, and $EIPV_7$ are placed in the right subtree. The regression tree is further split recursively. Both the left subtree and the right subtree are divided into two nodes to form a regression tree with 4 leaf nodes, as shown in Figure 1.

4.3 Managing Complexity

Note that it is possible to keep subdividing the EIPVs with walls until each chamber contains just one data point (one EIPV). Obviously, in such an extreme case the intra-chamber CPI variance goes to zero when there is only one CPI associated with an EIPV. For practical considerations, however, it is necessary to compromise between the *precision* of the model, i.e. the overall homogeneity of chambers, and the *complexity* of the tree, i.e. the number of final chambers. In our model we chose to restrict the maximum number of chambers to be no more than 50 ($1<=k<=50$), since for most of our applications going beyond 50 chambers does not reduce the per chamber CPI variance. For each k, we seek the optimal tree T_k that has k chambers.

For each chamber C of a tree T_k, we compute the mean CPI value v_C from the CPI value associated with each EIPV in that chamber. Finally, we use *cross-validation* [6], a well known technique to determine tree complexity, to select a k value that strikes the balance between precision and complexity. An intuitive and simplified description of this technique is presented now.

4.4 Cross Validation Algorithm

The cross validation algorithm divides the input data set, *(EIPV, CPI)*, into 10 parts. A regression tree is built using nine out of the ten parts (i.e. 90% of the input data) and the remaining 10% of the data set (*EIPV, CPI*) is subsequently inserted in the chambers of the regression tree. For each EIPV inserted, its estimated CPI is the mean CPI of the chamber (v_C) where it is inserted. The sum of the squared difference between estimated CPI and the EIPV's computed instantaneous CPI is called the cross validation error. Intuitively, cross validation error gives us an upper bound on the predictability of CPI using EIPs.

For the cross-validation process, let us first start with $k=1$, the number of chambers in the tree equals 1. We then repeat the following process for all k ($1<=k<=50$). The input data set (D), namely *(EIPV, CPI)* pairs, is randomly subdivided into 10 equal parts D_j $(j=1,\ldots,10)$.

We remove the j^{th} part D_j (10% of the EIPVs and their associated CPI values) from the data set and use the remaining 90% of the data to build a regression tree T_k, as described in Section 4.1. We repeatedly remove one of the 10 parts and build a regression tree using the remaining nine parts. Using all possible combinations of nine out of the 10 parts, 10 different regression trees are built. Thus, any tree T_k is built using only 90% of the input data and has *k* chambers (leaf nodes).

For a tree T_k, we first choose one EIPV (say *p*) from the 10% data that was left out from building that tree, and place *p* in the *most appropriate* chamber in that tree. For instance, if the root node of T_k is marked ($EIP_{root,}$ n_{root}) we look for the number of times EIP_{root} is executed in *p*. Remember that each EIPV contains one execution count entry for each unique EIP in the program, even if the count is zero. If the value is less than or equal to n_{root} we traverse the left node of the tree, otherwise we visit the right node. We recursively traverse the tree until we reach the leaf chamber, C_p, where *p* is placed. Then the CPI for *p* is predicted as v_{Cp}, the mean CPI of that chamber computed from the 90% of the data as described before. The squared difference between the computed instantaneous CPI associated with *p* and the predicted CPI is $E_{k,p} = (CPI(p)-v_{Cp})^2$. We then repeatedly compute the squared difference for each EIPV in the remaining 10% of the data. We compute the sum of all the squared differences to form $E_{k,i}$ *(i=1,…,10)* for a given regression tree T_k. We compute $E_{k,i}$ corresponding to each of the 10 regression trees that were built using 90% of the data.

The above procedure is repeated for each of the ten trees, for a given *k*. The partial sums of errors for the 10 trees are summed together and averaged over 10, the number of trees,

$$E_k = \frac{\sum_{i=1}^{10} E_{k,i}}{10}$$

The relative error is then considered for all *k* (number of leaf nodes) as

$$RE_k = \frac{E_k}{E}$$

where E is the variance of the total CPI population. The asymptotic value for the relative cross validation error $RE_{k=\infty}$ gives us an upper bound on predicting CPI using just EIPs. In this paper, we consider RE_{kopt} to be a good approximation of $RE_{k=\infty}$, if the relative error RE_{kopt} is within 0.5% of $RE_{k=\infty}$. Note that in this framework, we have repeatedly attempted to *predict* the behavior in CPI space of 10% of our data by using the behavior in the remaining 90% of the data, which is commonly considered as a good approximation of the tradeoff between precision and model complexity.

4.5 Interpreting Regression Tree Results

Before using the regression tree methodology, it is important to first look at the CPI variance of the sampled data. If the variance (E) is small to begin with, indicating a uniform CPI throughout execution, there is often no need to build elaborate regression functions for predicting CPI. A simple average is often enough; note that a regression tree with a single leaf and no split corresponds to the simple average estimator. On the other hand, if variance is sufficiently large, then the value of $RE_{k=\infty}$ provides many useful insights into the program behavior. It represents the amount of information the EIPVs provide for predicting CPI. For instance, $RE_{k=\infty}$=0.15 means that 85% (1-$RE_{k=\infty}$) of the CPI variance can be *explained* by the EIPVs. Furthermore, if the small $RE_{k=\infty}$ is reached with a small number of chambers, say k_{opt}<20, then the workload behavior exhibits a small number of dominant phases. On the other hand, when $RE_{k=\infty}$~1 then EIPVs have no relationship with CPI, or CPI cannot be predicted using only EIPVs. Finally, irrespective of the value $RE_{k=\infty}$, if this error limit is reached only with a large number of chambers, then it indicates that the relationship between EIPVs and CPI is not regulated by few dominant phases, and hence, it is unlikely that using a few samples will accurately represent the whole program behavior.

A comprehensive treatment of the regression tree technique is presented in [6]. In our implementation we use *rpart* from the R programming environment [7].

4.6 Regression Tree versus K-means Clustering

It is interesting to note the similarities and differences of regression trees with previously proposed clustering algorithms. In previous approaches EIPVs (or an equivalent control flow representation, e.g. BBVs [27]) are subdivided into "clusters" using K-means clustering algorithm. The fundamental difference between regression and K-means clustering lies in the amount of information used to compute the phases or clusters. With K-means clustering, the phases are determined *solely* using the EIPVs; in particular, CPI values are not used in determining the clusters. Thus, in K-means, it is *assumed*, rather than *proven*, that the points belonging to the same cluster have very similar CPI.

Since CPI does not drive the clustering process in K-means, it is possible that some clusters have high CPI variance although the EIPVs in the cluster are *similar*. To overcome this problem, recently stratified sampling [25] has been proposed, which uses more than one sample from those phases that have high CPI variance. Although stratified sampling does reduce the CPI variance, the fundamental assumption in k-means remains the same: CPI can be accurately predicted using just EIPVs.

When using regression trees, the "clustering" (i.e. the partitioning of the EIPV space into chambers) is driven by CPI variations between EIPV points being grouped into the same chamber. Therefore, *by construction*, the EIPVs

that ended up in the same chamber will have similar CPI values. In other words, the difference between the two approaches is that, in regression trees, CPI optimally drives the subdivision of EIPVs into chambers and, hence, can show the inherent limitation in predicting CPI with just EIPVs. The grouping of EIPVs into chambers in the regression tree is conceptually similar to grouping EIPVs into K-means clusters.

In order to quantify these differences, we compare the regression tree clustering to the K-means clustering for all of our workloads. We choose k-values independently from both schemes, where the k value is less than 50 and the performance predictability is minimized for each algorithm respectively. On average, the regression trees improve CPI predictability by 80% compared to K-means. Even though they both are clustered on EIP execution behavior, the lack of CPI information may hamper the performance of the K-means algorithm. The major focus of this research is not to compare regression trees to K-means, and hence the rest of this paper demonstrates the correlation between code flow and performance predictability.

5. ODB-C and SPECjAppServer

This section presents the results from regression tree analysis of transaction intensive server workloads ODB-C and SjAS. We show how the relative error (RE_k) for a regression tree T_k changes with increasing *k*, the number of chambers in the tree, and analyze the reasons for the observed behavior.

Figure 2 shows how relative error changes by increasing the number of chambers (*k*) in the regression trees for ODB-C and SjAS. For ODB-C, as the number of chambers in the regression tree increases, the relative cross validation error increases above one. For SjAS, as the number of chambers in the regression tree increases, the relative error remains flat at approximately 0.96, and the minimum error, $RE_{kopt,}$ is about 0.8 when *k*=3. In ODB-C EIPVs are entirely inadequate to explain CPI variance, while only 20% of the CPI variance in SjAS can be explained by EIPVs. Note that the regression trees are built using 90% of the data and the error is computed using the remaining 10% of the data. If the relationship between EIPVs and CPI is entirely random, any "optimal split" decision made on 90% of the data may not generalize when using the remaining 10% of the data. This can lead in some cases to more complex models performing worse than simple ones (RE>1)!

To understand the reasons for large relative errors in Figure 2, we present the EIP spread with time and CPI variations with time, in Figure 3, for a 60 second interval during the steady state execution of ODB-C and SjAS. Compared to SPEC CPU2K benchmarks, ODB-C and SjAS execute a large number of unique instructions that are rather uniformly distributed. For instance, the number of unique EIP samples collected by VTune (Y-axis in EIP spread graphs) in the 60 second interval is 23,891 in ODB-C and 31,478 in SjAS. On the other hand, the mcf benchmark from CPU2K suite has only 646 unique EIP samples in a 200 second interval. Furthermore, the CPI variance is only 0.01 for ODB-C and even for SjAS it is 0.03. The small CPI variance for the large number of unique EIPs indicates that the performance is independent of the code.

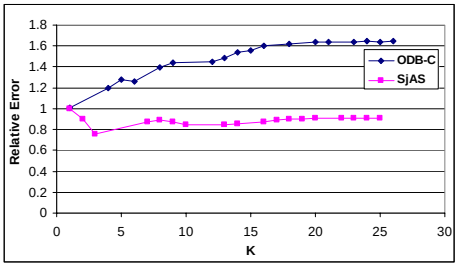
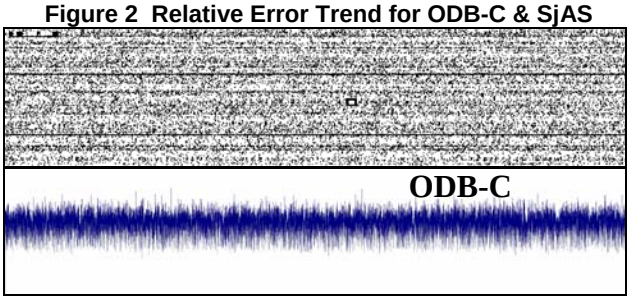
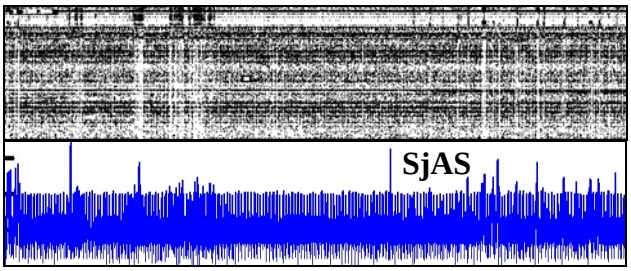


Figure 2 Relative Error Trend for ODB-C & SjAS



(a) ODB-C: EIP (top), CPI (bottom)



(b) SjAS: EIP (top), CPI (bottom)

Figure 3 EIP & CPI Spread of (a) ODB-C, and (b) SjAS

5.1 CPI Breakdown

To explain why CPI is independent of the code, we breakdown the instantaneous CPI into four components: time to execute instructions (WORK), I-cache and branch misprediction stalls (FE), D-cache miss stalls mostly due to L3 misses (EXE), and all the remaining backend stalls (OTHER). The embedded counters in an Intel Itanium 2 processor can measure the stall times and hence the CPI breakdown presented here is precise. Figure 4 and Figure 5 show the CPI breakdown for ODB-C and SjAS, respectively. In ODB-C, L3 cache miss stalls are responsible for more than 50% of the total CPI through out the entire execution interval. In SjAS also, L3 cache miss stalls account for 30-40% of the total CPI. Since L3 misses occur frequently and uniformly throughout the execution, CPI variations caused by other microarchitectural

bottlenecks are overwhelmed by the large L3 miss latency - 160 cycles in the Itanium 2 processor. This uniform CPI behavior leads us to conclude that random sampling is an effective mechanism to approximate CPI behavior.

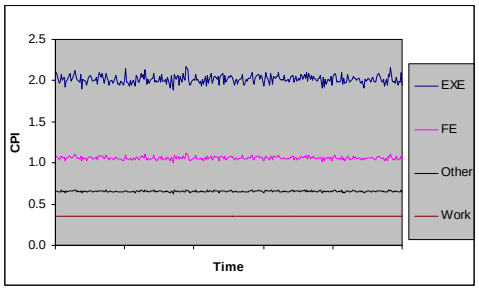


Figure 4 CPI Breakdown for ODB-C

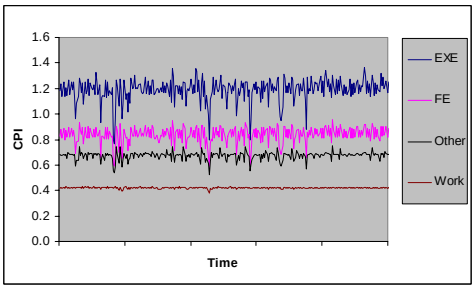


Figure 5 CPI Breakdown for SjAS

5.2 Threading Behavior

Another possible reason why the regression tree analysis shows the lack of relationship between EIPVs and CPI is the presence of multiple threads in ODB-C and SjAS. Server workloads typically have many separate lightweight and heavyweight threads of execution in order to hide large network and disk I/O latencies. Compared to SPEC CPU2K benchmarks, server workloads spend a significant portion of their execution time in the OS performing thread scheduling, managing disk I/O (in the case of ODB-C) and network I/O (in the case of SjAS). For instance, SPEC CPU2K benchmarks spend less than 1% of the execution time in the OS, while ODB-C spends nearly 15% of the execution time in the OS. Because of frequent disk or network I/O, threads voluntarily yield CPU, and there is frequent context switching as a result. ODB-C executes about 2600 context switches per second, while SjAS executes about 5000 context switches per second. In contrast, SPEC CPU2K benchmarks execute about 25 context switches per second.

We sampled the entire system when using the VTune based sampling process to collect the data. Hence the input data used in regression tree analysis contains the EIP samples from multiple threads, and even a single EIPV may have EIPs from many threads. It is possible that although there is a relationship between EIPs and CPI within each thread, this relationship may be hidden when multiple threads interact with each other and thus no phase behavior will be apparent. On the other hand, thread switching may reinforce the relationship if multiple threads are executing the same code.

The VTune sampling process tags each collected sample with a thread/process number that generated the sample. To approximate the impact of thread switching on phase behavior, we first separate the samples on a per thread basis, creating the EIPVs and the corresponding CPIs for each thread. We then use per thread EIPV and CPI data as input to the regression tree analysis. Figure 6 and Figure 7 shows the relative error with and without thread separation for ODB-C and SjAS, respectively. The thread-separated regression tree result is labeled *thread* and the original regression tree is labeled *nothread*. The relative error does decrease when threads are separated, indicating that per thread EIPVs can predict CPI better, although minimally. For instance, the relative error dips below one for ODB-C and below 0.8 for SjAS. However, for both of these workloads, even after separating threads EIPVs have little relationship with CPI. We conclude that for ODB-C and SjAS the large code size coupled with the dominating L3 cache misses reduces CPI predictability with just EIPVs.

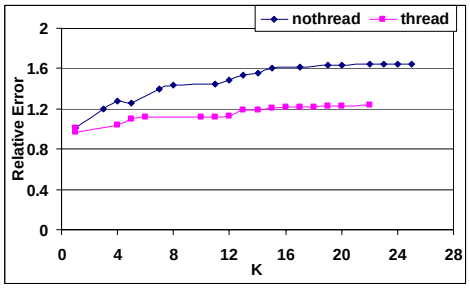


Figure 6 ODB-C Relative Error With & Without Threads

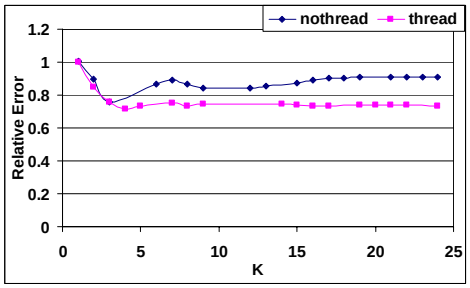


Figure 7 SjAS Relative Error With & Without Threads

6. ODB-H

Although we analyzed all 22 queries in ODB-H, in this section we focus on two categories of behaviors that are seen across all the ODB-H queries. The two categories are: queries with strong EIP-CPI relationship and queries with weak EIP-CPI relationship.

6.1 Strong EIP-CPI relationship

We use the query Q13 to represent the class of ODB-H queries where EIPVs can explain more than 85% of the CPI variations. This query determines the distribution of customers by the number of orders they have made. It counts and reports how many customers have no orders, how many have 1, 2, 3, etc. The three

dominant operations in this query are scan, join and sort of two large tables (about 7 GB of data). In other words, Q13 executes a small segment of code repeatedly over a large amount of data.

Figure 8 shows how relative error changes by increasing the number of chambers (k) in the regression tree for Q13. Unlike ODB-C and SjAS, as the number of chambers in the regression tree increases, the relative cross validation error decreases rapidly and the asymptotic value reaches 0.15 when k_{opt}=9. Hence 85% of the CPI variance in Q13 can be explained by EIPVs. Furthermore, we only need 9 chambers to optimally capture the CPI variance with EIPVs.

Q13 executes for 538 seconds, while the number of unique EIPs is only 4129 (Y-axis on Figure 9 (top)). The loopy execution behavior of Q13 results in a good CPI prediction using EIPVs. Figure 9 shows the EIP and CPI plots. From the visual inspection of these graphs one can see the strong EIP and CPI relationship.

It is worth mentioning that while ODB-H is also multithreaded there are significant differences between ODB-H threads and ODB-C threads. For instance, the context switching rate in ODB-H is less than that of ODB-C. Furthermore, in ODB-H, queries are broken into basic database operations, such as scan, sort, and join. Each query may execute multiple instances of the same operation and one thread is assigned per each instance. Hence, several identical threads may be operating concurrently, and therefore thread switching does not appear to be as detrimental.

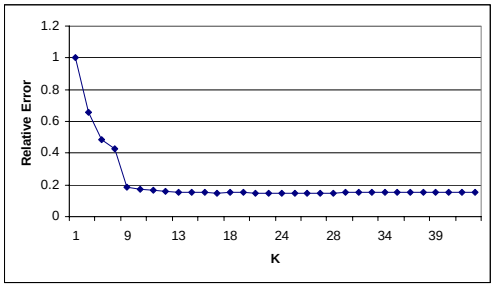


Figure 8 Relative Error Trend for Q13

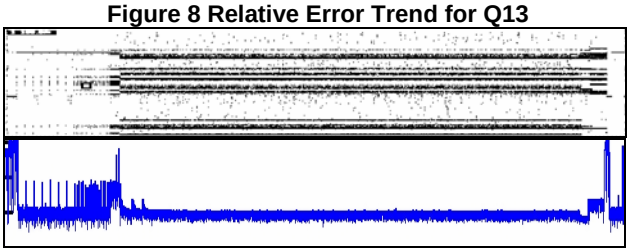


Figure 9 EIP(top) & CPI(bottom) Spread for Q13

6.2 Weak EIP-CPI relationship

The second category of queries demonstrates weak EIP-CPI relationship. We highlight the regression tree analysis for one such query, Q18, in Figure 10. The relative error goes above 1 and stays relatively flat at about 1.1 with increasing k. Hence, using EIPVs can not explain the CPI behavior in Q18.

This query finds a list of the top 100 customers who have ever placed large quantity orders, which is functionally similar to Q13. Both queries involve scan, join and sort operations on two identical tables. However, the Oracle query optimizer uses a sequential scan (access every row in a table) in Q13, and an index scan operation in Q18 where rows in a large table are accessed using a B-tree index. It is well known [31] that index based table scans can have a highly unpredictable behavior due to the randomness of the tree traversal. Hence, even though Q18 repeatedly executes a small code segment, the CPI varies significantly.

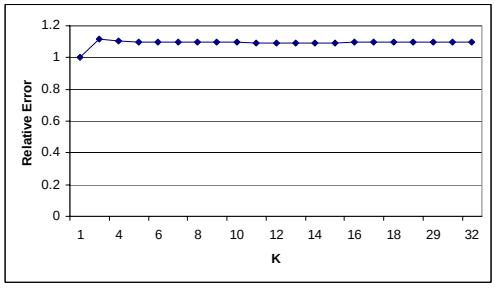


Figure 10 Relative Error Trend for Q18

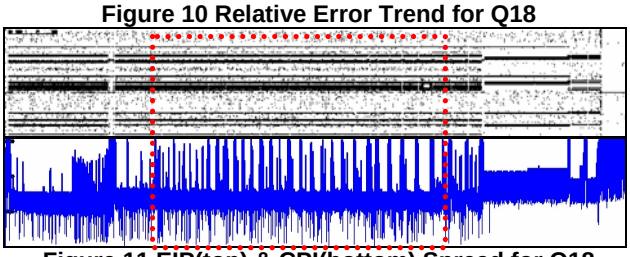


Figure 11 EIP(top) & CPI(bottom) Spread for Q18

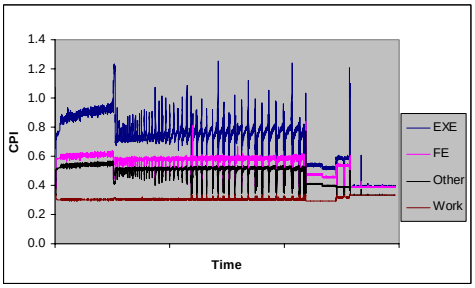


Figure 12: ODB-H: Q18 CPI Breakdown

Figure 11 shows the EIP and CPI variations with time for Q18. For the purpose of illustration, a dotted box is drawn around the EIP spread and the corresponding CPI variation graphs. Although the EIP spread shows that the same set of EIPs are accessed over time, the corresponding CPI graph shows significant variance, resulting in a poor CPI prediction using just EIPVs. While the CPI curves (Figure 11(bottom)) may indicate some apparent phases it is important to note that our approach is only trying to find correlations between EIPVs and CPI. Hence, even though there may be phases in CPI if they are not correlated with EIPVs the relative error will be high.

As further evidence, we present the CPI breakdown of Q18 in Figure 12. The graph shows that there is no

single microarchitectural bottleneck that dominates the CPI. Furthermore, the bottlenecks to performance change with time. For instance, within a short execution interval L3 cache miss stalls (EXE) dominate for a brief time while at other times front end stalls (due to branch mispredicts and I-cache misses) can replace L3 cache miss stalls as the primary bottleneck to performance.

7. Quadrant Classification of Benchmarks

Based on the results and analysis presented in Sections 5 and 6 we propose using *CPI predictability* in conjunction with *CPI variance* as the appropriate way to characterize the phase behavior of a benchmark. Figure 13 shows a conceptual two dimensional space. The X-dimension is the CPI variance and the Y-dimension is the relative error in predicting CPI with EIPVs, i.e. CPI predictability. The X-axis indicates the degree of CPI variance, with low variance to the left and high variance to the right. The Y-axis indicates the degree of predictable phase behavior based on the value of the relative error. In this paper we chose a CPI variance threshold of 0.01 for differentiating two regions along the X-dimension. We use a relative error of 0.15 as the threshold for differentiating the two regions along the Y-dimension. Using these two thresholds, we can identify four quadrants of workload behaviors as shown in Figure 13.

Benchmarks in Q-I have insignificant CPI variance and exhibit limited phase behavior, where CPI can not be explained using EIPVs. A Q-II benchmark has low CPI variance but still exhibits strong phase behavior, so even the low CPI variance can be explained using EIPVs. Similarly, benchmarks in Q-III and Q-IV both have high CPI variance, but their phase behavior ranges from weak to strong.

Table 2 shows the quadrant classification of all the SPEC CPU2K benchmarks, ODB-H queries, ODB-C and SjAS workloads. Surprisingly, 13 of the 26 SPEC benchmarks are categorized in Q-I along with ODB-C, where CPI variance can not be explained using EIPVs. The CPI variances of these benchmarks are extremely low and, hence, EIPVs can not predict/differentiate such small variations in CPI. This result may seem to contradict previous research; however, Dhodapkar and Smith [11] compared three different phase detection techniques and showed that a simple conditional branch count based phase detection correlates 83% of the time with basic block vectors [25][27][28]. Our results show low CPI variance makes it fairly easy to predict CPI within a small error margin and, hence, all three phase detection methods are likely to be equally effective. In fact, our analysis shows that simple sampling techniques, such as uniform sampling [30] with a few samples, work well even for a complex workload like ODB-C when CPI variance is low.

There are only five benchmarks in Q-II. These are the benchmarks where even subtle CPI changes are well captured by EIPVs and are excellent candidates for phase based trace sampling proposed in [25]. However, there is no clear advantage of using phase based trace sampling over uniform sampling for these workloads due to the insignificant CPI variance.

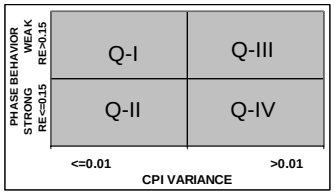


Figure 13 Quadrant-based Workload Characterization

Bmark	CPI var	$RE_{kopt}r$	Bmark	CPI var	$RE_{kopt}r$
ODB-C	0.00	1.01	Q18	0.65	1.00
twolf	0.00	1.00	SjAS	0.03	0.90
crafty	0.00	0.88	gcc	0.04	0.50
eon	0.00	0.80	vpr	0.07	0.40
Q1	0.01	0.78	bzip2	0.02	0.33
parser	0.01	0.71	gap	0.08	0.33
mesa	0.00	0.63	lucas	0.09	0.26
equake	0.00	0.61	vortex	0.03	0.24
gzip	0.01	0.60	Q7	0.04	0.19
galgel	0.00	0.54	perlbmk	0.02	0.18
applu	0.00	0.42	Q5	0.03	0.18
Q19	0.01	0.37	Q10	0.02	0.18
Q3	0.01	0.36	Q15	0.03	0.18
mgrid	0.00	0.35	Q9	0.02	0.17
wupwise	0.00	0.35	Q4	0.02	0.16
art	0.00	0.33			
sixtrack	0.00	0.30			
Q16	0.01	0.25			
Q17	0.01	0.15	Q22	0.04	0.15
ammp	0.01	0.11	Q21	0.02	0.15
Q2	0.01	0.09	Q13	0.02	0.15
facerec	0.01	0.06	Q8	0.02	0.12
Swim	0.01	0.04	Q20	0.03	0.12
			Q14	0.08	0.09
			apsi	0.03	0.09
			fma3d	0.05	0.08
			Q12	0.06	0.06
			Q6	0.03	0.06
			mcf	1.63	0.06
			Q11	0.02	0.04

Table 2 Benchmarks Classified into Quadrants

Q-III is an interesting quadrant. Seven of the SPEC CPU2K benchmarks, seven of the ODB-H queries, and SjAS are in Q-III, where CPI variance is high and the benchmarks have weak phase behavior. For instance, *gcc* and *gap* are known for their lack of phase behavior, and our classification concurs with these earlier findings, as these benchmarks are classified into Q-III. The regression tree analysis sheds new light that benchmarks such as *gcc* and *gap* are not just slightly more difficult to predict their CPI from control flow. Instead, the CPI of these benchmarks is determined by micro-architectural bottlenecks in the system (such as high branch misprediction rate in *gcc)* which may not correlate well

with EIPVs. Apart from CPU2K benchmarks, several ODB-H queries also inherently lack phase behavior. An interesting future research topic is to see if a much higher sampling rate of EIPs can capture the CPI variance.

Benchmarks in Q-IV have a high CPI variance and strong phase behavior. Hence these benchmarks are ideal candidates for phase based trace sampling. 12 (nine ODB-H queries and three SPEC) of the 49 benchmarks are in this quadrant. Using just a few samples based on phase analysis can capture the CPI behavior without needing to use uniform sampling, which may unnecessarily require a large number of small samples.

7.1 Classification Robustness

The partitioning of the workloads into four quadrants is clearly influenced by the choice of the two thresholds for CPI variance and RE. Our choice of the two thresholds is based on the design constraints under which this research is conducted. Varying these thresholds can shift some of the benchmarks to adjacent quadrants. In any case, regardless of specific values chosen for the thresholds, the quadrant based classification helps to better understand the different behaviors of the various workloads, and provides a quantitative means for comparing different benchmarks.

In order to show that the results presented in this paper are not significantly influenced by the in-order Itanium 2 processor, we analyzed EIP and CPI relationship on both a 2.3 GHz Intel Pentium 4 processor based system and a 2.0 GHz Intel Xeon processor based system. Due to the setup complexity of the server workloads, we only analyze a subset of the SPEC CPU2K benchmarks on these systems. Our results showed that CPI variance is higher on both systems when compared to the Itanium 2 based system. For benchmarks that have high cache miss rate, such as mcf, the CPI variance is the highest in the Pentium 4 based system which does not have a large L3 cache. Even with a larger CPI variance in the Pentium 4 based systems, we found that overall, compared to Itanium 2 based systems relative error is 30% better on Pentium 4 and 7% worse on Xeon. Thus, we conclude that quadrant classification is not simply an artifact of our base machine. Instead it shows the inherent relationship between EIP and CPI in our workloads.

In this work, we have fixed the EIPV size at 100 million instructions. To evaluate the impact of this parameter, we performed our regression analysis with an EIPV size of both 50 million and 10 million instructions, keeping the VTune sampling frequency unchanged. Relative to 100 million instruction, on average the CPI variance goes up 7% and 29% for the 50 million and 10 million respectively, corroborating prior research [30]. The Relative Errors were 13% and 14% higher for the 50 million and 10 million, respectively. As the size of the EIPV is reduced, both the CPI variance and relative error are increased, moving some of the Q-IV benchmarks into Q-III. We surmise that, when building EIPVs with fewer than 100 million instructions, it may be necessary to increase the sampling frequency to obtain enough EIP samples per EIPV.

8. Related Work

Database workloads have been the primary server benchmarks for most microarchitecture research on server processors. These benchmarks are analyzed using either simulation of a scaled down workload setup [9][26] or by monitoring a full scale system behavior using performance counters [8][20]. The results broadly show that database applications have large instruction and data memory footprints, suffer from frequent context switches, and have significant cache miss rates and branch mispredictions. These previously published results motivated our research to analyze the phase behavior of server workloads and to compare them with SPEC benchmarks.

Researchers have proposed several techniques to detect and exploit phase behavior of programs [11][12][25][27][28][29]. Broadly, their results showed that an application's EIP determines CPI, independent of the underlying machine configuration. However, they focused on analyzing the phase behavior of SPEC benchmarks while this paper focuses on server workloads. Most of the previous studies worked under the assumption that CPI can be accurately predicted from control flow; in this paper we present a methodology to find the theoretical upper bound on the accuracy of predicting the CPI using only EIPVs.

Sherwood *et al.*[27][28] proposed phase-based sampling that simulates a small number of (about 10) code segments, each with 10-100 million instructions. They used the execution frequency of basic blocks to form basic block vectors (BBV) over consecutive segments of 100M instructions. The BBVs are then grouped into *k* clusters using k-means clustering technique. Perelman *et al.* [25] presented a further refinement that proposed using more than one sample from those clusters that have high CPI variance. To reduce fast-forwarding time during simulation they evaluated algorithms for picking simulation points earlier in a program's execution. The fundamental premise of their work is that program performance (CPI) strongly correlates with BBVs. Our results show that some SPEC benchmarks do exhibit such relationship, which corroborates with their results. However, we show that for server workloads and several of the SPEC benchmarks, EIPVs alone are inadequate to accurately predict CPI. It is also important to note the BBVs are collected by tracking the execution of every basic block using full code profiling. Hence, it is possible that some of the information that was captured by sampling at a fine granularity may be lost in our VTune based sampling at one million instructions. It would be an

interesting future research topic to compare regression tree analysis using EIPVs and BBVs.

Eeckhout *et al.* [13] used statistical modeling to efficiently explore the workload space in order to reduce simulation time. Their analysis is based on collecting several characteristics, such as instruction mix and cache misses, over the entire program execution and then using principal component analysis to determine a reduced set of orthogonal parameters. Using this analysis they infer which program input pairs have similar behavior and prune redundant pairs for design exploration.

Wunderlich *et al.* [30] also used statistical approaches to reduce simulation time of SPEC benchmarks. Their strategy consists of taking numerous samples (50,000) of small size (1,000 instructions) at regular intervals during program execution. Their work presents a systematic way to compute the optimal frequency of sampling and the size of each sample that guarantees a specified error bound for the Instructions Per Cycle (IPC) estimate. We show that for some of the server workloads where the relationship between CPI and EIPVs is not strong, using statistical sampling is in fact a good way to improve simulation accuracy.

Dhodapkar and Smith [11] also identified program phase changes and represented each phase by a working set signature. These working set signatures are stored in a history table to recognize repeating signatures, which are then removed from design space exploration. Sun *et al.* [29] used a modified working set signature to compute the entropy of SpecJBB and ECPerf.

Duesterwald *et al.* [12] proposed monitoring several embedded event counters in IBM Power processors to derive relationships between different performance attributes: IPC, branch mispredictions, and cache misses. They observed that these metrics exhibit periodicity, which can be exploited in the design of on-line table based history predictors. The goal of their study was to design accurate table based predictors for programs with large variability.

9. Conclusions and Future Work

Recent studies have shown that most SPEC CPU2K benchmarks exhibit strong phase behavior, and CPI can be predicted accurately by just observing the EIPVs. This paper focuses on complex server workloads and analyzes whether their CPI behavior can also be accurately predicted by EIPVs. To quantify how accurately EIPVs can predict CPI, this paper uses regression trees that can optimally subdivide the EIPV space into groups, such that the CPIs of all the EIPVs in that group have the theoretically smallest possible variance.

This paper presents a thorough phase analysis using regression trees for three commercial workloads: ODB-C, ODB-H, and the SPECjAppServer (SjAS) benchmark. ODB-C and SjAS have a large code size, and L3 misses occur frequently and uniformly throughout the execution. The large L3 miss penalty overshadows stalls due to other microarchitectural bottlenecks. Hence, CPI is primarily determined by L3 misses and is independent of the EIPVs. On the other hand, several ODB-H queries exhibit *strong phase* behavior, where EIPVs can explain more than 85% of the CPI variance. These queries execute a small code segment repeatedly over a large data set and, hence, exhibit cyclic behavior across many different performance metrics. Interestingly, ODB-H query Q18 also executes small code segment repeatedly over a large data set, yet fails to exhibit phase behavior since the performance varies significantly with input data. We analyze thread interaction in server workloads to show that frequent thread switching has a relatively small impact on the CPI predictability with EIPVs. For workloads such as ODB-C and SjAS, the large code size coupled with the dominating L3 cache miss stall component precludes any possibility of predicting CPI with just EIPVs.

We classify the benchmarks into four quadrants based on the CPI variance and predictability of CPI using EIPVs. We show that server workloads and, surprisingly, even CPU2K benchmarks, exhibit a wide range of phase behaviors. Previous phase analysis techniques produced good phase-based sampling results for many CPU2K benchmarks, partly because some of the CPU2K benchmarks (in Q-II and Q-IV) truly exhibit strong phase behavior. However, a significant number of CPU2K benchmarks (in Q-I) lack phase behavior, but appear to give good clustering results, mainly because their CPI variance is extremely small. For these benchmarks, even a few random samples can adequately capture CPI behavior. Our experimental results indicate that no single sampling technique can be broadly applied to a large class of applications. We propose using quadrant based classification to better understand the wide range of workload behaviors and select the best-suited sampling technique to accurately capture the program behavior for each workload.

Our primary results presented are collected from Itanium-2 based systems. Prior work [15] did an extensive comparative study of the behavior of ODB-C on a Quad Xeon and a Quad Itanium server. Their results showed that 60% of the CPI can be attributed to L3 misses on both MP systems. They concluded that for benchmarks such as ODB-C (and ODB-H) that suffer large number of L3 misses, the underlying ISA has minimal impact on overall behavior; only major system level features, such as a different processor interconnect and different bus design, can impact their behavior. Therefore, we expect similar phase behavior of these complex workloads irrespective of the ISA.

In this paper we focused primarily on the correlation between CPI and EIPs and used microarchitectural event counts only to analyze the observed correlations. CPI is just one of the performance metrics used by processor

designers to evaluate the impact of alternative design choices. Measuring the correlation between CPI and EIPs provides a simple unifying approach for classifying the benchmarks. However, our analysis highlights the inability of just using EIPs in predicting performance. It is interesting to study how code flow impacts non-CPI metrics, such as cache misses and branch mispredictions. One future study is to analyze complex interactions between code path, CPI and the underlying microarchitectural events.

10. Acknowledgements

This work benefited greatly from the valuable insights provided by John Shen. We would like to thank Brad Calder for providing detailed comments on this work.

11. References

[1] VTune: http://www.intel.com/software/products/vtune/.

[2] TPC-H: http://www.tpc.org/tpch/default.asp

[3] EcPerf: http://ecperf.theserverside.com/ecperf/

[4] SPECjAppServer: http://www.specbench.org/osg/jAppServer

[5] J2EE: http://java.sun.com/j2ee/

[6] L. Breiman, J.H. Friedman, R.O. Olshen, and C.J. Stone. Classification and Regression Trees. Kluwer Publishers 1984.

[7] RPART: http://www.r-project.org/

[8] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th Intl. Conference on Very Large Data Bases*, pages 266–277, September 1999.

[9] L.A Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th Intl. Symposium on Computer Architecture*, pages 3–14, June 1998.

[10] P.T. Bickel and K.A. Doksum. Mathematical Statistics. Basic Ideas and Selected Topics. Vol I. Second Edition. Prentice Hall, 2001.

[11] A. Dhodapkar and J.E. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *Proceedings of the 29th Intl. Symposium on Computer Architecture*, pages 233–244, May 2002.

[12] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and Predicting Program Behavior and its Variability. In *Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 220-231, September 2003.

[13] L. Eeckhout, H. Vandierendonck, and K.D. Bosschere. Workload Design: Selecting Representative Program-Input Pairs. In *Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 83–94, September 2002.

[14] M. Franklin, W.P. Alexander, R. Jauhari, A.M.G. Maynard, and B.R. Olszewski. Commercial Workload Performance in the IBM Power2 Risc System/6000 Processor. IBM J. of Research and Development, 38(5): 555–561, 1994.

[15] R. Hankins, T. Diep, M. Annavaram, B. Hirano, H. Eri, H. Nueckel, and J.P. Shen. Scaling and Characterizing Database Workloads: Bridging the Gap between Research and Practice. In *Proceedings of the 36th Annual Intl. Symposium on Microarchitecture*, pages 151-162, December 2003.

[16] J.A. Hartigan and M.A. Wong. Algorithm AS136: A k-means Clustering Algorithm. In *Applied Statistics*, vol 28, pages 100–108, 1979.

[17] T. Hastie, R. Tibshirani, and J. H. Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics, 2001.

[18] M. D. Hill. Evaluating a $2M Commercial Server on a $2K PC and Related Challenges. In *the Seventh Workshop on Computer Architecture Evaluation using Commercial Workloads*, Feb 2004.

[19] L. Kaufman and P.J. Rousseeuw. Finding Groups in Data: An Introduction to Cluster Analysis. Wiley, New York, 1990.

[20] K. Keeton, D.A. Patterson, Y.Q. He, R.C. Raphael, and W.E. Baker. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *Proceedings of the 25th Intl. Symposium on Computer Architecture*, pages 15–26, June 1998.

[21] A.J. KleinOsowski and D.J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 1, June 2002.

[22] A. Y. Ng, M. Jordan, and Y. Weiss. On Spectral Clustering: Analysis and an Algorithm. Advances in Neural Information Processing Systems 14, MIT Press, Pages 849-856, 2002.

[23] S. Nussbaum and J.E. Smith. Modeling Superscalar Processors via Statistical Simulation. In *Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 15–24, September 2001.

[24] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of number of clusters. In *Proceedings of the 17th Intl. Conference on Machine Learning*, pages 727–734, June 2000.

[25] E. Perelman, G. Hamerly, and B. Calder. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 244-255, September 2003.

[26] P. Ranganathan, K. Gharachorloo, S.V. Adve and L.A. Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *Proceedings of the 8th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 307–318, October 1998.

[27] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, September 2001.

[28] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *Proceedings of the 30th Intl. Symposium on Computer Architecture*, pages 336–347, June 2003.

[29] M. Sun, J.E. Daly, H. Wang and J.P. Shen. Entropy-based Characterization of Program Phase Behaviors. In *the Seventh Workshop on Computer Architecture Evaluation using Commercial Workloads*, Feb 2004.

[30] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, and J.C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Intl. Symposium on Computer Architecture*, pages 84–95, June 2003.

[31] S. Chen, P.B. Gibbons and T.C. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD Intl. conference on Management of data*, pages 235-246, May 2001.

[32] B. Davies, J. Bouguet, M. Polito, and M. Annavaram. iPART : An Automated Phase Analysis and Recognition Tool. Technical Report IR-TR-2004-1-iPART, Intel Corporation, February 2004. ftp://download.intel.com/research/library/IR-TR-2004-1-iPART.pdf