Automatically Characterizing Large Scale Program Behavior

Timothy Sherwood Erez Perelman Greg Hamerly Brad Calder

Department of Computer Science and Engineering
University of California, San Diego

{sherwood,eperelma,ghamerly,calder}@cs.ucsd.edu

Abstract

Understanding program behavior is at the foundation of computer architecture and program optimization. Many programs have wildly different behavior on even the very largest of scales (over the complete execution of the program). This realization has ramifications for many architectural and compiler techniques, from thread scheduling, to feedback directed optimizations, to the way programs are simulated. However, in order to take advantage of time-varying behavior, we must first develop the analytical tools necessary to automatically and efficiently analyze program behavior over large sections of execution.

Our goal is to develop automatic techniques that are capable of finding and exploiting the Large Scale Behavior *of programs (behavior seen over billions of instructions). The first step towards this goal is the development of a hardware independent metric that can concisely summarize the behavior of an arbitrary section of execution in a program. To this end we examine the use of* Basic Block Vectors. *We quantify the effectiveness of Basic Block Vectors in capturing program behavior across several different architectural metrics, explore the large scale behavior of several programs, and develop a set of algorithms based on clustering capable of analyzing this behavior. We then demonstrate an application of this technology to automatically determine where to simulate for a program to help guide computer architecture research.*

1. INTRODUCTION

Programs can have wildly different behavior over their run time, and these behaviors can be seen even on the largest of scales. Understanding these large scale program behaviors can unlock many new optimizations. These range from new thread scheduling algorithms that make use of information on when a thread's behavior changes, to feedback directed optimizations targeted at not only the aggregate performance of the code but individual phases of execution, to creating simulations that accurately model full program behavior. To enable these optimizations, we must first develop the analytical tools necessary to automatically and efficiently analyze program behavior over large sections of execution.

In order to perform such an analysis we need to develop a hardware independent metric that can concisely summarize the behavior of an arbitrary section of execution in a program. In [19], we presented the use of *Basic Block Vectors* (BBV), which uses the structure of the program that is exercised during execution to determine where to simulate. A BBV represents the code blocks executed during a given interval of execution. Our goal was to find a single continuous window of executed instructions that match the whole program's execution, so that this smaller window of execution can be used for simulation instead of executing the program to completion. Using the BBVs provided us with a hardware independent way of finding this small representative window.

In this paper we examine the use of BBVs for analyzing large scale program behavior. We use BBVs to explore the large scale behavior of several programs and discover the ways in which common patterns, and code, repeat themselves over the course of execution. We quantify the effectiveness of basic block vectors in capturing this program behavior across several different architectural metrics (such as IPC, branch, and cache miss rates).

In addition to this, there is a need for a way of classifying these repeating patterns so that this information can be used for optimization. We show that this problem of classifying sections of execution is related to the problem of *clustering* from machine learning, and we develop an algorithm to quickly and effectively find these sections based on clustering. Our techniques automatically break the full execution of the program up into several sets, where the elements of each set are very similar. Once this classification is completed, analysis and optimization can be performed on a per-set basis.

We demonstrate an application of this cluster-based behavior analysis to simulation methodology for computer architecture research. By making use of clustering information we are able to accurately capture the behavior of a whole program by taking simulation results from representatives of each cluster and weighing them appropriately. This results in finding a set of simulation points that when combined accurately represents the target application and input, which in turn allows the behavior of even very complicated programs such as `gcc` to be captured with a small amount of simulation time. We provide simulation points (points in the program to start execution at) for Alpha binaries of all of the SPEC 2000 programs. In addition, we validate these simulation points with the IPC, branch, and cache miss rates found for complete execution of the SPEC 2000 programs.

The rest of the paper is laid out as follows. First, a summary of the methodology used in this research is described

ASPLOS X, 10/02, San Jose, CA, USA.

in Section 2. Section 3 presents a brief review of basic block vectors and an in depth look into the proposed techniques and algorithms for identifying large scale program behaviors, and an analysis of their use on several programs. Section 4 describes how clustering can be used to analyze program behavior, and describes the clustering methods used in detail. Section 5 examines the use of the techniques presented in Sections 3 and 4 on an example problem: finding where to simulate in a program to achieve results representative of full program behavior. Related work is discussed in Section 6, and the techniques presented are summarized in Section 7.

2. METHODOLOGY

In this paper we used both ATOM [21] and SimpleScalar 3.0c [3] to perform our analysis and gather our results for the Alpha AXP ISA. ATOM is used to quickly gather profiling information about the code executed for a program. SimpleScalar is used to validate the phase behavior we found when clustering our basic block profiles showing that this corresponds to the phase behavior in the programs performance and architecture metrics. The baseline microarchitecture model we simulated is detailed in Table 1. We simulate an aggressive 8-way dynamically scheduled microprocessor with a two level cache design. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch mis-prediction.

We analyze and simulated all of the SPEC 2000 benchmarks compiled for the Alpha ISA. The binaries we used in this study and how they were compiled can be found at: `http://www.simplescalar.com/`.

3. USING BASIC BLOCK VECTORS

A basic block is a section of code that is executed from start to finish with one entry and one exit. We use the frequencies with which basic blocks are executed as the metric to compare different sections of the application's execution to one another. The intuition behind this is that the behavior of the program at a given time is directly related to the code it is executing during that interval, and basic block distributions provide us with this information.

A program, when run for any interval of time, will execute each basic block a certain number of times. Knowing this information provides us with a fingerprint for that interval of execution, and tells us where in the code the application is spending its time. The basic idea is that knowing the basic block distribution for two different intervals gives us two separate fingerprints which we can then compare to find out how similar the intervals are to one another. If the fingerprints are similar, then the two intervals spend about the same amount of time in the same code, and the performance of those two intervals should be similar.

3.1 Basic Block Vector

A *Basic Block Vector* (BBV) is a single dimensional array, where there is a single element in the array for each static basic block in the program. For the results in this paper, the basic block vectors are collected in intervals of 100 million instructions throughout the execution of a program. At the end of each interval, the number of times each basic block is entered during the interval is recorded and a new count for each basic block begins for the next interval of 100 million instructions. Therefore, each element in the array is the count of how many times the corresponding basic block has been entered during an interval of execution, multiplied by the number of instructions in that basic block. By multiplying in the number of instructions in each basic block we insure that we weigh instructions the same regardless of whether they reside in a large or small basic block. We say that a Basic Block Vector which was gathered by counting basic block executions over an interval of $N \times 100$ million instructions, is a Basic Block Vector of duration N.

Because we are not interested in the actual count of basic block executions for a given interval, but rather the proportions between time spent in basic blocks, a BBV is normalized by having each element divided by the sum of all the elements in the vector.

3.2 Basic Block Vector Difference

In order to find patterns in the program we must first have some way of comparing two Basic Block Vectors. The operation we desire takes as input two Basic Block Vectors, and outputs a single number which tells us how close they are to each other. There are several ways of comparing two vectors to one another, such as taking the dot product or finding the Euclidean or Manhattan distance. In this paper we use both the Euclidean and Manhattan distances for comparing vectors.

The Euclidean distance can be found by treating each vector as a single point in D-dimensional space. The distance between two points is simply the square root of the sum of squares just as in $c^2 = a^2 + b^2$. The formula for computing the Euclidean distance of two vectors a and b in D-dimensional space is given by:

$$EuclideanDist(a,b) = \sqrt{\sum_{i=1}^{D} (a_i - b_i)^2}$$

The Manhattan distance on the other hand is the distance between two points if the only paths you can take are parallel to the axes. In two dimensions this is analogous to the distance traveled if you were to go by car through city blocks. This has the advantage that it weighs more heavily differences in each dimension (being closer in the x-dimension does not get you any closer in the y-dimension). The Manhattan distance is computed by summing the absolute value of the element-wise subtraction of two vectors. For vectors a and b in D-dimensional space, the distance can be computed as:

$$ManhattanDist(a,b) = \sum_{i=1}^{D} |a_i - b_i|$$

Because we have normalized all of the vectors, the Manhattan distance will always be a single number between 0 and 2 (because we normalize each BBV to sum to 1). This number can then be used to compare how closely related two intervals of execution are to one another. For the rest of this section we will be discussing distances in terms of Manhattan distance, because we found that it more accurately represented differences in our high-dimensional data. We present the Euclidean distance as it pertains to the clustering algorithms presented in Section 4, since it provides a more accurate representation for data with lower dimensions.

3.3 Basic Block Similarity Matrix

Now that we have a method of comparing intervals of program execution to one another, we can now concentrate on finding phase-based behavior. A phase of program behavior can be defined in several ways. Past definitions are built around the idea of a phase being a contiguous interval of exe-

Instruction Cache	8k 2-way set-associative, 32 byte blocks, 1 cycle latency
Data Cache	16k 4-way set-associative, 32 byte blocks, 2 cycle latency
Unified L2 Cache	1Meg 4-way set-associative, 32 byte blocks, 20 cycle latency
Memory	150 cycle round trip access
Branch Predictor	hybrid - 8-bit gshare w/ 8k 2-bit predictors + a 8k bimodal predictor
Out-of-Order Issue	out-of-order issue of up to 8 operations per cycle, 128 entry re-order buffer
Mechanism	load/store queue, loads may execute when all prior store addresses are known
Architecture Registers	32 integer, 32 floating point
Functional Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV
Virtual Memory	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

cution during which a measured program metric is relatively stable. We extend this notion of a phase to include all similar sections of execution regardless of temporal adjacency.

A key observation from this paper is that the phase behavior seen in any program metric is directly a function of the code being executed. Because of this we can use the comparison between the Basic Block Vectors as an approximate bound on how closely related any other metrics will be between those two intervals.

To find how intervals of execution relate to one another we create a *Basic Block Similarity Matrix*. The similarity matrix is an upper triangular $N \times N$ matrix, where N is the number of intervals in the program's execution. An entry at (x, y) in the matrix represents the Manhattan distance between the basic block vector at interval x and the basic block vector at interval y.

Figures 1(left and right) and 4(left) shows the similarity matrices for `gzip`, `bzip`, and `gcc` using the Manhattan distance. The diagonal of the matrix represents the program's execution over time from start to completion. The darker the points, the more similar the intervals are (the Manhattan distance is closer to 0), and the lighter they are the more different they are (the Manhattan distance is closer to 2).

The top left corner of each graph is the start of program execution and is the origin of the graph, $(0, 0)$, and the bottom right of the graph is the point $(N-1, N-1)$ where N is the number of intervals that the full program execution was divided up into. The way to interpret the graph is to start considering points along the diagonal axis drawn. Each point is perfectly similar to itself, so the points directly on the axis all are drawn dark. Starting from a given point on the diagonal axis of the graph, you can begin to compare how that point relates to it's neighbors forward and backward in execution by tracing horizontally or vertically. If you wish to compare a given interval x with the interval at $x + n$, you simply start at the point (x, x) on the graph and trace horizontally to the right until you reach $(x, x + n)$.

To examine the phase behavior of programs, let us first examine `gzip` because it has behavior on such a large scale that it is easy to see. If we examine an interval taken from 70 billion instructions into execution, in Figure 1 (left), this is directly in the middle of a large phase shown by the triangle block of dark color that surrounds this point. This means that this interval is very similar to it's neighbors both forward and backward in time. We can also see that the execution at 50 billion and 90 billion instructions is also very similar to the program behavior at 70 billion. We also note, while it may be hard to see in a printed version that the phase interval at 70 billion instructions is similar to the phases at interval 10 and 30 billion, but they are not as similar as to those around 50 and 90 billion. Compare this with the IPC and data cache miss rates for `gzip` shown in Figure 2. Overall, Figure 1(left) shows that the phase behavior seen in the similarity matrix lines up quite closely with the behavior of the program, with 5 large phases (the first 2 being different from the last 3) each divided by a small phase, where all of the small phases are very similar to each other.

The similarity matrix for `bzip` (shown on the right of Figure 1) is very interesting. `Bzip` has complicated behavior, with two large parts to it's execution, compression and decompression. This can readily be seen in the figure as the large dark triangular and square patches. The interesting thing about `bzip` is that even within each of these sections of execution there is complex behavior. This, as will be shown later, makes the behavior of `bzip` impossible to capture using a small contiguous section of execution.

A more complex case for finding phase behavior is `gcc`, which is shown on the left of Figure 4. This similarity matrix shows the results for `gcc` using the Manhattan distance. The similarity matrix on the right will be explained in more detail in Section 4.2.1. This figure shows that `gcc` does have some regular behavior. It shows that, even here, there is common code shared between sections of execution, such as the intervals around 13 billion and 36 billion. In fact the strong dark diagonal line cutting through the matrix indicates that there is good amount of repetition between offset segments of execution. By analyzing the graph we can see that interval x is very similar to interval $(x + 23.6B)$ for all x.

Figures 2 and 5 show the time varying behavior of `gzip` and `gcc`. The average IPC and data cache miss rate is shown for each 100 million interval of execution over the complete execution of the program. The time varying results graphically show the same phase behavior seen by looking at only the code executed. For example, the two phases for `gcc` at 13 billion and 36 billion, shown to be very similar in Figure 4, are shown to have the same IPC and data cache miss rate in Figure 5.

4. CLUSTERING

The basic block vectors provide a compact and representative summary of the program's behavior for intervals of execution. By examining the similarity between them, it is clear that there exists a high level pattern to each program's execution. In order to make use of this behavior we need to start by delineating a method of finding and representing the information. Because there are so many intervals of execution that are similar to one another, one efficient representation is to group the intervals together that have similar behavior. This problem is analogous to a *clustering* problem. Later, in Section 5, we demonstrate how we use the clusters we discover to find multiple simulation points for irregular programs or inputs like `gcc`. By simulating only a single representative from each cluster, we can accurately represent the whole program's execution.

4.1 Clustering Overview

The goal of clustering is to divide a set of points into groups

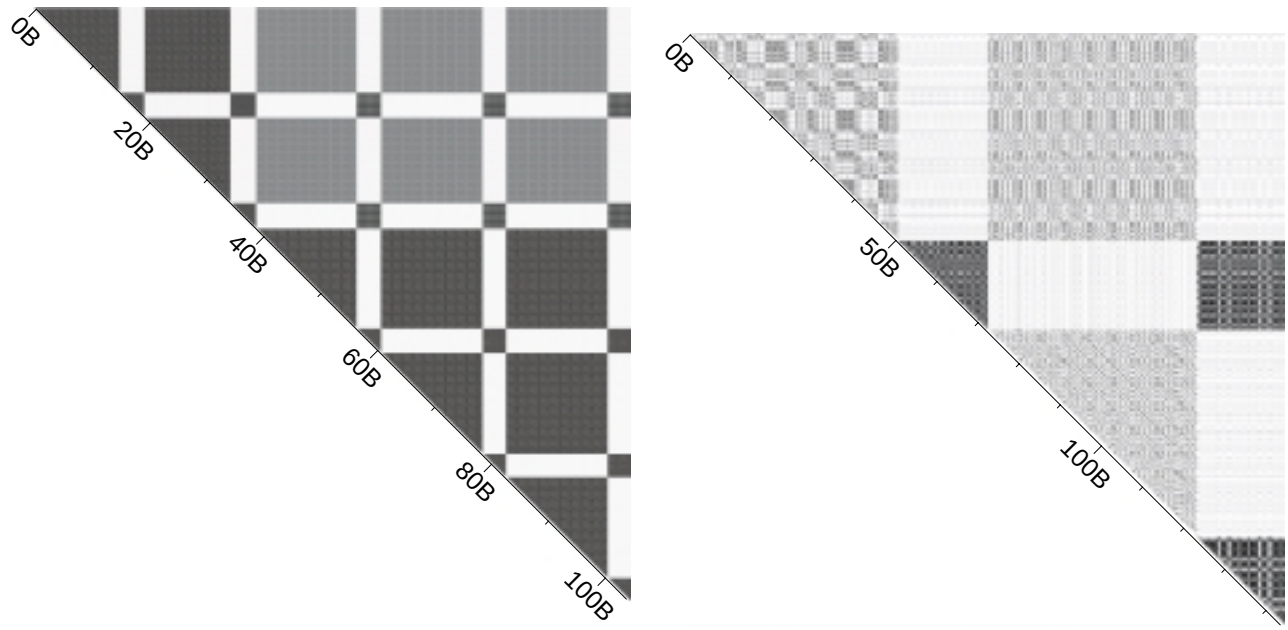

Figure 1: Basic block similarity matrix for the programs `gzip-graphic` (shown left) and `bzip-graphic` (shown right). The diagonal of the matrix represents the program's execution to completion with units in billions of instructions. The darker the points, the more similar the intervals are (the Manhattan distance is closer to 0), and the lighter the points the more different they are (the Manhattan distance is closer to 2).

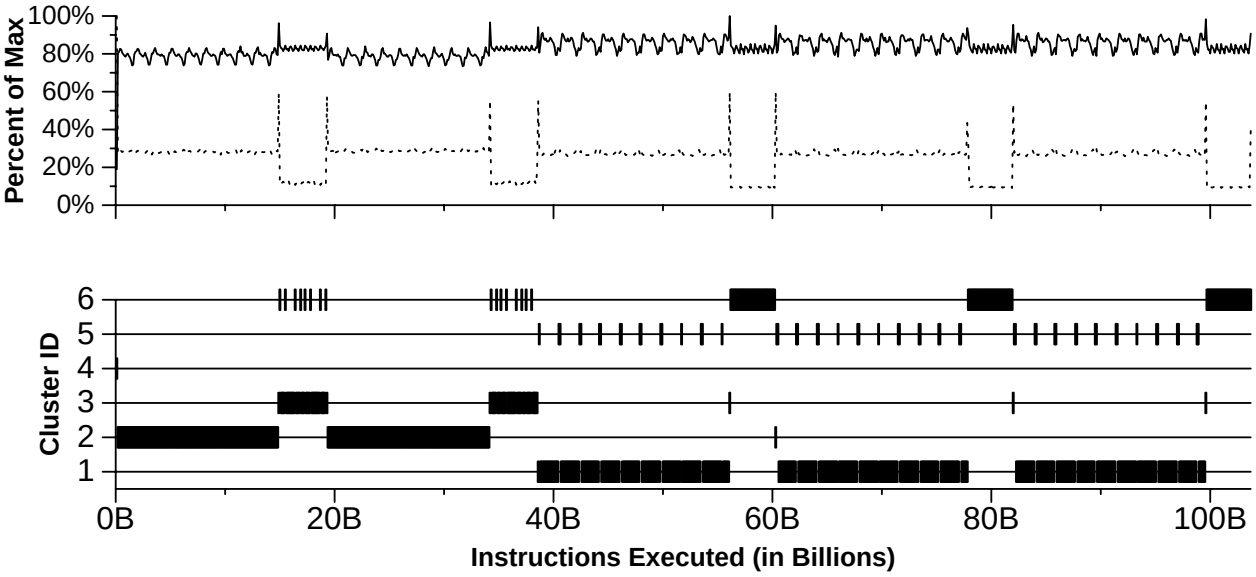

Figure 2: (top graph) Time varying graph for `gzip-graphic`. The average IPC (drawn with solid line) and L1 data cache miss rate (drawn with dotted line) are plotted for every interval (100 million instructions of execution) showing how these metrics vary over the program's execution. The x-axis represents the execution of the program over time, and the y-axis the percent of max value the metric had during execution. The results are non-accumulative.

Figure 3: (bottom graph) Cluster graph for `gzip-graphic`. The full run of the execution is partitioned into a set of 6 clusters. The x-axis is in instructions executed, and the graph shows for each interval of execution (every 100 million instructions), which cluster the interval was placed into.

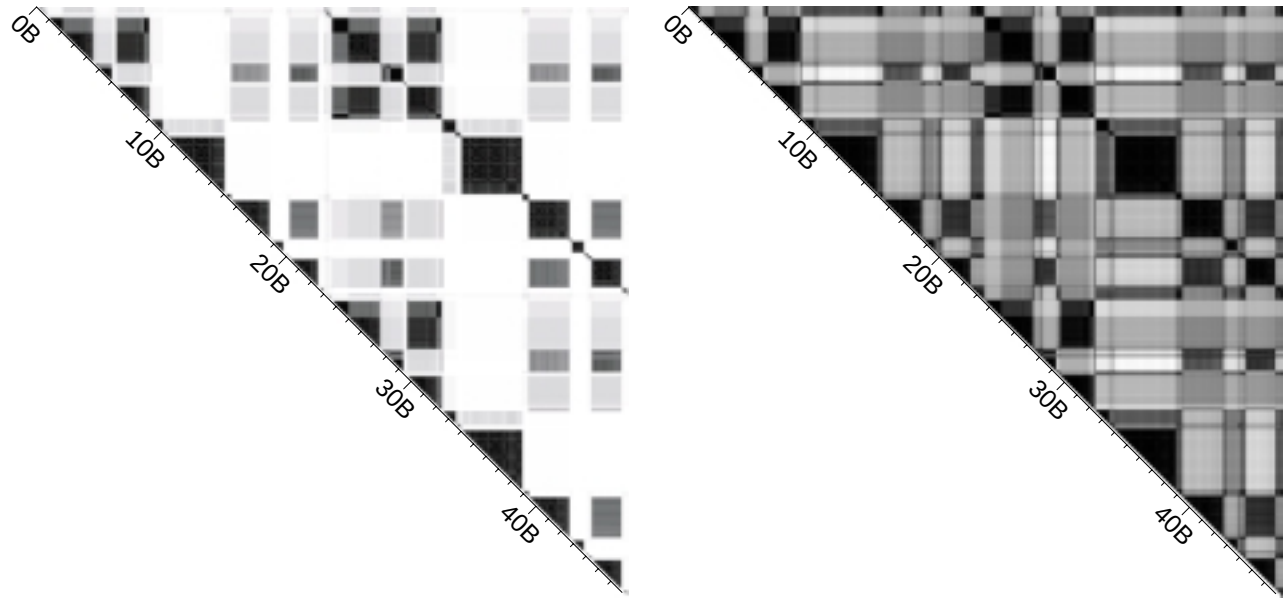

Figure 4: The original basic block similarity matrix for the program gcc (shown left), and the similarity matrix for gcc-166 drawn from projected data (on right). The figure on the left use the original basic block vectors (each of which has over 100,000 dimensions) and uses the Manhattan distance as a method of difference taking. The figure on the right uses projected data (down to 15 dimensions) and uses the Euclidean distance for difference taking.

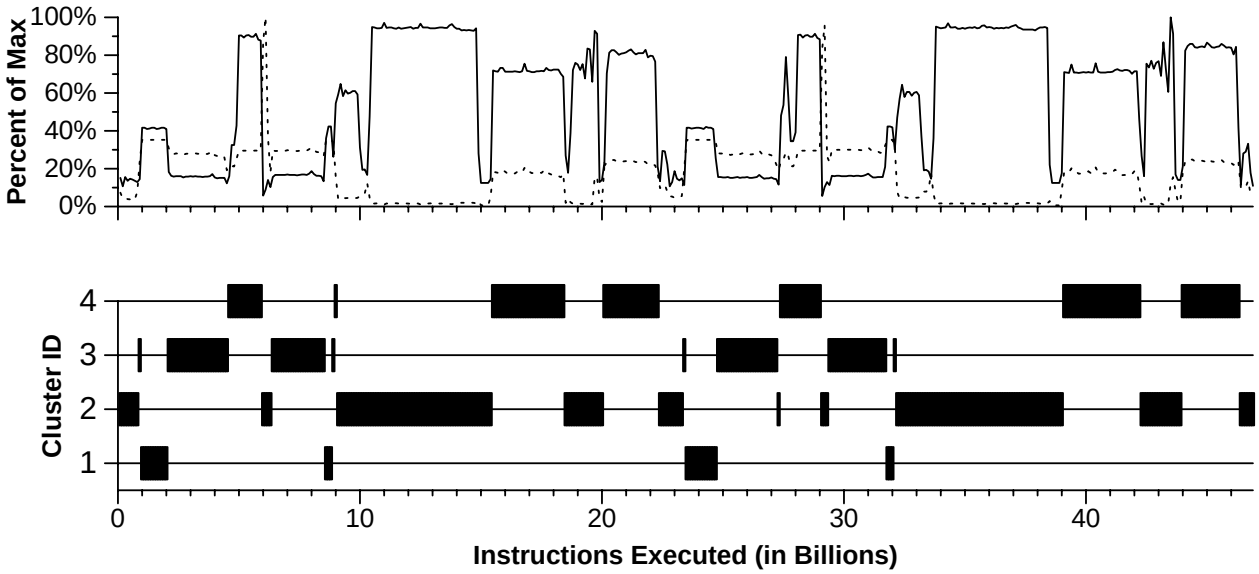

Figure 5: (top graph) Time varying graph for gcc-166. The average IPC (drawn with solid line) and L1 data cache miss rate (drawn with dotted line) are plotted for every interval (100 million instructions of execution) showing how these metrics vary over the program's execution. The x-axis represents the execution of the program over time, and the y-axis the percent of max value the metric had during execution. The results are non-accumulative.

Figure 6: (bottom graph) Cluster graph for gcc-166. The full run of the execution is partitioned into a set of 4 clusters. The x-axis is in instructions executed, and the graph shows for each interval of execution (every 100 million instructions), which cluster the interval was placed into.

such that points within each group are similar to one another (by some metric, often distance), and points in different groups are different from one another. This problem arises in other fields such as computer vision [10], document classification [22], and genomics [1], and as such it is an area of much active research. There are many clustering algorithms and many approaches to clustering. Classically, the two primary clustering approaches are Partitioning and Hierarchical:

Partitioning algorithms choose an initial solution and then use iterative updates to find a better solution. Popular algorithms such as k-means [14] and Gaussian Expectation-Maximization [2, pages 59–73] are in this family. These algorithms tend to have run time that is linear in the size of the dataset.

Hierarchical algorithms [9] either combine together similar points (called agglomerative clustering, and conceptually similar to Huffman encoding), or recursively divides the dataset into more groups (called divisive clustering). These algorithms tend to have run time that is quadratic in the size of the dataset.

4.2 Phase Finding Algorithm

For our algorithm, we use random linear projection followed by k-means. We choose to use the k-means clustering algorithm, since it is a very fast and simple algorithm that yields good results. To choose the value of k, we use the Bayesian Information Criterion (BIC) score [11, 17]. The following steps summarize our algorithm, and then several of the steps are explained in more detail:

1. Profile the basic blocks executed in each program to generate the basic block vectors for every 100 million instructions of execution.

2. Reduce the dimension of the BBV data to 15 dimensions using random linear projection.

3. Try the k-means clustering algorithm on the low-dimensional data for k values 1 to 10. Each run of k-means produces a clustering, which is a partition of the data into k different clusters.

4. For each clustering ($k = 1 \ldots 10$), score the fit of the clustering using the BIC. Choose the clustering with the smallest k, such that it's score is at least 90% as good as the best score.

4.2.1 *Random Projection*

For this clustering problem, we have to address the problem of dimensionality. All clustering algorithms suffer from the so-called "curse of dimensionality", which refers to the fact that it becomes extremely hard to cluster data as the number of dimensions increases. For the basic block vectors, the number of dimensions is the number of executed basic blocks in the program, which ranges from 2,756 to 102,038 for our experimental data, and could grow into the millions for very large programs. Another practical problem is that the running time of our clustering algorithm depends on the dimension of the data, making it slow if the dimension grows too large.

Two ways of reducing the dimension of data are dimension selection and dimension reduction. Dimension selection simply removes all but a small number of the dimensions of the data, based on a measure of goodness of each dimension for describing the data. However, this throws away a lot of data in the dimensions which are ignored. Dimension reduction reduces the number of dimensions by creating a new lower-dimensional space and then projecting each data point into the new space (where the new space's dimensions are not directly related to the old space's dimensions). This is analogous to taking a picture of 3 dimensional data at a random angle and projecting it onto a screen of 2 dimensions.

For this work we choose to use *random linear projection* [5] to create a new low-dimensional space into which we project the data. This is a simple and fast technique that is very effective at reducing the number of dimensions while retaining the properties of the data. There are two steps to reducing a dataset X (which is a matrix of basic block vectors and is of size $N_{intervals} \times D_{numbb}$, where D_{numbb} is the number of basic blocks in the program) down to D_{new} dimensions using random linear projection:

- Create a $D_{numbb} \times D_{new}$ projection matrix M by choosing a random value for each matrix entry between -1 and 1.

- Multiply X times M to obtain the new lower-dimensional dataset X' which will be of size $N_{intervals} \times D_{new}$.

For clustering programs, we found that using $D_{new} = 15$ dimensions is sufficient to still differentiate the different phases of execution. Figure 7 shows why we chose to project the data down to 15 dimensions. The graph shows the number of dimensions on the x-axis. The y-axis represents the k value found to be best on average, when the programs were projected down to the number of dimensions indicated by the x-axis. The best k is determined by the k with the highest BIC score, which is discussed in Section 4.2.3. The y-axis is shown as a percent of the maximum k seen for each program so that the curve can be examined independent of the actual number of clusters found for each program. The results show that for 15 dimensions the number of clusters found begins to stabilize and only climbs slightly. Similar results were also found using a different method of finding k in [6].

The advantages of using linear projections are twofold. First, creating new vectors with a low dimension of 15 is extremely fast and can even be done at simulation time. Secondly, using only 15 dimensions speeds up the k-means algorithm significantly, and reduces the memory requirements by several orders of magnitude over using the original basic block vectors.

Figure 4 shows the similarity matrix for `gcc` on the left using original BBVs, whereas the similarity matrix on the right shows the same matrix but on the data that has been projected down to 15 dimensions. For the reduced dimension data we use the Euclidean distance to measure differences, rather than the Manhattan distance used on the full data. After the projection, some information will be blurred, but overall the phases of execution that are very similar with full dimensions can still be seen to have a strong similarity with only 15 dimensions.

4.2.2 *K-means*

The k-means algorithm is an iterative optimization algorithm, which executes as two phases, which are repeated to convergence. The algorithm begins with a random assignment of k different centers, and begins its iterative process. The iterations are required because of the recursive nature of the algorithm; the cluster centers define the cluster membership for each data point, but the data point memberships define the cluster centers. Each point in the data belongs to, and can be considered a member of, a single cluster.

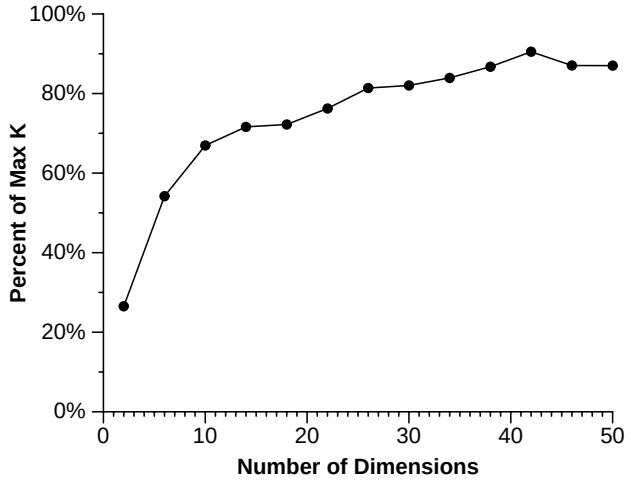

Figure 7: Motivation for random projection down to 15 dimensions (D=15). The x-axis is the number of dimensions of the projection, and the y-axis is the percent of the max number of clusters found for each program averaged over all spec programs. The results show that as you decrease the number of dimensions too far (the lowest point is two dimensions) the true clusters become collapsed on one another, and the algorithm cannot find as many clusters. By D=15 most of this effect has gone.

We initialize the k cluster centers by choosing k random points from the data to be clustered. After initialization, the k-means algorithm proceeds in two phases which are repeated until convergence:

- For each data point being clustered, compare its distance to each of the k cluster centers and assign it to (make it a member of) the cluster to which it is the closest.
- For each cluster center, change its position to the centroid of all of the points in its cluster (from the memberships just computed). The centroid is computed as the average of all the data points in the cluster.

This process is iterated until membership (and hence cluster centers) cease to change between iterations. At this point the algorithm terminates, and the output is a set of final cluster centers and a mapping of each point to the cluster that it belongs to. Since we have projected the data down to 15 dimensions, we can quickly generate the clusters for k-means with k from 1 to 10. In doing this, there are efficient algorithms for comparing the clusters that are formed for these different values of k, and choosing one that is good but still uses a small value for k is the next problem.

4.2.3 *Bayesian Information Criterion*

To compare and evaluate the different clusters formed for different k, we use the *Bayesian Information Criterion* (BIC) as a measure of the "goodness of fit" of a clustering to a dataset. More formally, the BIC is an approximation to the probability of the clustering given the data that has been clustered. Thus, the larger the BIC score, the higher the probability that the clustering being scored is a "good fit" to the data being clustered. We use the BIC formulation given in [17] for clustering with k-means, however other formulations of the BIC could also be used.

More formally, the BIC score is a penalized likelihood. There are two terms in the BIC: the likelihood and the penalty. The likelihood is a measure of how well the clustering models the data. To get the likelihood, each cluster is considered to be produced by a spherical Gaussian distribution, and the likelihood of the data in a cluster is the product of the probabilities of each point in the cluster given by the Gaussian. The likelihood for the whole dataset is just the product of the likelihoods for all clusters. However, the likelihood tends to increase without bound as more clusters are added. Therefore the second term is a penalty that offsets the likelihood growth based on the number of clusters. The BIC is formulated as

$$BIC(D,k) = l(D|k) - \frac{p_j}{2}\log(R)$$

where $l(D|k)$ is the likelihood, R is the number of points in the data, and p_j is the number of parameters to estimate, which is $(k-1)+dk+1$ for $(k-1)$ cluster probabilities, k cluster center estimates which each require d dimensions, and 1 variance estimate. To compute $l(D|k)$ we use

$$\begin{aligned} l(D|k) &= \sum_{i=1}^{k} -\frac{R_i}{2}\log(2\pi) - \frac{R_i d}{2}\log(\sigma^2) - \frac{R_i - 1}{2} \\ &\quad + R_i \log(R_i/R) \end{aligned}$$

where R_i is the number of points in the ith cluster, and σ^2 is the average variance of the Euclidean distance from each point to its cluster center.

For a given program and inputs, the BIC score is calculated for each k-means clustering, for k from 1 to N. We then choose the clustering that achieves a BIC score that is at least 90% of the spread between the largest and smallest BIC score that the algorithm has seen. Figure 8 shows the benefit of choosing a BIC with a high value and its relationship with the variance in IPC seen for that cluster. The y-axis shows the percent of IPC variance seen for a given clustering, and the corresponding BIC score the clustering received. Each point on the graph represents the average or max IPC variance for all points in the range of $\pm 5\%$ of the BIC score shown. The results show that picking clusterings that represent greater than 80% of the BIC score resulted in an IPC variance of less than 20% on average. The IPC variance was computed as the weighted sum of the IPC variance for each cluster, where the weight for a cluster is the number of points in that cluster. The IPC variance for each cluster is simply the variance of the IPC for all the points in that cluster.

4.3 Clusters and Phase Behavior

Figures 3 and 6 show the 6 clusters formed for `gzip` and the 4 clusters formed for `gcc`. The X-axis corresponds to the execution of the program in billions of instructions, and each interval (each of 100 million instructions) is tagged to be in one of the N clusters (labeled on the Y-axis). These figures, just as for Figures 1 and 4, show the execution of the programs to completion.

For `gzip`, the full run of the execution is partitioned into a set of 6 clusters. Looking to Figure 1(left) for comparison, we see that the cluster behavior captured by our tool lines up quite closely with the behavior of the program. The majority of the points are contained by clusters 1,2,3 and 6. Clusters 1 and 2 represent the large sections of execution which are similar to one another. Clusters 3 and 6 capture the smaller phases which lie in between these large phases, while cluster 5 contains a small subset of the larger phases, and cluster 4 represents the initialization phase.

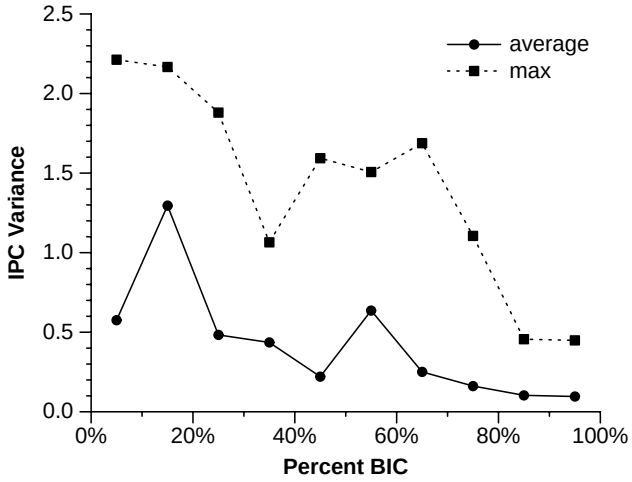

Figure 8: Plot of average IPC variance and max IPC variance versus the BIC. These results indicate that for our data, a clustering found to have a BIC score greater than 80% will have, on average, and IPC variance of less than 0.2.

In the cluster graph for `gcc`, shown in Figure 6, the run is now partitioned into 4 different clusters. Looking to Figure 4 for comparison, we see that even the more complicated behavior of `gcc` is captured correctly by our tool. Clusters 2 and 4 correspond to the dark boxes shown parallel to the diagonal axis. It should also be noted that the projection does introduce some degree of error into the clustering. For example, the first group of points in cluster 2 are not really that similar to the other points in the cluster. Comparing the two similarity matrices in Figure 4, shows the introduction of a dark band at (0,30) on the graph which was not in the original (un-projected) data. Despite these small errors, the clustering is still very good, and the impact of any such errors will be minimized in the next section.

5. FINDING SIMULATION POINTS

Modern computer architecture research relies heavily on cycle accurate simulation to help evaluate new architectural features. While the performance of processors continues to grow exponentially, the amount of complexity within a processor continues to grow at an even a faster rate. With each generation of processor more transistors are added, and more things are done in parallel on chip in a given cycle while at the same time cycle times continue to decrease. This growing gap between speed and complexity means that the time to simulate a constant amount of processor time is growing. It is already to the point that executing programs fully to completion in a detailed simulator is no longer feasible for architectural studies. Since detailed simulation takes a great deal of processing power, only a small subset of a whole program can be simulated.

SimpleScalar [3], one of the faster cycle-level simulators, can simulate around 400 million instructions per hour. Unfortunately many of the new SPEC 2000 programs execute for 300 billion instructions or more. At 400 million instructions per hour this will take approximately 1 month of CPU time.

Because it is only feasible to execute a small portion of the program, it is very important that the section simulated is an accurate representation of the program's behavior as a whole. The basic block vector and cluster analysis presented in Sections 3 and 4 will allow us to make sure that this is the case.

5.1 Single Simulation Points

In [19], we used basic block vectors to automatically find a single simulation point to potentially represent the complete execution of a program. A *Simulation Point* is a starting simulation place (in number of instructions executed from the start of execution) in a program's execution derived from our analysis. That algorithm creates a target basic block vector, which is a BBV that represents the complete execution of the program. The Manhattan distance between each interval BBV and the target BBV is computed. The BBV with the lowest Manhattan distance represents the single simulation point that executes the code closest to the complete execution of the program. This approach is used to calculate the long single simulation points (LongSP) described below.

In comparison, the single simulation point results in this paper are calculated by choosing the BBV that has the smallest Euclidean distance from the centroid of the whole dataset in the 15-dimensional space, a method which we find superior to the original method. The 15-dimensional centroid is formed by taking the average of each dimension over all intervals in the cluster.

Figure 9 shows the IPC estimated by executing only a single interval, all 100 million instructions long but chosen by different methods, for all SPEC 2000 programs. This is shown in comparison to the IPC found by executing the program to completion. The results are from SimpleScalar using the architecture model described in Section 2, and all fast forwarding is done so that all of the architecture structures are completely warmed up when starting simulation (no cold-start effect).

The first bar, labeled `none`, is the IPC found when executing only the first 100 million instructions from the start of execution (without any fast forwarding). The second bar, `FF-Billion` shows the results after blindly fast forwarding 1 billion instructions before starting simulation. The third bar, `SimPoint` shows the IPC using our single simulation point analysis described above, and the last bar shows the IPC of simulating the program to completion (labeled `Full`). Because these are actual IPC values, values which are closer to the `Full` bar are better.

The results in Figure 9 shows that the single simulation points are very close to the actual full execution of the program, especially when compared against the ad-hoc techniques. Starting simulation at the start of the program results in an average error of 210%, whereas blindly fast forwarding results in an average 80% IPC error. Using our single simulation point analysis we reduce the average IPC error to 18%. These results show that it is possible to reasonably capture the behavior of the most programs using a very small slice of execution.

Table 2 shows the actual simulation points chosen along with the program counter (PC) and procedure name corresponding to the start of the interval. If an input is not attached to the program name, then the default ref input was used. Columns 2 through 4 are in terms of the number of intervals (each 100 million instruction long). The first column is the number of instructions executed by the program, on the specific input, when run to completion. The second column shows the end of initialization phase calculated as described in [19]. The third column shows the single simulation point automatically chosen as described above. This simu-

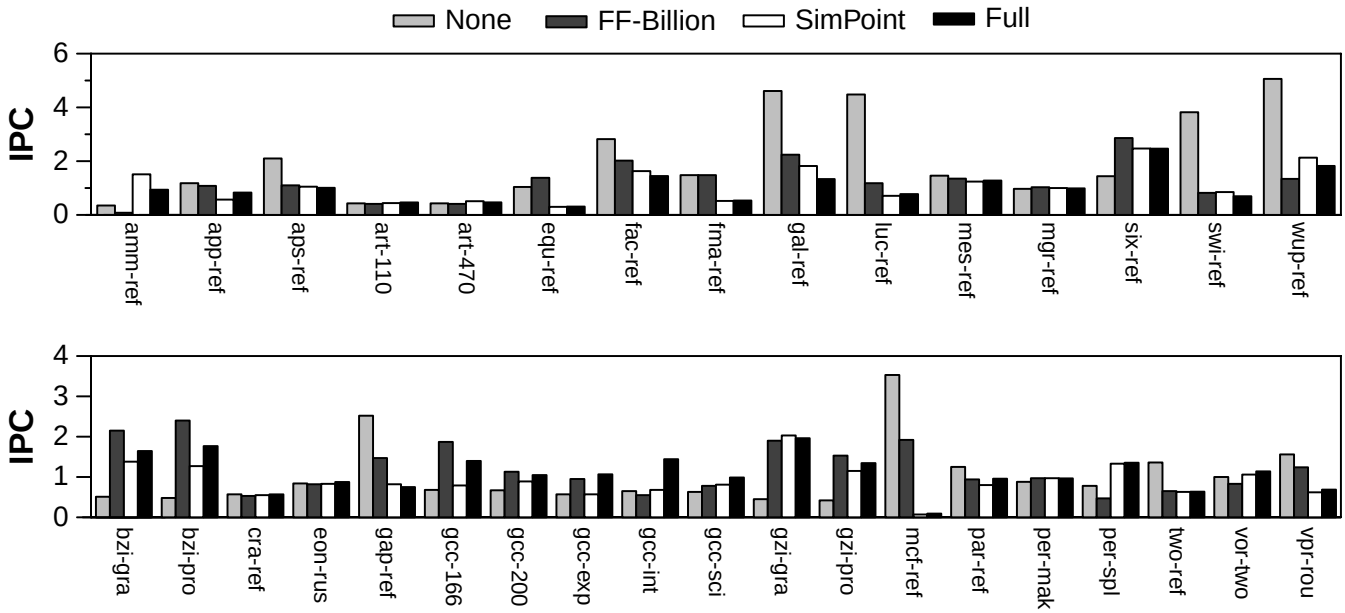

Figure 9: Simulation results starting simulation at the start of the program (none), blindly fast forwarding 1 billion instructions, using a single simulation point, and the IPC of the full execution of the program.

lation point is used to fast forward to the point of desired execution. Some simulators, debuggers, or tracing environments (e.g., gdb) provide the ability to fast forward based upon a program PC, and the number of times that PC was executed. We therefore, provide the instruction PC for the start of the simulation point, the procedure that PC occurred in, and the number of times that PC has to be executed in order to arrive at the desired simulation point.

These results show that a single simulation point can be accurate for many programs, but there is still a significant amount of error for programs like `bzip`, `gzip` and `gcc`. This occurs because there are many different phases of execution in these programs, and a single simulation point will not accurately represent all of the different phases. To address this, we used our clustering analysis to find multiple simulation points to accurately capture these programs behavior, which we describe next.

5.2 Multiple Simulation Points

To support multiple simulation points, the simulator can be run from start to stop, only performing detailed simulation on the selected intervals. Or the simulation can be broken down into N simulations, where N is the number of clusters found via analysis, and each simulation is run separately. This has the further benefit of breaking the simulation down into parallel components that can be distributed across many processors. This is the methodology we use in our simulator. For both cases results from the separate simulation points need to be weighed and combined to arrive at overall performance for the program [4]. Care must be taken to combine statistics correctly (simply averaging will give incorrect results for statistics such as rates).

Knowing the clustering alone is not sufficient to enable multiple point simulation because the cluster centers do not correspond to actual intervals of execution. Instead, we must first pick a representative for each cluster that will be used to approximate the behavior of the the full cluster. In order to pick this representative, we choose for each cluster the actual interval that is closest to the center (centroid) of the cluster. In addition to this, we weigh any use of this representative by the size of the cluster it is representing. If a cluster has only one point, it's representative will only have a small impact on the overall outcome of the program.

Table 2 shows the multiple simulation points found for all of the SPEC 2000 benchmarks. For these results we limited the number of clusters to be at most six for all but the most complex programs. This was done, in order to limit the number of simulation points, which also limits the amount of warmup time needed to perform the overall simulation. The cluster formation algorithm in Section 4 takes as an input parameter the max number of clusters to be allowed. Each simulation point contains two numbers. The first number is the location of the simulation point in 100s of millions of instructions. The second number in parentheses is the weight for that simulation point, which is used to create an overall combined metric. Each simulation point corresponds to 100 million instructions.

Figure 10 shows the IPC results for multiple simulation points. The first bar shows our single simulation points simulating for 100 million instructions. The second bar `LongSP` chooses a single simulation point, but the length of simulation is identical to the length used for multiple simulation points (which may go up to 1 billion instructions). This is to provide a fair comparison between the single simulation points and multiple. The `Multiple` bar shows results using the multiple simulation points, and the final bar is IPC for full simulation. As in Figure 9, the closer the bar is to `Full`, the better.

The results show that the average IPC error rate is reduced to 3% using multiple simulation points, which is down from 17% using the long single simulation point. This is significantly lower than the average 80% error seen for blindly fast forwarding. The benefits can be most clearly seen in the programs `bzip`, `gcc`, `ammp`, and `galgel`. The reason that the long contiguous simulation points do not do much better is that they are constrained to only sample at one place in the program. For many programs this is sufficient, but for those with interesting long term behavior, such as `bzip`, it is

name	Len	Init	SP	PC	Proc Name	Multiple SimPoints				
ammp	3265	24	109	026834	mm_fv_update.	3026(13.8) 1607(12.6)	1774(31) 2437(4.9)	595(15.3) 3112(11.5)	1068(1.3) 2480(2.2)	2128(7.4)
applu	2238	3	2180	018520	buts_	624(22.1) 1507(14.5)	1625(22.5)	1956(18.8)	2234(6.6)	1380(15.5)
apsi	3479	3	3409	0380ac	dctdxf_	2107(5.6)	2863(14)	1007(70.7)	896(7.7)	1618(2)
art-110	417	75	341	00fbb0	match	82(42.9)	255(41.2)	50(15.8)		
art-470	450	83	366	00f5d0	match	300(36.2)	46(14.7)	236(49.1)		
bzip2-graphic	1435	4	719	012a5c	spec_putc	168(11.7) 519(11.6)	1042(3.7) 872(8.2)	430(7.5) 195(5.6)	762(16.2) 148(2)	106(15.3) 1435(18.2)
bzip2-program	1249	4	459	00ddd0	sortIt	140(11) 1005(7)	468(12.3) 94(6.9)	78(6.2) 606(14)	990(16) 859(14.6)	445(7.4) 341(4.7)
bzip2-source	1088	4	978	00d774	qSort3	395(16) 177(34.7)	511(4.3)	64(29.1)	488(7.3)	530(8.6)
crafty	1918	462	775	021730	SwapXray	123(25)	510(19.7)	664(22.7)	1123(32.5)	
eon-rushmeier	578	140	404	04e1b4	viewingHit	260(6.6)	238(23.7)	337(20.9)	435(35.6)	216(13.1)
equake	1315	35	813	012410	phi0	874(12.2) 62(11.6)	1292(36.7)	463(12.2)	336(24.1)	3(3.2)
facerec	2682	356	376	02d1f4	graphroutines_lo.	1976(60.1)	1528(2.5)	1935(3.9)	1398(29.2)	348(4.3)
fma3d	2683	192	2542	0e3140	scatter_element.	112(7) 509(13)	209(0.6)	842(68.4)	1600(11)	47(0.1)
galgel	4093	3	2492	02db00	syshtn_	3511(5.5) 2181(29)	2081(11) 2161(3.3)	3466(11.2) 1017(5.5)	516(31.6)	2141(2.7)
gap	2695	639	675	050750	CollectGarb	1114(8.2)	1196(58.1)	88(12.7)	2189(14)	2609(7.1)
gcc-166	469	61	390	0d157c	gen_rtx	238(6.4)	149(42.2)	30(21.3)	404(30.1)	
gcc-200	1086	151	737	0ceb04	refers_to_regno.	8(45.8)	587(17.9)	921(10.9)	575(14.5)	1011(11)
gcc-expr	120	27	37	191fd0	validate_change	63(12.5) 88(5)	81(15.8)	42(16.7)	25(4.2)	9(45.8)
gcc-integrate	131	14	5	1198e0	find_single_use.	118(9.2) 73(17.6)	41(27.5)	102(21.4)	9(20.6)	57(3.8)
gcc-scilab	620	139	208	100d54	insert	255(54.2)	39(9.5)	231(13.2)	379(15.8)	170(7.3)
gzip-graphic	1037	158	654	009c00	fill_window	961(45.4) 566(13.4)	87(28.5)	373(7.3)	1(0.1)	461(5.2)
gzip-log	395	91	266	00d280	inflate_codes	207(24.1)	171(16.5)	157(16.7)	330(23.5)	71(19.2)
gzip-program	1688	112	1190	009660	longest_match	228(22.7)	779(21.4)	472(9.1)	1410(20.4)	594(26.4)
gzip-random	821	152	624	00a14c	deflate	484(0.9) 1(0.1)	625(0.2)	580(51)	811(16.8)	200(30.9)
gzip-source	843	68	335	00a224	deflate	248(14.5) 720(2.5)	327(13.2)	167(17.7)	656(27.8)	373(24.4)
lucas	1423	11	546	021ef0	fft_square_	982(21.4)	602(10.7)	1370(21.4)	458(28)	524(18.6)
mcf	618	15	554	00911c	price_out_impl k	268(39.6) 143(3.9)	425(11)	205(30.1)	468(4.5)	316(10.8)
mesa	2816	6	1136	0a30f0	general_textured.	1846(35.3)	2806(0.7)	398(35.3)	977(28.8)	
mgrid	4191	21	3293	0160f0	resid_	43(24.2)	3459(22.8)	807(20.1)	3110(16.3)	2476(16.6)
parser	5467	388	1147	01edfc	region_valid	3342(25.1)	1771(29.8)	5102(19.7)	2008(19.4)	4772(6)
perlbmk-diff	399	56	142	07f974	regmatch	6(1) 239(31.8)	355(62.7)	11(0.5)	397(0.8)	12(3.3)
perlbmk-make	20	3	12	08268c	Perl_runops_st.	1(5)	20(20)	6(75)		
perlbmk-perf	290	69	6	08268c	Perl_runops_st.	39(59.3)	207(40.7)			
perlbmk-split	1108	162	451	07fc98	regmatch	704(44.9)	596(9.1)	232(21.7)	461(21.8)	501(2.6)
sixtrack	4709	250	3044	167894	thin6d_	6(1.7)	1719(98.3)			
swim	2258	3	2080	019130	calc1_	1951(29.8)	38(14)	777(24.7)	710(13.8)	2101(17.8)
twolf	3464	7	1067	041094	ucxx1	312(17)	2888(11.3)	3268(11.7)	961(20.4)	2054(39.5)
vortex-one	1189	36	272	06289c	Mem_GetWord	536(17.1)	366(23.3)	115(8.2)	1068(17.2)	878(34.2)
vortex-three	1330	177	565	0336a8	Part_Delete	934(25.4) 485(25.4)	1129(11.4)	96(8.9)	47(11.1)	586(17.8)
vortex-two	1386	206	1025	05e6fc	Mem_NewRegion	635(7.6) 397(23.2)	752(24.5)	554(21.9)	930(7.4)	360(15.3)
vpr-place	1122	4	593	0224ec	get_non_update.	166(25.5) 547(27.9)	857(21.6)	1(0.2)	362(12.8)	1057(12)
vpr-route	840	12	477	025c80	get_heap_head	559(29.9)	89(28)	353(23.8)	3(2.6)	490(15.7)
wupwise	3496	11	3238	01d680	zgemm_	1811(43.3)	91(8)	3055(43.2)	1524(5.4)	

Table 2: Single simulation points for SPEC 2000 benchmarks. Columns 2 through 4 are in terms of 100 million instruction executed. The length of full execution is shown, as well as the end of initialization. SP is the single simulation point using the approach in this paper. The procedure in which the simulation point occurred and its PC are also shown. The last 6 digits of PC of each SimPoint is given in hex, so the address is formed from 120xxxxxx. Procedure names that end in "." were truncated due to space. The rest of the columns list the multiple simulation points found in 100s of millions. The first number is the starting place of the simulation point relative to the *start* of execution. The second number shows the weight given to the cluster that simulation point was taken from, and is used when weighing the final results of the simulation.

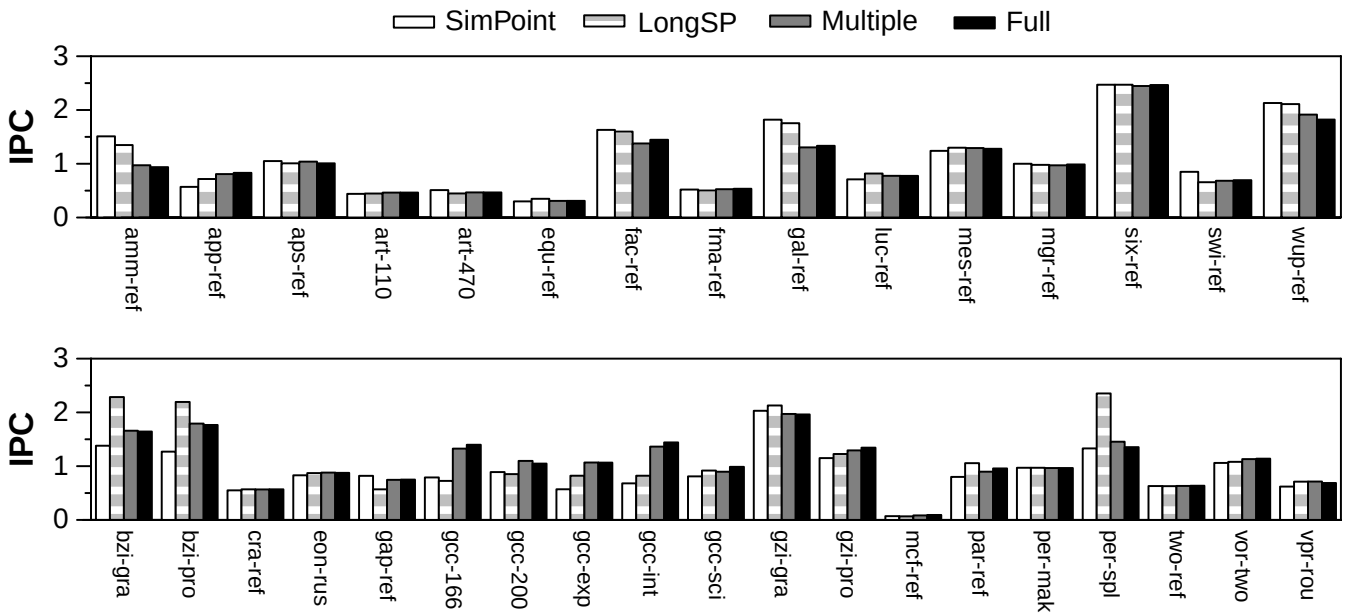

Figure 10: Multiple simulation point results. Simulation results are shown for using a single simulation point simulating for 100 million instructions, LongSP chooses a single simulation point simulating for the same length of execution as the multiple point simulation, simulation using multiple simulation points, and the full execution of the program.

impossible to approximate the full behavior.

Figure 11 is the average over all of the floating point programs (top graph) and integer programs (bottom graph). Errors for IPC, branch miss rate, instruction and data cache miss rates, and the unified L2 cache miss rate for the architecture presented in Section 2 are shown. The errors are with respect to these metrics for the full length of simulation using SimpleScalar. Results are shown for starting simulation at the start of the program `None`, blindly fast forwarding a billion instructions `FF-Billion`, single simulation points of duration 1 (`SimPoint`) and k (`LongSP`), and multiple simulation points (`Multiple`).

The first thing to note is that using the just a single small simulation point performs quite well on average across all of the metrics when compared to blindly fast-forwarding. Even though a single SimPoint does well, it is clearly beaten by using the clustering based scheme presented in this paper across all of the metrics examined. One thing that stands out on the graphs is that the error rate of the instruction cache and L2 cache appear to be high (especially for the integer programs) despite the fact that our technique is doing quite well in terms of overall performance. This is due to the fact that we present here an arithmetic mean of the errors, and there are several programs that have high error rates due to the very small number of cache misses. If there are 10 misses in the whole program, and we estimate there to be 100, that will result in a error of 10X. We point to the overall IPC as the most important metric for evaluation as it implicitly weighs each of the metrics by it's relative importance.

6. RELATED WORK

Time Varying Behavior of Programs: In [18], we provided a first attempt at showing the periodic patterns for all of the SPEC 95 programs, and how these vary over time for cache behavior, branch prediction, value prediction, address prediction, IPC and RUU occupancy.

Training Inputs and Finding Smaller Representative Inputs: One approach for reducing the simulation time is to use the training or test inputs from the SPEC benchmark suite. For many of the benchmarks, these inputs are either (1) still too long to fully simulate, or (2) too short and place too much emphasis on the startup and shutdown parts of the program's execution, or (3) inaccurately estimate behavior such as cache accesses do to decreased working set size.

KleinOsowski et. al [12], have developed a technique where they manually reduce the input sets of programs. The input sets were developed using a range of approaches from truncating of the input files to modification of source code to reduce the number of times frequent loops were traversed. For these input sets they develop, they make sure that they have similar results in terms of IPC, cache, and instruction mix.

Fast Forwarding and Check-pointing: Historically researchers have simulated from the start of the application, but this usually does not represent the majority of the program's behavior because it is still in the initialization phase. Recently researchers have started to *fast-forward* to a given point in execution, and then start their simulation from there, ideally skipping over the initialization code to an area of code representative of the whole. During fast-forward the simulator simply needs to act as a functional simulator, and may take full advantage of optimizations like direct execution. After the fast-forward point has been reached, the simulator switches to full cycle level simulation.

After fast-forwarding, the architecture state to be simulated is still cold, and a warmup time is needed in order to start collecting representative results. Efficiently warming up execution only requires references immediately proceeding the start of simulation. Haskins and Skadron [7] examined probabilistically determining the minimum set of fast-forward transactions that must be executed for warm up to accurately produce state as it would have appeared had the entire fast-forward interval been used for warm up [7]. They

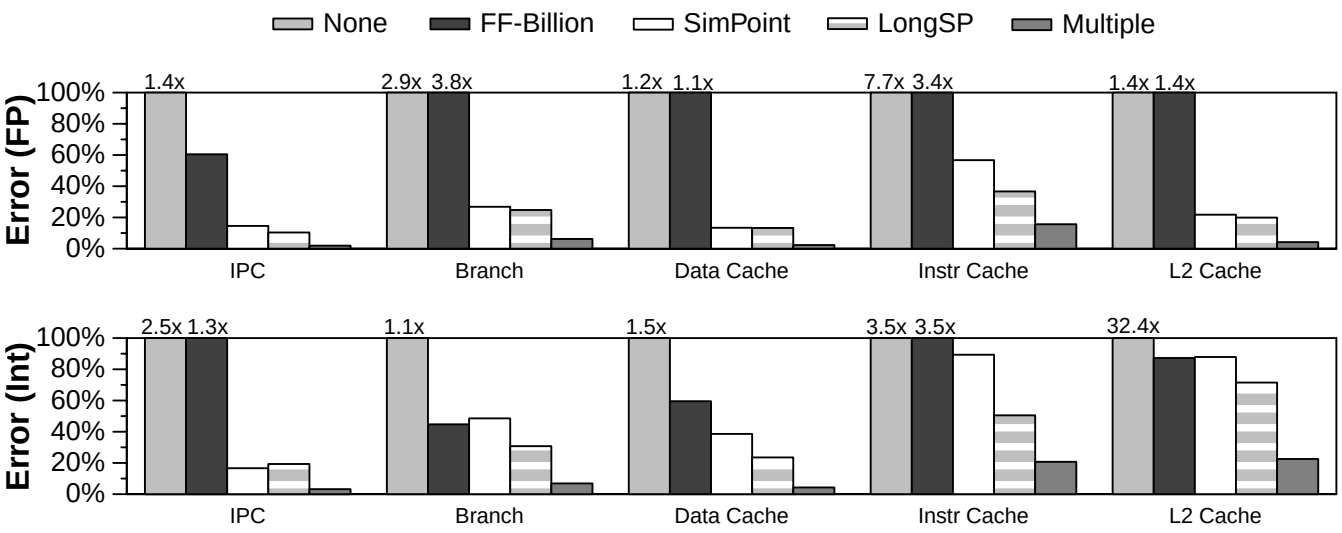

Figure 11: Average error results for the SPEC 2000 floating point (top) and integer (bottom) benchmarks for IPC, branch misprediction, instruction, data and unified L2 cache miss rates.

recently examined using reuse analysis to determine how far before full simulation warmup needs to occur [8].

An alternative to fast forwarding is to use check-pointing to start the simulation of a program at a specific point. With check-pointing, code is executed to a given point in the program and the state is saved, or checkpointed, so that other simulation runs can start there. In this way the initialization section can be run just one time, and there is no need to fast forward past it each time. The architectural state (e.g., caches, register file, branch prediction, etc) can either be stored in the trace (if they are not going to change across simulation runs) or can be warmed up in a manner similar to described above.

Automatically Finding Where to Simulate: Our work is based upon the basic block distribution analysis in [19] as described in prior sections. Recent work on finding simulation points for data cache simulations is presented by Lafage and Seznec [13]. They proposed a technique to gather statistics over the complete execution of the program and use them to choose a representative slice of the program. They evaluate two metrics, one which captures memory spatial locality and one which captures memory temporal locality. They further propose to create specialized metrics such as instruction mix, control transfer, instruction characterization, and distribution of data dependency distances to further quantify the behavior of the both the program's full execution and the execution of samples.

Statistical Sampling: Several different techniques have been proposed for sampling to estimate the behavior of the program as a whole. These techniques take a number of contiguous execution samples, referred to as clusters in [4], across the whole execution of the program. These clusters are spread out throughout the execution of the program in an attempt to provide a representative section of the application being simulated. Conte et. al [4] formed multiple simulation points by randomly picking intervals of execution, and then examining how these fit to the overall execution of the program for several architecture metrics (IPC and branch and data cache statistics). Our work is complementary to this, where we provide a fast and metric independent approach for picking multiple simulation points based just on basic block vector similarity. When an architect gets a new binary to examine they can use our approach to quickly find the simulation points, and then validate these with detailed simulation in parallel with using the binary.

Statistical Simulation: Another technique to improve simulation time is to use statistical simulation [16]. Using statistical simulation, the application is run once and a synthetic trace is generated that attempts to capture the whole program behavior. The trace captures such characteristics as basic block size, typical register dependencies and cache misses. This trace is then run for sometimes as little as 50-100,000 cycles on a much faster simulator. Nussbaum and Smith [15] also examined generating synthetic traces and using these for simulation and was proposed for fast design space exploration. We believe the techniques presented in this paper are complementary to the techniques of Oskin et al. and Nussbaum and Smith in that more accurate profiles can be determined using our techniques, and instead of attempting to characterize the program as a whole it can be characterized on a per-phase basis.

7. SUMMARY

At the heart of computer architecture and program optimization is the need for understanding program behavior. As we have shown, many programs have wildly different behavior on even the very largest of scales (over the full lifetime of the program). While these changes in behavior are drastic, they are not without order, even in very complex applications such as `gcc`. In order to help future compiler and architecture researchers in exploiting this large scale behavior, we have developed a set of analytical tools that are capable of automatically and efficiently analyzing program behavior over large sections of execution.

The development of the analysis is founded on a hardware independent metric, *Basic Block Vectors*, that can concisely summarize the behavior of an arbitrary section of execution in a program. We showed that by using Basic Block Vectors one can capture the behavior of programs as defined by several architectural metrics (such as IPC, and branch and cache miss rates).

Using this framework, we examine the large scale behavior of several complex programs like `gzip`, `bzip`, and `gcc`, and find interesting patterns in their execution over time. The

behavior that we find shows that code and program behavior repeat over time. For example, in the input we examined in detail for `gcc` we see that program behavior repeats itself every 23.6 billion instructions. Developing techniques that automatically capture behavior on this scale is useful for architectural, system level, and runtime optimizations. We present an algorithm based on the identification of clusters of basic block vectors that can find these repeating program behaviors and group them into sets for further analysis. For two of the programs `gzip` and `gcc` we show how the clustering algorithm results line up nicely with the similarity matrix and correlate with the time varying IPC and data cache miss rates.

It is increasingly common for computer architects and compiler designers to use a small section of a benchmark to represent the whole program during the design and evaluation of a system. This leads to the problem of finding sections of the program's execution that will accurately represent the behavior of the full program. We show how our clustering analysis can be used to automatically find multiple simulation points to reduce simulation time and to accurately model full program behavior. We call this clustering tool to find single and multiple simulation points *SimPoint*. SimPoint along with additional simulation point data can be found at: `http://www.cs.ucsd.edu/~calder/simpoint/`. For the SPEC 2000 programs, we found that starting simulation at the start of the program results in an average error of 210% when compared to the full simulation of the program, whereas blindly fast forwarding resulted in an average 80% IPC error. Using a single simulation point found, using our basic block vector analysis, resulted in an average 17% IPC error. When using the clustering algorithm to create multiple simulation points we saw an average IPC error of 3%.

Automatically identifying the phase behavior using clustering is beneficial for architecture, compiler, and operating system optimizations. To this end, we have used the notion of basic block vectors and a random projection to create an efficient technique for identifying phases on-the-fly [20], which can be efficiently implemented in hardware or software. Besides identifying phases, this approach can predict not only when a phase change is about to occur, but to which phase it is about to transition. We believe that using phase information can lead to new compiler optimizations with code tailored to different phases of execution, multi-threaded architecture scheduling, power management, and other resource distribution problems controlled by software, hardware or the operating system.

Acknowledgments

We would like to thank Suleyman Sair and Chris Weaver for their assistance with SimpleScalar, as well as Mark Oskin and the anonymous reviewers for providing helpful comments on this paper. This work was funded in part by DARPA/ITO under contract number DABT63-98-C-0045 and NSF CAREER grant No. CCR-9733278.

8. REFERENCES

[1] A. Ben-Dor, R. Shamir, and Z. Yakhini. Clustering gene expression patterns. *Journal of Computational Biology*, 6:281–297, 1999.

[2] C. M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

[3] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[4] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, October 1996.

[5] S. Dasgupta. Experiments with random projection. In *Uncertainty in Artificial Intelligence: Proceedings of the Sixteenth Conference (UAI-2000)*, pages 143–151, San Francisco, CA, 2000. Morgan Kaufmann Publishers.

[6] G. Hamerly and C. Elkan. Learning the k in k-means. Technical Report CS2002-0716, University of California, San Diego, 2002.

[7] J. Haskins and K. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *Proceedings of the 2001 International Conference on Computer Design*, September 2001.

[8] J. Haskins and K. Skadron. Memory reference reuse latency: Accelerating sampled microarchitecture simulations. Technical Report CS-2002-19, U of Virginia, July 2002.

[9] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[10] J.-M. Jolion, P. Meer, and S. Bataouche. Robust clustering with applications in computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(8):791–802, 1991.

[11] R. E. Kass and L. Wasserman. A reference Bayesian test for nested hypotheses and its relationship to the schwarz criterion. *Journal of the American Statistical Association*, 90(431):928–934, 1995.

[12] A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja. Adapting the spec 2000 benchmark suite for simulation-based computer architecture research. In *Proceedings of the International Conference on Computer Design*, September 2000.

[13] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Workload Characterization of Emerging Applications, Kluwer Academic Publishers*, September 2000.

[14] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. LeCam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, CA, 1967. University of California Press.

[15] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[16] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[17] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734. Morgan Kaufmann, San Francisco, CA, 2000.

[18] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[19] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[20] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. Technical Report CS2002-0710, UC San Diego, June 2002.

[21] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[22] O. Zamir and O. Etzioni. Web document clustering: A feasibility demonstration. In *Research and Development in Information Retrieval*, pages 46–54, 1998.