Workload Reduction for Multi-Input Profile-Directed Optimization

Abstract

Profile-directed optimization is an effective technique to improve program performance, but it may result in program performance and compiler behavior that is sensitive to both the selection of inputs used for training and the actual input in each run of the program. Cross-validation over a workload of inputs can address the input-sensitivity problem, but introduces the need to select a representative workload of minimal size from the population of available inputs. We present a compiler-centric clustering methodology to group similar inputs so that redundant inputs can be eliminated from the training workload. Input similarity is determined based on the compile-time code transformations made by the compiler after training separately on each input. Differences between inputs are weighted by a performance metric based on cross-validation in order to account for code transformation differences that have little impact on performance. We introduce the $CrossError$ metric that allows the exploration of correlations between transformations based on the results of clustering. The methodology is applied to several SPEC [14] benchmark programs, and illustrated using selected case studies.

1. Introduction

Profile-directed optimization, or profile directed feedback (PDF), is an effective compilation technique to improve program performance[1] in terms of execution speed. For static (or offline) PDF, an instrumented version of the program is created by the compiler to measure program characteristics such as the execution frequencies of basic blocks and functions, function-pointer targets, dynamic memory allocation amounts, or divisor values. These values are recorded in a profile file, which is used by the compiler to inform code transformations during a subsequent optimizing compilation to create the final program executable. Dynamic PDF performs similar data-collection and code optimization operations during program execution, facilitated by just-in-time compilation. This work considers static PDF where training and execution occur in different program invocations.

Compiler vendors routinely assist clients in using the compiler to improve the performance of the client's programs. In cases involving the most important customers, members of the compiler development team may be involved in this process, and may implement improvements in the compiler to better optimize the client's code. PDF is one option for improving program performance. However, as Berube and Amaral reported, when PDF is used, compiler behavior and program performance can be sensitive to both the training input(s) used to generate the profile and the input(s) used to measure the final performance [2].

In order to reliably measure the performance of the program (and consequently the benefits of PDF

[1] Throughout this paper the term *performance* will refer to the execution speed of a program on a workload of inputs.

and of code transformations based on PDF), a cross-validation strategy should be used over the workload of inputs. For important applications, the client can supply in-house correctness or performance testing inputs, along with end-user input, that cover a range of use-cases. However, consider the case where a client provides hundreds, or thousands, of inputs to the compiler team. Using such a large set of inputs is computationally prohibitive, and is not suitable for repeated performance evaluations as the compiler evolves. Each additional input used for training, and/or evaluation, increases the time required to evaluate performance due to additional training runs, program compilations and execution time measurements. Also, if the compiler supports the simultaneous use of multiple training inputs, the client cannot be expected to use a very large set of inputs for training in their build process.

Therefore, a minimal set of representative inputs is required to reduce the time consumed by performance evaluation, without compromising evaluation quality. The method used to reduce the workload should not rely on human intuition or the experience of experts. For large, complicated programs, predicting the interactions between the program, the compiler, data inputs, and the underlying computer architecture is likely impossible, even for an expert of all the involved components. Thus, an automatic workload-reduction technique is useful for both compiler designers, who may not be experts regarding the client's program or it's inputs, as well as for the client, who may not be an expert regarding the compiler. Furthermore, the method should measure how representative the selected inputs are of the full workload, and thus provide a quantitative estimate of the trade-off between workload size and workload accuracy.

This work presents a compiler-centric methodology to reduce the size of the workload needed for proper evaluation of the performance improvements achieved by an optimizing compiler for a given application. Input similarity is based on the code-transformation decisions made by the compiler according to the profile generated for each input. Inputs are clustered based on the variations they induce in code transformations. This clustering produces groups of inputs to which the compiler responds in a similar fashion, and thus identifies redundancy in the training, and testing, workloads.

Furthermore, we present a novel metric to compare different clusterings on related data. This metric allows for intuitive investigation of the correlation between individual transformations.

The goal of this technique is not the immediate maximization of program performance, and certainly never to maximize program performance on a particular input. Rather, the goal is to reduce a large workload to a representative set of inputs, enabling time-efficient improvement of the compiler by the development team, and ultimately improving program performance on the full workload of inputs.

In the next section, we present related work on workload reduction and identify why existing techniques are not appropriate in the context of cross-validation for PDF compilation. §3 presents the details

of our metrics and methodology, while experimental design is discussed in §4. We present results and analysis in §5 and conclude in §6.

2. Related Work

Input characterization and workload reduction are not new problems. Computer architecture research is largely simulation-based, necessitating small workloads of representative programs using minimally-sized inputs. The architectural metrics of benchmark programs are repeatedly scrutinized for redundancy, while smaller inputs are compared with large inputs. Alternatively, some work bypasses program behavior and examines the inputs directly.

Shen and Mao propose the XICL language to allow programmers to formally describe how to extract the important properties of an input directly [11]. A feature selection process removes correlated features, and basic-block frequency counts are predicted using an input-behavior model constructed by regression. However, the programmer must understand the code, the input, and the compiler, in order to hypothesize important features, and then determine a procedure to automatically extract those features from an arbitral input. Furthermore, the system only predicts basic-block execution frequencies. While critical to many current transformations, these frequencies do not inform any value specialization transformations.

Maxiaguine *et al.* examine the variability of input streams for the system-level design of multimedia system-on-chip devices [9]. They reduce the input set to corner cases and the best-case and worst-case scenarios, the critical concerns of real-time data processing systems. However, input characterization that does not take the run-time characteristics of the program into account does not provide much useful information in the context of PDF.

Gove and Spracklen test how well the SPEC2006 training inputs represent the reference workloads [6]. They find that in almost every case, based only on the correspondence of function execution frequencies and branch behaviors, the training workload is highly representative of the reference workload.

Most input characterization and workload reduction research aims to reduce the time required for detailed architectural simulation, without compromising the applicability of simulation results. Similarity metrics in this area are based on architecture-level program characteristics (instructions per branch, cache miss rates, *etc.*). Most techniques use clustering to choose a representative subset of the full workload.

KleinOsiwski and Lilja created the MinneSPEC benchmark suite from the SPEC2000 suite by reducing the sizes of the input data [8]. The reference inputs were truncated, sampled, replaced with the test or train input, or run with a modified command line. However, the authors warn that these reduced inputs do not always conserve all the program characteristics of the original inputs, and should be used with caution. Furthermore, while program execution time on each input is of some concern in the context of PDF, the computation required for cross-validation scales with the number of inputs. Consequently, reducing the size of

the workload is of greater importance.

Vandierendonck *et al.* cluster SPEC2000 benchmarks based on performance numbers reported on the SPEC web-site [15]. Rank analysis validates the clustering results, but simply predicting the relative performance of machines does not provide enough information to determine the reasons for these performance differences, nor to illuminate opportunities to improve performance in the future.

A prevailing methodology to select representative program-input pairs uses Principal Component Analysis (PCA) to reduce the dimensionality of program-characteristic data, and then clusters the data in the resulting space. Eeckhout *et al.* employ this methodology with the SPEC95 and TCP benchmarks [5]. They find that while there is significant redundancy between the program-input pairs, the behavior of some programs is significantly impacted by the choice of input. Phansalkar *et al.* find that the SPEC2006 benchmark suite [10] is more varied than earlier versions of the suite, though some redundancy still exists. Later, Eeckhout *et al.* find that Independent Component Analysis produces more accurate clusters and smaller representative sets than PCA for the SPEC2000 and MinneSPEC benchmark suites [4].

Alternately, Hoste *et al.* use a correlation reduction technique with a genetic algorithm on michroarchitecturally-independent program characteristics [7]. They find that this technique provides superior results than PCA and clustering for emerging benchmarks, while the results are more easily interpreted because the dimensions of the similarity space are the measured characteristics. In particular, two apparently similar programs according to micro-architecturally dependent characteristics may be significantly different, as many program behaviors can produce similar performance counter values. While this observation is to be expected, it's implications are paramount to our study. Input similarity is dependent on the level where similarity is measured, which must match it's intended use. Consequently, it is essential to this study that input similarity be determined at the compiler level.

Sherwood *et al.* propose SimPoint, a tool to that identifies representative program phases that can be sampled to make predictions about a full simulation [12]. Phases are detected using Basic-Block Vectors (BBV) containing basic block execution counts for 100 million instruction intervals. Random projection reduces the dimensionality of the BBVs. The Manhattan or Euclidean distance is the similarity metric for K-means clustering. However, sampling does not reduce the number of inputs in a workload.

The work presented here differs from previous work in several ways. In the preceding works, dimensionality-reducing techniques, such as PCA, are applied before the similarity metric is calculated. The primary motivation for reducing dimensionality is to minimize the correlations between dimensions, and thus reduce bias toward redundant characteristics in the clustering. In this work, there are relatively few measured characteristics (transformations), and deter-

mining if they are correlated is one goal of the study. Thus, correlations between transformations are investigated directly.

In most studies, only the (small number of) inputs provided with the benchmark are considered: The emphasis is reducing the simulation time for the benchmark suite, and thus minimizing the number of program-input pairs that must be simulated. The use of program-input pairs not withstanding, the focus of workload reduction for architectural simulation is to select the *programs* that are representative of the suite. In this work, the focus is on finding the representative inputs from a workload *for a particular program*.

Finally, this work takes a uniquely compiler-oriented perspective. The primary point of interest is how training inputs interact with a compiler; how different training inputs result in different code transformations by a profile-directed compiler. While variations in code transformations may change architecture-level program characteristics, these metrics may be too far removed from the compiler to quickly assist designers at improving the compiler.

3. Clustering Methodology

Consider a program p with a workload of inputs $\mathcal{W} = \{n_1, n_2, \ldots, n_m\}$, and an optimizing compiler with a set $\mathcal{T} = \{T_1, T_2, \ldots, T_s\}$ of profile-directed transformations. Each input in $\mathcal{W}$ is used to generate a profile of an execution of p. When the compiler uses the profile generated by input n_i it will potentially either apply transformations from $\mathcal{T}$ to different locations of p, or apply such transformations with different frequencies, than when the compiler uses a profile generated by input $n_j \neq n_i$. The goal of this paper is to cluster inputs in $\mathcal{W}$ based on how similarly the compiler applies the transformations in $\mathcal{T}$ to p when using each of the profiles generated by each input. The calculated similarities are held in a similarity matrix, which is the input for the clustering algorithm.

3.1 Input Features and Similarity

Let $\mathcal{L}_{T_i}$ be the set of program locations where transformation T_i may be applied to p. A transformation vector V_{T_i,n_j} records how many times T_i is applied at each location in $\mathcal{L}_{T_i}$ when the compiler is guided by the profile generated by training on input n_j. V_{T_i,n_j} has $|\mathcal{L}_{T_i}|$ elements. $\mathcal{L}_{T_i}$ is the union of all locations where T_i is applied when the profile from each input is used, as well as the locations from a baseline compilation that does not use PDF information.

Each investigated transformation T_i is instrumented in the compiler to collect *(location, frequency)* pairs that indicate each time T_i is applied. Each location $l_c \in \mathcal{L}_{T_i}$ is identified, as uniquely as possible, using source-code line numbers and expression identifiers. Most transformations are all-or-nothing transformations: With the profile generated using input n_i, a 1 is recorded in position l_c of V_{T_i,n_j} if T_i is applied at l_c; a 0 is recorded if T_i is not applied at l_c. For loop unrolling, the unroll factor is recorded as the frequency in the transformation vector. Due to code replication, a transformation may be applied multiple times in indistinguishable locations. In these cases, the frequency values from aliased locations are accumulated at the

appropriate index of V_{T_i,n_j}.

Manhattan distance is used as the similarity metric between transformation vectors. The Manhattan distance between two vectors is the sum of the absolute value of their index-wise differences. Geometrically, the Manhattan distance is the length of the shortest path from two points if travel is restricted to movement along a unit grid. Thus, in the context of transformation vectors, the Manhattan distance counts the number of differences between two vectors.

The difference matrix D_{T_i} for transformation T_i is an $m \times m$, symmetric, matrix that encodes the pairwise Manhattan distances between the m V_{T_i,n_j} vectors:

$$D_{T_i}[x, y] = Manhattan(V_{T_i,n_x}, V_{T_i,n_y})$$

In order to account for all transformations during clustering, a combined difference matrix includes the data from each $T_i \in \mathcal{T}$. Each D_{T_i} is normalized by dividing it's element by the dimension of $\mathcal{L}_{T_i}$. Subsequently, the combined difference matrix D is created by point-wise summing the normalized matrices:

$$D = \sum_{T_i \in \mathcal{T}} \frac{D_{T_i}}{|\mathcal{L}_{T_i}|}$$

Normalization allows each transformation to have equal weight in D, even if the number of transformation sites differs by orders of magnitude between transformations. Thus, a single transformation will not dominate the combined difference score.

3.2 Performance Weighting

Compilers make a large number of transformation decisions while compiling a program. The interactions between these decisions is complex and can result in unexpected performance results. A single decision may have a large impact on performance, while many others may not make any measurable difference. In order to take program performance into account, and to attempt to filter out the inconsequential differences in transformation decisions, the elements of the difference matrix are weighted by a pair-wise performance factor. A running example will illustrate how the weights are determined. Table 1 presents a small set of actual execution times. The input names are omitted for clarity since this example has no relation to the results in §5. Each table row gives execution times on the listed input. Each column indicates the training input. Baseline indicates that PDF is not used.

	Training Input				
Data	**A**	**B**	**C**	**D**	**baseline**
A	-	59.07	62.08	58.74	61.76
B	71.29	-	74.15	70.09	73.35
C	4.34	4.14	-	4.14	4.29
D	110.14	108.65	115.17	-	116.89

Table 1: Running-times (in seconds)

	A	**B**	**C**	**D**
A	0	86.55	91.11	90.05
B	86.55	0	9.38	0.06
C	91.11	9.38	0	9.14
D	90.05	0.06	9.14	0

Table 2: Difference matrix D

Table 2 presents D for the example inputs, using data from the seven transformation discussed later in §4. Training on A produces significantly different transformation decisions than the other inputs, while training on B or D results in nearly the same decisions.

The baseline program is run on each input of $\mathcal{W}$ to provide reference time measurements. Similarly, each of the PDF-optimized programs are run on $\mathcal{W}$,

	Training Input				
Data	**A**	**B**	**C**	**D**	$log(t_{bl,v})$
A	-	0.96	1.01	0.95	4.12
B	0.97	-	1.01	0.96	4.30
C	1.01	0.97	-	0.97	1.46
D	0.94	0.93	0.99	-	4.76
LNP	0.96	0.94	1.00	0.96	

Table 3: Normalized run-time ($t_{u,v}/t_{bl,v}$), log-weights ($\log(t_{bl,v})$), and LNP values

excluding the training input for that program. A log-weighted normalized workload running time is calculated for each copy of the program. Given two inputs $u, v \in \mathcal{W}$ with $u \neq v$, let $t_{u,v}$ be the average running time of the program trained on input u executing using input v. $t_{bl,v}$ denotes the baseline running time for input v. For example, the value of $t_{B,D}$ in Table 1 is 108.65s. A log-weighted normalized performance (LNP) vector summarizes the workload performance for each training input u in $\mathcal{W}$:

$$LNP[u] = \frac{\sum_{v \in \mathcal{W}, v \neq u} \left(\frac{t_{u,v}}{t_{bl,v}} \times \log(t_{bl,v}) \right)}{\sum_{v \in \mathcal{W}, v \neq u} \log(t_{bl,v})}$$

Workload performance should be a weighted average, such that the weight assigned to relative performance compared to the baseline on any individual input is in relation to the execution time of that input. Furthermore, the performance measure should follow a cross-validation methodology, as discussed by Berube and Amaral [3]. Thus, $LNP[u]$ excludes values where the training and evaluation input are the same. In the situation of acquiring many inputs from a client, there is no control over the running-times of the inputs. Consequently, the execution times for various inputs may vary significantly. A long-running input should not unduly influence the workload performance metric. Log-weighting addresses this issue, while normalizing by the weights ensures comparability between different LNP values. Table 3 shows each execution time from Table 1 relative to the baseline ($t_{u,v}/t_{bl,v}$), along with $log(t_{bl,v})$, to illustrate the logarithm's effect on the weights. C still has a smaller weight than the other inputs, but the other three inputs are assigned similar weights, even though processing input D takes nearly twice as long as input A. The final LNP values are listed in the last row of Table 3.

	A	**B**	**C**	**D**
A	0	19.82	37.21	9.31
B	19.82	0	57.77	10.41
C	37.21	57.77	0	46.87
D	9.31	10.41	46.87	0

Table 4: Performance Weight PW (x1000)

	A	**B**	**C**	**D**
A	0	1.716	3.391	0.838
B	1.716	0	0.542	0.001
C	3.391	0.542	0	0.428
D	0.838	0.001	0.428	0

Table 5: Weighted difference matrix $\overline{D}$

The definition of LNP presumes that all runtimes will be longer than 1 second. With shorter times, adding 1 to each time keeps the logarithm positive.

LNP is used to calculate the performance weight matrix PW, which is used to weigh the transformation-vector differences between inputs in a difference matrix. Weighing these differences based on performance helps to identify when the differences in transformation decisions impact program performance, and filters out cases where different decisions

	A	**B**	**C**	**D**
A	3.39	1.86	0.00	2.55
B	1.68	3.39	2.85	3.39
C	0.00	2.85	3.39	2.96
D	2.55	3.39	2.96	3.39

Table 6: Similarity matrix $\overline{S}$ (Table 5)

have little practical effect. Clustering requires a symmetric matrix, thus PW must also be symmetric.

$$PW[x,y] = \frac{max(LNP[x], LNP[y])}{min(LNP[x], LNP[y])} - 1$$

As the difference between the LNP scores for a pair of programs reduces to 0, so does the weight assigned to their difference scores. Table 4 shows the PW matrix for the example. The largest PW values correspond to C, as expected. Unlike training on C, training on the other inputs results in a performance improvement. Therefore, some portion of the transformation decisions the compiler made differently when using C's profile compared to the other profiles have a significant (negative) impact on performance. Similarly, the differences between the other inputs are less important, but still impact performance. Complex interactions between transformation decisions make inferring the performance impact of any individual transformation difficult. Therefore, when individual transformations are investigated, D_{T_i} is not weighted by PW.

The combined difference matrix D is pointwise-weighted by PW to generate $\overline{D}$. For $0 < x \leq m$ and $0 < y \leq m$:

$$\overline{D}[x,y] = D[x,y] \times PW[x,y]$$

$\overline{D}$ for the example is shown in Table 5. Consider the columns for input A in Table 2 and Table 5. When the large, but fairly similar, differences in D are weighed by PW to create $\overline{D}$, A remains similar to D, but become different than C. This change indicates that while both D and C had a similar number of transformation differences when compared to A, the differences between A and D have less impact on performance.

Thus far, input similarity data has been presented in a difference matrix. However, the clustering technique requires a similarity matrix as input. As implied by its name, a similarity matrix measures input similarity rather than difference. Any difference matrix D can be converted to a similarity matrix S by subtracting each element of D from the maximum element in the D. For $0 \leq x < m$ and $0 \leq y < m$:

$$\begin{aligned} M &= \max(D[x,y]) \\ S[x,y] &= M - D[x,y] \end{aligned}$$

$\overline{S}$ denotes the similarity matrix for $\overline{D}$. $\overline{S}$ for the example is shown in Table 6. The lowest values in the similarity matrix indicate that the strongest combination of transformation differences and performance differences occur between A and C. In a manual study this result would indicate that the decisions made by the compiler using PDF from inputs A and C warrant closer examination. However, a complete analysis should consider the similarity between every pair of inputs, which is what clustering provides.

3.3 Clustering

The goal of clustering is to group inputs to which the compiler responds similarly. By comparing each V_{T_i,n_k}, $\overline{D}$ leverages the expertise and experience built into the compiler to indirectly identify which aspects of the profiles are important. Therefore, clustering $\overline{S}$ will group inputs with similar important runtime behaviors together, as determined by the compiler.

For clarity and simplicity, further discussion will use a graph interpretation of matrices. Each data input is a vertex in a complete, undirected graph. The similarity matrix S_{T_a} for a transformation T_a contains the edge weights for the graph. Clustering can be interpreted as breaking the graph into several disconnected maximal cliques by removing edges. The goal is to maximize the sum of the weights on the remaining edges. Unfortunately, the true maximum is not known, making the choice of an upper bound against which to judge a particular clustering problematic.

However, minimizing an error metric measuring the dissimilarity in the graph after partitioning is simpler and algorithmically convenient — zero is a known, constant, lower bound on the minimum value of a sum of edge weights. The error is calculated by applying the partitioning of the similarity matrix to the corresponding difference matrix.

A clustering $C_{T_a,k} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k\}$ is a set of k disjoint vertex partitions that covers the similarity matrix S_{T_a}. Each partition $\mathcal{P}_i\{v_1, v_2, \ldots, v_n\}$ is a set of vertices from S_{T_a}. Given $C_{T_a,k}$ and the difference matrix D_{T_a} used to compute S_{T_a}, the clustering error $Mismatch(C_{T_a,k}, D_{T_a})$ is defined as the sum of the edge weights in all clusters of $C_{T_a,k}$:

$$Mismatch(C_{T_a,k}, D_{T_a}) = \sum_{\mathcal{P} \in C_{T_a,k}} \left(\sum_{v_i, v_j \in \mathcal{P}} D_{T_a}[v_i, v_j] \right)$$

$Mismatch$ is applicable to clusterings based on any difference matrix: D, $\overline{D}$, and D_{T_a} are all valid parameters, provided that the clustering used S, $\overline{S}$, or S_{T_a} respectively. For the combined matrices D or $\overline{D}$, the clustering will be denoted C or $\overline{C}$, respectively.

3.4 ϵ-Greedy Spectral Clustering

A modified version of recursive 2-way spectral clustering [13] is employed. Spectral clustering is convenient because it relies entirely on the similarity matrix, and does not use the raw vectors to recompute the similarity matrix as the partitioning progresses. The standard formulation of spectral clustering presents two methods to obtain the partitioning: recursive greedy 2-way splitting, or a direct method using multiple eigenvectors of the similarity matrix. The direct method is preferable in most cases because it is computationally efficient and takes the desired number of clusters as an input. However, to produce k clusters, the similarity matrix must have k distinct eigenvalues. The 2-way partitioning must compute the eigenvectors for each subpartition instead of just the eigenvectors of the full similarity matrix. Furthermore, the 2-way partitioning relies on a data-sensitive stopping criteria rather than having direct control over the number of clusters.

Unfortunately, the direct method is unsuitable for this study because an $m \times m$ similarity matrix frequently does not have m distinct eigenvalues. Therefore, we modify the 2-way partitioning method to parameterize the number of clusters generated, and mitigate the sub-optimality of a purely greedy algorithm.

As in the original formulation, cuts are selected in a best-first order. Each cut splits one of the current partitions in two. To select the best cut, a local similarity matrix is created using the rows/columns of to the nodes in the partition. The elements of the partition

Algorithm 1: ϵ-Greedy Spectral Clustering

```
1  C = randomPartitioning(S);
2  for i ← 1 to N do
3      Partition.length = 1;
4      Partition[0] = W;
5      while Partition.length < k do
6          maxCutValue = 0;
7          for i ← 0 to Partition.length do
8              if rand() < ε1
9                  cut[i] = SpectralCut(Partition[i]);
10             else
11                 cut[i] = RandomCut(Partition[i]);
12             endif
13             if cut[i].NcutValue > maxCutValue
14                 maxCutValue = cut[i].NcutValue;
15                 maxCut = i;
16             endif
17         end
18         if rand() < ε2
19             p = Random(Partition.length);
20         else
21             p = maxCut;
22         endif
23         [partA,partB] = Partition[p].applyCut(cut[p]);
24         Partition[p] = partA;
25         Partition.add(partB);
26     end
27     if Mismatch(Partition, D) < Mismatch(C, D)
28         C = Partition;
29     endif
30 end
31 return C;
```

are ordered by their projection onto the 2^{nd} eigenvector of the local similarity matrix. The partition is cut at each point along the ordered list of partition elements, and a cut value is determined. The cut corresponding to the smallest cut value is selected.

Cut values are calculated using Ncut. Ncut solves a relaxed version of the NP-Complete minimum cut problem. Therefore, cuts that are selected based on the Ncut value are not guaranteed to minimize $Mismatch$. Moreover, greedy algorithms can result in very sub-optimal solutions unless specific conditions are met. For graph partitioning, a greedy algorithm has no optimality guarantees. In particular, when purely greedy 2-way partitioning is used, the $Mismatch$ does not always monotonically decrease as the number of clusters increases.

Therefore, we employ the classic search technique of injecting a random component to the greedy partitioning, and then iterate the search. In this case, simulated annealing is not appropriate because the solution space is not smooth and has many local minima. We use a fixed number of iterations with a constant probability of making a random choice in each iteration. As shown in Algorithm 1, the clustering result C is initialized randomly (line 1). A fixed, pre-determined number of iterations, N, are used to search for the best clustering (line 2). Partition is a vector, thus *Partition.length* is the number of partitions.

Each iteration of partitioning proceeds as follows: Initially, Partition has a single partition containing every vertex of S (line 4). Two-way partitioning is then iterated to produce k partitions. In order to select which of the $Partition.length < k$ current partitions to split, and how to split it, a cut is proposed for each partition (lines 7-17), and the Ncut value recorded (lines 9, 11). However, a random cut will be proposed with probability ϵ_1, and a greedy cut calculated by spectral clustering with probability $(1 - \epsilon_1)$ (line 8). The partition with the lowest Ncut value is selected to be cut with probability ϵ_2, while a random partition is cut with probability $(1 - \epsilon_2)$ (line 18). Applying the selected cut (from line 9 or 11) creates two new parti-

tions (line 23), which replace the split partition in the Partition vector (lines 24,25). Partitioning continues until there are k partitions. At this point, the Partition object is one possible k-clustering of S.

$Mismatch$(Partition,D) is calculated at the end of each partitioning. If $Mismatch$ for Partition is less than the previous best clustering, the clustering result is updated (line 27). After N the iterations of partitioning are complete, the best clustering, C, is reported.

3.5 $CrossError$: Comparing Clusterings

When both the compiler and the application are evolving, how often should the input clustering be recomputed? The more significant the gradual changes due to development work are on the clustering, the more frequently the clustering process should be repeated to ensure that the clustering remains relevant. Furthermore, if a transformation T_i generates clustering results that are representative of several transformations, then generating transformation vectors only for T_i can reduce compilation time and save time and effort invested by developers into analyzing and interpreting clustering data. To address both of these issues, the clustering methodology must provide a measure of clustering similarity based on different data-sets.

The clustering similarity measure cannot simply compare error curves or cluster members. Comparing error curves is only useful for different clustering methods using the same data. Different data sets are not equally difficult to cluster — a good k-clustering of one data set may also be a good k-clustering of another data set, even if the $Mismatch$s are different. Comparing the members of the resulting clusters may be more informative, but does not indicate the importance of the observed differences. Several near-optimal clusterings may exist, with cluster composition differing only for (possibly many) insignificant elements. We propose a novel metric, $CrossError$, to allow quantitative clustering comparison.

$CrossError$ requires that each matrix contain the same amount of potential error, so that all $Mismatch$ measurements share the same range. Transformation vector dimensions depend on the transformation, the compiler, and the program, leading to different ranges of possible Manhattan distances. The original matrices result in error values without context, which are both incomparable and difficult to interpret. Therefore, each similarity matrix is normalized by dividing it by the sum of it's elements:

$$\widetilde{S}[x,y] = \frac{S[x,y]}{sum(S)}$$

Normalization makes each edge weight proportional to the total weight in the graph. Consequently, $Mismatch$ and $CrossError$ values are a percent of the total possible error, a more intuitive metric that enables comparisons between error values. Clustering uses relative edge weights and is not influenced by this uniform scaling. Henceforth, normalization always precedes clustering, but we omit the $\widetilde{S}$ symbol to streamline the notation.

The $CrossError$ metric quantitatively measures the differences between two k-clusterings of the same workload, using different edge weights. If the $CrossError$ is low, one similarity matrix, and con-

sequently the clustering based on that matrix, may be representative of the other.

Given code transformations T_a and T_b, with difference matrices D_{T_a} and D_{T_b}, and their clusterings $C_{T_a,k}$ and $C_{T_b,k}$, the $CrossError$ metric is computed:

$$CrossError(C_{T_a,k}/C_{T_b,k}, D_{T_b}) = Mismatch(C_{T_a,k}, D_{T_b}) - Mismatch(C_{T_b,k}, D_{T_b})$$

$CrossError$ evaluates $C_{T_a,k}$ using $C_{T_b,k}$ as a baseline. If $C_{T_b,k}$ is an optimal clustering of $\overline{S}_{T_b}$, then $Mismatch(C_{T_b,k}, D_{T_b})$ is the minimum error for D_{T_b} and $Mismatch(C_{T_a,k}, D_{T_b}) \geq Mismatch(C_{T_b,k}, D_{T_b})$. Assuming effective clustering, $C_{T_b,k}$ estimates the minimum error for D_{T_b}, which is a property of D_{T_b} and k only that is invariant with respect to the clustering. Thus, $CrossError$ measures (or estimates, for sub-optimal clustering) the *extra* error incurred by using the alternate clustering $C_{T_a,k}$.

Consider $Mismatch$ over the range of k. For small k, clustering separates the greatest differences in D. As k increases, less significant differences are separated until each partition contains identical elements. Therefore, regardless of k, if $C_{T_a,k}$ is a good clustering of S_{T_b}, $CrossError(C_{T_a,k}/C_{T_b,k}, D_{T_b})$ will be low, implying that T_a provides input similarity data that is representative of the input similarity data provided by T_b. The curve for $CrossError(C_{T_a,k}/C_{T_b,k}, D_{T_b})$ over the range of k provides a quantitative measure of the strength of the representativeness relationship.

$CrossError$ is not symmetric. As such, if T_a is judged to be representative of T_b, the reciprocal relationship is not implied. For example, transformation T_b might have little impact on the program, and consequently expose few differences between inputs.

Alternately, $CrossError$ can be used to compare clusterings for similar programs.[2] In this case, the transformation T_a and workload $\mathcal{W}$ are held fixed, and the programs p and q are compared:

$$CrossError(C^p_{T_a,k}/C^q_{T_a,k}, D^q_{T_a}) = Mismatch(C^p_{T_a,k}, D^q_{T_a}) - Mismatch(C^q_{T_a,k}, D^q_{T_a})$$

4. Experimental Design

We investigate workload clustering for a range of programs. Clustering is performed for both individual transformations and combined multi-transformation data. These clusterings are analyzed to identify the impact of performance filtering, the correlations between transformations and the significance of the input-processing source code on input similarity.

All programs are compiled using a development snapshot of the IBM XL 8.0 compiler that is instrumented to output transformation vectors. Performance evaluation is performed using a dedicated machine running AIX on a POWER4 processor. Five runs are used for each program on each input, and the average of these runs is used as t_{n_i} when calculating LNP_n.

All experiments are performed using `gzip`, `bzip2`, `VPR`, `crafty`, `MCF`, and `GAP` from the SPEC CPU2000 suite, as well as `gcc` from the SPEC

[2] A practical use for such comparison is to evaluate if two versions of the same program produce similar clustering of inputs.

CPU2006 suite [14]. VPR performs two digital design tasks, logic placement and circuit routing, which take different input files and exercise different portions of the code. Therefore, VPR is used for these two tasks separately and identified as vpr.place and vpr.route.

Each program uses a workload of inputs. The provided inputs from both CPU2000 and CPU2006 are used when possible (MCF, bzip2, and gcc). Furthermore, the CPU2000 program workloads are augmented with inputs made available by Berube and Amaral [1]. The gcc workload is augmented with source code from the CPU2000 benchmark programs gzip, mesa, parser, and twolf. Gzip and bzip2 use a common workload that is the union of their available inputs.

4.1 Transformations

During the training process, transformation vectors are collected for the transformations that are the primary consumers of profile information. Inlining and loop transformations use profile information to order the transformation opportunities by expected profitability (*i.e.*, hottest first). The following transformations are instrumented:

Early Inlining Inlining at the beginning of the optimization phase focused on removing calls to small functions to enable subsequent transformations.

Late Inlining Inlining after other high-level transformations such as loop nest optimizations and function-pointer specialization, focused on removing function call overhead.

Loop Unrolling The loop unrolling factor is based on the number of memory streams identified in the loop, the number of hardware-supported memory streams, and the number of instructions in the unrolled loop. Ideally, unrolling should activate all the hardware memory stream prefetching units without overflowing the instruction cache.

Loop Unroll-and-Jam Loop unrolling, loop peeling and loop fusion for the inner loops in loop nests, to create perfectly-nested loops.

Specialization transformations use value profiling to replicate code segments, replacing the use of variables with constants. A test ensures that the run-time variable value matches the specialized value:

Memory Allocation Specialization Memory allocation library calls with variable memory block sizes are specialized with a constant memory size. These specialized allocations use a pooled memory allocator that increases the spatial locality of memory accesses and reduces the overhead of memory allocation and deallocation.

Function-Pointer Specialization Indirect function calls are converted to direct function calls. Removing indirection enables other transformations, *e.g.*, inlining. Furthermore, function call overhead is reduced on architectures where a branch and direct call is less expensive than an indirect call.

Value Specialization Integer division and modulus operations with a variable denominator are replaced with constant-denominator versions for frequently observed denominator values. Constant denominators allow for the generation of more ef-

ficient code using various architecture-dependent instruction-level transformations.

4.2 Clustering Comparison

ϵ-Greedy clustering is performed using a variety of similarity matrices:

- Each transformation from §4.1, individually, without performance-weighting. The clusterings for a transformation T_a is referred to as C_{T_a}.
- The combined, performance-weighted similarity matrix $\overline{S}$, as discussed in §3.2. The clustering generated from $\overline{S}$ is referred to as $\overline{C}$.
- The combined, but not performance-weighted, similarity matrix S produced directly from D. These clusterings are referred to as C.

Clusterings are compared in several ways. These comparisons are not intended to support strong claims about particular transformations, benchmark programs, or data inputs. Rather, they serve as single-point case studies that illustrate the types of questions that cluster comparison can help answer. Three variables that influence clustering results are investigated:

Performance Performance-weighting using PW serves as a filter to remove the impact of transformation decisions that are distinct but yet have little impact on program running time. Comparing the clustering with and without performance weighting indicates the impact of this filtering.

Transformations Clustering based on one transformation may predict the clustering based on another, particularly if the transformations are closely related. If the clustering for T_b can be predicted by the clustering for T_a, then T_b could be omitted from the analysis.

Algorithms `Bzip2` and `gzip` share an identical workload, but are very different algorithms. Comparing the clusterings for the two programs suggests the degree to which input similarity can be considered independently of the code processing the input. This information may be significant when using a reduced workload for evaluation in the case of rapid program development with frequent and significant code changes.

5. Results

Reliable clustering depends on setting the clustering algorithm parameters to appropriate values. Once the ϵ-greedy clusterer has been tuned[3], the workload for each benchmark program is clustered for each transformation listed in §4.1, along with the combined matrices D and $\overline{D}$, for each possible number of clusters.

5.1 Impact of Performance Weighting

Section 3.2 justified the use of performance weighting as a means of taking program performance into account when clustering inputs. Compiler decisions that have a larger impact on the program's performance should be given more weight when clustering inputs than decisions that have little effect on performance. But does performance weighting change the resulting input clusters? Comparing the $CrossError$ between

[3] In this study, $\epsilon_1 = \epsilon_2 = 0.5, N = 1000$

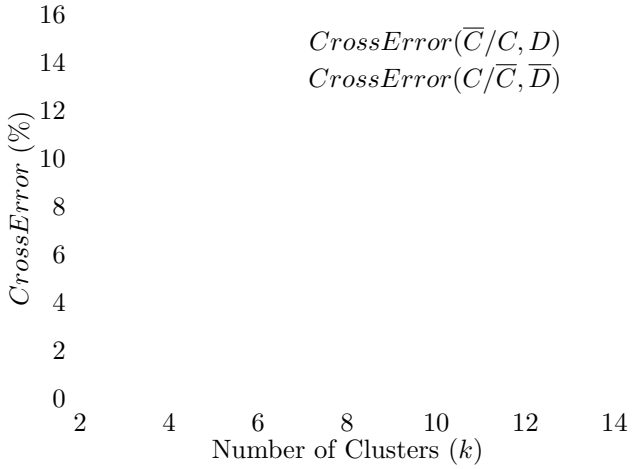

Figure 1: Comparison of $CrossError$ using weighted (D) and unweighed ($\overline{D}$) difference matrices.

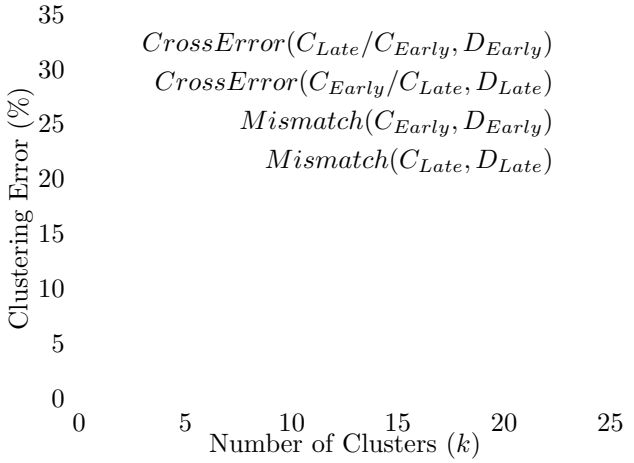

Figure 2: Clustering error using the early-inlining and late-inlining clusterings from VPR routing

unweighted and weighted clusters should answer this question. The results in Figure 1 are typical of this comparison: D and $\overline{D}$ are, respectively, the weighted and unweighted combined difference matrices for the seven transformations described in §4.1 for `gcc`, and C and $\overline{C}$ are their corresponding clusterings.

$CrossError(\overline{C}/C, D)$ evaluates the weighted clustering with the unweighted difference matrix. As illustrated by the evaluation of `gcc` in Figure 1, an unweighted clustering C evaluated with a weighted difference matrix $\overline{D}$ generally results in less $CrossError$ than $\overline{C}$ evaluated with D. This result indicates that D is more representative of $\overline{D}$ than vice-versa. Thus, in this case, performance-weighting filters out performance-irrelevant differences in the data that would otherwise influence the clustering. The additional error under the $CrossError$ curves illustrates that $\overline{D}$ is not a scaled version of D; in that case, the clustering results would be equivalent, and both $CrossError$ curves would be near 0. Instead, the performance-weighting has changed which pairs of inputs are most similar or most different from each other, thus changing the clustering in a meaningful way.

5.2 Impact of Different Transformations

Section 4.2 discussed the possibility that several code transformations in the compiler could yield similar clusterings, and thus some of these transformations could be eliminated from future input clusterings. This section illustrates how to evaluate the similarity of two transformations in relation to the clustering of inputs.

We generated $CrossError$ graphs for each possible pairing of clusterings from the 7 transformations listed in §4.1 plus the combined matrices D and $\overline{D}$, for each of the 7 benchmark programs. Careful examination of these 252 graphs suggests that transformations are generally not representative of others. Furthermore, the combined matrices tend not to be good representatives of any individual transformation, nor is any single transformation representative of either combined case. The greatest correlation exists between the two inlining transformations, but even in this case, the correlation is usually not very strong.

The $Mismatch$ between a clustering, such as C_{Early}, and its corresponding difference matrix, D_{Early}, is a measurement of the differences that exist within nodes that are clustered together in C_{Early}. Figure 2 plots $Mismatch(C_{Early}, D_{Early})$ and $Mismatch(C_{Late}, D_{Late})$ as a function of the number of clusters k for the VPR routing benchmark. This plot is typical of such curves between early inlining and late inlining for most benchmarks. As the plot shows, while four clusters are sufficient to separate virtually all the differences between inputs with respect to early inlining, the $Mismatch$ curve for late inlining has a long tail, indicating that many more differences exist amongst the inputs when late inlining is considered. Thus, early inlining would be a poor representative of late inlining.

Figure 2 also plots the $CrossError$ curves that evaluate how well the clustering based on late inlining, C_{Late}, represents that data in the early inlining difference matrix D_{Early} and vice-versa. $CrossError(C_{Early}/C_{Late}, D_{Late})$ stays level in the 3%-3.5% range from 4 clusters to 11 clusters. Over this range, $CrossError$ is on average 97% of $Mismatch$, indicating that applying C_{Early} to the late inlining data results in about twice as much error than using C_{Late}. The level $CrossError$ curve through this range indicates that even with additional clusters, early inlining does not provide any information to enable the extra clusters to better separate the inputs.

On the other hand, since late inlining has more information regarding input dissimilarity, it could be the case that this information is a superset of the information provided by early inlining. However, from 4 to 8 clusters, $CrossError(C_{Late}/C_{Early}, D_{Early})$ tracks just below $Mismatch(C_{late}, D_{Late})$. Even at 8 clusters, the $CrossError$ is only slightly less than $Mismatch(C_{late}, D_{Late})$ at 3 clusters. The late-inlining clustering does not separate the few differences that do exist in the early-inlining data. Therefore, in this case, late inlining is not a good representative of early inlining, despite the conceptual similarity of the transformations. We observed similar patterns between early and late inlining for most benchmarks.

An alternate way to try to establish the correlation between the two inlining transformations would be to take each pair of inputs as a data point and use the early-inlining Manhattan distance between the inputs for one axis and their late-inlining Manhattan distance for the other axis. The coefficient of correlation calculated this way is 0.99, which would indicate a very high degree of correlation in the data. The high coefficient of correlation can be attributed to data points falling into two clusters, with one cluster occurring very far from the others. Consequently, at the scale of the data, the clusters become two points and display a linear relationship. However, looking at each of the two groups of data individually, the data does not exhibit any linear or recognizable non-linear relationship. A systematic study of the same form of data across all transformation pairings and all programs suggests that the coefficient of correlation is usually not a good indicator of a representativeness relation-

ship between a pair of transformations.

5.3 Input Similarities for Different Programs

If two application programs p and q can use the same set of training inputs — such is the case, for instance, when the application is a compiler, a translator, or a data compressing program — then it may be interesting to find out if the clustering generated for p is also a good clustering for q. A more practical use of this evaluation is if p and q are two versions of the same program.

As a case study, we use `Bzip2` and `gzip`, two data compression programs that use different compression algorithms. The workload of inputs consists of the SPEC2000 and SPEC2006 inputs for `bzip2`, the SPEC2000 inputs for `gzip`, and the additional inputs [1]. When different inputs with the same name exist in the two SPEC benchmark suites, 2000 or 2k6 is appended to the name as appropriate. Many of the input names clearly indicate the file format of the data (*jpeg, compressed, html, mp3, mpeg, random, pdf, xml). Of the rest, gap and program are compiled programs; source is a tarball of source code; log and reuters are ASCII text documents; graphic is a TIFF image; combined contains parts of various other SPEC inputs; and docs is a tarball of documents from various office-application suites. Baseline refers to the case without PDF. Table 7 shows clusters for the two programs using the combined similarity matrix. `Bzip2`'s first clustering with less than 2% error occurs with 6 clusters; this same condition is met with 4 clusters for `gzip`. The clusters are arranged horizontally by similarity, with each distinct input in a row of clusters on a separate line. When the transformation from 4 to 6 clusters splits a cluster, the new cluster is listed vertically adjacent to the cluster it was split from.

5.3.1 Selecting a Good Clustering

Comparing the two clusterings for `gzip` in Table 7, the two largest clusters are split in the transition from 4 to 6 clusters. Cluster 5 is split to create cluster 6, and cluster 7 is split to create cluster 8. However, this is not the case with `bzip2`. The largest cluster, cluster 7, remains unchanged, while the combined2k6 input is split out to it's own cluster from cluster 5. Additionally, 2 of the 5 inputs in cluster 9 are split off to create cluster 10.

The $Mismatch$ error metric generally encourages splitting large clusters, since this action removes the most edges from the graph. However, when the edge weights are not similar, the benefit of splitting smaller clusters or splitting a cluster into unequally-sized parts increases. In the case of `gzip`, the edge weights are likely all of similar value. By splitting the two largest clusters, $9 \times 5 + 3 \times 4 = 57$ edges are removed, and the error is reduced from 0.66% to 0.16% of total error. Therefore, given the low error and the behavior of subsequent clustering, it is reasonable to conclude that similar inputs are grouped together for `gzip` by 4 clusters. In fact, checking the difference matrix shows that the `gap` input and baseline compilation are the major points of difference for `gzip`. Baseline compilation is expected to be significantly different than PDF compilation, since the compiler has far less data

Data	`gzip`		`bzip2`	
Clusters **Error**	**4** **0.66**	**6** **0.16**	**4** **6.25**	**6** **1.71**
1	baseline	baseline		
2			xml	xml
3	gap	gap		
4				combined2k6
5	byoudoin-jpg chicken-jpg compressed dryer-jpg liberty-jpg graphic mpeg	byoudoin-jpg chicken-jpg liberty-jpg	byoudoin-jpg chicken-jpg compressed dryer-jpg combined2k6 mpeg jpeg program2000	byoudoin-jpg chicken-jpg compressed dryer-jpg mpeg jpeg program2000
6		compressed dryer-jpg graphic mpeg		
7	combined2k6 docs html jpeg log mp3 pdf program2000 program2k6 random reuters source2000 source2k6 xml	combined2k6 html log program2000 random reuters source2000 source2k6	 gap html log mp3 pdf program2k6 reuters source2000 source2k6	 gap html log mp3 pdf program2k6 reuters source2000 source2k6
8		docs jpeg mp3 pdf program2k6		
9			docs graphic liberty-jpg random baseline	docs graphic random
10				liberty-jpg baseline

Table 7: Clusterings for the combined `bzip2` and `gzip` matrices

on which to base transformation decisions, and performs some transformations differently without PDF data. Consequently, only very small training inputs (*e.g.*, SPEC test inputs) that do not exercise the majority of the program code would be expected to share similarity with the baseline. Once these two points have been split off into separate clusters, clustering error is 3.21% of total error. Nonetheless, the 4-cluster result is likely a better basis for workload reduction than the 3-cluster result because the reduction in error is still proportionally large (error is reduced by 80%).

On the other hand, the additional partitioning from 4 to 6 clusters for `bzip2` removes only $7 + 6 = 13$ edges, but reduces error from 6.25% to 1.71% of total error. From the error value and partitioning behavior, it is difficult to evaluate the quality of the clustering. Even at 6 clusters, the inputs for `bzip2` may not be sufficiently partitioned. Error is slowly reduced as more clusters are created, and the partitioning never clearly converges to largest-cluster splitting. For example, cluster 7 remains unchanged until it is split to form the 9^{th} cluster (error at that point is still 0.70).

When have the significant differences between inputs been accounted for, and when has over-clustering begun? How can a good number of clusters be selected? When the answers to these questions are not clear, a compiler designer should consult more than the final clustering results to make a decision as to how many clusters represents a "good" clustering. First, the similarity (or difference) matrix shows quantitatively how similar the inputs are to each other. Table 8 and

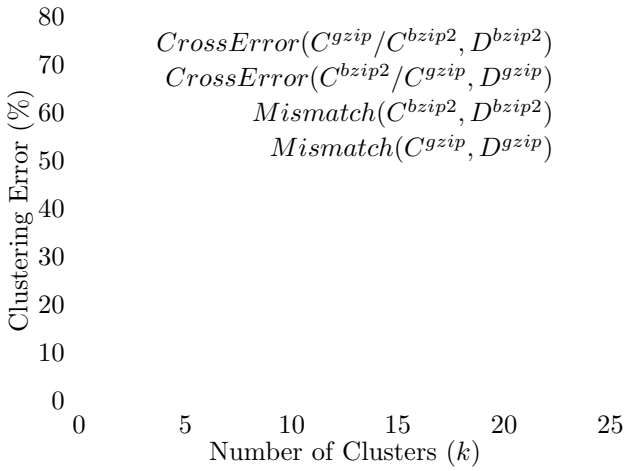

Figure 3: $Mismatch$ and $CrossError$ using the combined matrix of `bzip2` and `gzip`

Table 9 show the difference matrix for clusters 5 and 7 of the 4-cluster partitioning of `bzip2`, respectively. These tables contain only the upper-triangular section of the symmetric matrix to improve the clarity of data presentation. The `combined2k6` input in cluster 5 is significantly different from the other inputs in that cluster, so it should be split off into it's own cluster. However, the inputs in cluster 7 are all quite similar to each other, and further partitioning of these inputs is probably not necessary. Therefore, the 5-cluster partitioning (where `combined2k6` is split off) provides the desired result. In the case where the combined difference matrix does not present strong evidence, the difference matrices of the individual transformations may be consulted. Finally, investigating the individual transformations that differ in the transformation vectors may not only provide important information on how much inputs differ from each other, but also offer clues about why those differences occur.

chicken-jpg	**combined2k6**	compressed	dryer-jpg	jpeg	mpeg	program2000	
0.28	**13.84**	2.07	1.59	0.71	0.72	0.01	byoudoin-jpg
	7.24	2.27	1.03	0.28	1.12	0.25	chicken-jpg
		43.9	**8.07**	**3.16**	**28.8**	**13.5**	**combined2k6**
			0.36	0.54	1.28	1.63	compressed
				0.12	3.01	1.26	dryer-jpg
					1.97	0.50	jpeg
						1.47	mpeg

Table 8: Difference matrix of cluster 5 in Table 7 for the 4-cluster partitioning of `bzip2`

html	log	mp3	pdf	program2k6	reuters	source2000	source2k6	
3.69	1.80	0.56	1.01	0.61	0.92	0.89	0.80	gap
	1.60	0.95	0.20	0.69	1.01	0.73	1.06	html
		0.66	0.41	0.43	1.05	0.39	1.10	log
			0.17	0.05	1.86	0.02	1.24	mp3
				0.32	0.89	0.12	1.38	pdf
					1.39	0.03	1.33	program2k6
						1.41	0.05	reuters
							0.94	source2000

Table 9: Difference matrix of cluster 7 in Table 7 for the 4-cluster partitioning of `bzip2`

5.3.2 Clustering Comparison

There are several common sets of inputs between the `bzip2` and `gzip` clusterings: `bzip2` clusters 4 and 5 both overlap `gzip` cluster 5, and `bzip2` cluster 6 has 6 members in common with `gzip` cluster 6. However, in all cases, these partially-overlapping clusters have several members that are not common between the two programs. $CrossError$ provides a mechanism to evaluate the significance of the similarities (and differences) between these alternate clusterings. Figure 3 shows the $Mismatch$ curves for both programs, along with the corresponding $CrossError$ curves. Recall that $CrossError$ measures the amount of *additional* error incurred by using an alternate clustering.

By 5 clusters, the $Mismatch$ for both programs is near the minimum error. Furthermore, as discussed above, 5 clusters provides a good clustering for both programs. However, at the same point, the $CrossError$ for both programs is more than 18%. Furthermore, even as the number of clusters increases, the $CrossError$s reduce slowly. This evidence indicates that, from the compiler's perspective, input similarity cannot be adequately measured outside of the context of the program processing the inputs.

6. Conclusion

PDF is an important tool for minimizing program execution time. Unfortunately, the standard practice of PDF can be sensitive to input diversity, and thus

a cross-validation strategy is require for performance evaluation. However, selecting an appropriate workload for cross-validation is challenging: The workload must cover all the important aspects of the program, while also minimizing the number of inputs in the workload. This paper illustrates a clustering technique to select this small subset of inputs from the large number of inputs available to a compiler designer. A similarity matrix is constructed from transformation vectors, information extracted directly from the compiler regarding differences between inputs. This matrix is weighted by a cross-validation based performance metric in order to filter out those differences that do not impact performance. Once the workload has been clustered, the $Mismatch$ curve presents a quantitative measure of the tradeoff between how many inputs are selected and how representative these inputs are of the full workload. Finally, $CrossError$ provides a means to investigate correlations between transformations.

Rather than developing a case study for a single application with a large workload, this study explores the proposed techniques across a range of standard benchmark programs with moderately sized workloads. The variety of programs used provides an environment where the generality of the proposed methodology can be tested, and trends may be observed in transformation correlations. Furthermore, since the benchmark programs and several of the inputs are well-known by compiler researchers and developers, this work is more accessible than a study using a program encumbered by proprietary source code and confidential data.

Acknowledgments

This research is supported by fellowships and grants from the Natural Sciences and Engineering Research Council of Canada (NSERC), the Informatics Circle of Research Excellence (iCORE), the Canadian Foundation for innovation (CFI), and Alberta Ingenuity. Thanks to Dan Lizotte for discussions and advice related to clustering.

Trademarks

References

[1] P. Berube. Additional FDO inputs. http://www.cs.ualberta.ca/ berube/compiler/fdo/inputs.shtml.

[2] P. Berube and J. N. Amaral. Aestimo: a feedback-directed optimization evaluation tool. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 251 – 260, Austin, Texas, March 2006. IEEE Computer Society.

[3] P. Berube and J. N. Amaral. Benchmark design for robust profile-directed optimization. In *Standard Performance Evaluation Corporation (SPEC) Workshop*, Austin, Texas, USA, January 2007.

[4] L. Eeckhout, R. Sundareswara, J. J. Yi, D. J. Lilja, and P. Schrater. Accurate statistical approaches for generating representative workload compositions. In *International Symposium on Workload Characterization (IISWC)*, pages 56–66. IEEE Computer Society, October 2005.

[5] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Workload design: Selecting representative program-input pairs. In *Parallel Architectures and*

Compilation Techniques (PACT), page 83, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[6] D. Gove and L. Spracklen. Evaluating the correspondence between training and reference workloads in SPEC CPU2006. In *SIGARCH Computer Architecture News*, volume 35, pages 122–129, New York, NY, USA, 2007. ACM.

[7] K. Hoste and L. Eeckhout. Comparing benchmarks using key microarchitecture-independent characteristics. In *International Symposium on Workload Characterization (IISWC)*, pages 83–92, San Jose, CA, October 2006. IEEE Computer Society.

[8] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. In *Computer Architecture Letters*, volume 1. IEEE Computer Society, June 2002.

[9] A. Maxiaguine, Y. Liu, S. Chakraborty, and W. T. Ooi. Identifying ”representative” workloads in designing MpSoC platforms for media processing. In *Embedded Systems for Real-Time Multimedia (ESTImedia)*, pages 41– 46, September 2004.

[10] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *SIGARCH Computer Architecture News*, volume 35, pages 412–423, New York, NY, USA, 2007. ACM.

[11] X. Shen and F. Mao. Modeling relations between inputs and dynamic behavior for general programs. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Urbana, IL, USA, October 2007.

[12] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, New York, NY, USA, 2002. ACM.

[13] J. Shi and J. Malik. Normalized cuts and image segmentation. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 22, pages 888–905, August 2000.

[14] Standard Performance Evaluation Corporation. SPEC: The standard performance evaluation corporation. http://www.spec.org/.

[15] H. Vandierendonck and K. D. Bosschere. Experiments with subsetting benchmark suites. In *International Workshop on Workload Characterization (WWC)*, pages 55–62. IEEE Computer Society, October 2004.