Approximate Graph Clustering for Program Characterization

JOHN DEMME and SIMHA SETHUMADHAVAN, Columbia University

An important aspect of system optimization research is the discovery of program traits or behaviors. In this paper, we present an automated method of program characterization which is able to examine and cluster program graphs, i.e., dynamic data graphs or control flow graphs. Our novel approximate graph clustering technology allows users to find groups of program fragments which contain similar code idioms or patterns in data reuse, control flow, and context. Patterns of this nature have several potential applications including development of new static or dynamic optimizations to be implemented in software or in hardware.

For the SPEC CPU 2006 suite of benchmarks, our results show that approximate graph clustering is effective at grouping behaviorally similar functions. Graph based clustering also produces clusters that are more homogeneous than previously proposed non-graph based clustering methods. Further qualitative analysis of the clustered functions shows that our approach is also able to identify some frequent unexploited program behaviors. These results suggest that our approximate graph clustering methods could be very useful for program characterization.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms: Design, Algorithms, Performance

ACM Reference Format:
Demme, J. and Sethumadhavan, S. 2012. Approximate graph clustering for program characterization. ACM Trans. Architec. Code Optim. 8, 4, Article 21 (January 2012), 21 pages.
DOI = 10.1145/2086696.2086700 http://doi.acm.org/10.1145/2086696.2086700

1. INTRODUCTION

Identifying interesting program execution behaviors is important for creating optimized, secure systems. Today, program characterization is a laborious and increasingly time consuming process due to a combination of factors including the growth in number of applications, the wide variety of platforms these applications run on, and difficulty in the characterization process. To see some of the challenges in program characterization consider the case of SPEC benchmarks. Figure 1 plots the number of functions responsible for a certain fraction of the execution time. For example, five functions from *each* SPEC INT benchmark (of which there are 11), contribute to, on average, 67% of execution time. If SPEC FP is examined as well, the number of functions grows from 55 to 135. In fact, examining all of the functions in SPEC responsible for any non-trivial amount of execution time (at least 1%) requires characterizing about 300 functions comprising about 14,000 lines of code. This is a significant amount of code but is still small in comparison to real-life codes. The last data set shown in figure 1 is coverage data from the V8 Javascript Engine, a production library used in the Chrome web browser. The V8 profiling data indicate that the amount of code that must be

The research that was conducted at CASTL was funded by grants from DARPA, AFRL (FA8750-10-2-0253, FA9950-09-1-0389), the NSF CAREER program and gifts from Microsoft Research and Columbia University.
Authors' address: J. Demme and S. Sethumadhavan, Columbia University, New York; email: {jdd, simha}@cs.columbia.edu.

DOI 10.1145/2086696.2086700 http://doi.acm.org/10.1145/2086696.2086700

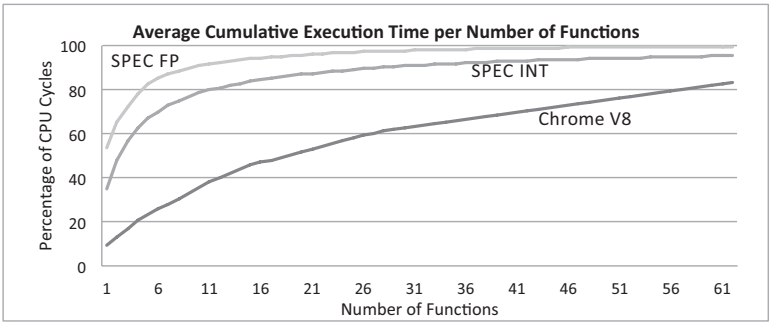

Fig. 1. This chart plots cumulative program execution time (on the Y-axis) along with number of functions contributing to the execution time (on the X-axis). Measurements on 11 SPEC INT benchmarks and 16 SPEC FP (both averaged in the figure) and the V8 Javascript engine distributions all demonstrate that *many* functions contribute to total execution time significantly. This spread presents a significant challenge for program characterization.

characterized is very large—nearly 30% of its functions must be examined to cover 90% of its execution. Given this immense scale, automated program characterization could be much more comprehensive than manual characterization, potentially yielding better results.

To ameliorate these challenges researchers have proposed a wide variety of technologies including sophisticated code profiling, fast simulation techniques, and machine learning with performance counter data [Namolaru et al. 2010; Dubach et al. 2007; Demme and Sethumadhavan 2011; Eeckhout 2010]. Some researchers have even proposed crowdsourcing approaches [Fursin and Temam 2009; Fursin et al. 2008]. In this paper we propose a new, complementary technique to study program behavior. Our program characterization technique allows automatic identification of unique code behaviors across a code base by *approximately clustering program graphs*. By clustering similar graphs, we expose similar control and dataflow patterns in software, allowing one representative sample from each cluster to be studied rather than all graphs.

While we are not the first to observe the benefit of clustering [Joshi et al. 2006], prior approaches have focused on clustering of non-graphical formats such instruction frequency, or microarchitecture dependent features such as cache misses or IPC measured from performance counters. Our technique is the first to propose clustering on program graphs and thus enables microarchitecture independent characterization of programs. Futher, graphical intermediate representations are a semantic step closer to algorithmic description and thus may offer more fundamental insights into program traits and lead to more comprehensive program characterization.

In this paper we adapt a decade old advance in identifying similarity in graphs [Melnik et al. 2002] to create approximate graph clustering for program characterization. Traditional graph similarity methods such as isomorphism can determine if two graphs match, but are not useful for clustering because even very similar programs can produce slightly different graphs. Approximate graph clustering, on the other hand, instead of providing discrete answers, produces a continuous measure for similarity based on the number and content of nodes and edges and graph topologies. This continuous measure lends itself to grouping graphs using known clustering techniques.

To evaluate the usefulness of graph clustering we compare it to other known non-graphical, microarchitecture dependent and independent information. We measure the effectiveness by applying and comparing the effect of *existing* optimizations to functions in clusters. We hypothesize that if functions in these clusters react homogeneously

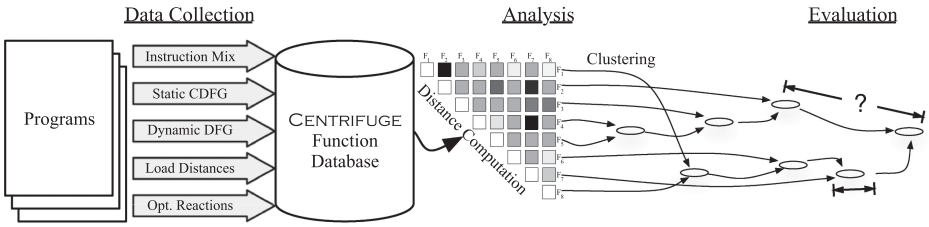

Fig. 2. CENTRIFUGE involves three stages: data collection, analysis and evaluation. In data collection, information on function behavior is assembled. Analysis uses this information to compute distances between each function, in turn using this distance data to cluster the functions. Finally, in the evaluation stage we examine the clusters to determine their quality.

to optimization then they share common behaviors or characteristics. Homogeneous reactions also suggest that exemplars from the clusters could be used to develop the original optimizations.

The results of our evaluation on the SPEC benchmark suite show that clustering programs based on dynamic dataflow graphs produces much better clusters than clustering with instruction mixes (non-graphical) or other static/dynamic run time characteristics (non-graphical, microarchitecture dependent). We also qualitatively examined our best clusters to investigate if new distinct behaviors could be identified in the clusters. We were able to locate distinct behaviors in some clusters, but not all. This is likely because the SPEC benchmark is intended to be diverse so little obvious redundancy exists. Our results suggest that automated behavior identification may be possible with graphical clustering.

Interesting characteristics found with our techniques could assist in the discovery of a variety of optimizations in software, hardware or some hybrid of the two. These optimizations could be purely serial in nature or take advantage of parallelism. For instance, if one was to build clusters using data dependence information, resulting clusters might highlight data parallelism patterns ripe for optimization. As another example, we find an interesting pattern in our results (Table III). As we discuss later, this very common pattern could be further optimized.

The rest of this paper is structured as follows: Section 2 defines our clustering framework, called CENTRIFUGE, and details our implementation. We explain our experimental setup for computing results in Section 3. These results are then presented and analyzed Section 4. Section 5 discusses related work. We conclude with a discussion of results, implications and future work in Section 6.

2. METHODOLOGY

Our clustering framework, CENTRIFUGE supports three primary steps: building program representations, clustering and evaluation. The first stage involves collecting static and dynamic characteristics of functions into approximate representations of the functions. The second stage involves two processes: comparison and clustering. Comparison involves examining the approximate representation of different functions to judge their similarity. This similarity information is then used to cluster the functions into similar groups. At the end of the second stage a user could study functions from each cluster. In the evaluation stage—the third and final stage—the contents of each cluster can be used to analyze the quality of the clusterings. This section provides details about each of these stages and is intended to give an overview of the processes, desiderata and tradeoffs relevant to each stage.

2.1. Representations

The first task in identifying code behaviors is to find static and/or dynamic features that capture function behavior. Good features will capture all characteristics that influence performance significantly. In this paper, we include traditional static and dynamic characteristics that are believed to influence performance such as static instruction mix, static control/data flow graphs and dynamic data flow graphs. In addition we included two new features: dynamic data flow graphs augmented with locality information and a representation based on how well functions perform on existing hardware using existing optimizations. Each of these representations is described below.

(1) *Static Instruction Mix.* Static instruction mix can be useful as an indicator of how functions react to optimizations. For instance, clustering based on instruction mix may group math heavy functions or data movement functions in different clusters, and optimizations (such as vectorization or optimizations that use string movements) may operate on functions in different clusters differently. We compute the static instruction mix by compiling a program without any optimization and then categorizing the instructions into one of following categories: control transfer, arithmetic, logical, and memory. The fraction of instructions in each category are represented in a vector for each function in the code base.

(2) *Static CDFG.* Static control/data flow graphs are commonly used in compiler analyses. These graphs can capture the parallelism that is available in a function (or program) and thus may be good candidates to represent function behavior. As is common in compiler analyses, we use basic blocks as vertices in the flow graphs. We further annotate our graph vertices with basic block instruction mixes and edges with the dependence type (control, must pointer, may pointer, register). These dependencies were generated using the LLVM [Lattner 2002] compiler.

(3) *Dynamic DFG.* Static CDFGs include static memory dependence information. This dependence information is computed using alias analysis algorithms, which have been shown to be imprecise [Mock et al. 2001]. In the worst case, overly conservative analysis yields completely connected dependence graphs. Since there is only one complete graph given a number of vertices, this poor analysis effectively reduces the amount of information about the function. Dynamic data flow graphs, on the other hand, can be more detailed than Static CDFGs because they represent only *observed* dependence edges instead of potential dependence edges. Further, we can annotate these graphs with other dynamic information. In this paper, we use dependence observation frequencies (the number of times a data dependence is observed between two basic blocks divided by the number of times the consuming basic block is executed) to add dependence edge weights to our Dynamic DFGs. To annotate these graphs' vertices, we compute the basic block's dependence chain length (the length of the longest producer-consumer chain of instructions within a basic block), number of integer instructions, number of floating point instructions, and number of memory loads, all of which are expressed as a fraction of total instructions. Dynamic features like execution count can also be used, though we have found that this can increase sensitivity to input bias.

(4) *Dynamic DFG with Load Distance.* It has been known for long time that Data locality is an important determinant of performance along with parallelism [Arvind and Iannucci 1987]. While the CDFG and Dynamic DFG are likely to be good at capturing parallelism constraints, they do not explicitly capture locality. To capture locality, we measure the number of dynamic instructions since each load's address was last accessed in the program (measured with PIN on executables with no optimizations). This is similar to reuse distance [Ding and Zhong 2003], except we count instructions instead of memory accesses. For example, if a called function accesses an address X which was last written by the calling function, a very low load distance will result. If, however, address X was last written during program initialization and has not been read since, a very

high load distance will result, reflecting the improbability of the location being in the processor cache. We then compute the average and standard deviation of the logarithms of all load distances in a function and use these two measures as indicators of reuse.

(5) *Function Optimization Reactions.* In addition to the above representations, we can use existing optimizations to cluster similar functions. The intuition is that functions that are similar will be affected to the same degree when a set of optimizations are applied to them. Observing the optimization reactions, therefore, may give us an indirect indication of function characteristics. We construct this model by applying various combinations of known optimizations to a function, measuring the change in execution time for that function, and computing the function's optimization reaction, as defined in Eq. (7). The optimization reactions are then used to cluster functions.

2.2. Comparison Methods

Now that we have the features to create the clusters, the next step is to determine how these features should be compared. We formalize the comparison by defining distance functions: for features that are represented in vector formats (e.g., static instruction mix), we use the Euclidean distance between the two vectors to measure their similarity. In our initial experiments Euclidean distance yielded better results than alternatives (like Manhattan distance). For two vectors A and B both with length n, Euclidean distance is defined as:

$$\sqrt{\sum_{i=1}^{n}(A_i - B_i)^2} \tag{1}$$

Comparison of the graphical representations (e.g., static CDFG, dynamic DFG, locality-annotated Dynamic DFGs) is more complex, deserving further explanation.

Traditional graph comparison algorithms such as isomorphism and subgraph isomorphism produce boolean "match" or "no match" results when two graphs are compared and are not useful for computing the *approximate* similarity between two functions represented as graphs. Instead, we use an approximate graph comparison technique [Melnik et al. 2002] to compute the distance between two graphs. Determining the distance between two graphs involves two steps: mapping and scoring. In the mapping stage, we pair nodes from the two input graphs. That is, for each vertex in graph A, we find the vertex in graph B which is most similar, both in terms of the local information (basic block instruction mix) and neighborhood information (the similarity of connected vertices). The scoring phase then uses this mapping to compute vertex and graph structural similarity. With our graphical features, this process corresponds to finding similar basic blocks and then using the edge information (presence or absence of edge, and edge annotations) to judge the similarity of two functions.

Mapping. The mapping phase has been adapted from the approach described by Melnik et al. as "Similarity Flooding" [Melnik et al. 2002]. Although we use a similar set of steps, we have modified the ways in which values are normalized in various phases. The basic idea is to construct a new graph which contains all possible pairs of vertices and edges derived from edges in the two original graphs.[1] We assign a similarity value to each vertex (described shortly) in this graph and then iteratively propagate these values along edges. Upon convergence, each value represents the similarity of a pair of vertices which is sensitive to both the basic blocks' functional similarity (as determined by instruction mix distance) and the two graphs' structural similarity. We can then use these values to determine the optimal mapping. The precise details follow in addition to an example in Figure 3.

[1]This is known as a product graph. Specifically, we use the Tensor product graph.

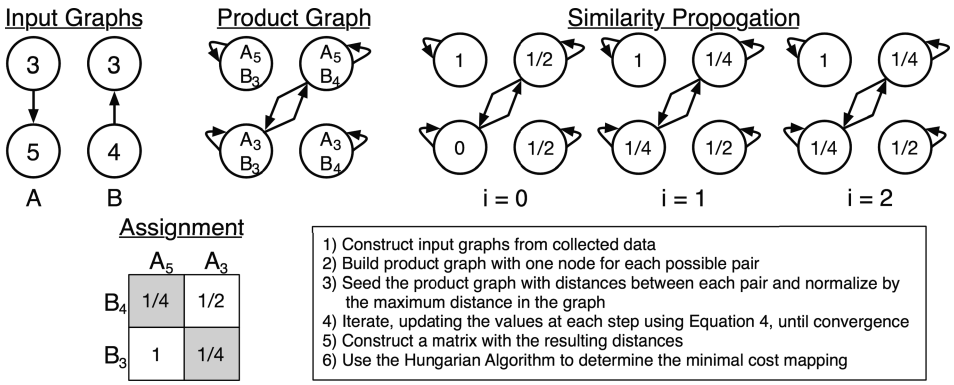

Fig. 3. Simple example of the mapping stage, a variant of Similarity Flooding. In this example, we use unity edge weights and scalar (instead of vector) values in each vertex.

Formally, we begin by forming a tensor graph product of two function graphs to be matched. For weighted graphs (V_A, E_A) and (V_B, E_B), we form the tensor graph product (V_P, E_P):

$$\begin{aligned} V_p &\equiv \{AxB; A \in V_A, B \in V_B\} \\ E_p &\equiv \{((AxB)x(A'xB'), W_a \cdot W_B); \\ &\quad (AxA', W_A) \in E_A, (BxB', W_B) \in E_B\} \end{aligned} \tag{2}$$

This product graph contains all the possible mappings, so the final mapping is intuitively a restricted subset of this graph.

Next, we define a primitive product graph vertex similarity function, $D(v)$. In our case, each product graph vertex represents a pair of basic blocks—one from each input graph—so we use the Manhattan distance between the basic blocks' instruction mixes. (In our initial experiments, Manhattan distance yielded better results than the more popular Euclidean distance.) For two vectors A and B each of length n, Manhattan distance is defined as:

$$\sum_{i=1}^{n} |A_i - B_i| \tag{3}$$

Step three involves propagating the similarity scores along the graph edges, in a manner similar to that described by Melnik et al. [2002]. This is an iterative calculation wherein at each step, i, we compute each vertex value M_j^i for each vertex j in V using the set of adjacent vertices and edge weights, E_j and W_j:

$$\begin{aligned} M_j^0 &= D(V_j) \\ M_j^i &= \frac{D(V_j)}{max(D(V_{i-1})) \cdot \sum W_j} \sum_{a \in E_j} W_j^a \cdot M_a^{i-1} \end{aligned} \tag{4}$$

The intuition behind this step is that if a node represents a good match and its neighbors are also good matches, then that node is probably a very good match and hence its score improves.[2]

The output from this similarity propagation step is a matrix of vertex-vertex pair scores which reflect both the primitive basic block and structural similarity. Determining the optimal mapping is thus a matter of selecting the set of pairs which are overall

[2]Detailed explanation of the Similarity Flooding algorithm is outside the scope of this paper. Please refer to Melnik et al. [2002] for more information.

most similar. This is an instance of the assignment problem and we use the well known Hungarian Algorithm [Kuhn 1955] to solve it.

Scoring. Once the optimal mapping is established, we merge the two input graphs and score the merged graph. There are a number of scoring methods one could use. One might use the scores assigned by the previous mapping step. However, the scores from the mapping stage have been heavily affected by scores from potential pairs which turned out to be bad matches. (Recall that the product graph through which scores are propagated contains all possible pairs from which we select a small subset.) As a result, the scores from the mapping phase reflect the fitness of the selected vertex pairs relative to all the possible vertex pairs. The score we want, however, is the similarity of the two input graphs relative to other graphs.

We have designed an alternative scoring metric which essentially takes the basic block Manhattan distance and combines it with the structural mismatch which we define as the normalized quantity of unmatched edges. For each merged vertex, i in V, we produce the basic block instruction mix Manhattan distance, D_i, the number of common shared edges, S_i, and the number of non-shared edges, E_i. For vertices with no counterpart, we use $\sum_{d \in D_i} d$ for D_i and set $S_i = E_i$. We compute the score as:

$$\frac{1}{|V|} \sum_{i \in V} \left(D_i + \frac{E_i}{S_i + E_i} \right) \tag{5}$$

2.3. Clustering Methods

Equipped with quantitative comparisons for each representation, CENTRIFUGE can use well-known clustering methods such as agglomerative clustering, k-means, or SOM. In general, a clustering algorithm will attempt to group most similar functions together based on item distances or similarity of their features.

Some of these clustering techniques such as agglomerative clustering are hierarchical and produce dendrograms as the output instead of discrete clusters. The dendrograms can be converted to flat clusters by *thresholding*—cutting off growth of a cluster at a particular maximum diameter or a predetermined average internal distance. Thresholding creates a tradeoff between cluster size and the similarity of the contained functions. Higher thresholds typically create larger clusters with more dissimilarity than lower thresholds.

The user can also use an alternative property called savings instead of thresholding. *Savings* indicates the reduction in number of functions to be examined after clustering. Formally, the *savings* yielded from clustering f functions into c clusters is $(1 - \frac{c}{f}) \cdot 100$. For example, if 50 functions are clustered 20 groups, then only 20 representative functions must be studied. This results in a 60% savings in the number of functions to be understood. The savings and threshold metrics are related because a larger threshold results in larger savings. Importantly, the savings metric can be compared across different representations and distance functions whereas threshold cannot since it is specific to the distance function. Additionally, parameterizing function organization via savings allows the user to gain better insight into functions' similarities by adjusting the savings level. Different clusters are shown at each savings level, indicating the distribution of similarity for various thresholds.

For this paper, we use average linkage agglomerative clustering with the following four representations and distance functions.

(1) *Instruction Mix.* Euclidean distance
(2) *Static CDFG.* Similarity flooding-based graph distance defined in Section 2.2 using unity edge weights.
(3) *Dynamic DFG.* Similarity flooding-based graph distance defined in Section 2.2.

(4) *Locality Annotated Dynamic DFG.* We must combine the score from Similarity Flooding-based graph distance (defined in Section 2.2) for the Dynamic DFG with the load distance information detailed in Section 2.1. This data is represented by five numbers: the graph distance, g, for each function, the logarithms of the mean of the load distances, M_a and M_b along with logarithms of the standard deviations of the load distances, S_a and S_b. We combine them as such:

$$g + |M_a - M_b| + |S_a - S_b| \tag{6}$$

Our fifth representation, optimization reaction, also uses agglomerative clustering, but instead uses Ward's linkage [Ward 1963]. Ward's linkage clustering attempts to minimize the variance of vector data within clusters. In other words, each time a cluster is selected for a particular data point, the cluster with the least increase in internal average variance is selected. Since our evaluation will eventually compute internal standard deviations, we can expect this clustering technique to create better clusters.

Optimization Reaction Metric. Since profiling yields absolute times, we cannot directly compare optimization affects without some normalization of the times. Throughout this paper, we use *optimization reaction*, defined below, to normalize all execution times to the range [0, 1]. If $T(f, o, i)$ is the runtime for function f when compiled with optimization o, executed with input set i and O is set of all profiled optimizations, we define the optimization reaction:

$$R(f, o, i) \equiv \frac{T(f, o, i) - min_{j \in O} T(f, j, i)}{max_{j \in O} T(f, j, i) - min_{j \in O} T(f, j, i)} \tag{7}$$

Normalizing the optimizations allows functions to be compared based on their relative effectiveness rather than raw speedup. The optimization reaction metric allows us to judge a reaction to an optimization by rank, so functions which are both sped up *best* by a particular optimization will have similar reactions for that optimization.

Parameter Tuning. In all of these comparison methods, some information may be more important than others for similarity comparison i.e., the number of memory instructions may influence performance more than the number of math instruction. To account for this, we can add additional parameters to each distance function allowing them to weight information differently. For instance, when we compute vector distances, we can instead use a weighted vector distance, changing the importance of each vector field. We add parameters for two representationss—Dynamic DFGs and Dynamic DFGs with Load Distances—in the following places.

(1) *Vector Distances.* All vector distance calculations (in both similarity flooding and scoring) get weights for each vector field.
(2) *Similarity Flooding.* In the mapping phase, similarity scores propagate through the product graph (Eq. (2)). At each iteration, these propogated scores are combined with local scores, creating a trade off between local and neighborhood similarity (Eq. (4)). We add weights to each input before adding them.
(3) *Graph Scoring.* After mapping, a merged graph is created and scored. During this scoring, several factors are combined to create a single similarity score. Here, we add importance weights when combining local and neighborhood similarity (similar to the last point), plus a weights to use for local and neighborhood similarity for unmatched vertices.
(4) *Load Distances.* When combining data flow graphs with locality information (Eq. (6)), we add three pieces of information: graph similarity, difference in log load distance means and difference in standard deviations. When tuned, each term is given a weight.

In total, we have 12 parameters which must be tuned. Given the extreme nonlinearity of graph mapping, scoring and clustering, it is not possible to back-calculate optimal parameter values. Instead, we must use black-box optimization. In particular, we implemented simulated annealing, splitting our set of functions into 1/3 for training and 2/3 for testing. As an objective function, we measure the variance of reaction to optimization within the generated clusters. As such, simulated annealing with this objective function will attempt to minimize the variance of reaction to optimization within the generated clusters.

2.4. Evaluation Strategy for CENTRIFUGE

We have now defined various representations, distance functions and clustering algorithms. Combined, these are used to create a clustering of functions. In this paper, we compare functions across programs. This function granularity is neither the only possible granularity nor the optimal granularity, and virtually any block of code could be compared to any other. However, we chose function granularity for two reasons: First, functions are the granularity at which programmers generally think. Since we are attempting to find patterns in programmers' code, functions are a reasonable code unit. Second, identifying appropriate code sequences from an entire program is a difficult analysis problem. Other work [Pan and Eigenmann 2008; Triantafyllis et al. 2006] explores this problem and should integrated into CENTRIFUGE in the future.

To determine the utility of each representation, we must examine the resulting clustering and judge its quality. We can quantitatively measure quality by studying optimizations that have already been discovered; in particular, we wish to determine if our clustering could have been useful in discovering an optimization which already exists. If the functions in each cluster tend to react similarly to an existing optimization, then those clusters are significant with respect to that optimization. As a result, having those clusters may have been useful in developing the optimization. This metric can easily be quantified by measuring the functions' reactions to various optimizations. With this insight it is also fairly straightforward to come up with lower bound and upper bounds for the quality of the clusters. We can then compare various quality metrics about each representation's clustering to these bounds.

Random Clustering. It is reasonable to expect a clustering to do no worse than randomly grouping functions. To estimate this worst case, we randomly generate similarity distances between all pairs of functions in the code base and use these distances along with average linkage agglomerative clustering to generate clusters. We would expect all properties of the functions in each of these clusters to be random selections of properties from the global set of functions.

Ideal Clustering. Clusters built using the same information upon which they are evaluated should represent an upper bound. We will evaluate clusters based on the consistency with which their functions react to optimization, so we use the evaluation data and Ward's linkage agglomerative clustering to build this loose upper bound. (Loose due to the heuristic nature of agglomerative clustering in this context.) It is important to note that this is also an unfair upper bound because it assumes complete knowledge whereas other representations, by definition, are incomplete approximations of function behavior. With complete knowledge, this ideal clustering can account for random variations in runtime as well as measurement error.

3. EXPERIMENTAL METHODOLOGY

3.1. Data Collection

We collected static and dynamic execution characteristics data from functions in SPEC CPU 2006 suite. The static information (viz., instruction mix and CDFG

representations) were collected using a custom LLVM pass operating on code compiled to bitcode with "-O0 -g" options. Optimization is mostly turned off as we want to analyze code which is closely related to the original source code; optimizations distort this relationhip. We used the "mem2reg"[3] and "basicaa"[4] optimization and analysis passes before invoking our own. Dynamic data flow graphs and load distances were collected using custom PIN [Luk et al. 2005] tools.

Function Profiling Framework. To measure the reactions of functions to optimization, we used performance counters via LiMiT [Demme and Sethumadhavan 2011] to measure the execution times of functions precisely. To account for random variation, we execute programs three times and use the average.

Function Pruning. Data is collected about SPEC functions from four different tools (viz., LLVM, GCC, PIN and profiling tools), each of which have some limitations and caveats e.g., measurement errors of very short functions e.g., less than 10k cycles. To fairly evaluate our representations, we need to execute CENTRIFUGE with a set of functions for which all of this data is available and accurate. As a result of pruning we are left with 628 functions, 1/3 of which are selected for training and the remaining 2/3 are used for evaluation in the following section.

3.2. Evaluation

To evaluate the merit of the CENTRIFUGE methodology and the success of each representation we have proposed, we must judge the utility of its results. To do so, we take a retrospective approach: if the clusters which CENTRIFUGE produces react homogeneously to *existing* optimizations, then there exists some relationship between the clusters and these existing optimizations. As a result, we speculate that these clusters *may* have been useful in discovering these optimizations.

Existing Optimizations. We select four of GCC's optimizations (unswitch-loops, predictive-commoning, gcse-after-reload and tree-vectorize) which can be applied beyond the -O2 level. These are all of the optimizations which are turned on by '-O3' with the exception of inlining-based optimization, which we cannot use since we are measuring each function. We then create 15 different combinations of these four[5] and use them with -O2. We measured the effect of these fifteen different combinations of optimization flags on SPEC.

By evaluating CENTRIFUGE against advanced optimizations (which often have minimal or negative effect on functions) and simultaneously combining them with basic optimizations we make our task both more difficult and more realistic. Newer optimizations are likely to be relatively complex and/or less-than-universally applicable as much of the "low-hanging fruit" has been realized in the simpler optimizations. (Indeed this is the case, as evidenced by the data in Table I, which shows that our selected optimizations have significant effect on a small percentage of our experimental functions.) Should our representations work well only with simpler optimizations, they are less likely to be useful in the future. However, if our clustering is effective with advanced optimizations which are often not effective themselves, this implies that our method has very good resolution.

Additional Clusterings. To provide context for the results, we artificially generate two additional clusterings—random and ideal—which represent loose lower and upper

[3]The mem2reg optimization is necessary to create sane LLVM bitcode from LLVM-GCC's output, which converts all stack variables to pointers instead of using LLVM's SSA form.

[4]Although LLVM has other alias analysis passes, we observed no difference in behavior.

[5]GCC does not allow optimization re-ordering; each optimization can simply be on or off, so $2^4 - 1 = 15$ since we don't use only '-O2'.

Table I.

Optimizations	(Percent of Functions) Speedup ≥ 15%	Slowdown ≥ 15%
-O2 -fpredictive-commoning -fgcse-after-reload	3.1%	6.9%
-O2 -fpredictive-commoning -ftree-vectorize	3.9%	9.1%
-O2 -fpredictive-commoning	2.7%	6.8%
-O2 -fgcse-after-reload	1.9%	9.6%
-O2 -funswitch-loops -fpredictive-commoning -fgcse-after-reload -ftree-vectorize	25.5%	4.4%
-O2 -ftree-vectorize	3.0%	7.5%
-O2 -fpredictive-commoning -fgcse-after-reload -ftree-vectorize	4.1%	9.0%
-O2 -fgcse-after-reload -ftree-vectorize	3.3%	10.2%
-O2 -funswitch-loops -fgcse-after-reload -ftree-vectorize	7.4%	7.5%
-O2 -funswitch-loops	5.0%	7.7%
-O2 -funswitch-loops -ftree-vectorize	33.2%	5.3%
-O2 -funswitch-loops -fpredictive-commoning -fgcse-after-reload	2.5%	8.0%
-O2 -funswitch-loops -fgcse-after-reload	2.5%	7.9%
-O2 -funswitch-loops -fpredictive-commoning	24.4%	3.3%
-O2 -funswitch-loops -fpredictive-commoning -ftree-vectorize	16.2%	9.1%

Of our 628 selected functions, this table shows the percentage of programs that were sped up or slowed down a significant percent (15%) over optimization with just '-O2'.

bounds. Random clusters are produced using a Gaussian random distance function to judge the distance between functions. The ideal clusters were built using the profiling data collected for evaluation. Since they are built and evaluated on the same data, they are close to optimal. This optimality is not guaranteed, however, due to the heuristic nature of agglomerative clustering. Additionally, we construct other clusters using randomly selected subsets of the evaluation data. These "Existing (1/2)" and "Existing (1/3)" clusterings use one-half and one-third of the optimizations, respectively, for construction but are evaluated on all of the optimizations. They are intended to judge how well optimization reaction data generalizes to other optimizations.

Savings. In our cluster analyses, we parameterize groups of clusters by savings. Recall that "savings" indicates the reduction in number of functions to be examined after clustering and is defined as $(1 - c/f) \cdot 100$ for f functions grouped into c clusters. The savings level also represents a trade off: the higher the savings level, there are fewer clusters, but the functions in each cluster are less likely to be similar. At the extreme high end, all functions are in a single cluster, and thus there is no interesting information. At the extreme low end, each cluster contains one function, so there is also no interesting information. The CENTRIFUGE user must determine an appropriate point.

4. RESULTS

We design statistical tests to answer the following two questions and qualitatively evaluate two more.

(1) Are similarity distances (as determined by each representation) within random and ideal clusters different from global similarity distances? If a representation captures characteristics pertinent to the optimizations, we expect that representation to produce small distances between functions in ideal clusters. As a sanity check, we expect distances within random clusters to be little different from the global set of distances. (Section 4.1)
(2) How consistently does optimization affect functions in each cluster? We would expect functions in clusters which are relevant with respect to existing optimizations to react similarly to optimization. (Section 4.2)

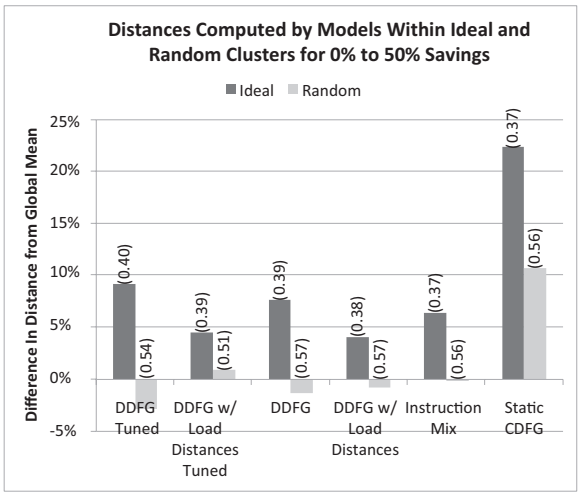

Fig. 4. To measure the acuity of our representations we compare the average distance between clustered functions (in the 0% to 50% range) to the global average distance. We expect the random clusters' averages to be approximately zero and the ideal clusters' averages to be greater than that. Measurements show that random clusters tend to be zero or negative whereas ideal clusters have high differences (T-Test significance values are in parenthesis). Results indicate that our representations' distances are not correlated with random clusters, but are correlated with ideal clusters.

(3) After tuning, what features are most effective? (Section 4.3)
(4) Qualitatively, what type of clusters are produced? (Section 4.4)

4.1. Evaluating Distances in Ideal and Random Clusters

We first wish to determine if our representations' judgment of similarity is interesting with respect to existing optimizations. This test uses each representation's distance function but not the clusters produced using these distances, only the random and ideal clusters. As such, it allows us to test for significance using a minimal amount of clustering and thus eliminate a potential source of error.

For each representation, we compute the average distance between all function pairs. This average global distance indicates how far apart—on average—functions usually are for each representation. We then examine the ideal and random clusterings at each savings level (0%–50%). For each cluster, we look at the distances between each pair of functions, as determined by each representation. We compute an average of these internal distances and subtract that number from the representation's global average. If the cluster is significant with respect to the representation, the cluster's internal distance average will be small and thus this difference will be high. We also run Student's T-Test [Student 1908] to determine the significance of this difference. We further calculate weighted (by cluster size) averages of the mean differences and average T-Test probabilities for each savings range.

The summary results of this analysis are shown in Figure 4. As expected, differences in the random clusters tend to be closer to zero than in the ideal clusters.[6] This result indicates that there is no correlation between random clusters and our representation distances. The fact that differences in ideal clusters are higher indicates that representation distances tend to be smaller in these clusters than the average

[6]In some cases, the bars for random clusters are visibly above or below zero. This is because the distributions are skewed, so random selection is more likely be closer to the global median rather than the global average. The more important comparison is the difference between the two bars.

global distance. Further, the T-Test probability values are smaller for the ideal clusters, showing a greater chance that these differences are significant. On the whole, this test shows that all of the proposed representations have a closer relationship to ideal clusters than random clusters. Although tempting, we cannot determine from this difference data which of the representations is superior. Although normalized using distance averages, these distances are not guaranteed to have similar distributions, so we cannot determine the significance of these values relative to each other.

4.2. Clustering Quality and Implications

Next, we evaluate the consistency of optimization reactions within each cluster. In contrast to the previous test of distances in random and ideal clusters, this test evaluates the clusters generated by each of our representations in addition to the two artificial ones—random and ideal—allowing us to directly compare all of them. To test this consistency, we compute the standard deviation of reaction to optimization (as defined in Eq. (7)) for all functions within each cluster. The intuition behind this metric is simple: good clusters should contain functions which react similarly to optimization. To present these data, we compute the reaction standard deviation for each cluster and average across all the clusters at each savings level.

Figure 5 shows the consistency of optimization reaction for each representation described in this paper, plus the ideal and random clusters. As expected, the consistency of the clusters tends to decrease with savings because the clusters must grow in size, forcing functions with decreasing similarity into the same clusters. This is a direct result of the tradeoff discussed in the above "savings" paragraph.

There are several other interesting things to note in these results.

(1) These results clearly demonstrate that instruction mixes and "Existing (1/3)" are largely worthless for clustering because they are barely better than random.
(2) As expected, using a substantial amount of the evaluation data (Existing 1/2) produces good results. When this quantity is decreased to one-third, however, the results are little better than random. This result implies that optimization reactions themselves are poor predictors of similarity because they do not generalize to other optimizations.
(3) Few of the representations perform well in the very low savings range (0% to 25%) compared to ideal. Although subtle, this affirms a widely-held belief that performance is extremely difficult to predict accurately. Although several of the representations are able to predict large performance changes, none can do so at the accuracy required to perform well in this regime.
(4) Static CDFG fairs very poorly overall but perform well in the small savings range (0% to 25%). This result implies that Static CDFGs are very useful for identifying identical functions, but poor for gauging approximate function similarity. We speculate that this is due to weaknesses in alias analysis and as a result, Static CDFGs contain too many edges to be useful.

Overall, our proposed representations are a mixed bag: some perform well and some do not. In general, representations utilizing dynamic data flow graphs perform well; the area under the tuned dynamic data flow graph representation curve is 80% closer to the area under the ideal curve than that of the random curve! This result is encouraging and confirms the utility of CENTRIFUGE: clusters of functions which react similarly to optimization can be built using generic representations.

Discussion of Optimizations. We can also examine our results' relationship to the optimizations we are using for evaluation—predictive-commoning, tree-vectorize, unswitch-loops, and gcse-after-reload. Predictive commoning examines loops and

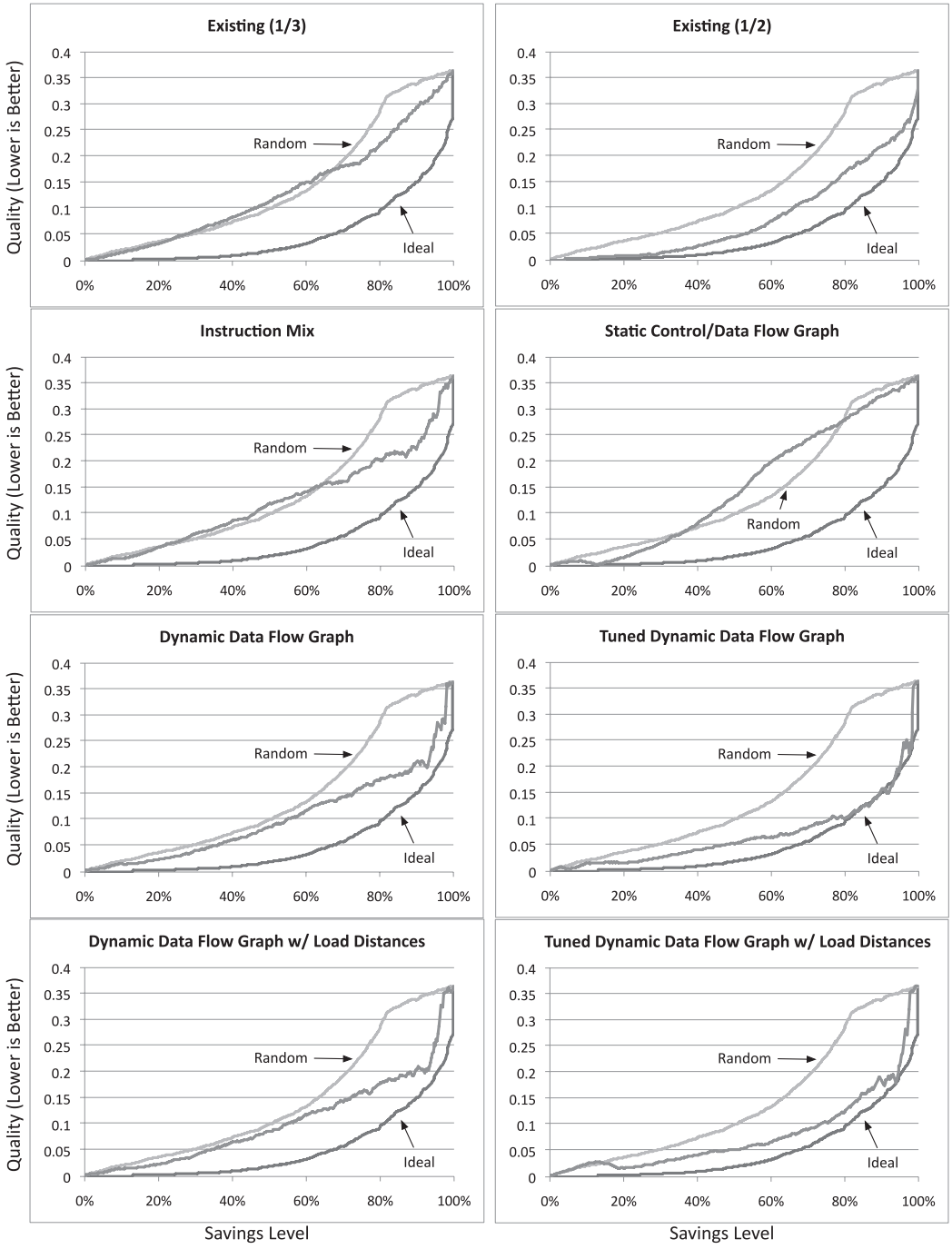

Fig. 5. Consistency of optimization reactions for clusters from various representations. Consistency is calculated as standard deviation, so lower numbers are better. Each clustering is shown relative to loose upper and lower bounds, "ideal" and "random". Although none of the representations perform perfectly, we see that "Tuned Dynamic Data Flow Graph" performs very well.

Table II.

Parameter	DDFG	DDFG w/ LD
Basic Block Annotations		
Dependence Chain Length	1.0	0
Number of Int Instructions	1.0	0.68
Number of FP Instructions	0.64	0.95
Number of Memory Loads	0	0.26
Graph Mapping		
Neighborhood Score (vs. local similarity score)	0.46	1.0
Graph Scoring		
Difference in Local Similarity	0.05	0
Difference in Edges	1.0	0.54
Unmatched Annotation Penalty	0	0.18
Unmatched Edge Penalty	0.64	0.33
Load Distances		
Graph Similarity Score	—	0.45
Difference in LD Means	—	0.84
Difference in LD Std. Dev.	—	0

Parameter values for dynamic data flow graphs and dynamic data flow graphs with load distances after tuning them via simulated annealing. Area under the curves of Figure 5 was used as the objective function to minimize.

attempts to pull out redundant computations. Tree vectorization is GCC's auto vectorizer, intended to create parallel SSE code from normal loops. Unswitch loops pulls conditional statements out of loops allowing the loop to be optimized with less hindrance. Finally, GCSE after reload invokes another subexpression elimination with the goal of eliminating redundant loads. In each case, there are characteristics in the code which determine whether or not the optimization can be applied and how performance will be affected. The results of some of our representations imply that they are capturing some of these characteristics whereas others are not.

With regard to our DDFG (dynamic data flow graph) representation, what features relevant to these optimizations could it be capturing? All of these optimizations deal with movement (or elimination) of operations and thus the optimizations sensitive to both control and data dependences—if certain dependencies exist, the optimization cannot be applied. Alternatively, if certain dependence patterns do exist, the first three optimizations may be applied and strongly affect performance.

We also add load distances to indicate data locality for one representation. Although it does not seem to have a strong effect on our results, it is possible that it could related strongly to unswitch loops and GCSE optimizations. Both eliminate redundant operations (which are likely to have memory operations); since these operations are likely to occur often and are guaranteed to access the same memory, they will likely have very good locality. As such, low load distances may weakly indicate that these optimizations would apply.

Lastly, it is not surprising that instruction mix does poorly; merely knowing that a function has memory loads or integer calculations tells us nothing about whether or not the operations can be moved.

Although our results are specific to these four optimizations, we suspect that dependencies are key to most complex optimization, thus our DDFG representations are likely to scale well.

4.3. Evaluation of Parameter Tuning

As discussed in Section 2.3, "Parameter Tuning", we use simulated annealing to optimize parameters used for our graph comparisons—dynamic data flow graphs and dynamic data flow graphs with load distances (locality information). The resulting weights are shown in Table II. Though potentially interesting, these results have no

Table III.

Cluster #1

```
int CCTK_NumTimeLevelsFromVarI (int var) {
 return ((0 <= var && var < total_variables) ?
  groups[group_of_variable[var]].n_timelevels:
    -1);
}
int CCTK_GroupTypeFromVarI (int var) {
 return ((0 <= var && var < total_variables) ?
  groups[group_of_variable[var]].gtype :
    -1)
}
```

```
int ETree_frontSize (ETree *etree, int J) {
 if ( etree == NULL || J < 0 ||
      J >= etree->nfront ) {
    fprintf(stderr, "\n fatal error in"
            "ETree_frontSize(%p,%d)\n"
            " bad input\n", etree, J);
    exit(-1);
 }
 return(etree->nodwghtsIV->vec[J]);
}
```

An example cluster in which all functions are guarded accessors. The first two are from 436.cactusADM and the last is from 454.calculix. In these clusters, conditions are checked before returning data from a structure. The error case, however, is different—two return error codes, one aborts. Error cases are uncommon, so they have little effect on performance, and may provide an (de)optimization opportunity.

guarantee of optimality as they are found in continuous in 12-dimensional space. Although not optimal these weights were used to generate the clusters evaluated and presented here and evaluate better than untuned representations. There are several interesting observations to make about the weights.

(1) The dependence chain length within a basic block is hugely important to gauging similarity unless load distances are being considered. Further, the number of memory loads in each basic block are not a good indicator of similarity unless load distances are also considered.
(2) Although the similarity between basic blocks is used in the graph mapping stage, they are not used in scoring. Instead, the graphical edge similarity (number of matched/unmatched edges) is used. As a result, graphs of different sizes are penalized far less with out tuned models than untuned.
(3) When integrating load distances, the difference in standard deviations is not used, however the difference in log load distance averages is given nearly twice the weight as the graph similarity. This indicates that data locality is important in judging similarity.

4.4. Qualitative Evaluation of Cluster Results

The results presented in Figure 5 show that tuned dynamic data flow graphs tend to do a reasonable job clustering functions which will react similarly to optimization. So, what sort of functions get grouped together? To answer this question, we examine clusters produced around the 15% to 20% savings level. In some clusters, we see functions which are nearly duplicates. Others have obvious patterns—these *may* indicate potential widely-applicable opportunities for optimization. Other clusters contain functions which do not appear similar, yet react similarly to optimization and have similar data flow graphs. Here are some examples.

Near-Duplicates. Programs like GCC tend to have many functions which are autogenerated and thus nearly identical. Many of these near-duplicates are very small functions (often constructors) with a single basic block. These clusters are largely uninteresting as they are best optimized via inlining and in-context optimization.

Guarded Accessors. By far, the most common pattern we see clustered is a pattern we call "guarded accessors". Although class field accessors are thought to be a pattern used primarily in object-oriented languages, we also see in C that many functions are created to conditionally get or mutate a data structure. For an example, see Table III.

Table IV.

Cluster #2

```
tree nreverse (tree t) {
  tree prev = 0, decl, next;
  for (decl = t; decl; decl = next) {
     next = TREE_CHAIN (decl);
     TREE_CHAIN (decl) = prev;
     prev = decl;
  }
  return prev;
}
```

```
void type_hash_add (unsigned int hashcode,
                    tree type) {
  struct type_hash *h;   void **loc;

  h = (struct type_hash *)
        ggc_alloc (sizeof (struct type_hash));
  h->hash = hashcode;
  h->type = type;
  loc = htab_find_slot_with_hash(type_hash_table,
              h, hashcode, INSERT);
  *(struct type_hash **) loc = h;
}
```

An example cluster in which functions react similarly to optimization, but do not appear similar. Despite the dissimilarity, these two functions have identical dynamic data flow graphs and react similarly to optimization.

These clusters represent a common behavior that one might be able to optimize. First, all of them have a very uncommon branch case—the error conditions—where performance does not matter. Second, the conditions being checked have few side effects (with the exception of NULL checks), so they can be evaluated in any order. These uncommon cases may be detected via profiling. Alternatively, it may be reasonable to assume simple return values like -1 or not returning (like the exit call) are uncommon cases. Further, the functions shown here are not directly affected much by the four optimizations we are applying, and thus may represent a new optimization opportunity.

Non-Intuitive Clusters. There is another set of clusters which contain functions which react similarly to optimization, but it is not intuitively (or obviously) clear why. Table IV shows an example—one function which reverses a list and another that inserts a record into a hash table. While one has a loop (which is likely not unrolled as it is pointer chasing) the other has no loop in either its body nor function calls. One allocates memory and makes a function call, the other is a terminal in the call chain. What do these functions have in common? First, their dynamic data flow graphs are identical, have very similar edge weights and their basic block annotations are similar—they have very few integer and floating point calculations but have memory loads in some basic blocks. The similar memory patterns mean that similar data placements, layouts or prefetching strategies may work similarly on both functions. This class of clusters is probably the most interesting; it shows similarity that likely would not have been recognized during manual inspection of the code, yet our generic representation and profiling of optimized code shows similar behavior and reactions to optimization.

5. RELATED WORK

Related research topics include performance prediction mechanisms, machine learning applications in optimization, program behavior analysis and code mining for topic analysis and microarchitectural enhancement, and computational kernel classification.

Several works [Dubach et al. 2007; Cavazos et al. 2006; Hoste et al. 2006] attempt to predict the reaction of code to various program transformations using code features, profiling information from subsets of possible program transformations and dynamic program characteristics (like instruction mix and strides), respectively. These works, however, are based entirely on overall program speedup and whole program analysis. While there are some similarities to our work (in spirit) none of these works discuss clustering based on the program features. These works are largely motivated by the

problem of determining if/when an optimization should be applied during compilation and not for characterizing program behavior.

A large body of work [Leather et al. 2009; Cavazos 2008; Alvincz and Glesner 2009; Stephenson and Amarasinghe 2005; Stephenson et al. 2003; Agakov et al. 2006; Fursin and Temam 2009] attempts to apply machine learning techniques to compiler optimizations. The bulk of this work attempts to improve existing optimizations and their associated heuristics via machine learning or find better combinations of program optimizations (phase ordering.) While some of them implicitly use clustering, none of these cluster on graphical formats which this work shows to be advantageous.

The software engineering community has several works [Kuhn et al. 2007; Ugurel et al. 2002; Tangsripairoj and Samadzadeh 2005] in which functions are clustered to identify functions which have similar keywords or are semantically similar. These efforts use textual analysis, so there is little reason to believe these analysis techniques could be relevant to program characterization as minor changes in source code (e.g., changing variable names) do not typically affect optimization or performance behavior.

The software engineering community has also long worked to identify "copy and paste" code. Many approaches [Li et al. 2006; Kamiya et al. 2002; Baxter et al. 1998; Gabel et al. 2008; Krinke 2001; Pham et al. 2009; Kontogiannis 1993; Basit and Jarzabek 2009; Smith and Horwitz 2009] have been developed, nearly all of which rely solely on static code analysis. Typically, these tools are not designed to calculate similarity—they only detect when code has been copied, thus any graph matching algorithms they use do not require the same level of approximation our approch provides.

In the architecture community there is also some relevent work in both benchmark selection and program graph mining. Several papers [Joshi et al. 2006; Phansalkar et al. 2007; Eeckhout et al. 2003] use analysis to find redundancy in sets of benchmark programs. These techniques can be used for benchmark selection however they operate at the granularity of an entire program. As a result, this work is largely complimentary and in fact was indirectly used to select benchmarks for this paper as [Phansalkar et al. 2007] was used to create SPEC06. Another set of papers [Hormati et al. 2007; Clark et al. 2003; Clark et al. 2006] use program mining to assist in instruction set customization. In this work, Clark et al. examine and find common patterns in graphs, however their techniques work to find very small patterns—several instructions—only rather than function-granularity patterns and idioms.

Another effort to recognize patterns in code is XARK [Arenaz et al. 2008]. XARK's is able to classify loop structures into several categories of computational kernels types such as inductions, maps and scalar assignments. CENTRIFUGE is distinctly different as it computes approximate similarity and hierarchically clusters functions. Other work [Gupta et al. 2010] on design pattern mining uses inexact graph matching, an approach similar to ours. It uses a different approximate graph matching algorithm, however, and operates on UML graphs rather than automatically collected data.

6. CONCLUSION

In this paper, we proposed clustering on graphical intermediate program representations for program characterization. We used approximate graph similarity to drive our graph clustering. To evaluate the effectiveness of such clustering we designed a framework called CENTRIFUGE that groups functions based on common static and dynamic characteristics. We have shown that functions grouped by graphical properties tend to react similarly to several existing optimizations. These results indicate that (1) it is possible to classify code snippets into behavioral groups which react similarly to optimization and (2) that clustering on graphical representations produces better results compared to static non-graphical formats. Further, based on manual analysis of some of clustered functions we determine that there is potential for discovering interesting

patterns and thus our techniques may actually be useful for program characterization. This process of automatic behavior discovery could be an important step towards automatic optimization discovery.

ACKNOWLEDGMENTS

We thank Prof. Alfred Aho, Melanie Kambadur, anonymous reviewers, and members of the Computer Architecture and Security Technologies Lab (CASTL) at Columbia University for their feedback on this work.

REFERENCES

AGAKOV, F., BONILLA, E., CAVAZOS, J., FRANKE, B., FURSIN, G., O'BOYLE, M. F. P., THOMSON, J., TOUSSAINT, M., AND WILLIAMS, C. K. I. 2006. Using machine learning to focus iterative optimization. In *CGO'06: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Los Alamitos, CA, 295–305.

ALVINCZ, L. AND GLESNER, S. 2009. Breaking the curse of static analyses: Making compilers intelligent via machine learning. In *Proceedings of the SMART'09 Workshop*.

ARENAZ, M., TOURIÑO, J., AND DOALLO, R. 2008. Xark: An extensible framework for automatic recognition of computational kernels. *ACM Trans. Program. Lang. Syst. 30*, 6, Article 32.

ARVIND AND IANNUCCI, R. A. 1987. Two fundamental issues in multiprocessing. In *Parallel Computing in Science and Engineering*, 61–88.

BASIT, H. A. AND JARZABEK, S. 2009. A data mining approach for detecting higher-level clones in software. *IEEE Trans. Softw. Eng. 35,* 4, 497–514.

BAXTER, I. D., YAHIN, A., MOURA, L., SANT'ANNA, M., AND BIER, L. 1998. Clone detection using abstract syntax trees. In *ICSM'98: Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, Los Alamitos, CA, 368.

CAVAZOS, J. 2008. Intelligent compilers. In *Proceedings of the IEEE International Conference on Cluster Computing*. 360–368.

CAVAZOS, J., DUBACH, C., AGAKOV, F., BONILLA, E., O'BOYLE, M. F. P., FURSIN, G., AND TEMAM, O. 2006. Automatic performance model construction for the fast software exploration of new hardware designs. In *CASES'06: Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM, New York, 24–34.

CLARK, N., HORMATI, A., MAHLKE, S., AND YEHIA, S. 2006. Scalable subgraph mapping for acyclic computation accelerators. In *CASES'06: Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM, New York, 147–157.

CLARK, N., ZHONG, H., AND MAHLKE, S. 2003. Processor acceleration through automated instruction set customization. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Los Alamitos, CA, 129.

DEMME, J. AND SETHUMADHAVAN, S. 2011. Rapid identification of architectural bottlenecks via precise event counting. In *Proceeding of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, 353–364.

DING, C. AND ZHONG, Y. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*. ACM, New York, 245–257.

DUBACH, C., CAVAZOS, J., FRANKE, B., FURSIN, G., O'BOYLE, M. F., AND TEMAM, O. 2007. Fast compiler optimisation evaluation using code-feature based performance prediction. In *CF'07: Proceedings of the 4th International Conference on Computing Frontiers*. ACM, New York, 131–142.

EECKHOUT, L. 2010. Computer architecture performance evaluation methods. In *Synthesis Lectures on Computer Architecture*, Morgan Claypool.

EECKHOUT, L., VANDIERENDONCK, H., AND BOSSCHERE, K. D. 2003. Quantifying the impact of input data sets on program behavior and its applications. *J. Instruction-Level Parall. 5*, 1–33.

FURSIN, G., MIRANDA, C., TEMAM, O., NAMOLARU, M., YOM-TOV, E., ZAKS, A., MENDELSON, B., BARNARD, P., ASHTON, E., COURTOIS, E., BODIN, F., BONILLA, E., THOMSON, J., LEATHER, H., WILLIAMS, C., AND O'BOYLE, M. 2008. Milepost gcc: Machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*.

FURSIN, G. AND TEMAM, O. 2009. Collective optimization. In *HiPEAC'09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*. Springer-Verlag, Berlin, 34–49.

GABEL, M., JIANG, L., AND SU, Z. 2008. Scalable detection of semantic clones. In *ICSE'08: Proceedings of the 30th International Conference on Software Engineering*. ACM, New York, 321–330.

GUPTA, M., SINGH RAO, R., AND TRIPATHI, A. 2010. Design pattern detection using inexact graph matching. In *Proceedings of the International Conference on Communication and Computational Intelligence (INCOCCI)*. 211–217.

HORMATI, A., CLARK, N., AND MAHLKE, S. 2007. Exploiting narrow accelerators with data-centric subgraph mapping. In *Proceedings of the International Symposium on Code Generation and Optimization. CGO'07*. 341–353.

HOSTE, K., PHANSALKAR, A., EECKHOUT, L., GEORGES, A., JOHN, L. K., AND DE BOSSCHERE, K. 2006. Performance prediction based on inherent program similarity. In *PACT'06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*. ACM, New York, 114–122.

JOSHI, A., PHANSALKAR, A., EECKHOUT, L., AND JOHN, L. 2006. Measuring benchmark similarity using inherent program characteristics. *Comput. IEEE Trans. 55,* 6, 769–782.

KAMIYA, T., KUSUMOTO, S., AND INOUE, K. 2002. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng. 28,* 7, 654–670.

KONTOGIANNIS, K. 1993. Program representation and behavioural matching for localizing similar code fragments. In *CASCON'93: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 194–205.

KRINKE, J. 2001. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*. 301–309.

KUHN, A., DUCASSE, S., AND GÍRBA, T. 2007. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol. 49,* 3, 230–243.

KUHN, H. W. 1955. The hungarian method for the assignment problem. *Naval Resear. Logistics Quart.* 83–97.

LATTNER, C. 2002. LLVM: An Infrastructure for multi-stage optimization. M.S. thesis, Computer Science Department University of Illinois at Urbana-Champaign. http://www.llvm.org.

LEATHER, H., BONILLA, E., AND O'BOYLE, M. 2009. Automatic feature generation for machine learning based optimizing compilation. In *CGO'09: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, Los Alamitos, CA, 81–91.

LI, Z., LU, S., MYAGMAR, S., AND ZHOU, Y. 2006. CP-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng. 32,* 3, 176–192.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. PIN: Building customized program analysis tools with dynamic instrumentation. In *PLDI'05: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 190–200.

MELNIK, S., GARCIA-MOLINA, H., AND RAHM, E. 2002. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings of the 18th International Conference on Data Engineering*. 117–128.

MOCK, M., DAS, M., CHAMBERS, C., AND EGGERS, S. J. 2001. Dynamic points-to sets: a comparison with static analyses and potential applications in program understanding and optimization. In *PASTE'01: Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM, New York, 66–72.

NAMOLARU, M., COHEN, A., FURSIN, G., ZAKS, A., AND FREUND, A. 2010. Practical aggregation of semantical program properties for machine learning based optimization. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'10)*.

PAN, Z. AND EIGENMANN, R. 2008. PEAK—A fast and effective performance tuning system via compiler optimization orchestration. *ACM Trans. Program. Lang. Syst. 30*, 3, Article 17.

PHAM, N. H., NGUYEN, H. A., NGUYEN, T. T., AL-KOFAHI, J. M., AND NGUYEN, T. N. 2009. Complete and accurate clone detection in graph-based models. In *ICSE'09: Proceedings of the IEEE 31st International Conference on Software Engineering*. IEEE Computer Society, Los Alamitos, CA, 276–286.

PHANSALKAR, A., JOSHI, A., AND JOHN, L. K. 2007. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM, New York, 412–423.

SMITH, R. AND HORWITZ, S. 2009. Detecting and measuring similarity in code clones. In *Proceedings of the International Workshop on Software Clones (IWSC)*.

STEPHENSON, M. AND AMARASINGHE, S. 2005. Predicting unroll factors using supervised classification. In *CGO'05: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Los Alamitos, CA, 123–134.

STEPHENSON, M., AMARASINGHE, S., MARTIN, M., AND O'REILLY, U.-M. 2003. Meta optimization: Improving compiler heuristics with machine learning. *SIGPLAN Not. 38,* 5, 77–90.

STUDENT. 1908. The probable error of a mean. *Biometrika*, 1–25.

TANGSRIPAIROJ, S. AND SAMADZADEH, M. H. 2005. Organizing and visualizing software repositories using the growing hierarchical self-organizing map. In *SAC'05: Proceedings of the ACM Symposium on Applied Computing*. ACM, New York, 1539–1545.

TRIANTAFYLLIS, S., BRIDGES, M. J., RAMAN, E., OTTONI, G., AND AUGUST, D. I. 2006. A framework for unrestricted whole-program optimization. In *PLDI'06: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 61–71.

UGUREL, S., KROVETZ, R., AND GILES, C. L. 2002. What's the code? automatic classification of source code archives. In *KDD'02: Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, 632–638.

WARD, J. H. J. 1963. Hierarchical grouping to optimize an objective function. *J. Amer. Statis. Assn.* 236–244.

Received July 2011; revised October 2011 and December 2011; accepted December 2011