Using Graph-Based Program Characterization for Predictive Modeling

Eunjung Park, John Cavazos
Department of Computer and Information Sciences
University of Delaware
{epark,cavazos}@cis.udel.edu

Marco A. Alvarez
Computer Science Department
Utah State University
marco.alvarez@usu.edu

Abstract

Using machine learning has proven effective at choosing the right set of optimizations for a particular program. For machine learning techniques to be most effective, compiler writers have to develop expressive means of characterizing the program being optimized. The current state-of-the-art techniques for characterizing programs include using a fixed-length feature vector of either source code features extracted during compile time or performance counters collected when running the program. For the problem of identifying optimizations to apply, models constructed using performance counter characterizations of a program have been shown to outperform models constructed using source code features. However, collecting performance counters requires running the program multiple times, and this "dynamic" method of characterizing programs can be specific to inputs of the program. It would be preferable to have a method of characterizing programs that is as expressive as performance counter features, but that is "static" like source code features and therefore does not require running the program.

In this paper, we introduce a novel way of characterizing programs using a *graph-based characterization*, which uses the program's intermediate representation and an adapted learning algorithm to predict good optimization sequences. To evaluate different characterization techniques, we focus on loop-intensive programs and construct prediction models that drive polyhedral optimizations, such as auto-parallelism and various loop transformation.

We show that our graph-based characterization technique outperforms three current state-of-the-art characterization techniques found in the literature. By using the sequences predicted to be the best by our graph-based model, we achieved up to 73% of the speedup achievable in our search space for a particular platform, whereas we could only achieve up to 59% by other state-of-the-art techniques we evaluated.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages; D.3.4 [*Programming Languages*]: ProcessorsCompilers, Optimization; I.2.6 [*Artificial intelligence*]: Learning

General Terms Performance, Experimentation, Languages

Keywords compiler optimization, iterative compilation, machine learning, support vector machine, graph-based program characterization

CGO'12 Mar 31 – April 4, 2011, San Jose, CA, USA.
Copyright © 2011 ACM [to be supplied]. . . $10.00

1. Introduction

Using a well-constructed machine learning model to choose optimizations for a specific program has repeatedly been shown to outperform the most aggressive optimization levels in open-source and commercial compilers [6, 10, 13, 15, 16, 19, 20, 22, 25, 30]. However, to use machine learning effectively, it is critical to use expressive features that characterize programs well and that strongly correlate to beneficial optimization sequences for the target program. Previous work has introduced a variety of different ways to characterize programs based on either static or dynamic program characteristics. Static characterizations of a program includes collecting features from a program's source code or intermediate representation [16, 19]. Methods to dynamically characterize a program have consisted of either using performance counters or a technique known as "reactions." Performance counters can be used to collect information, such as cache behavior, functional unit usage, and dynamic instruction mix from the running program [10, 15, 22]. Reactions is a technique to dynamically characterize a program by selecting specific program transformations to apply to a program and using the resulting speedups to characterize a program's behavior [9, 27].

In previous work, static or dynamic features have been represented as structured data, usually as fixed-length feature vectors. Also, previous work has shown that models using dynamic characterizations out-perform models using static characterizations [10]. However, dynamic characterizations have disadvantages over static characterizations. To collect this dynamic information from a program, the application must be run at least once, which increases training time to construct prediction models and adds an additional cumbersome profiling step to the compilation process. Moreover, dynamic characterizations are sensitive to a program's input because the information was collected during a program run.

In this paper, we introduce a novel method of using the program's graph-based intermediate representation (IR) and an adapted machine learning algorithm to predict optimization sequences that will benefit a program. A program's graph-based IR is a static characterization technique because it is collected during the compilation of the program. Also, our learning technique uses the topology of the IR, and we therefore represent the IR in an unstructured manner, i.e., not using a fixed-length feature vector. We compared the method introduced in this paper to three state-of-the-art characterization methods from the literature. Our graph-based characterization methods gives significantly better average performance over the other three characterization methods we evaluated.

This paper is organized as follows. In Section 2, we describe the different program characterization techniques, including our proposed graph-based characterization method with a motivating example of using our proposed technique. In Section 3, we give an

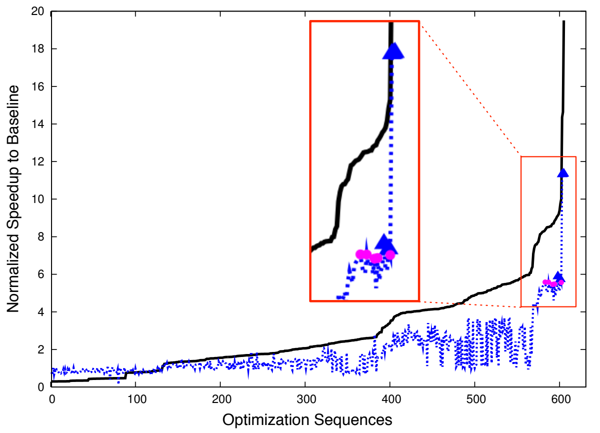


Figure 1. This graph shows actual speedups (solid line) for 600 sequences applied to the program 2MM. We also show predicted speedups (dotted line) for those same sequences obtained from a prediction model that is not trained with 2MM and that uses our graph-based characterization. The x-axis shows optimization sequences sorted by increasing order of actual speedup, and the y-axis shows speedup obtained after applying a sequence over some baseline (ICC -fast). Triangles correspond to the top 5 predicted sequences, and circles correspond to next best 5 predicted sequences.

overview of our solution giving more details of how we construct and use the prediction models. We explain our experiment setup in Section 4 and demonstrate results and present analysis in Section 5. In Section 6, we explain and compare it to related work. Section 7 presents our conclusions and future work.

2. Characterizing the Program

Compiler researchers have proposed to use machine learning models to focus search on beneficial areas of the optimization search space [6, 10, 11, 14, 20, 25]. An important step in using these models is to characterize (or construct features for) the programs being optimized. Finding the "right" set of features and a good representation for this information is one of the most challenging, and probably the least automatable, part of the process.

Compiler researchers have used fixed-length representations of the program's source code features and intermediate representation [6, 20, 25]. These representations are straight-forward to extract from a program and can be collected during compilation time. However, these representations are less expressive and are often out-performed by more expensive dynamic techniques. Other researchers have proposed using dynamic characterizations of programs; however, techniques (e.g., performance counters [10] and reactions [15, 27]) are expensive and require running the program, which limits their practical use.

In this section, we motivate the applicability of using the program's intermediate representation as unstructured data as input for machine learning for finding good optimization sequences.

2.1 Different Ways to Characterize the Program

Program characterization techniques can be grouped into two categories: static or dynamic methods. The benefit of using a static characterization is that we do not need to run a program. We can capture these features during the compilation process or by parsing the source code itself. Previous work on static features has focused mainly on collecting source code features, such as instruction mix, loop nest depth, etc [6]. Additionally, there has been work on collecting simple summary statistics derived from a program's intermediate representation, e.g., number of nodes and edges, number of phi nodes, etc. [16]. Previous work has put this static information into fixed-length vectors, which is essential for traditional supervised learning techniques. However, summarizing this data into fixed-length vectors causes vital information to be lost, such as the control flow through the program.

More recently, researchers have looked into characterizing programs using dynamic information, e.g., using performance counters or optimization reactions. Using performance counters, we can collect multiple characteristics of the running program, such as the behavior of the cache at each level, number of stall cycles, or the mix of different instructions executing, etc. Another method of dynamically characterizing a program is through *reactions*. Here, we record the performance impact from a fixed set of different optimization sequences [9, 15]. Thus, we run a program *K* number of times by optimizing it with *K* different optimization sequences, and then obtain different speedups for each sequence. These reactions are used to characterize the program. These dynamic techniques have been shown to empirically out-perform static features. However, collecting dynamic information requires running the program at least once and in many cases multiple times to collect these features.

In this paper, we propose a novel technique to characterize a program using its *graph-based* intermediate representation (IR). In particular, we use the program's control flow graph (CFG) as the basis for our graph-based characterization. We also include in this characterization information about each instruction at each CFG node. We convert this characterization into a format that can be fed into an adapted machine learning algorithm. This is a static characterization and therefore does not require running the program. The main difference between our technique and previous static techniques is that we encode the topology of the program's IR into our characterization. Given this novel characterization, we can produce models that predict optimization sequences that out-perform sequences predicted by models using other characterization techniques. We also experimented with other graph-based IRs for program characterization, and we present these results in Section 5.3.

Figure 1 shows the predictions for one program from a model that uses our graph-based characterization technique. The solid line shows actual speedup (sorted) optimizing 2MM from the PolyBench suite [3] with random optimization sequences. We randomly generated 600 different optimization sequences consisting of different loop/parallelization transformations from the PoCC source-to-source compiler [2], and we used the ICC compiler with option -fast as the backend compiler to produce an executable from the optimized output source code of PoCC. To obtain predictions for a model that uses our graph-based characterization, we performed leave-one-out cross-validation to construct a model. We constructed the model by training it with the PolyBench programs and by leaving out the benchmark 2MM. Our training data consisted of each program's graph-based CFG characterization, the above mentioned 600 optimization sequences, and the speedups obtained from these sequences. We constructed our models using support vector machines (SVM). We discuss training of our models more thoroughly in Section 3.

We observe in Figure 1 that the predicted speedup follows the trend of the actual speedup similarly although not perfectly. This means there is a high probability that we will get a performance improvement by using a sequence (x-value) with a high predicted speedup (y-value). If we look closer at the top 10 predicted sequences and their predicted y-values (marked by blue triangles and purple circles in the figure), these sequences are among the best sequences in terms of their actual speedup. In particular, the best predicted sequence from CFG model is the actual best sequence, which means we achieve a speedup of around 19×. In contrast, when we train a model for the same machine, but use performance counters instead of our CFG characterization, we only achieved around 4× if we use the best predicted optimization sequence. This preliminary experiment shows the potential of using a graph-based representations to characterize programs for predictive modeling.

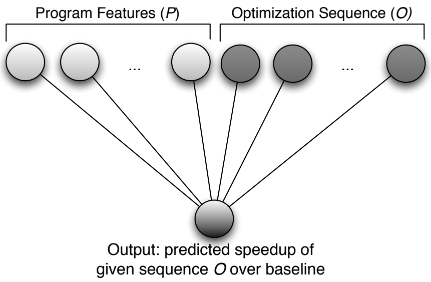

Program Features (*P*)
Optimization Sequence (*O*)
...
...
Output: predicted speedup of
given sequence *O* over baseline

Figure 2. Speedup Predictor

3. Automatically constructing a model

This section describes our prediction model and a detailed description of the different characterization methods we evaluated. We also give an explanation of the machine learning algorithm used for our graph-based characterization techniques.

3.1 Speedup Prediction Model

The prediction model used in this paper is shown in Figure 2. This model has been used in recent work [9, 13, 22] to predict good optimization sequences for various different compilers. We refer to this model as the *speedup predictor* because it takes as input a program's characterization (*P*) and an optimization sequence (*O*), and it outputs the predicted speedup over some baseline for that optimization sequence. In this paper, we use the baseline of PoCC with no optimization and the backend compiler with its most aggressive setting -fast for ICC and -O3 for GCC). We evaluate our different characterization methods using the same set of optimization sequences *O* and build a different speedup prediction model for each different characterization method.

We evaluate our different models (characterization methods) by using them in two different scenarios. First, we evaluate them in a *non-iterative* scenario, where we use only the best predicted optimization sequence from each model. This is the typical scenario, in which a developer uses a compiler. Second, we evaluate the different models in an *iterative* scenario, where we optimize our programs with the *N*-best predicted optimization sequences, and we report the speedup obtained by the sequence giving us the best speedup. For this paper, we used $N = 5$ since we observed only a gradual performance improvement after 5 sequences.

3.2 Program characteristics

We collect four different kinds of program features (shown in Figure 4.) to evaluate our different program characterization techniques.

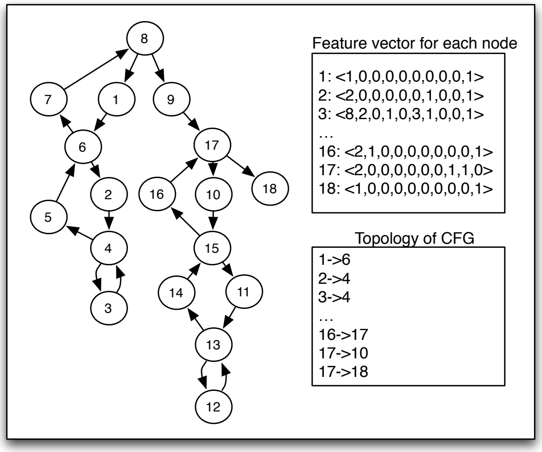
Performance Counters (PC) We collect all PC events available on our target architectures including data/instruction cache behavior, TLB, instruction types, etc. The full list of PCs used in this paper are shown in Table 1, and all values are normalized by total number of instruction in a given architecture. To collect the PC information, we run the program 56 times collecting one counter during each run of the program. Note that we can collect counters using multiplexing, but this may introduce noise. The full list of counter events used are shown in Table 1, and we show the process of collecting PC information on the top-left of Figure 4.

Category of PCs	List of PCs selected
Cache Line Access	CA-CLN, CA-ITV, CA-SHR
Level 1 Cache	L1-DCA, L1-DCH, L1-DCM, L1-ICA, L1-ICH, L1-ICM, L1-LDM, L1-STM, L1-TCA, L1-TCM
Level 2 & 3 Cache	L2-DCA, L2-DCM, L2-DCR, L2-DCW, L2-ICA, L2-ICH, L2-ICM, L2-LDM, L2-STM, L2-TCH L2-TCR, L2-TCW, L2/L3-TCA, L2/L3-TCM
Branch Related	BR-CN, BR-INS, BR-MSP, BR-NTK, BR-PRC, BR-TKN, BR-UCN
Floating Point	DP/FP/SP-OPS, FDV/FML/FP-INS
Interrupt/Stall	HW-INT, RES-STL
TLB	TLB-DM, TLB-IM, TLB-SD, TLB-TL
Total Cycle/Insts.	TOT-CYC, TOT-IIS, TOT-INS
Load/Store Insts.	LD-INS, SR-INS
SIMD Insts.	VEC-DP, VEC-INS, VEC-SP

Table 1. Performance counters (PC): We collected 56 different performance counters available using PAPI library to characterize a program

ft1	Number of Instructions
ft2	Number of Add instruction
ft3	Number of Sub instruction
ft4	Number of Mult instruction
ft5	Number of Div instruction
ft6	Number of Load instruction
ft7	Number of Store instruction
ft8	Number of Comparisons
ft9	Number of Conditional Branches
ft10	Number of Unconditional Branches

Table 2. Control Flow Graph (CFG): We collected 10 different features for each node (basic block) in CFG. We used MinIR to collect this information.



Feature vector for each node

1: <1,0,0,0,0,0,0,0,0,1>
2: <2,0,0,0,0,0,1,0,0,1>
3: <8,2,0,1,0,3,1,0,0,1>
…
16: <2,1,0,0,0,0,0,0,0,1>
17: <2,0,0,0,0,0,0,1,1,0>
18: <1,0,0,0,0,0,0,0,0,1>

Topology of CFG

1->6
2->4
3->4
…
16->17
17->10
17->18

Figure 3. This figure depicts the CFG characterization for 2MM, one of PolyBench programs. We did not include start and exit nodes in CFG since these nodes are empty (do not include instructions).

Reactions We randomly selected a small set *K* of different optimizations sequences and collect the performance impact for each of these sequences on each program. The performance impact (speedup or degradation) of these sequences serves as a signature for a specific program and are used as input to a prediction model. We used *K*=5 for this paper. We show the process of collecting these reactions on the top-right of Figure 4.

Source Code Features (SRC) We depict the process of extracting source code features from programs on the bottom-left of Figure 4. We collect different source code features from the Milepost GCC [16] framework. It provides different static features including properties of the different instructions and variables used in each

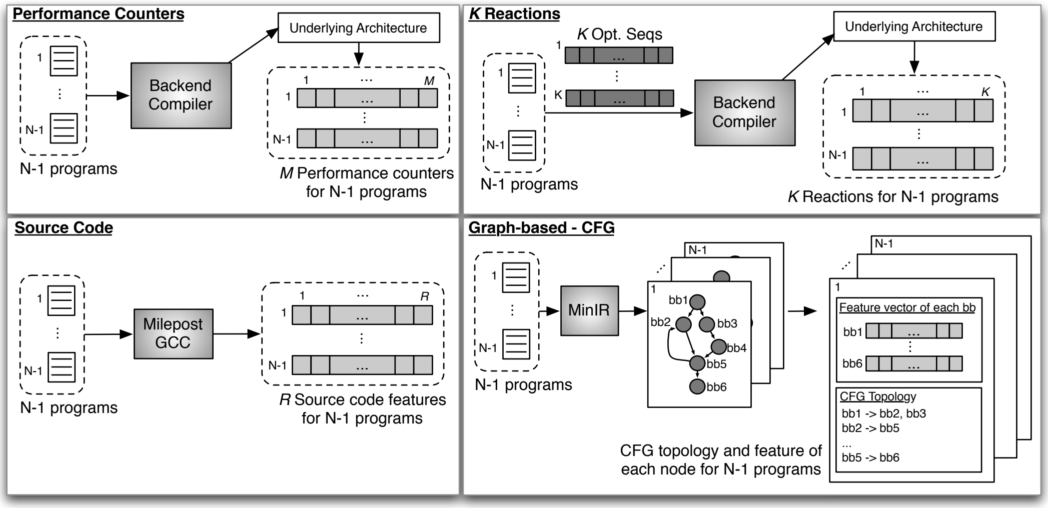


Figure 4. Collecting Different Program Characterizations

function as well as different summary statistics pertaining to basic blocks and edges. We used 34 features from Milepost GCC, which were all the relevant features for our programs, i.e., we removed features that were always zero for all our programs.

Graph-based characterization (CFG) We show the process of extracting our graph-based characterization on the bottom-right of Figure 4. We use MinIR [1] to extract control flow graphs from each of our programs. Before generating our CFGs, we run SSA on the programs to give us the property of having one definition for each variable. From the CFG, we generate graph-based characterization, which includes 1) a feature vector for each basic block in the CFG as shown in Table 2 and 2) a list of directed edges in the graph. MinIR can also be used to extract other graph-based IRs, including the dominator tree, def-use-chain, etc. We describe a preliminary investigation using other graph-based IRs in Section 5.3. Figure 3 shows an example of the CFG of the 2MM program and the extracted graph-based characterization from the graph. Note that an important difference between this characterization and previous static characterizations is the presence of topological information of the graph, which is used by an adapted machine learning algorithm. Also, we collect information per basic blocks, not just a summarization across the entire function, thus providing more fined-grained information to the learning algorithm.

Optimization	Value Used
Unrolling Factor	0 (no unrolling), 2, 4, 8
Loop Fusion	nofuse, maxfuse, smartfuse
Loop Tiling	1 (no tiling), 32
Parallelization	on, off
Vectorization	on, off

Table 3. List of optimization used in this paper. For loop fusion and tiling, our optimization sequences directed which loops to fuse and/or tile. If parallelization is turned on, we parallelize the outer loop. If vectorization is turned on, we vectorize the inner loop.

3.3 Building and Using a Prediction Model

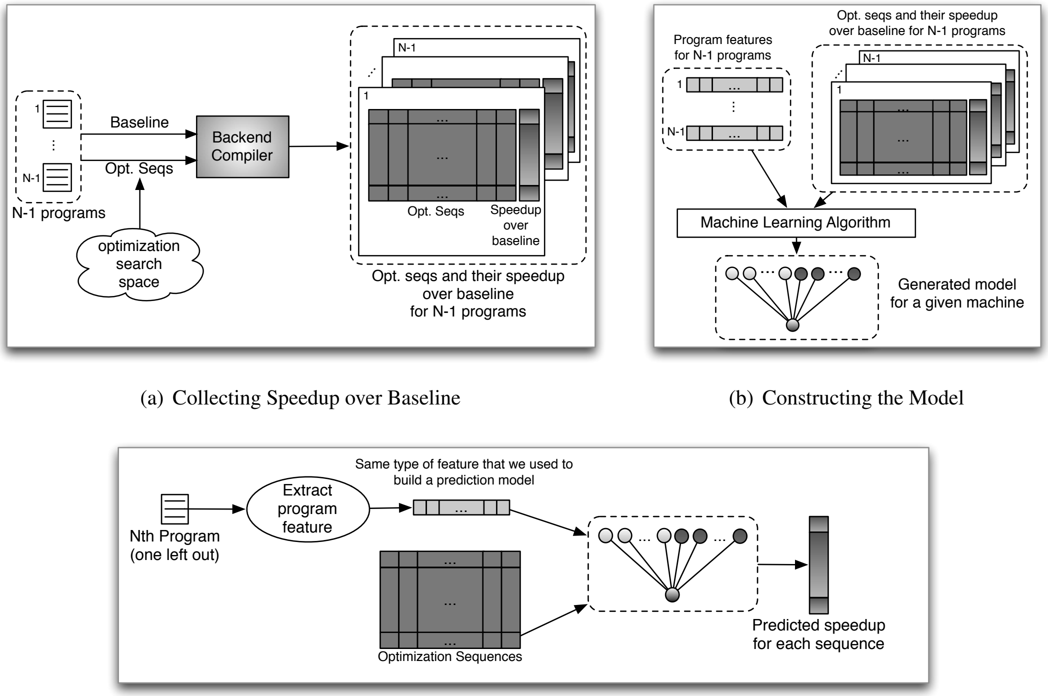
Collecting Speedup over Baseline Table 3 gives a list of the optimizations used to create our optimization search space. We focus on optimizing loops, thus our search space consists of different loop transformations and different level of parallelism, including thread-level and instruction-level parallelism. We exhaustively generated all optimization sequences from this optimization space giving a maximum of 600 optimization sequences, and we evaluate each sequence on our benchmark suite. We collected the speedup obtained for each optimization sequence over the baseline of applying no PoCC optimizations. The process of collecting speedups for each sequence on each program is shown in Figure 5(a). The characterization of each program along with the optimization sequences and their corresponding speedups on each program are used to construct the training data.

Constructing the Model Once the training data is constructed, it can be fed to a learning algorithm that will automatically induce a prediction model as shown in Figure 5(b). We used support vector machines (SVMs) [23] to construct our predictive models. SVMs are a class of machine learning algorithms that can be used for both classification and regression. SVMs use *kernel* functions to transform the training data into a different, linearly-separable feature space, and then a linear classifier is constructed that separates the points into multiple classes.

Using the Model for Unseen Program Figure 5(c) depicts how the model is used on an unseen program to get predictions of the optimizations that will work best for that program. We order the predicted speedups to determine which sequence is predicted best, and apply it to the unseen program. We train our models using *l*eave-one-out cross-validation, i.e., given N programs, and we train a model on N-1 programs and test the model on the Nth program left out.

3.4 Graph-based SVM Kernels

The use of kernel functions is very attractive because the input data does not always need to be expressed as feature vectors. In our control flow graphs, we would like SVMs to perform structural comparisons between different graphs. Note that we cannot simply flatten this information into feature vectors, because this would remove important information about the structure of the graphs. Such information is useful because it allows the learning algorithm to effectively capture the similarities between two different programs.


Baseline
Backend Compiler
Opt. Seqs
1
N-1
N-1 programs
optimization search space
Opt. Seqs
Speedup over baseline
Opt. seqs and their speedup over baseline for N-1 programs
Program features for N-1 programs
Opt. seqs and their speedup over baseline for N-1 programs
Machine Learning Algorithm
Generated model for a given machine
Same type of feature that we used to build a prediction model
Nth Program (one left out)
Extract program feature
Optimization Sequences
Predicted speedup for each sequence

(a) Collecting Speedup over Baseline

(b) Constructing the Model

(c) Using the Model for Unseen Program

Figure 5. This figure depicts how we construct and use our prediction models. This process uses leave-one-out cross validation with *N* programs. We collect speedup over baseline for *N-1* programs from each sequence in optimization search space as shown in (a). Then, we construct the model from the program's characterization, the optimization sequences, and their corresponding speedups (b). We use the model on an unseen program by collecting the program's characterization. We then feed this characterization along with the optimization sequences, and the model gives us predicted speedups for those sequences (c).

Thus, we are interested in kernel functions that can take discrete structures data as inputs, e.g., control flow graphs.

Along these lines, a family of convolution kernels [18] has been introduced as a general framework for building kernels over discrete structures. This general framework is based on the idea of the decomposition of objects into sub-parts in such a way that a kernel function for a pair of objects can be defined as a convolution of kernel functions defined over their sub-parts. Simple closure properties of valid kernel functions (positive semi-definite) allow the definition of kernels in this way. This finding has made great impact, and defining kernels for discrete structures has become an important topic in machine learning [8, 17, 24, 28]. By changing the decomposition, several types of kernels for graphs have been proposed so far, including shortest paths, walks, sub-trees, and cyclic patterns.

Among all recent developments, graph kernels based on shortest paths [7] are a remarkable class of kernel functions. They retain expressivity while comparing graphs at acceptable polynomial time. This class of kernels is also attractive because of their wide applicability. Contrary to other graph kernels, shortest path graph kernels can deal with labeled graphs, where the nodes and edges can be labeled by real values.

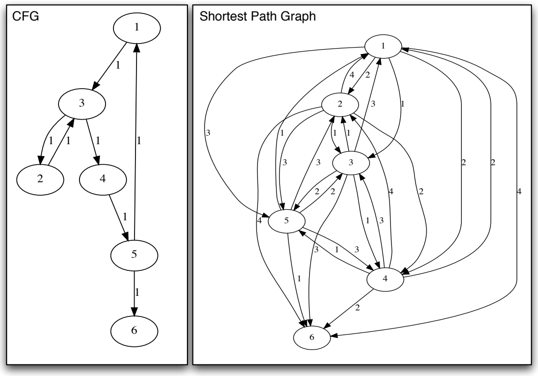


Figure 6. The control flow graph for GESUMMV and its corresponding shortest-path graph. Note: We choose the simplest CFG from the PolyBench benchmark suite so that the shortest-path graph might be readable.

Shortest Path Graph Kernel Roughly speaking, the basic idea of a shortest path graph kernel [7] is to quantify the number of common shortest paths in two input graphs. To this end, prior to the kernel computation, the original graphs must be transformed into shortest path graphs. Given a graph $G = \langle V, E \rangle$, its shortest path graph is denoted by $G_{sp} = \langle V', E' \rangle$, where $V' = V$ and nodes in V' are connected by edges $e' = (u', v')$ if there is a path between nodes u and v in the original graph. Edges are labeled with the length of the shortest distance between u and v in the original graph. This transformation can be performed using any all-pairs shortest path algorithm. In particular, we use the Floyd-Warshall algorithm because it is straightforward and has time complexity of $O(n^3)$. An example of an original CFG and its corresponding shortest-path graph is shown in Figure 6.

Once the original graphs G_1 and G_2 are transformed into shortest path graphs, the shortest path graph kernel function for a pair of graphs $G_{sp1} = \langle V_1, E_1 \rangle$ and $G_{sp2} = \langle V_2, E_2 \rangle$ is defined as follows:

$$K_{sp}(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{walk}(e_1, e_2)$$

where k_{walk} is a kernel function for comparing two walks. A walk includes an edge and its two end nodes. Let e_1 be the edge connecting nodes v_1 and w_1, and e_2 be the edge connecting nodes v_2 and w_2, then k_{walk} is defined as:

$$k_{walk}(e_1, e_2) = k_{node}(v_1, v_2) \cdot k_{edge}(e_1, e_2) \cdot k_{node}(w_1, w_2)$$

where k_{node} is a kernel function for comparing two nodes. Once our control flow graphs have feature vectors at every node, we use the well-known Gaussian kernel [23] for calculating k_{node}. On the other hand, k_{edge} is a kernel function for comparing two edges. We use a Brownian bridge kernel [23] that returns the highest value when two edges have identical length, and 0 when the edges differ in length more than a constant c[1] as shown below:

$$k_{edge}(e_1, e_2) = max(0, c - |weight(e_1) - weight(e_2)|)$$

4. Experimental Setup

This section describes our experimental setup, including the machine configurations and compilers we used. We evaluated each characterization technique on two different machines: a Nehalem 2-socket 4-cores Intel Xeon 5620 (16 H/W threads) with 12MB L3 cache and a Q9650 4-cores Intel Core 2 Quad Q9650 4 H/W threads) with 12MB L2 cache. Our optimization search space came from the PoCC source-to-source polyhedral compiler [2]; therefore, we needed a backend compiler to produce executable code from our optimized codes. We experimented with two back-end compilers for each machine: GCC and ICC. We used GCC V4.5 and ICC V11.0 for the Nehalem and GCC V4.4 and ICC V10.1 for the Q9650. We show performance improvements compared to the following baselines: -fast for ICC and -O3 for GCC. In summary, we provide experiment results for four different machine/compiler configurations: (a) Nehalem-GCC, (b) Nehalem-ICC, (c) Q9650-GCC, and (d) Q9650-ICC.

We used PolyBench V2.1 [3] benchmark suite to evaluate our characterization techniques. Polybench consists of 30 scientific kernels and applications. We flushed the cache before every run to reduce the amount of variability, and we took the average of multiple runs of the same code.

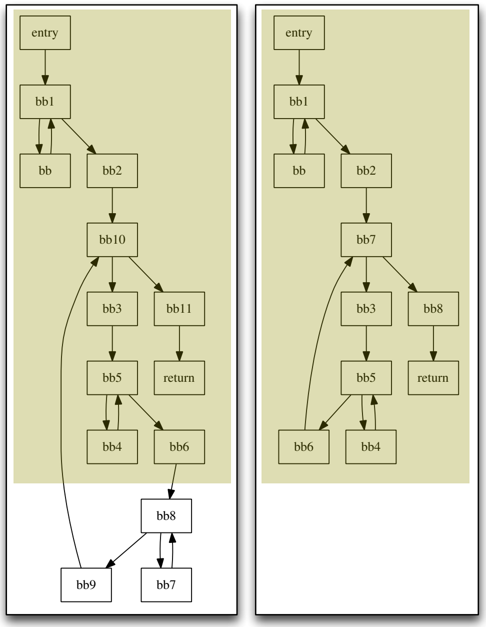
To build our models, we used SVMs in Weka [5] (for performance counters, reactions, and source code features) and Shogun [4] with Gaussian kernel (for graph-based characterization). To evaluate our different program characterization techniques, we used leave-one-out-cross validation over our set of kernel benchmarks.

5. Experimental Results

The experimental results for our four different machine/compiler pairs are given in Tables 4 through 7. PC refers to a performance counters characterization, 5R refers to using reactions of five optimization sequences, SRC refers to using Milepost source code features, and CFG refers to our graph-based characterization using the program's control flow graph. Note that for the reactions characterizations, more than five reactions did not significantly improve the results of that model. The first five columns show the results for an *non-iterative* scenario, where we use only the best predicted sequence from our different models. We term this scenario 1-shot. The next five columns correspond to using our models in an iterative fashion by using the top ten best predicted sequences and reporting the results of the sequences that gave the best actual speedup. We term this scenario 5-shot. The last column, referred to as OPT, gives the maximum performance achievable by exhaustively evaluating all optimizations in our search space. We also include Random, which gives results for randomly choosing 1 and 5 optimization sequences to evaluate for each program, averaged over 30 random trials. These tables provide speedups over their baseline followed by a percentage between parantheses indicating the percent of OPT achieved.

5.1 Graph-based Characterization and other Static Characterizations

This section compares our graph-based characterization (CFG) to Milepost features (SRC). Milepost contains static source code and intermediate representation characteristics represented in a fixed-length feature vector. For the four machine/compiler configurations, CFG significantly outperforms SRC both in a non-iterative (1-shot) and an iterative (5-shot) scenario. Some programs like 2mm, 3mm, and and jacobi-2d almost always achieved better performance given CFG versus SRC, regardless of the machine or backend compiler used. We observed that some programs with similar CFGs often have the same best optimization sequence. For example, atax and bicg have similar CFG structures (as shown in Figure 7) and their best sequences are the same for both machine-compiler configurations using ICC and very similar for the other the machine-compiler configurations using GCC. This can explain the good performance obtained when using the best predicted sequence for atax and bicg in Nehalem-ICC.



(a) atax (b) bicg

Figure 7. Control Flow Graph of atax and bicg. We observe that these two programs have very similar structure in their CFGs. We highlight the similarity between the two CFGs.

Non-Iterative Scenario (1-shot) For the non-iterative scenario, we observed that CFG outperformed SRC in more than half of the programs and in the average performance improvements for all machine-compiler configurations. For 1-shot results for the Nehalem-GCC configuration (Table 4), CFG achieved 5.7×

[1] In this paper, we chose $c = 2$ after 10-fold cross-validation

	1-Shot					5-shot					
Benchmark	Random	PC	5R	SRC	CFG	Random	PC	5R	SRC	CFG	OPT
2mm	2.7×(10%)	16.0×(63%)	13.8×(54%)	16.0×(63%)	25.0×(100%)	14.3×(56%)	16.0×(63%)	13.8×(54%)	16.0×(63%)	25.1×(100%)	25.1×
3mm	3.0×(10%)	13.9×(49%)	14.8×(53%)	13.9×(49%)	25.1×(89%)	10.0×(35%)	14.0×(50%)	19.8×(70%)	14.9×(53%)	25.4×(90%)	28.0×
adi	0.9×(24%)	3.3×(95%)	3.4×(98%)	3.4×(98%)	1.0×(29%)	2.3×(68%)	3.4×(98%)	3.4×(98%)	3.4×(98%)	1.0×(29%)	3.5×
atax	0.3×(15%)	2.0×(90%)	2.0×(90%)	1.8×(82%)	1.9×(86%)	1.0×(48%)	2.2×(100%)	2.0×(91%)	2.0×(90%)	2.2×(100%)	2.2×
bicg	0.6×(31%)	1.7×(81%)	1.7×(81%)	1.7×(80%)	0.8×(39%)	1.2×(58%)	2.1×(99%)	2.1×(99%)	2.1×(99%)	0.8×(39%)	2.1×
cholesky	1.1×(96%)	1.1×(100%)	1.1×(100%)	1.1×(100%)	1.1×(100%)	1.1×(99%)	1.1×(100%)	1.1×(100%)	1.1×(100%)	1.1×(100%)	1.1×
correlation	3.9×(20%)	11.1×(57%)	13.7×(71%)	15.3×(79%)	18.8×(97%)	10.3×(53%)	11.4×(59%)	15.2×(78%)	15.3×(79%)	19.2×(100%)	19.2×
covariance	2.3×(11%)	11.4×(54%)	15.8×(75%)	8.2×(39%)	20.6×(98%)	10.5×(50%)	11.5×(55%)	15.8×(75%)	15.8×(75%)	20.6×(98%)	21.0×
doitgen	4.3×(20%)	3.1×(14%)	2.4×(11%)	3.0×(14%)	5.8×(27%)	8.8×(42%)	3.3×(15%)	2.4×(11%)	3.1×(14%)	6.4×(30%)	20.9×
durbin	1.0×(99%)	1.0×(99%)	1.0×(99%)	1.0×(99%)	1.0×(98%)	1.0×(99%)	1.0×(99%)	1.0×(99%)	1.0×(100%)	1.0×(99%)	1.0×
dynprog	0.6×(59%)	0.4×(39%)	0.6×(58%)	0.6×(58%)	0.4×(45%)	0.8×(83%)	0.4×(45%)	0.8×(83%)	0.6×(60%)	0.5×(57%)	0.9×
fdtd-2d	1.1×(21%)	1.0×(20%)	0.9×(17%)	0.9×(16%)	5.2×(100%)	2.1×(41%)	3.7×(71%)	1.0×(19%)	3.7×(71%)	5.2×(100%)	5.2×
fdtd-apml	2.5×(31%)	0.8×(9%)	4.1×(49%)	7.0×(84%)	2.8×(34%)	5.4×(65%)	0.8×(9%)	4.5×(54%)	7.3×(89%)	3.9×(47%)	8.2×
gaussfilter	1.1×(31%)	1.8×(52%)	1.7×(49%)	0.8×(24%)	0.8×(23%)	1.9×(54%)	1.8×(52%)	2.1×(60%)	0.8×(24%)	0.8×(24%)	3.5×
gemm	3.9×(14%)	11.7×(42%)	20.2×(72%)	11.6×(41%)	19.7×(70%)	11.6×(41%)	12.1×(43%)	20.2×(72%)	11.9×(43%)	27.6×(99%)	27.8×
gemver	2.9×(36%)	7.0×(89%)	6.7×(86%)	7.0×(89%)	0.6×(7%)	5.0×(63%)	7.0×(89%)	7.0×(89%)	7.0×(89%)	5.9×(76%)	7.8×
gesummv	1.1×(48%)	1.5×(65%)	1.6×(67%)	1.5×(64%)	1.6×(68%)	1.8×(77%)	1.6×(67%)	1.6×(68%)	1.6×(67%)	2.1×(90%)	2.3×
gramschm	7.6×(29%)	18.9×(74%)	19.0×(74%)	18.9×(74%)	13.6×(53%)	16.0×(63%)	19.0×(74%)	19.0×(74%)	18.9×(74%)	18.7×(73%)	25.4×
jacobi-1d	0.9×(30%)	2.9×(95%)	0.4×(12%)	3.0×(100%)	2.9×(95%)	1.4×(47%)	2.9×(95%)	2.9×(95%)	3.0×(100%)	3.0×(100%)	3.0×
jacobi-2d	0.7×(10%)	0.7×(10%)	0.5×(7%)	0.7×(10%)	3.3×(50%)	1.9×(28%)	2.7×(41%)	0.7×(10%)	0.7×(10%)	3.3×(51%)	6.5×
lu	0.5×(8%)	0.2×(3%)	6.2×(100%)	3.5×(57%)	0.9×(15%)	1.3×(21%)	0.2×(3%)	6.2×(100%)	3.5×(57%)	3.5×(57%)	6.2×
ludcmp	1.1×(97%)	1.1×(99%)	1.1×(99%)	1.1×(99%)	1.1×(100%)	1.1×(98%)	1.1×(99%)	1.1×(99%)	1.1×(99%)	1.1×(100%)	1.1×
mvt	4.5×(34%)	11.0×(83%)	11.0×(83%)	11.0×(83%)	1.0×(7%)	9.2×(70%)	12.2×(93%)	11.4×(86%)	11.4×(86%)	11.5×(87%)	13.1×
reg-detect	0.8×(41%)	0.6×(29%)	0.6×(30%)	1.3×(66%)	0.2×(12%)	1.4×(72%)	1.1×(55%)	1.6×(86%)	1.9×(96%)	0.4×(20%)	1.9×
seidel	2.0×(28%)	0.9×(13%)	0.8×(11%)	7.1×(100%)	1.0×(14%)	4.0×(56%)	7.1×(100%)	0.8×(11%)	7.1×(100%)	7.1×(100%)	7.1×
symm	1.0×(99%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×
syr2k	1.9×(19%)	4.2×(42%)	4.9×(49%)	4.9×(49%)	9.9×(99%)	5.7×(57%)	7.6×(76%)	4.9×(49%)	7.2×(72%)	10.0×(100%)	10.0×
syrk	3.0×(21%)	6.1×(45%)	3.2×(23%)	6.4×(47%)	1.3×(9%)	7.6×(55%)	6.9×(50%)	6.4×(47%)	6.9×(50%)	3.9×(28%)	13.5×
trisolv	0.5×(25%)	0.2×(11%)	0.2×(11%)	0.3×(15%)	0.8×(37%)	1.3×(66%)	1.4×(71%)	1.4×(71%)	1.8×(91%)	1.9×(95%)	2.0×
trmm	0.8×(56%)	0.8×(58%)	0.8×(58%)	0.8×(58%)	0.8×(62%)	1.0×(76%)	0.8×(58%)	1.3×(96%)	1.3×(96%)	0.8×(62%)	1.4×
AVG	2.0×(21%)	4.6×(50%)	5.2×(57%)	5.2×(56%)	5.7×(63%)	4.7×(52%)	5.2×(57%)	5.8×(63%)	5.8×(63%)	7.2×(79%)	9.1×

Table 4. Nehalem with GCC V4.5 Backend Compiler

speedup (63% of OPT) on average using CFG versus 5.2× speedup (56% of OPT) using SRC features. CFG fares equally well against SRC on other machine/compiler configurations for the 1-shot scenario. On the Nehalem-ICC configuration (Table 5), average performance for CFG and SRC are 3.0× and 2.7×, respectively. The average performance on the Q9650-GCC configuration (Table 6) for CFG is 2.7× speedup versus 2.2× speedup for SRC. And finally, on the Q9650-ICC configuration (Table 6), CFG gives the best improvement over SRC, achieving on average 2.8× speedup versus 1.8× speedup for SRC. We also observed CFG often gives the best speedups across all characterizations for all machine-compiler configurations. For example, CFG give the best speedups versus all other characterizations for 13 programs for both the Nehalem-GCC and Nehalem-ICC configurations, while SRC gives the best results for only 5 programs for both those configurations. For Q9650-GCC and Q9650-ICC configurations, CFG gives the best speedups for 10 and 9 programs, respectively. The SRC characterization gives the best speedups across the other characterizations for only 4 and 5 programs, respectively.

Iterative Scenario (5-shot): CFG also does well in an iterative scenario by allowing a model to get closer than SRC to the maximum available speedup with the top 5 predicted sequences. For example, Nehalem-GCC achieved 79% of OPT using CFG compared to 63% of OPT using SRC. We emphasize that we already achieved 63% of OPT using CFG in a non-iterative scenario (1-shot model), while SRC achieves the same result after 5 iterations. For the other machine-compiler configurations, CFG outperforms SRC by a significant amount. We also achieved better speedups over all characterizations for a larger number of programs using CFG over SRC. For example, for both Nehalem configurations, CFG gives the best results for 13 and 14 programs, while SRC gives the best results for only 3 and 4 programs.

Summary Note that Milepost features (SRC) include information about basic blocks and edges; however, this information is summarized over the entire function and put into a fixed-length vector representation. In contrast, CFG provides topological information about the program's intermediate representation. The improved performance of using a graph-based representation over a fixed-length feature vector representation points to the additional benefit of including topology and a specialized learning algorithm that can use this topological information.

There were some programs (e.g., gemver) that achieved better performance using the SRC characterization versus CFG for some of our machine-compiler configurations. SRC has additional static features, which we did not implement in our CFG characterization that may be helpful for this particular program. As future work, we will investigate adding additional features to improve our graph-based characterization techniques.

5.2 Graph-based Characterization and Dynamic Characterizations

We now compare CFG to two state-of-the art dynamic characterization techniques: performance counters (PC) and reactions (5R). For our different machine-compiler combinations, CFG significantly outperforms these two dynamic characterizations in both non-iterative and iterative scenarios.

Non-Iterative Scenario (1-shot) For the Nehalem-GCC configuration (Table 4) and 1-shot, CFG achieved 63% of OPT on average, whereas PC and 5R achieved 50% and 57% of OPT, respectively. Not only did CFG perform well on average, models using this characterization give us good predictions on many programs that were poorly optimized with models using the dynamic characterizations. For this configuration, jacobi-2d and fdtd-2d achieved less than 20% of OPT for PC and 5R; however, we achieved much better performance using CFG for those programs by achieving 50%

	1-Shot					5-shot					
Benchmark	Random	PC	5R	SRC	CFG	Random	PC	5R	SRC	CFG	OPT
2mm	1.0×(7%)	4.0×(29%)	2.8×(21%)	2.8×(21%)	12.8×(96%)	3.6×(26%)	4.0×(29%)	4.5×(33%)	4.0×(29%)	13.4×(100%)	13.4×
3mm	0.7×(5%)	3.1×(23%)	3.0×(22%)	4.0×(30%)	13.3×(99%)	2.4×(18%)	3.1×(23%)	4.3×(32%)	4.0×(30%)	13.4×(100%)	13.4×
adi	1.0×(25%)	2.1×(56%)	2.1×(56%)	2.0×(53%)	0.9×(22%)	1.7×(45%)	2.3×(60%)	2.3×(60%)	2.3×(60%)	2.6×(67%)	3.8×
atax	0.6×(21%)	1.9×(75%)	2.2×(84%)	2.2×(84%)	1.9×(75%)	1.1×(42%)	2.2×(84%)	2.2×(84%)	2.2×(84%)	2.2×(84%)	2.6×
bicg	0.4×(25%)	1.4×(84%)	1.4×(84%)	1.4×(84%)	1.7×(100%)	0.8×(46%)	1.4×(84%)	1.4×(84%)	1.4×(84%)	1.7×(100%)	1.7×
cholesky	0.7×(71%)	0.6×(54%)	1.0×(97%)	0.6×(54%)	1.0×(97%)	0.9×(86%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×
correlation	1.0×(12%)	6.3×(74%)	6.3×(74%)	6.4×(75%)	6.3×(74%)	4.6×(54%)	6.4×(75%)	6.4×(75%)	6.4×(75%)	8.3×(97%)	8.5×
covariance	1.9×(20%)	5.9×(64%)	6.4×(70%)	6.4×(70%)	0.7×(8%)	4.8×(53%)	6.4×(70%)	6.4×(70%)	6.4×(70%)	8.8×(97%)	9.0×
doitgen	2.1×(16%)	1.5×(11%)	1.5×(11%)	1.5×(11%)	12.7×(98%)	4.9×(38%)	7.5×(57%)	4.8×(37%)	1.5×(11%)	12.7×(98%)	12.9×
durbin	1.0×(99%)	1.0×(99%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×
dynprog	0.7×(63%)	0.9×(82%)	0.6×(52%)	0.6×(52%)	0.3×(25%)	0.9×(77%)	1.0×(86%)	0.9×(81%)	0.6×(54%)	0.5×(44%)	1.1×
fdtd-2d	0.2×(23%)	0.2×(20%)	0.2×(20%)	0.2×(20%)	0.2×(20%)	0.4×(36%)	0.2×(20%)	0.2×(21%)	0.2×(21%)	0.2×(20%)	1.0×
fdtd-apml	1.6×(27%)	1.6×(26%)	2.1×(36%)	1.6×(26%)	1.3×(21%)	4.6×(77%)	2.1×(35%)	2.1×(36%)	1.8×(29%)	1.6×(26%)	6.0×
gaussfilter	2.4×(27%)	3.9×(46%)	3.9×(46%)	5.5×(65%)	1.7×(19%)	4.5×(53%)	4.4×(52%)	4.4×(52%)	5.5×(65%)	1.9×(22%)	8.5×
gemm	1.2×(8%)	2.3×(17%)	2.6×(20%)	3.9×(29%)	0.9×(6%)	3.5×(26%)	2.6×(20%)	4.0×(30%)	4.3×(32%)	11.3×(86%)	13.1×
gemver	0.5×(29%)	1.3×(77%)	1.5×(84%)	1.5×(86%)	1.6×(92%)	1.1×(60%)	1.7×(99%)	1.7×(99%)	1.7×(99%)	1.6×(92%)	1.7×
gesummv	0.5×(49%)	0.8×(80%)	0.8×(80%)	0.8×(79%)	1.0×(99%)	1.0×(91%)	0.8×(80%)	0.8×(80%)	0.8×(81%)	1.0×(99%)	1.1×
gramschm	6.5×(29%)	17.7×(79%)	18.0×(80%)	17.8×(79%)	7.1×(32%)	17.6×(78%)	17.9×(80%)	18.0×(80%)	17.9×(80%)	7.2×(32%)	22.3×
jacobi-1d	1.3×(14%)	9.1×(100%)	9.1×(100%)	7.9×(87%)	1.0×(10%)	3.2×(35%)	9.1×(100%)	9.1×(100%)	9.1×(100%)	7.9×(87%)	9.1×
jacobi-2d	0.8×(18%)	0.4×(9%)	0.4×(9%)	0.4×(10%)	3.6×(86%)	1.4×(33%)	0.4×(10%)	0.5×(12%)	0.4×(10%)	3.6×(86%)	4.2×
lu	0.4×(8%)	0.1×(3%)	0.8×(18%)	4.2×(89%)	0.7×(13%)	1.2×(26%)	0.8×(18%)	4.2×(89%)	4.2×(89%)	1.2×(26%)	4.7×
ludcmp	1.0×(97%)	1.0×(100%)	1.0×(98%)	1.0×(96%)	1.0×(98%)	1.0×(99%)	1.0×(100%)	1.0×(98%)	1.0×(100%)	1.0×(98%)	1.0×
mvt	0.5×(25%)	1.4×(68%)	1.7×(84%)	1.7×(80%)	2.0×(97%)	1.2×(56%)	1.5×(71%)	1.7×(84%)	1.7×(84%)	2.1×(100%)	2.1×
reg-detect	0.6×(43%)	0.5×(35%)	0.7×(47%)	1.0×(67%)	0.3×(19%)	1.0×(70%)	1.1×(81%)	1.0×(71%)	1.2×(87%)	0.3×(19%)	1.4×
seidel	2.0×(16%)	1.0×(8%)	0.9×(7%)	1.0×(8%)	9.0×(72%)	7.1×(57%)	8.6×(69%)	5.6×(45%)	1.1×(8%)	10.1×(82%)	12.3×
symm	1.0×(99%)	1.0×(99%)	1.0×(99%)	1.0×(99%)	1.0×(99%)	1.0×(100%)	1.0×(100%)	1.0×(99%)	1.0×(100%)	1.0×(100%)	1.0×
syr2k	0.4×(37%)	0.8×(77%)	1.0×(100%)	0.4×(39%)	1.0×(100%)	0.8×(76%)	0.8×(77%)	1.0×(100%)	0.8×(77%)	1.0×(100%)	1.0×
syrk	0.2×(16%)	0.6×(58%)	0.3×(33%)	0.6×(58%)	1.0×(96%)	0.5×(45%)	0.6×(58%)	0.6×(58%)	0.6×(58%)	1.0×(99%)	1.1×
trisolv	1.0×(36%)	0.3×(9%)	0.3×(9%)	0.7×(24%)	1.0×(36%)	1.6×(58%)	2.0×(73%)	2.0×(73%)	2.6×(95%)	2.6×(95%)	2.7×
trmm	0.9×(16%)	0.9×(16%)	0.9×(16%)	0.9×(16%)	1.0×(18%)	1.8×(34%)	5.4×(100%)	5.4×(100%)	1.0×(18%)	1.0×(18%)	5.4×
AVG	1.1×(20%)	2.5×(43%)	2.5×(45%)	2.7×(48%)	3.0×(53%)	2.7×(48%)	3.3×(58%)	3.3×(59%)	2.9×(52%)	4.1×(73%)	5.6×

Table 5. Nehalem with ICC v11.0 Backend Compiler

for jacobi-2d and 100% for fdtd-2d. For the Nehalem-ICC configuration (Table 5), CFG achieved much better performance improvement than both PC and 5R on average. CFG achieved 53% of the maximum available performance, with PC and 5R achieving 45% and 48%, respectively. For the Q9650-GCC configuration (Table 6), CFG also performed better than the two dynamic characterization techniques by achieving 74% of OPT, while PC and 5R only achieved 59%. Finally, for the Nehalem-ICC configuration (Table 5), we again achieved better performance improvement for 1-shot using CFG. CFG achieves 60% of the available performance while PC and 5R achieve 46% and 40%, respectively.

Iterative Scenario (5-shot) For the iterative scenario, CFG achieves at least 73% of OPT on average on the Nehalem-ICC configuration and up to 88% of OPT on the Nehalem-GCC configuration. Both 5R and PC perform well; 5R slightly outperforms PC for both Nehalem configurations, and it performs as well as PC for both Q9650 configurations. However, CFG significantly outperformed the two dynamic characterizations for all machine-compiler configurations. In summary, CFG achieves results closer to the space-optimal performance than both dynamic characterization techniques for our iterative approach.

Summary We noticed that CFG outperforms PC and 5R in the average performance improvement for all machine-compiler configurations. Moreover, CFG is a static technique and unlike the dynamic characterization techniques does not require running the compiled program at all. Thus, our experimental results indicate a potential in using graph-based characterzations such as CFG in predictive modeling.

5.3 Evaluation of Different IRs

We performed additional experiments with a different type of IR to statically characterize our programs. We used the program's dominator tree (DOM), often used for static analysis in compilers, e.g., in building SSA form. We used the same prediction model and simply the program characterization that the model was using to the program's DOM. Table 8 shows the results when using DOM compared to using CFG. DOM is competitive with CFG, even outperforming CFG on the Nehalem-ICC configuration. Still, CFG significantly out-performs DOM on the Nehalem-GCC configuration and remains competitive on all other configurations. Therefore, we favor the CFG over DOM. We tried to create graphs that combined CFG and DOM, but the results were sometimes worse than just using CFG and DOM alone. We believe combining other types of graph-based IRs that give additional (not redundant) information will help improve our results. For example, as we mentioned in Section 5.2, we plan to look at a combination of CFG with the data flow graph (DFG) in future work.

	1-Shot		5-shot	
Configuration	CFG	DOM	CFG	DOM
Nehalem-GCC	5.7×(63%)	5.1×(57%)	7.2×(79%)	6.6×(72%)
Nehalem-ICC	3.0×(53%)	3.0×(54%)	4.1×(73%)	4.2×(75%)
Q9650-GCC	2.7×(74%)	2.7×(74%)	3.2×(88%)	3.2×(88%)
Q9650-ICC	2.8×(60%)	2.7×(60%)	3.5×(76%)	3.4×(75%)

Table 8. Average speedup achieved on 4 different test configuration. DOM is the prediction model based on dominator tree of the program.

5.4 Using Models in Iterative Compilation

We also evaluated how our different characterization techniques would perform for larger iterative compilation evaluations. We evaluated the top 100 predicted optimization sequences for each characterization method and measured the performance improvement for each pair of machine/compiler configuration. We produced line graphs showing best actual performance (y-axis) achieved as compared to the number of predicted sequences evaluated

			1-Shot					5-shot			
Benchmark	Random	PC	5R	SRC	CFG	Random	PC	5R	SRC	CFG	OPT
2mm	1.6×(27%)	3.7×(63%)	3.8×(64%)	3.8×(64%)	5.6×(96%)	3.5×(59%)	3.8×(65%)	3.8×(65%)	3.8×(65%)	5.8×(100%)	5.8×
3mm	1.3×(12%)	0.7×(6%)	4.1×(39%)	4.1×(39%)	10.5×(99%)	4.1×(38%)	4.1×(39%)	4.2×(39%)	4.2×(39%)	10.6×(100%)	10.6×
adi	0.5×(34%)	1.1×(76%)	1.1×(82%)	1.1×(82%)	1.0×(74%)	0.7×(51%)	1.1×(82%)	1.4×(100%)	1.1×(82%)	1.0×(74%)	1.4×
atax	0.3×(31%)	0.7×(70%)	1.0×(98%)	0.7×(70%)	0.5×(50%)	0.6×(57%)	0.7×(71%)	1.0×(98%)	0.7×(71%)	0.5×(50%)	1.0×
bicg	0.3×(25%)	0.6×(52%)	1.0×(100%)	0.5×(50%)	0.3×(26%)	0.5×(51%)	0.6×(52%)	1.0×(100%)	0.6×(52%)	0.3×(31%)	1.0×
cholesky	1.0×(97%)	1.0×(96%)	1.0×(95%)	1.0×(95%)	1.0×(96%)	1.0×(98%)	1.0×(98%)	1.0×(97%)	1.0×(97%)	1.0×(97%)	1.1×
correlation	4.2×(29%)	7.2×(49%)	7.5×(52%)	7.8×(54%)	13.8×(95%)	8.7×(60%)	7.3×(50%)	8.4×(58%)	8.1×(56%)	14.4×(100%)	14.4×
covariance	3.1×(20%)	10.9×(70%)	8.1×(52%)	8.1×(52%)	14.6×(94%)	7.5×(48%)	10.9×(70%)	8.1×(52%)	8.1×(52%)	15.4×(100%)	15.4×
doitgen	1.7×(29%)	2.5×(43%)	2.5×(43%)	2.4×(42%)	2.2×(37%)	3.4×(58%)	2.5×(43%)	2.5×(43%)	2.5×(43%)	5.6×(97%)	5.8×
durbin	1.0×(96%)	1.0×(98%)	1.0×(97%)	1.0×(97%)	1.0×(98%)	1.0×(98%)	1.0×(99%)	1.0×(99%)	1.0×(99%)	1.0×(98%)	1.0×
dynprog	0.5×(53%)	0.2×(23%)	0.4×(41%)	0.2×(19%)	0.3×(27%)	0.8×(83%)	0.6×(59%)	0.4×(44%)	0.3×(28%)	0.3×(28%)	1.0×
fdtd-2d	1.3×(35%)	2.0×(52%)	2.0×(52%)	2.0×(52%)	2.0×(52%)	2.3×(59%)	3.1×(83%)	3.0×(78%)	3.0×(78%)	3.1×(83%)	3.8×
fdtd-apml	1.5×(43%)	0.8×(21%)	1.9×(54%)	3.3×(94%)	0.8×(21%)	2.6×(75%)	3.5×(98%)	2.1×(59%)	3.3×(94%)	2.7×(76%)	3.5×
gaussfilter	0.6×(42%)	0.8×(58%)	0.8×(57%)	0.6×(40%)	0.2×(14%)	0.9×(60%)	0.8×(58%)	0.8×(58%)	0.6×(40%)	0.3×(22%)	1.4×
gemm	1.6×(37%)	3.3×(77%)	3.2×(77%)	3.3×(77%)	4.1×(96%)	3.1×(72%)	3.3×(78%)	3.3×(77%)	3.3×(78%)	4.2×(100%)	4.2×
gemver	1.3×(59%)	2.2×(99%)	2.2×(100%)	2.0×(90%)	1.4×(62%)	1.6×(73%)	2.2×(99%)	2.2×(100%)	2.2×(100%)	1.5×(69%)	2.2×
gesummv	0.9×(46%)	1.1×(58%)	1.1×(58%)	1.8×(93%)	1.1×(59%)	1.5×(80%)	1.9×(100%)	1.9×(100%)	1.8×(93%)	1.1×(61%)	1.9×
gramschm	3.4×(42%)	3.1×(39%)	3.3×(41%)	3.4×(42%)	4.9×(61%)	5.6×(70%)	3.4×(42%)	3.4×(42%)	3.4×(42%)	5.8×(73%)	8.0×
jacobi-1d	0.2×(23%)	0.1×(6%)	0.2×(16%)	0.3×(31%)	0.2×(17%)	0.5×(53%)	0.3×(31%)	1.0×(97%)	0.3×(31%)	0.3×(31%)	1.0×
jacobi-2d	1.3×(43%)	2.5×(85%)	2.5×(85%)	2.5×(85%)	2.7×(94%)	2.0×(69%)	2.5×(85%)	2.5×(85%)	2.5×(85%)	2.9×(100%)	2.9×
lu	0.5×(15%)	0.2×(5%)	3.1×(93%)	0.6×(19%)	0.5×(13%)	1.3×(37%)	0.6×(17%)	3.1×(93%)	3.2×(96%)	1.2×(34%)	3.4×
ludcmp	1.0×(99%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×
mvt	1.3×(55%)	2.2×(93%)	2.2×(93%)	2.2×(93%)	2.0×(85%)	1.9×(80%)	2.2×(94%)	2.2×(93%)	2.2×(93%)	2.2×(93%)	2.3×
reg-detect	0.9×(37%)	1.1×(45%)	0.2×(10%)	2.0×(78%)	0.4×(15%)	1.7×(68%)	2.0×(78%)	2.5×(99%)	2.5×(100%)	1.6×(65%)	2.5×
seidel	1.1×(38%)	0.8×(29%)	0.9×(33%)	0.9×(32%)	0.9×(30%)	1.9×(66%)	2.5×(89%)	0.9×(33%)	0.9×(32%)	2.8×(100%)	2.8×
symm	1.0×(97%)	1.0×(100%)	1.0×(97%)	1.0×(98%)	1.0×(98%)	1.0×(99%)	1.0×(100%)	1.0×(99%)	1.0×(99%)	1.0×(99%)	1.0×
syr2k	1.1×(27%)	4.1×(99%)	3.9×(94%)	3.9×(94%)	2.1×(51%)	2.6×(63%)	4.1×(99%)	4.0×(96%)	4.1×(99%)	3.8×(92%)	4.1×
syrk	1.7×(47%)	2.9×(81%)	3.3×(91%)	3.3×(91%)	3.5×(97%)	2.8×(79%)	3.3×(92%)	3.5×(97%)	3.5×(97%)	3.6×(99%)	3.6×
trisolv	0.6×(60%)	0.6×(64%)	0.6×(64%)	0.1×(15%)	0.9×(95%)	0.9×(94%)	0.9×(94%)	1.0×(100%)	1.0×(99%)	0.9×(95%)	1.0×
trmm	0.6×(54%)	0.6×(58%)	0.6×(58%)	0.6×(58%)	0.6×(57%)	0.8×(78%)	1.0×(100%)	1.0×(100%)	0.6×(61%)	0.6×(61%)	1.0×
AVG	1.2×(34%)	2.0×(54%)	2.2×(59%)	2.2×(59%)	2.7×(74%)	2.3×(61%)	2.4×(66%)	2.4×(66%)	2.4×(65%)	3.2×(88%)	3.7×

Table 6. Q9650 with GCC V4.4 Backend Compiler

(graphs not shown due to space constraints). The CFG line always had a steeper curve compared to the other three characterization techniques. Also, the CFG line was typically always above the lines of the other characterization methods. For the Q9650 configurations, PC only slightly out-performed CFG after many iterations (>40).

6. Related Work

There has also been some research on improving the program characterization to be used with machine learning for selecting good optimizations. In particular, Leather *et al.* [19] used compiler's IR and genetic programming to construct automatically new features from the GCC RTL representation of loops to improve a machine learning algorithm's performance on loop unrolling. However, the static features discovered are those that can be summarized into a fixed-length feature vector. Also, their technique only outperforms static source code features (such as, SRC) by only a couple of percent on average. Fursin *et al.* [16] also use the program's intermediate representation along with source code information in the Milepost GCC project [16]. These features are used to construct models that predict good optimization strategies according to metrics desired by the user (e.g., performance or code size). The authors collect summary statistics about the different instructions and from the control flow graph for each function, but again, these features are summarized into a fixed-length feature vector.

Wang *et al.* [29] also used an intermediate representation called the streaming graph to extract static program features. In this work, they focus on streaming programs, and they constructed a model that automatically predicts the ideal partitioning structure of each streaming program. Their program feature includes two sets of feature, one is the summary characteristics of streaming program, e.g., instruction mix, and other characteristics of critical path extracted from stream IR. Again, these features are summarized in a fixed-length feature vector. They developed a tool that automatically generates small training examples for this predictive model.

In contrast to these previous works, we extract topological information from the program's control flow graph, and we provide fine-grained statistics in our graph-based characterization corresponding to each node in the CFG.

Demme *et al.* [12] propose to cluster similar functions in a program based on various program characterizations including static instruction mix, different graph-based intermediate representations, and functions' reactions to a set of optimizations. To determine the similarity of two graphs, the authors use a technique called "flooding-based graph distance" to score the similarity of two graphs. They show the potential of using graph-based representations to characterize input programs by demonstrating that clustering functions based on their graphical representations is more effective in classifying their behavior/reaction on very small set of optimization combinations (15) out of four optimizations. In our work, we demonstrate the potential of using graph-based characterization on a much larger set of optimization combinations (hundreds of optimization combinations) by building a prediction model using machine learning algorithm, specifically SVM.

Recent research has focused on using dynamic program features to improve the performance of models used to predict good optimization sequences. For example, performance counters have been used by Cavazos *et al.* [10] for PathScale EkoPath compiler to predict good sequences to use for that compiler. Parello *et al.* [21] also used performance counters to identify "anomalies" and iteratively apply optimizations to a program. These techniques have been shown to out-perform static characterization methods that use fixed-length feature vectors. Unfortunately, these dynamic techniques require running the program to collect these features. These features may also be specific to program inputs, since they were collected from one or more runs of the program.

	1-Shot					5-shot					
Benchmark	Random	PC	5R	SRC	CFG	Random	PC	5R	SRC	CFG	OPT
2mm	3.1×(15%)	3.8×(19%)	5.2×(26%)	5.3×(27%)	19.5×(100%)	6.2×(31%)	7.6×(39%)	7.6×(39%)	14.4×(73%)	19.5×(100%)	19.5×
3mm	2.0×(8%)	4.3×(17%)	6.8×(27%)	8.9×(36%)	24.7×(99%)	6.4×(25%)	5.9×(23%)	8.9×(36%)	8.9×(36%)	24.8×(100%)	24.8×
adi	0.8×(42%)	1.4×(75%)	0.6×(30%)	0.9×(49%)	1.0×(52%)	1.3×(69%)	1.4×(75%)	0.6×(31%)	1.0×(54%)	1.0×(54%)	1.9×
atax	0.8×(30%)	1.8×(70%)	1.7×(68%)	2.1×(80%)	2.5×(100%)	1.7×(67%)	2.1×(82%)	1.8×(69%)	2.1×(83%)	2.5×(100%)	2.5×
bicg	0.5×(34%)	1.3×(83%)	1.3×(80%)	1.3×(80%)	1.0×(64%)	1.0×(64%)	1.3×(83%)	1.5×(96%)	1.3×(80%)	1.5×(96%)	1.6×
cholesky	0.8×(70%)	1.1×(99%)	0.6×(56%)	0.6×(56%)	1.1×(100%)	1.0×(87%)	1.1×(99%)	1.1×(100%)	1.1×(99%)	1.1×(100%)	1.1×
correlation	3.3×(49%)	2.7×(40%)	4.5×(67%)	2.6×(38%)	1.4×(21%)	4.9×(73%)	5.1×(76%)	4.9×(73%)	4.2×(64%)	4.5×(67%)	6.6×
covariance	3.1×(59%)	3.0×(57%)	4.0×(75%)	3.7×(70%)	4.0×(77%)	4.0×(77%)	4.4×(83%)	4.2×(79%)	3.7×(70%)	4.0×(77%)	5.2×
doitgen	1.0×(33%)	0.8×(28%)	0.7×(24%)	1.4×(50%)	1.4×(49%)	1.5×(51%)	1.5×(52%)	1.4×(50%)	1.5×(53%)	2.1×(73%)	2.9×
durbin	1.0×(92%)	0.7×(67%)	1.0×(94%)	1.0×(94%)	1.0×(94%)	1.0×(97%)	1.0×(95%)	1.0×(95%)	1.0×(98%)	1.0×(96%)	1.1×
dynprog	0.5×(29%)	0.9×(52%)	0.3×(16%)	0.2×(11%)	0.2×(13%)	0.9×(48%)	0.9×(52%)	0.3×(17%)	0.2×(11%)	0.2×(13%)	1.8×
fdtd-2d	1.5×(45%)	3.1×(98%)	3.1×(99%)	2.5×(78%)	0.5×(17%)	2.6×(80%)	3.1×(98%)	3.2×(99%)	3.1×(98%)	2.7×(86%)	3.2×
fdtd-apml	1.0×(35%)	1.7×(63%)	0.9×(32%)	0.4×(15%)	0.4×(14%)	2.0×(75%)	2.6×(98%)	1.6×(58%)	0.4×(15%)	0.6×(20%)	2.7×
gaussfilter	1.7×(37%)	1.4×(30%)	2.1×(46%)	1.4×(30%)	1.1×(24%)	2.4×(52%)	2.2×(49%)	2.2×(49%)	1.5×(33%)	1.8×(39%)	4.5×
gemm	0.7×(24%)	0.8×(28%)	1.3×(47%)	0.8×(29%)	1.6×(57%)	1.4×(49%)	1.1×(39%)	1.4×(51%)	0.8×(29%)	2.2×(79%)	2.8×
gemver	0.9×(61%)	0.5×(35%)	1.2×(83%)	1.3×(89%)	0.5×(34%)	1.2×(80%)	1.5×(99%)	1.3×(85%)	1.3×(89%)	0.8×(51%)	1.5×
gesummv	0.9×(52%)	0.9×(50%)	0.9×(50%)	1.4×(78%)	0.7×(37%)	1.2×(67%)	1.8×(100%)	0.9×(50%)	1.5×(88%)	0.8×(46%)	1.8×
gramschm	6.8×(29%)	6.0×(26%)	6.1×(26%)	1.4×(6%)	7.7×(33%)	13.1×(57%)	13.2×(57%)	6.1×(26%)	6.0×(26%)	13.2×(57%)	22.9×
jacobi-1d	2.3×(48%)	2.1×(44%)	3.5×(75%)	3.8×(80%)	1.1×(22%)	3.7×(77%)	3.8×(80%)	3.5×(75%)	3.8×(80%)	2.8×(59%)	4.7×
jacobi-2d	2.9×(47%)	3.9×(62%)	4.8×(76%)	3.9×(63%)	2.7×(43%)	4.1×(66%)	5.4×(87%)	5.5×(89%)	5.4×(87%)	5.4×(87%)	6.2×
lu	0.4×(12%)	0.8×(22%)	3.5×(100%)	0.7×(19%)	0.1×(4%)	1.2×(34%)	0.8×(22%)	3.5×(100%)	1.7×(47%)	0.1×(4%)	3.5×
ludcmp	1.0×(98%)	1.0×(99%)	1.0×(100%)	1.0×(99%)	1.0×(100%)	1.0×(99%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×
mvt	1.1×(59%)	0.5×(26%)	1.7×(89%)	1.5×(77%)	1.6×(81%)	1.6×(81%)	1.5×(79%)	1.9×(96%)	1.9×(96%)	1.8×(94%)	1.9×
reg-detect	0.6×(58%)	0.3×(29%)	1.0×(90%)	0.9×(82%)	0.9×(82%)	0.9×(82%)	1.0×(85%)	1.0×(91%)	0.9×(82%)	0.9×(82%)	1.1×
seidel	1.0×(44%)	1.0×(42%)	1.1×(45%)	1.1×(44%)	1.1×(46%)	1.5×(64%)	1.7×(72%)	1.3×(54%)	1.2×(51%)	1.9×(80%)	2.4×
symm	0.9×(86%)	0.8×(82%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(97%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×(100%)	1.0×
syr2k	0.2×(21%)	1.0×(100%)	1.0×(100%)	0.9×(87%)	0.9×(87%)	0.5×(51%)	1.0×(100%)	1.0×(100%)	0.9×(88%)	0.9×(87%)	1.0×
syrk	0.2×(15%)	0.8×(77%)	1.0×(93%)	0.2×(15%)	0.1×(5%)	0.5×(44%)	1.1×(100%)	1.0×(96%)	0.9×(79%)	1.1×(100%)	1.1×
trisolv	1.0×(30%)	1.3×(41%)	1.3×(41%)	1.9×(59%)	1.0×(31%)	1.6×(52%)	1.9×(59%)	1.9×(59%)	1.9×(59%)	1.5×(47%)	3.1×
trmm	0.8×(51%)	1.0×(63%)	1.0×(63%)	0.8×(50%)	1.0×(63%)	1.0×(67%)	1.0×(63%)	1.0×(63%)	0.8×(53%)	1.0×(63%)	1.6×
AVG	1.4×(31%)	1.7×(37%)	2.1×(46%)	1.8×(40%)	2.8×(60%)	2.4×(52%)	2.6×(57%)	2.5×(53%)	2.5×(55%)	3.5×(76%)	4.6×

Table 7. Q9650 with ICC v10.1 Backend Compiler

There has been an attempt to use a hybrid of dynamic and static features together for predictive modeling. Tournavitis *et al.* [26] used both static and dynamic features to control an auto-parallelizing compiler. They first used a profiling-based approach to reveal application parallelism, and then they used a machine learning-based model to decide whether to parallelize a given loop or not. Their prediction model was constructed using an SVM classifier based on both the dynamic and static program features, including instruction count, memory access count, and loop iteration count of loop body.

Our optimization search space also includes auto-parallelism, however, our model predicts the speedup for auto-parallelism being used along with other important loop transformations. The idea of using a hybrid characterization of both static and dynamic features is interesting, and one we will explore in future work.

7. Conclusion

In this paper, we address the problem of developing an expressive program characterization technique for predicting compiler optimizations by using the program's control flow graph. Moreover, we evaluate this technique along with three state-of-the-art techniques, which are *source code features*, *performance counters*, and *reactions*. Prediction models were constructed using support vector machines, and these models were used predict auto-parallelization and loop transformations in the polyhedral compiler PoCC. We evaluated our characterization techniques on a large set of scientific kernels that came from the PolyBench suite using leave-one-out cross-validation. From our experiment results, our graph-based characterization approach performed significantly better than other characterization approaches. Specifically, our approach achieved up to 74% of the maximum speedup available in our search space for a non-iterative scenario. When used in an iterative scenario, models that use our graph-based characterization obtained up to 88% on average of the maximum available speedup in our search space in just 5 iterations.

For future work, we expect to evaluate our graph-based characterization technique using other machine learning algorithms to build our prediction models. In addition, we also plan to investigate additional graph-based IRs, in particular, the PDG, which includes both control and data dependencies in one graph. Further, we plan to extend our testbed to different domains of applications, different compilers, and larger optimization spaces.

Acknowledgments

We thank Louis-Noël Pouchet, Julien Le Guen, Stéphane Zuckerman, and anonymous reviewers for their feedback and help on this work. This work was funded in part by the U.S. National Science Foundation through Career award 0953667 and the DARPA Computer Science Study Group (CSSG).

References

[1] MINimal IR space. http://www.assembla.com/wiki/show/minir-dev.

[2] PoCC. http://www.cse.ohio-state.edu/ pouchet/software/pocc.

[3] Polybench v2.1. http://www-roc.inria.fr/ pouchet/software/polybench.

[4] Shogun. http://www.shogun-toolbox.org/.

[5] Weka 3. http://www.cs.waikato.ac.nz/ml/weka.

[6] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 295–305, 2006.

[7] K. M. Borgwardt and H. P. Kriegel. Shortest-path kernels on graphs. In *IEEE International Conference on Data Mining (ICDM)*, pages 74–81, 2005.

[8] H. Bunke and K. Riesen. A family of novel graph kernels for structural pattern recognition. In *Iberoamerican Congress on Pattern Recognition (CIARP)*, pages 20–31, 2007.

[9] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. F. O'Boyle, G. Fursin, and O. Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*, October 2006.

[10] J. Cavazos, G. Fursin, F. V. Agakov, E. V. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 185–197, 2007.

[11] F. de Mesmay, Y. Voronenko, and M. Püschel. Offline library adaptation using automatically generated heuristics. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[12] J. Demme and S. Sethumadhavan. Approximate graph clustering for program characterization. In *Proceedings of the International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*, January 2012.

[13] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the International Conference on Computing Frontiers (CF)*, pages 131–142, 2007.

[14] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and M. F. O'Boyle. Portable compiler optimization across embedded programs and microarchitectures using machine learning. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2009.

[15] G. Fursin and O. Temam. Collective optimization. In *Proceedings of the International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*, January 2009.

[16] G. Fursin, Y. Kashnikov, A. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. Williams, and M. O'Boyle. Milepost GCC: machine learning enabled self-tuning compiler. *International Journal of Parallel Programming (IJPP)*, 39: 296–327, 2011.

[17] T. Gärtner. A survey of kernels for structured data. *SIGKDD Exploration Newsletter*, 5:49–58, 2003.

[18] D. Haussler. Convolution kernels on discrete structures. Technical Report UCSC-CRL-99-10, UC Santa Cruz, 1999.

[19] H. Leather, E. Bonilla, and M. O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 81–91, 2009.

[20] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA)*, pages 41–50, 2002.

[21] D. Parello, O. Temam, A. Cohen, and J.-M. Verdun. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC)*, pages 15–, 2004.

[22] E. Park, L.-N. Pouchet, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 119–129, 2011.

[23] B. Schölkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, 2001.

[24] B. Schölkopf, K. Tsuda, and J. P. Vert. *Kernel Methods in Computational Biology*. MIT Press, 2004.

[25] M. Stephenson and S. P. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 123–134, 2005.

[26] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 177–187, 2009.

[27] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 204–215, 2003.

[28] S. V. N. Vishwanathan, N. Schraudolph, R. Kondor, and K. Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 11:1201–1242, 2010.

[29] Z. Wang and M. F. O'Boyle. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 307–318, 2010.

[30] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. E. Eichenberger, and K. O'Brien. Automatic creation of tile size selection models. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 190–199, 2010.