Selecting Computer Architectures by Means of Control-Flow-Graph Mining

Frank Eichinger and Klemens Böhm

Institute for Program Structures and Data Organisation (IPD)
Universität Karlsruhe (TH), Germany, {eichinger, boehm}@ipd.uka.de

Abstract Deciding which computer architecture provides the best performance for a certain program is an important problem in hardware design and benchmarking. While previous approaches require expensive simulations or program executions, we propose an approach which solely relies on program analysis. We correlate substructures of the control-flow graphs representing the individual functions with the runtime on certain systems. This leads to a prediction framework based on graph mining, classification and classifier fusion. In our evaluation with the SPEC CPU 2000 and 2006 benchmarks, we predict the faster system out of two with high accuracy and achieve significant speedups in execution time.

1 Introduction

The question which computer architecture is best suited for a certain application is of major importance in hardware design and benchmarking. Think of a new scientific tool for which hardware is needed. It is not clear which hardware is most appropriate. Other developments give way to similar questions: With heterogeneous multicore processors, one has to decide at runtime on which processors to execute a certain program. Reconfigurable hardware allows to change the hardware at runtime. These upcoming technologies motivate studying dependencies between program characteristics and computer architectures as well.

To deal with the problem which architecture provides the best performance for a certain application, several approaches have been used, ranging from executions and simulations to analytical models and program analysis. At first sight, it seems feasible to assess the performance of a program by *executions* on the systems in question. But this requires to have access to the machines, and porting the program to them can be expensive. *Simulations* of processor architectures require detailed information on the architectures to choose from and might be very time-consuming. As modern computer architectures have an extreme complexity, *analytical models* describing them are hard to establish and may be unreliable. Some recent approaches make use of *program analysis.* The intuition is that similar programs display a similar runtime behaviour when executed on the same machine. Execution times for a number of programs are known for many systems, e.g., from benchmarks suites. [1, 2] compare similarities of programs based on execution properties such as the CPU instruction mix. These properties are architecture-dependent, but independent of the implementation used. To obtain them, program executions or simulations are necessary.

In this article we investigate another method to find the best computer architecture which does not require any execution or simulation of the application in question. Likewise, we assume that similar applications have a similar execution behaviour – but have consciously decided not to measure any runtime-related characteristics. Instead, we entirely rely on published execution times of benchmark programs. In our approach, we define similarity using structural characteristics of the control-flow graphs (CFGs) [3] of the underlying functions. Our research question is to investigate how well they describe the performance-related characteristics of a program, and if they can be used for performance predictions. In contrast to software metrics like lines of code and statements used, CFGs do not have any potentially distracting characteristics which depend on language specifics, such as the language used or the programming style of the developers. In more detail, we derive structural features from CFGs by means of frequent subgraph mining. The resulting subgraph features characterise a function and can train a classifier which predicts the best architecture for a given application.

The solution just outlined requires a number of contributions at different stages of the analysis process:

Representation of Control-Flow Graphs. To derive subgraph features from CFGs, the nodes of the graphs have to be labelled with information relevant for performance analysis. So far, nodes represent blocks of source code. We have to turn them into concise categorical labels which graph mining algorithms can use. However, such a labelling is not obvious. We propose a labelling scheme with information that is relevant for performance.

Mining Large Graphs. Once we have derived suitable CFGs, mining them is another challenge, due to the size of some of them. We develop an efficient technique consisting of two steps: We first mine a subset of the graphs that are 'easy to mine' and then inspect the remaining graphs. Our technique provides guarantees for the support values achieved.

Classification Framework. We propose a classification setting for our specific context. This is necessary: CFG based information is available at the function level, as we will explain, while we want to choose the best architecture for a program as a whole. We propose a framework that first learns at the function level, before we turn our classification model into a predictor for the architecture where a given program performs best.

Our experimental evaluation is based on the SPEC CPU 2000 and 2006 benchmark suites. The main result is that, for 'relatively similar' computer architectures to choose from, our approach achieves an average prediction accuracy of 69% when choosing between two systems. This also shows the existence of remarkably strong relationships between CFGs and runtime behaviour.

Paper outline: Section 2 presents related work, Section 3 describes CFG representations, and Section 4 says how we mine them. Section 5 describes the prediction framework, Section 6 our results. Section 7 concludes.

2 Related Work

In the areas of computer architecture, high-performance computing and benchmarking, different approaches have been investigated to predict the runtime of applications. Many of them make use of intelligent data-analysis techniques.

As mentioned, analytic models can assess the performance of software on certain machines. For distributed MPI (message passing interface) programs, Kühnemann et al. developed a compiler tool which helps deriving such a model [4]. It builds on source-code analysis and properties of the underlying machines. These properties include the execution times of basic arithmetic and logical operations, which have to be derived for the machines in question. This requires access to the machines or at least a detailed knowledge. The approach then creates a runtime-function model. Another analytical model approach, in the area of superscalar processors, is [5]. Karkhanis et al. use architecture-dependent information such as statistics of branch mispredictions and cache misses to build a performance-prediction model. While predictions are good, the approach requires time-consuming executions to obtain the characteristics used.

The approach which probably is most similar to ours is [1, 2]. Joshi et al. use program characteristics to make statements on the similarities of programs [1]. In contrast to [4, 5], they do not use microarchitecture-dependent measures to characterise programs, but microarchitecture-independent ones, such as the instruction mix and branch probabilities. This limits the approach to a certain instruction-set architecture and a specific compiler. Furthermore, generating the measures requires simulation or execution. Based on [1], Hoste et al. use program-similarity measures and predict performance with programs from the SPEC CPU 2000 benchmark suite [2]. They then normalise the microarchitecture-independent characteristics with techniques such as principal-component analysis. These normalised measures represent a point in the so-called benchmark space for every program. The performance of an unknown application is then predicted as the weighted average of execution times of programs in the neighbourhood. To obtain enough reference points with known performance measures, the authors rely on programs from a benchmark suite. Like our approach, [2] can determine the best platform for an application. Its advantage is that predictions tend to be more accurate than ours. This is achieved by the limitation to a certain instruction-set architecture and by executing or simulating the application in question on an existing platform. Our approach in turn has no such limitation. It uses only measures generated from the source code and does not require any simulation or execution of the program in question.

İpek et al. do not only predict performance for different systems [6] but also contribute to hardware design. The number of design alternatives in computer architecture is huge, and it is hard to develop a good architecture for certain applications. The combination of design parameters is often described as a point in a design space, as is done in [6]. The authors simulate sampled points in design spaces corresponding to the memory hierarchy and to chip multiprocessors, which then serve as input for neural networks. They then use these networks for performance predictions of new computer-architecture designs.

Our approach is also related to work in the field of *graph classification*. One of the first studies, with an application in chemistry, is [7]. The authors propose a graph-classification framework which consists of three steps: (1) search for frequent subgraphs which are then used as binary features indicating if a certain subgraph is included in a graph, (2) a feature-selection strategy to reduce the dimensionality and (3) a model-learning step. Our approach is similar in that frequent subgraphs are generated which serve as features to learn a classification model. However, our application does not require feature selection, but a more complex approach to integrate classifications to a prediction for a program.

3 Control-Flow-Graph Representation

Control-Flow-Graph Generation. Control-flow graphs (CFGs) [3] are a common program representation in compiler technology. They are static in nature and can be derived from source code. They represent all control flows which can possibly occur. The nodes of a CFG stand for *basic blocks* of code, i.e., sequences of statements without any branches. The edges represent the possible control flows, i.e., edges back to previous nodes for loops and different branches for condition statements. This paper studies the usual setting where one CFG describes a single function.

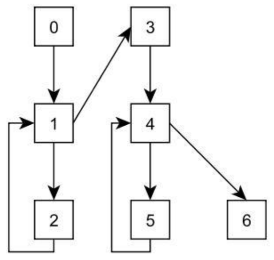
For our work, it is important to define an architecture-independent representation of CFGs. In particular, some compiler optimisations affect the structure of the CFGs, e.g., loop unrolling. This might vary when making optimisations for different architectures. Therefore, we use the *GNU compiler collection* (*gcc*) to obtain CFGs using the `-O0`-flag, which prevents the compiler from making any optimisations. However, the *gcc* normalises the source code by using canonical constructs for artefacts which can be expressed in several ways in the programming language. This normalisation is an advantage, as the same algorithms tend to be expressed in the same way, even if the source-code representations vary.

```
# BLOCK 0
InitAinfo (ainfo);                                    // Call
nseq.0 = (unsigned int) nseq;
D.4949 = nseq.0 * 4;
D.4950 = sre_malloc (&"alignio.c"[0], 50, D.4949);  // Call
aseq = (char * *) D.4950;
idx = 0;
goto BLOCK 1;

# BLOCK 1
if (idx < nseq) goto BLOCK 2; else goto BLOCK 3;

# BLOCK 2
idx.1 = (unsigned int) idx;
// ...
D.4957 = sre_malloc (&"alignio.c"[0], 52, D.4956);  // Call
D.4958 = (char *) D.4957;
*D.4954 = D.4958;
idx = idx + 1;
goto BLOCK 1;
```

(a)



(b)

Figure 1. Example control-flow graph (CFG).

Figure 1 is an example of a CFG: (a) is a part of the function `AllocAlignment` from the SPEC program 456.hmmer in a (simplified) intermediate representation derived with the *gcc*. (b) is the CFG derived from the function. In the intermediate representation, `if` and `goto`-statements represent loops.

In addition to the nodes displayed in Figure 1(b), some CFG representations introduce additional entry and exit nodes which do not represent any code. They do not represent any performance-related information. To obtain a more concise graph representation, we do not make use of such nodes.

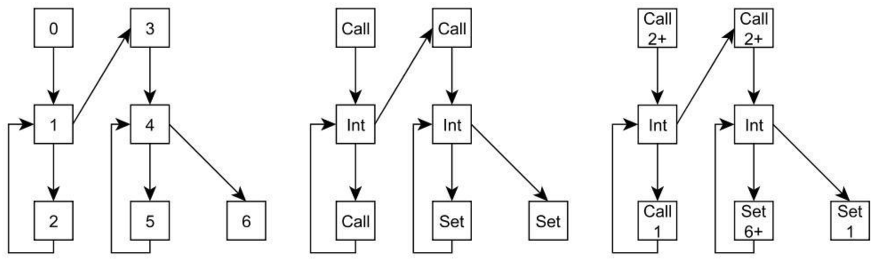
Node Labelling. To mine CFGs, it seems that one could analyse the pure graph structure ignoring the content of the nodes. However, such an approach would lose a lot of (performance-related) information. To avoid this, we propose the following mapping of source code to node labels:

– *FP* for blocks containing *floating-point operations.*
– *Call* for blocks without *FP* operations but *calls of other functions.*
– *Set* for blocks without *FP* or *Call* operations but *load/store* operations.
– *Int* for blocks containing none of the above (simple *integer* ALU operations).

So far, the labelling scheme leaves aside the actual number of statements in a node, which might be important as well. Furthermore, the different labels are quite imbalanced: *FP* is assigned to only 3% of the nodes from CFGs in the SPEC CPU 2000 and 2006 programs used (see Table 1), *Call* is assigned to 19%, *Set* to 61% and *Int* to 17%. Large sets of nodes with the same label, as well as only few different ones, have a negative effect on the performance of graph-mining algorithms (cf. [8]). We therefore propose a more fine-grained labelling scheme: We divide the blocks labelled with *Call* into blocks containing one function call, $Call_1$, and blocks with two or more calls, $Call_{2+}$. As there is a larger variety in the number of load/store operations, and the *Set* class of labels is the largest, we divide it in four labels. Each of them has approximately the same number of corresponding nodes. Blocks with one load/store operation are labelled Set_1, blocks with two with Set_2, those with three to five with $Set_{3\text{-}5}$ and those with more than five Set_{6+}. In preliminary experiments, graph mining was one order of magnitude faster with the fine-grained labels, while the accuracy of predictions did not decrease. Figure 2 provides examples of the labelling schemes.

CPU 2000			**CPU 2006**		
164.gzip	183.equake	255.vortex	400.perlbench	436.cactus-ADM	464.h264ref
175.vpr	186.crafty	256.bzip2	401.bzip2	445.gobmk	470.lbm
176.gcc	188.ammp	300.twolf	403.gcc	454.calculix	481.wrf
177.mesa	197.parser		429.mcf	456.hmmer	482.sphinx3
179.art	253.perlbmk		433.milc	458.sjeng	
181.mcf	254.gap		435.gromacs	462.lib-quantum	

Table 1. SPEC benchmark programs used.

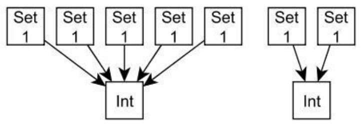


(a) orig. CFG from Fig. 1 (b) coarse-grained (c) fine-grained

Figure 2. Examples of CFG node labelling schemes.

4 Control-Flow-Graph Mining

For our experiments (see Section 6) we use the C/C++ programs from the SPEC CPU 2000 and 2006 benchmarks suites listed in Table 1. This results in a set G_{CFG} of approximately 27,000 CFGs belonging to the 31 programs. The graphs have an average size of 22 nodes, with high variance. Approximately 30% of the graphs consist of one node only, while roughly 12% have more than 32 nodes, including a few with more than 1,000 nodes. As graphs with a single node do not contain any information which is useful for our scheme, we omit these graphs. This reduces the size of G_{CFG} to approximately 19,000 graphs.



(a) (b)

Figure 3. Illustration of CFGs with problematic node degrees.

Though the average graphs in G_{CFG} are not challenging from a state-of-the-art graph-mining perspective, the large CFGs do lead to scalability problems. We can mine the entire graph set with, say, the *gSpan* algorithm [9] in a reasonable time, but only with relatively high minimum support values (*minSup*). In preliminary experiments, this leads to the discovery of very small substructures, i.e., with a maximum of two nodes only. Further, the *CloseGraph* algorithm [10] is not helpful in our case. This is because there rarely are closed graphs (with exactly the same support) which offer pruning opportunities. In preliminary experiments, *CloseGraph* even increased the runtime because of the search for closed structures. – To obtain larger subgraph patterns from our CFG dataset G_{CFG} by means of frequent subgraph mining, we inspect the larger graphs further. We observe that they frequently contain nodes with a high degree. This causes the scalability problems. Node *Int* in Figure 3(a) serves as an illustration. In many cases, bulky `switch-case` statements which lead to many outgoing edges in one node and many incoming edges in another one cause these high degrees. Typically, programs treat many different `case` branches similarly. Therefore, the corresponding nodes frequently have the same labels (Set_1 in Figure 3(a)). The problem with these situations is the number of potential embeddings. As an example, we want to find out the embeddings of the graph in Figure 3(b) in the graph in Figure 3(a). Algorithms like *gSpan* search for all such embeddings (subgraph isomorphisms), which is NP-complete [11]. In the example, there are 20 distinct embeddings. G_{CFG} contains nodes with a degree of 720, which leads to extreme numbers of possible embeddings. To conclude, the graphs in G_{CFG} with high node degrees prohibit mining of G_{CFG} with reasonably low *minSup* values. Early experiments with roughly the same data set, but with the largest graphs excluded, have led to encouraging results in turn. We therefore propose the following mining steps:

1. Mining of all graphs G_{small} smaller than a certain threshold t_{size}, resulting in a set of frequent subgraphs SG. ($G_{\text{small}} := \{g \in G_{\text{CFG}} \mid size(g) \leq t_{\text{size}}\}$)
2. Search for subgraph isomorphisms of the subgraphs SG within the large graphs G_{large} which have been omitted in Step 1. ($G_{\text{large}} := G_{\text{CFG}} \setminus G_{\text{small}}$)
3. Unified representation of all graphs in G_{CFG} with feature vectors.

Before we can derive frequent subgraphs (SG) in Step 1, we first systematically identify combinations of the support in G_{small} ($minSup_{\text{small}}$) and the size threshold (t_{size}), which let us mine the data in reasonable time. We do this by means of preliminary mining runs. In general, one wants to have a low $minSup_{\text{small}}$, to facilitate finding large and significant subgraphs, and a high t_{size}. This is to ensure that only few patterns, namely those only included in G_{large}, are missed. With our dataset, we found a t_{size} of 32 (corresponding to G_{large} with a size of 12% of $|G_{\text{CFG}}|$) and a $minSup_{\text{small}}$ of 1.7% to be good values. We assume that similar t_{size} values can be found when mining other CFG datasets, since our numbers are based on a sample of 27,000 CFGs. We then mine G_{small} with the *ParSeMiS* implementation[1] of the *gSpan* algorithm [9], which is a state-of-the-art algorithm in frequent subgraph mining. This results in a set of subgraphs SG which are frequent within G_{small}.

In Step 2 we determine which graphs in G_{large} contain the subgraphs in SG by means of a subgraph-isomorphism test. Although this problem is NP-complete [11], we benefit from properties of our specific dataset. E.g., there are no cliques larger than three nodes, and the average node degree of 3.4 is relatively low. Therefore, this step is less expensive in terms of runtime than Step 1.

In Step 3, we represent every CFG in G_{CFG} with a feature vector. Such a vector contains one bit for every subgraph in SG. A bit states if the respective subgraph is included in the CFG.[2] As the subgraphs in SG have a minimum size

[1] `http://www2.informatik.uni-erlangen.de/Forschung/Projekte/ParSeMiS/`

[2] In preliminary experiments we have used the numbers of embeddings. This has not yielded better results. However, generating boolean features makes the subgraph-isomorphism test in Step 2 much easier.

of one edge, single nodes are not included. We believe that these nodes provide important information as well. We therefore extend the vector with single nodes labelled with the label classes described in Section 3. For these features, we use integers representing their number of occurrences. This allows for a more precise description of the operations contained in a CFG, i.e., in a function. Summing up, we represent every CFG $g \in G_{\text{CFG}}$ with the following vector:

$$g := (sg_1, sg_2, ..., sg_n, FP, Call_1, Call_{2+}, Set_1, Set_2, Set_{3\text{-}5}, Set_{6+}, Int)$$

where $sg_1, sg_2, ..., sg_n \in SG$ are boolean features, $|SG| = n$, and FP, $Call_1$, $Call_{2+}$, Set_1, Set_2, $Set_{3\text{-}5}$, Set_{6+} and Int are integers.

Our technique based on mining G_{small} bears the risk that certain subgraphs may not be found, namely those contained in G_{large}. In the worst case, a subgraph sg is contained in every graph in G_{large}, corresponding to 12% of $|G_{\text{CFG}}|$ in our case, but hardly misses the minimum support when only looking at G_{small}. In other words, sg becomes a part of the result set SG if it has a support of 13.5% in G_{CFG}. More formally, we can guarantee to find all subgraphs with the following minimum support in G_{CFG}:

$$minSup_{\text{guarantee}} = \frac{|G_{\text{large}}|}{|G_{\text{CFG}}|} + \frac{|G_{\text{small}}|}{|G_{\text{CFG}}|} \cdot minSup_{\text{small}}$$

In our dataset, $minSup_{\text{guarantee}}$ is 13.5%. However, we find many more subgraphs as we are mining with a much lower $minSup_{\text{small}}$ in G_{small}. A direct mining of G_{CFG} with a $minSup$ of 13.5% was not possible due to scalability problems – the lowest $minSup$ value possible in preliminary experiments was 20%. Our approach succeeds due to the relatively small fraction of large graphs. It is applicable to other datasets as well, but only if the share of large graphs is similar.

5 Classification Framework

We now describe the subsequent classification process, to predict the best computer architecture for a given program. We formulate this prediction as the selection between a number of architectures. The architectures are the classes in this setting. In the following, we focus on the prediction of the faster one of two architectures. This is due to the limited number of architectures with data available, as we will explain. However, with more training data, we do not see any problems when choosing from an arbitrary number of systems.

For the classification, we are faced with the challenge that our substructure based feature vectors are descriptions at the function level, while we are interested in predictions for a program as a whole. At the same time, SPEC publishes runtimes of several systems – this information is at the level of programs as well. Potentially helpful information on the execution of functions, such as the number of calls and the execution time, is not available. This situation is completely natural, and this is why we propose an approach that is supposed to work without that information. In the following, we develop an approach which does not need any more runtime-related information than the one typically available.

One way to do program-level predictions is to aggregate the information contained in the feature vectors to the program level. Then a classification model could be learned with this data. As one program consists of many functions (typically hundreds to thousands in the benchmarks), such an aggregation would lose potentially important fine-grained information. Further, it would force us to learn a model based on only few tuples (programs). The problem is the limited availability of systems which are evaluated with more than one benchmark suite. E.g., a system evaluated with SPEC CPU 2006 is rarely evaluated with the now outdated CPU 2000 benchmark suite as well. The number of benchmark programs whose execution time for the same machines is known is therefore limited in practice – and deriving this information would be tedious.

Hence, we propose a classification framework containing a simplification which might seem unusual or 'simplistic' at first sight: To learn a classifier at the function level, we assign the fastest architecture for the program as a whole to all feature vectors describing its CFGs (functions). This simplification, caused by a lack of any respective information, clearly does not take the characteristics of the different functions of a program into account. It also ignores the potentially imbalanced distribution of execution times of the individual functions. However, our hope is that the large amount of function-level training data compensates these issues, and we will show this. Once the classification model is learned, we use it to classify functions from programs with unknown runtime behaviour. We aggregate these predictions to the program level with majority vote.

To learn a prediction model, any classification technique can be used in principle. We have carried out preliminary experiments with support vector machines, neural networks and decision trees, and the results were best with the latter. We therefore deploy the C5.0 algorithm, a successor of C4.5 [12]. Our implementation in the SPSS Clementine data-mining suite lets us specify weights for every tuple during the learning process, to emphasise certain tuples. With our approach, we weight every feature vector with two factors:

1. One class might consist of many more tuples than the other one in the learning data set. As this typically leads to an increased number of predictions of the larger class, we increase the weight of the under-represented one. We use the fraction of the number of functions in the larger class divided by the one of the under-represented class as the weight.
2. The difference in runtime of some programs on the two machines considered might be large, while it is marginal with other programs. To give a higher influence to a program with very different execution times, we use the ratio of the execution time of the slower machine to the one of the faster machine as another weight for the feature vectors of a program.

To fuse the classifications on the function level, we use the majority-vote technique [13]. This is standard to combine multiple classifications. In extensive experiments, we have evaluated alternative weights for learning, as well as different combination schemes. In particular, we have examined the two weights mentioned – as well as other ones – as weights for the majority-vote scheme.

System 1	**System 2**	**System 3**
Bull SAS NovaScale B280	Dell Precision 380	HP Proliant BL465c
Intel Xeon E5335,	Intel Pentium 4 670	AMD Opteron 2220
QuadCore, 2.0 GHz	SingleCore, 3.8 GHz	DualCore, 2.8 GHz
2x4 MB L2, 8 GB RAM	2 MB L2, 2 GB RAM	2x1 MB L2, 16 GB RAM
System 4	**System 5**	**System 6**
Intel D975XBX Motherboard	FSC Celsius V830	IBM BladeCenter LS41
Intel Pentium EE 965	AMD Opteron 256	AMD Opteron 8220
DualCore, 3.7 GHz	SingleCore, 3.0 GHz	DualCore, 2.8 GHz
2x2 MB L2, 4 GB RAM	1 MB L2, 2 GB RAM	2x1 MB L2, 32 GB RAM

Table 2. Systems used for runtime experiments.

Note that, while the graph-mining step of our approach may be time-consuming, it only takes place once, in order to build the classification model. The prediction for a new program is much faster, once the model is built.

6 Experiments

In this section we present our experiments and results. For the programs listed in Table 1, the runtimes on a number of systems are published on the SPEC homepage[3]. We make use of this data and use a subset of the systems available, listed in Table 2. Not every system is evaluated with the older CPU 2000 and the more recent CPU 2006 benchmark. Therefore, our selection is motivated by the availability of runtimes for both benchmarks. Although there would have been a few more systems available, we have only used the ones mentioned. They allow us to set up experiments where the balance of systems being fastest with some programs and systems being fastest with other programs is almost equal. This eases the data-mining process. In reality, equality is not necessary when enough data is available. Our experiments cover single-, dual- and quadcore architectures as well as different memory hierarchies and processors, see Table 3.

We use the classification framework as described in Section 5, along with 2-fold cross-validation. We use partitions which are stratified with respect to the class and consist of roughly equal numbers of functions. For evaluation we derive the accuracy, i.e., the percentage of programs with correct prediction, and the speedup in terms of execution time. To obtain the latter, we first calculate the total runtime of all programs, each one on the machine predicted. This allows us to derive a percentage, 'speedup reached'. 0% is achieved when the slowest architecture is always selected, 100% if the predictor assigns the fastest architecture to every program. Table 3 contains both measures as well.

We achieve an accuracy of 69% on average. This indeed corresponds to a speedup. On average, we reach 71% of the speedup that would have been possible in theory. Further, our results show that there is a strong relationship

[3] `http://www.spec.org/benchmarks.html`

Exp.	Platforms	Processors	accuracy	speedup reached
1	System 5 vs. System 2	Opteron vs. Pentium 4	71.0%	83.2%
2	System 3 vs. System 4	Opteron vs. Pentium EE	64.5%	58.6%
3	System 6 vs. System 1	Opteron vs. Xeon	71.0%	72.6%

Table 3. Experiments and results.

between CFGs and runtime behaviour. Although it would be interesting, we do not compare our results to approaches making use of execution properties, such as [2]. Such a comparison is not possible, since [2] uses other benchmarks and target machines, as well as other evaluation metrics.

To savour our experimental results, one should take several points into account. The programs in the SPEC CPU benchmarks are relatively similar in the sense that they are all compute-intensive (not I/O or memory-intensive). The systems considered are relatively similar as well. All of them are off-the-shelf systems, differing mainly in their configuration. However, it does not affect runtime by much if, say, the number of processors or the size of RAM changes. The programs considered do not use multiple threads and always fit in memory. The only architectural difference of some significance is the instruction set used, i.e., *x86* in the Xeon and Pentium systems and *x86-64* in the Opteron systems.

7 Conclusion and Future Work

In the computer industry, it is important to know which platform provides the best performance for a given program. Most approaches proposed so far require in-depth knowledge of the systems or of runtime-related characteristics. One must obtain them using expensive simulations or executions.

This paper has proposed an approach solely based on the static analysis of programs and on runtime data from benchmark executions, which is available online. It analyses the control flow graphs (CFGs) of the functions. Based on graph-mining results, it correlates programs with similar CFG substructures and assumes that their runtime is similar as well. This leads to our prediction framework for learning at function level and a classifier-fusion technique to derive program-level predictions. Our framework can predict the runtime behaviour of programs on the target platforms. Though our approach to assign the best architecture for a program as a whole to its classes might be unusual and somewhat risky, it is beneficial according to our evaluation. In experiments with the SPEC CPU 2000 and 2006 benchmarks we obtain an accuracy of 69% on average.

From a graph-mining perspective, we propose a technique which can deal with situations when the usual approach does not scale, e.g., because of high node degrees. Our technique leaves aside few graphs in the graph-mining step which are 'problematic'. Then, it maps the results to the graphs we left out before. We provide guarantees on the overall support.

One aspect of our future work is to improve the prediction quality further. We currently investigate the usage of software metrics which provide additional information on a function. We also investigate program-dependence graphs [14]. They feature data dependencies in addition to control-flow information. Such information might help regarding certain aspects of computer architectures, e.g., pipelining and register usage. However, the graphs are much larger than CFGs. Another aspect is a further investigation from the computer architecture point of view. Rather than correlating properties from source-code representations, e.g., CFG substructures, with architectures as a whole, we are interested in ties with micro-architectural details, such as the cache architecture. Such insights would be of enormous help when designing hardware for specific applications.

Acknowledgments

We thank Dietmar Hauf for much help with all aspects of this study and Wolfgang Karl and David Kramer for their guidance regarding computer architecture.

References

[1] Joshi, A., Phansalkar, A., Eeckhout, L., John, L.: Measuring Benchmark Similarity Using Inherent Program Characteristics. IEEE Trans. Comput. **55**(6) (2006) 769–782
[2] Hoste, K., Phansalkar, A., Eeckhout, L., Georges, A., John, L.K., Bosschere, K.D.: Performance Prediction Based on Inherent Program Similarity. In: Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT). (2006)
[3] Allen, F.E.: Control Flow Analysis. In: Proc. Symposium on Compiler Optimization. SIGPLAN Notices (1970) 1–19
[4] Kühnemann, M., Rauber, T., Runger, G.: A Source Code Analyzer for Performance Prediction. In: Proc. Int. Symposium on Parallel and Distributed Processing. (2004)
[5] Karkhanis, T.S., Smith, J.E.: A First-Order Superscalar Processor Model. SIGARCH Comput. Archit. News **32**(2) (2004) 338
[6] İpek, E., McKee, S.A., Singh, K., Caruana, R., de Supinski, B.R., Schulz, M.: Efficient Architectural Design Space Exploration via Predictive Modeling. ACM Trans. Archit. Code Optim. **4**(4) (2008) 1–34
[7] Deshpande, M., Kuramochi, M., Wale, N.: Frequent Substructure-Based Approaches for Classifying Chemical Compounds. IEEE Trans. Knowl. Data Eng. **17**(8) (2005) 1036–1050
[8] Chakrabarti, D., Faloutsos, C.: Graph Mining: Laws, Generators, and Algorithms. ACM Comput. Surv. **38**(1) (2006) 2
[9] Yan, X., Han, J.: gSpan: Graph-Based Substructure Pattern Mining. In: Proc. Int. Conf. on Data Mining (ICDM). (2002)
[10] Yan, X., Han, J.: CloseGraph: Mining Closed Frequent Graph Patterns. In: Proc. Int. Conf. on Knowledge Discovery and Data Mining (KDD). (2003)
[11] Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)
[12] Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
[13] Kuncheva, L.I.: Combining Pattern Classifiers: Methods and Algorithms. John Wiley & Sons (2004)
[14] Ottenstein, K.J., Ottenstein, L.M.: The Program Dependence Graph in a Software Development Environment. SIGSOFT Softw. Eng. Notes **9**(3) (1984) 177–184