From Source Code to Runtime Behaviour: Software Metrics Help to Select the Computer Architecture.

Frank Eichinger, David Kramer, Klemens Böhm and Wolfgang Karl

Abstract The decision which hardware platform to use for a certain application is an important problem in computer architecture. This paper reports on a study where a data-mining approach is used for this decision. It relies purely on source-code characteristics, to avoid potentially expensive program executions. One challenge in this context is that one cannot infer how often functions that are part of the application are typically executed. The main insight of this study is twofold: (a) Source-code characteristics are sufficient nevertheless. (b) Linking individual functions with the runtime behaviour of the program as a whole yields good predictions. In other words, while individual data objects from the training set may be quite inaccurate, the resulting model is not.

1 Introduction

The question which computer architecture is best suited for a certain application is of outstanding importance in the computer industry. With the continuous refining of computer architectures, this problem becomes even more challenging. Think of the high degree and various forms of parallelism (multicores), heterogeneity due to application-specific accelerators, interconnection technology on the chip, or the memory hierarchy. The design space is huge and leads to a broad variety of processor architectures. It is not at all obvious which architecture is best suited for a specific application. For example, due to the branch-prediction unit, an application with predictable branches benefits from a long pipeline, while a shorter pipeline is bet-

Frank Eichinger, Klemens Böhm
Institute for Program Structures and Data Organisation (IPD)
Universität Karlsruhe (TH), Germany, e-mail: `{eichinger,boehm}@ipd.uka.de`

David Kramer, Wolfgang Karl
Institute for Computer Science and Engineering (ITEC)
Universität Karlsruhe (TH), Germany, e-mail: `{kramer,karl}@ira.uka.de`

ter for unpredictable branch behaviour. The question which architecture yields the best performance is particularly important for high-performance computing where an expensive system is purchased for a few or even only one application.

Traditional approaches use experimental executions, simulations or analytical models to identify the best computer architecture for a given application. For instance, when a computing centre plans to procure a new cluster for a specific application, one way to do so is to compare the runtime behaviour of this application on different platforms. This obviously is time-consuming and expensive, and the platforms in question must be available in the first place. Similar arguments apply to state-of-the-art simulation approaches: In-depth simulation is time-consuming, in particular with machine models that are sophisticated. Finally, due to the increasing complexity of computer systems, establishing analytical models of the computer architectures in question is extremely hard. This may lead to a relatively poor reliability of these models, compared to experimental executions and simulations. In consequence, techniques are sought which help to decide between several platforms for a specific application. Ideally, such techniques should not require any execution or simulation and should be based on an analysis of the application in question. Some approaches exist which can make a decision between several platforms [5, 8]. They rely on the assumption that similar programs perform alike when executed on the same machine. However, in addition to measures deduced from the source code, these approaches make use of runtime-related characteristics, such as branch probabilities or instruction counts. To generate these characteristics, simulations or program runs on real hardware are necessary.

This article reports on the results of a study that investigates another method to determine the best computer architecture for a given application. The method likewise assumes that similar applications have similar execution behaviour. But in contrast to the previous work, we have consciously decided not to take any runtime-related information of the application in question into account. In this current study we characterise the application entirely by means of measures gained from the source code. In other words, we hypothesise that there is a strong correlation between program properties encoded in the source code and the execution behaviour, and that this correlation can be exploited. This hypothesis may appear to be unsettling – taking only source-code characteristics into account obviously is much less informative than runtime behaviour! In particular, it is difficult to impossible to infer how often a certain function is typically executed. Another issue is that source-code metrics, i.e., existing measures that quantify characteristics of the source code, typically are defined on the function level rather than on the level of entire programs, while we are interested in predictions for programs as a whole. Having said this, the method examined here is a data-mining approach with the following distinctive feature: It links individual functions with the runtime behaviour of the program as a whole. Even though this approach clearly is simplistic, i.e., the characterisation of individual functions may be *very* inaccurate, it yields a prediction accuracy for entire programs which is surprisingly high. In retrospect, our explanation is as follows: Since applications typically consist of a large number of functions, there is a lot of training data which, on average, compensates for that simplification. I.e., we provide evidence that source-code characteristics alone are indeed helpful to predict a good computer architecture. More specifically, our contributions are as follows:

Software Metrics. The software-engineering community has proposed a number of software metrics in order to represent source-code characteristics and properties. Originally, these metrics have been cast as quality measures rather than as performance indicators. Preliminary investigations of ours have revealed that measures based on the control flow of functions are particularly promising to predict runtime behaviour. Consequently, we define and derive a number of metrics, such as graph invariants, based on the control-flow graphs (CFGs) [1] of the functions. We use these metrics in addition to more common ones.

Classification Framework. We propose a classification setting for our specific context and evaluate it. This setting is not obvious: While most metrics are available at the function level, we want to choose the best architecture for a program as a whole. Instead of potentially lossy aggregation approaches, we propose a framework where we first learn at the function level before deploying classifier-fusion techniques to come up with predictions at the program level.

Evaluation. Our case study features an evaluation using five systems from the online database of the SPEC CPU 2000 and 2006 benchmark suites. The results are that, for 'relatively similar' computer architectures to choose from, and with the runtime behaviour of only few programs used as training data, our approach achieves an average prediction accuracy of 78% when choosing between two systems.

Correlation of Software Metrics and Runtime Behaviour. Our main concern, from a 'research' perspective, has been to confirm (and to exploit) the relationship between source-code properties and runtime behaviour, on different platforms. Besides the fact that the approach investigated here does indeed yield a statement regarding the computer architecture best suited, our evaluation shows that the correlation between source-code properties and runtime behaviour is remarkably strong.

Paper outline: Section 2 presents related work, Section 3 describes the process of acquiring software metrics, before we describe the data-mining process in Section 4. Section 5 presents our results, which are discussed in Section 6. Section 7 concludes.

2 Related Work

In the past, various approaches to predict the runtime or the runtime behaviour of given applications have been proposed. Newer approaches propose the use of machine-learning approaches for this prediction.

In [2, 16] the authors use multilayer neural networks to predict the performance of the multigrid solver SMG 2000 on a BlueGene/L cluster. The parameter space includes the cluster configuration as well as the size of the grid used. The training set used consists of performance results on an actual platform using a collection of random points from the parameter space. In contrast to our approach, these approaches can only be used to predict the performance of a parametrised application on a cluster with different configurations. In addition, they require time-consuming training-data generation.

Another possible use of neural networks is described in [6]. Here, İpek et al. use neural networks to predict the performance of points in the design space. Neural networks are used to approximate the design space and to create a model of it. The model built predicts the performance of points with high accuracy and has been applied to memory hierarchy and chip-multiprocessor design spaces.

To ease the generation of analytical models of complex high-performance systems, Kühnemann et al. have developed a compiler tool for automated runtime prediction of parallel MPI programs [9]. The tool analyses the source code of MPI programs to create an appropriate runtime-function model for the communication overhead and for the computation. Properties of the underlying machine are needed for proper prediction of the computation effort.

Another method for performance prediction is [8] from Joshi et al. They use inherent program characteristics to measure the similarity between programs. Instead of using microarchitecture-dependent measures for characterisation, such as cycles per instruction, cache-miss rate or runtimes, they use microarchitecture-independent ones. These measures include the instruction mix, the size of the working set and branch probabilities. To generate the measures, either simulation or execution of the application is necessary. Based on [8], the authors exploit the similarity between programs for performance prediction of applications in the SPEC CPU 2000 benchmark suite [5]. They use microarchitecture-independent characteristics and performance numbers from an application to build a so-called benchmark space. To predict the performance of an application, the developer has to compute a point in the benchmark space using the same characteristics. Comparing our approach to [5] reveals that both approaches can predict the runtime-behaviour of an application in question on given platforms and have advantages and disadvantages. [5] uses runtime-related microarchitecture-independent characteristics in the prediction process. The advantage is that predictions are likely to be more precise. A drawback is that the execution of the application on an existing platform or a detailed simulation is necessary. Saveedra and Smith [14] use a similar approach as proposed in [5], but they use program and machine characteristics to estimate the performance of a given Fortran program on an arbitrary machine. A drawback of all these approaches is the usage of architecture-dependent characteristics which are time-consuming to create. Our approach in turn does not require such characteristics.

Finally, [3] studies the same problem as this current paper, but with a different approach based on graph mining and control-flow graphs. The technique described here yields better results.

3 Software-Metric Data

In this study we try to predict the best-performing platform by means of standard data-mining techniques. More precisely, we only use software characteristics derived from the source code, but no runtime or platform-related information. In order to use source-code metrics as input for data-mining algorithms, we describe the software entities with feature vectors of software-metrics values. The software-engineering community has been very active in defining metrics based on source code [7]. These metrics are primarily used to quantify the quality and the maintainability of applications and have not been intended to characterise runtime behaviour. However, we deploy a number of these metrics as well as some metrics defined by ourselves exactly to this end. Source-code metrics cover various aspects of software, e.g., statements used, source-code quality, complexity and understandability. We decided not to use any of the numerous metrics dealing with source-code size and understandability, such as the various lines of code (LOC) measures or any measure concerned with comments. These metrics strongly depend on the coding scheme used and do not have any impact on the program complexity and therefore on the runtime behaviour. Besides these measures, we do not exclude any other metric a priori. This is because we are not aware of any previous experience in predicting runtime behaviour based on source-code metrics. Even if some metrics such as McCabe's cyclomatic complexity [11] are debatable [15], we leave it to the data-mining algorithm to decide which metrics are useful for our purpose.

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

Table 1 SPEC benchmark programs used.

In order to derive source-code metrics from the benchmark programs, we employ a standard tool from software engineering: RSM from M Squared Technologies LLC. We derive the metrics and characteristics for every function in every program from the SPEC CPU 2000 and 2006 benchmark suites. As these benchmarks have been assembled with the intention to cover a broad variety of different domains, they are a good basis for the classification of new programs. – The RSM tool does not provide any metrics from Fortran source code. We therefore limit our experiments to the C and C++ benchmark programs. Table 1 lists all 31 programs we use for our experiments.

RSM delivers a huge variety of metrics, in particular *counts*, *quality measures* and *complexity measures*.[1] The ones we use for our purpose are listed in Table 2. *Counts* refer to simple counts of statements and syntactical elements such as braces and brackets. The *quality measures* refer to counts of certain kinds of program quality, which could also have an impact on execution behaviour. For example, one of

[1] See the RSM documentation for details on specific metrics:
`http://msquaredtechnologies.com/m2rsm/docs/rsm_metrics.htm`

these counts is increased whenever a variable is assigned to a literal value. Finally, the *complexity measures* describe the complexity of the function interface and the cyclomatic complexity [11] of the underlying control-flow graph [1].

Counts	memory_free_count	notice_50_count	freq_sdev
abort_count	open_brace_count	notice_50_percent	freq_sum
break_count	open_bracket_count	notice_50_type	loop_depth_max
case_count	open_paren_count	notice_119_count	loop_depth_mean
class_count	return_count	notice_119_percent	loop_depth_sdev
close_brace_count	switch_count	notice_119_type	loop_depth_sum
close_bracket_count	typedef_count	notice_all_count	loops_max
close_paren_count	union_count		loops_mean
const_count	while_count	**Complexity Measures**	loops_sdev
default_count		cyclomatic_complexity	loops_sum
define_count	**Quality Measures**	interface_complexity	nodes
do_count	notice_22_count	interface_params	num_pred_max
else_count	notice_22_percent	interface_returns	num_pred_mean
enum_count	notice_22_type	total_complexity	num_pred_sdev
exit_count	notice_27_count		num_pred_sum
for_count	notice_27_percent	**CFG Measures**	num_succ_max
goto_count	notice_27_type	back_edges_max	num_succ_mean
if_count	notice_28_count	back_edges_mean	num_succ_sdev
include_count	notice_28_percent	back_edges_sdev	num_succ_sum
inline_function_count	notice_28_type	back_edges_sum	record_count
literal_strings_count	notice_44_count	edges	registers
macros_count	notice_44_percent	freq_max	
memory_alloc_count	notice_44_type	freq_mean	

Table 2 Source code and control-flow-graph (CFG) measures used.

The set of metrics from RSM includes only a few measures regarding the structure and the complexity of the application. As observed in preliminary experiments, measures based on the control flow of functions might be important when predicting runtime behaviour. Therefore, we have decided to use more metrics than those provided by RSM and to derive measures from control-flow graphs [1]. Such graphs are widely used in software engineering and are a common way of representing code in compilers internally. Basic blocks of code without any jump statements are the nodes, and the control dependencies between these blocks are the edges. We use the front-end of the GNU Compiler Collection (gcc) to derive control-flow graphs of all functions. From these graphs, we calculate some graph invariants, *CFG Measures* (cf. Table 2), such as the number of nodes and edges of a control-flow graph, the number of loops and the aggregated in- and out-degrees of the nodes of the graph. We use these measures as further metrics generated purely from the source code.

The metrics used (cf. Table 2) are certainly not an exhaustive set of metrics defined by the software-engineering community. For example, we do not take object-oriented metrics for the C++ programs into account. (Most of our programs are C programs.) However, our goal rather is to demonstrate and to make use of the correlation of source-code properties and runtime behaviour and not to investigate the usefulness of any metric possible. In Section 5 we will demonstrate that the metrics used are well suited, and that the results are useful.

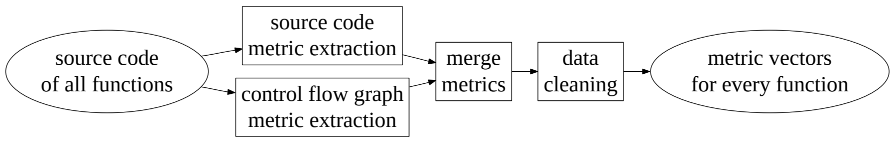

Fig. 1 Metric-generation workflow.

Before actually using the data we do some data cleaning in order to ease the data-mining process. We do this by eliminating measures containing null values in more than 95% of all functions as well as measures displaying always the same value. We deem these attributes irrelevant as they do not influence the quality predictions but increase runtime. Figure 1 summarises the metric-generation process.

4 Data-Mining Process

A naïve way to predict the runtime behaviour of a given program is to describe every program using a set of metrics and to train a machine-learning algorithm on it. This approach is not practical. One reason is that most source-code metrics are defined on the function level rather than on the program level (cf. Section 3). The definition of program-level metrics would certainly be possible, but would require new tools, and – more important – such metrics would represent a very coarse view on the application. As an example, a count of `if`-statements for an entire program would rather be a statement on the total size of the program than on the complexity of its functions. Another possibility would be aggregating function-level measures to the program level. Such an approach, e.g., the arithmetic mean of *counts* of possibly thousands of functions belonging to one program, would lead to imprecise predictions due to the loss of potentially important fine-grained information. Another reason which opposes direct learning on program level is that suitable runtime information is usually only available for a relatively small number of programs. Learning with such small datasets typically is not feasible. This is because it is hard to generalise from tens of programs in order to learn a hopefully universal prediction model. The limited number of programs available is due to the huge costs of executing, say, hundreds of programs on a number of different platforms. We for our part use 31 benchmark programs (listed in Table 1) where the runtime information is available (cf. Section 5). In the following we develop an approach which works well with a number of programs of this magnitude. Note that we only make use of execution times (and no further runtime-related measures) of the benchmark programs in our learning dataset. To classify a program, no execution of the program is necessary – measures are only derived from the source code.

To address the problems discussed when learning on the program level, we have decided to perform machine learning on the function level. On the one side, this approach is feasible since the software metrics are available at this level of detail. On the other side, this approach leads to a new challenge: the labelling of the target class, which is required for every tuple in the training set (i.e., for every function). As we avoid executions or simulations in our scenario, we only know the target class at the program level. We therefore resort to the following simplification: Each function inherits the fastest platform from the program it is part of as its target class. Clearly, this approach ignores the characteristics of the different functions. But nevertheless, we hypothesise that it yields predictions of acceptable quality. The hope is that there are not too many functions that are untypical for the performance of the program, and a large number of functions in the training set will compensate those functions.

$Program/Function$	$Metric_1$	$Metric_2$	$\cdots$	$Metric_n$	$Platform$
$Program_A/function_1$	3	7	$\cdots$	34	*System 5*
$Program_A/function_2$	2	7	$\cdots$	45	*System 5*
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	*System 5*
$Program_A/function_{\|A\|}$	3	7	$\cdots$	24	*System 5*
$Program_B/function_1$	3	4	$\cdots$	42	*System 2*
$Program_B/function_2$	6	4	$\cdots$	61	*System 2*
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	*System 2*
$Program_B/function_{\|B\|}$	1	4	$\cdots$	23	*System 2*
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

Table 3 Example learning dataset.

Table 3 is an example of the datasets used. The rows correspond to each function ($function_i$) of every benchmark program ($Program_A, Program_B, \ldots$). For our experiments we use the benchmark programs listed in Table 1. The columns correspond to the source-code metrics, which we compute for every function ($Metric_1$, ..., $Metric_n$). The column *Platform* contains the target class, which is the same for all functions of a program. In the experiments we use all metrics enumerated in Table 2. The example dataset in Table 3 contains *System 2* and *System 5* as examples of two possible target platforms.

Using a dataset as in Table 3, we learn a prediction model to classify data without class information (*Platform*). In other words, such a model can make predictions for each function in isolation. To obtain a prediction for a program as a whole, which consists of a number of functions, these predictions need to be integrated. We for our part use the majority-vote technique, a standard scheme to combine multiple classifications [10]. Experiments with other combination techniques such as the usage of weights have lead to results which, on average, are not better than majority vote in our specific context.

In summary, our prediction approach consists of two steps. In the first step we learn a classification model (Figure 2). This is based on a training dataset as shown in Table 3, consisting of source-code metrics at the function level (cf. Section 3) and target systems derived from the execution times at the program level.

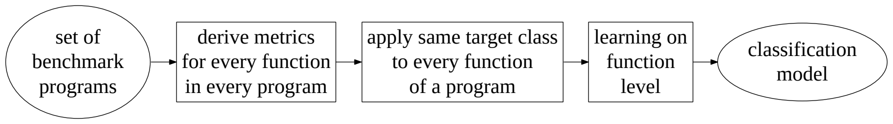
set of benchmark programs
derive metrics for every function in every program
apply same target class to every function of a program
learning on function level
classification model

Fig. 2 Learning workflow.

In the second step we predict the platform best suited for applications with unknown runtime behaviour (Figure 3). This prediction at the function level is based on the same metrics as used for learning. Afterwards, we merge these results into one overall prediction at the program level.

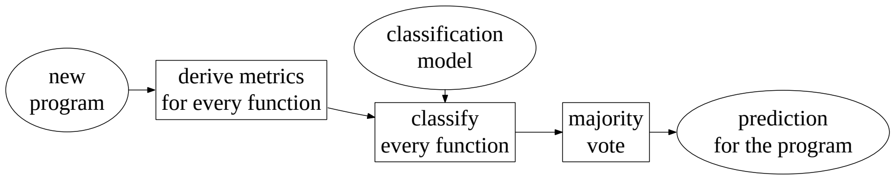

Fig. 3 Prediction workflow.

Besides the selection problem studied here using classification and classifier fusion techniques, the direct prediction of the runtime on a certain machine is a related, but different problem. At first sight, this could be done similarly using regression techniques. However, the runtime not only depends on the program given, but also on the parameters used and the data processed. Such information is available for certain configurations of, say, benchmark-program runs, but not for new programs in general. Hence, we limit our study to that selection problem.

5 Experiments

In order to demonstrate the usefulness of our approach and to show the correlations between properties observed in the source code and the runtime behaviour on different platforms, we perform a case study utilising the SPEC CPU benchmark programs as described in Section 3. As the SPEC CPU benchmarks are broad, i.e., cover many application domains, we have purposefully decided to deploy this benchmark, as opposed to any other set of training examples, e.g., home-grown programs.

The runtimes of the programs are published on the SPEC homepage[2], and we make use of this data. We decided to use a subset of the systems available, listed in Table 4. For these systems, runtime data is available for both, the CPU 2000 and the CPU 2006 benchmark. The systems cover single-, dual- and quadcore architectures as well as different memory hierarchies and processors: Intel Xeon, Intel Pentium 4,

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

Intel Pentium EE and AMD Opteron. There would have been a few more systems available. As we want to run experiments where one system is fastest with some programs and another system is fastest with other programs, it would not be reasonable to include systems performing (almost) always better than all other systems.

System	Vendor & Processor	Processor-Type & Memory
System 1	Bull SAS NovaScale B280	QuadCore
	Intel Xeon E5335, 2.0 GHz	2 x 4 MB L2-Cache, 8 GB PC2-5300
System 2	Dell Precision 380	SingleCore
	Intel Pentium 4 670, 3.8 GHz	2 MB L2-Cache, 2 GB PC2-4200
System 3	HP Proliant BL465c	DualCore
	AMD Opteron 2220, 2.8 GHz	2 x 1 MB L2-Cache, 16 GB PC2-5300
System 4	Intel D975XBX motherboard	DualCore, HT
	Intel Pentium EE 965, 3.7 GHz	2 x 2 MB L2-Cache, 4 GB PC2-5300
System 5	FSC CELSIUS V830	SingleCore
	AMD Opteron 256, 3.0 GHz	1 MB L2-Cache, 2 GB PC3200

Table 4 Systems used for runtime experiments.

Based on the systems considered, we set up a number of experiments. We evaluate the performance of our approach by predicting the fastest platform for a benchmark program. In each experiment, we take care that different processor models are used, and that all systems are best suited for a significant number of programs. More specifically, in order to ease data mining and the comparison of the results, the experiments feature situations where the distribution of the systems being fastest is as balanced as possible. Table 5 lists the experiments with the systems compared.

Experiment	Platforms	Processors
Experiment 1	System 5 vs. System 2	Opteron vs. Pentium 4
Experiment 2	System 3 vs. System 4	Opteron vs. Pentium EE
Experiment 3	System 3 vs. System 1	Opteron vs. Xeon
Experiment 4	System 3 vs. System 1 vs. System 4	Opteron vs. Xeon vs. Pentium EE

Table 5 Experiments.

Experiments 1, 2 and 3 are binary prediction problems where the task is to chose one out of two platforms, Experiment 4 is a three-class prediction problem. We limit ourselves to these experiments and do not run experiments where to choose between all systems. This is because the training data from the 31 programs available would not provide enough learning examples to choose from more than three systems. In the following, we show that predictions with two or three target systems are possible. We do not expect any difficulties when choosing from more systems when more training examples are available.

We evaluate our approach using different learning algorithms such as neural networks, support vector machines and decision trees. As we have achieved the best results using the C5.0 decision-tree algorithm (a variant of the well known C4.5 algorithm [13]) implemented in the SPSS Clementine data-mining suite, we will focus on the C5.0 classifier in the following. However, the results with other classifiers are not significantly different.

We conduct all experiments using stratified 2-fold-cross-validation: We use half of the programs for learning and the other half for testing, in two iterations. We deem this evaluation scheme adequate for our dataset consisting of 31 programs for learning and classification. We then derive the *accuracy*, i.e., the percentage of programs with correct prediction, as well as the *speedup*. Here, the *speedup* is the improvement in execution time over the average execution time on all systems in the experiment. Averaging the execution time of the systems is a fair baseline, as it mimics random selection of the underlying system. The *speedup* can be compared to the highest possible speedup, $\mathit{speedup}_{\max}$. This is the improvement in execution time when selecting the fastest platform for each benchmark program. In our scenario, the *speedup* measure is more significant than *accuracy*. To illustrate, predicting a system slightly worse than the best one would decrease *accuracy* but would affect the *speedup* only slightly. This is consistent with our goal to select fast architectures. We have consciously decided not to consider any error or confidence level information in our evaluation. This is because some decisions made by our majority-vote scheme might be tight. This is natural in our setting where some individual functions might be misclassified. We expect the large number of data tuples to compensate this effect, as discussed before. Table 6 contains our experimental results.

Experiment	*accuracy*	*speedup*	$\mathit{speedup}_{\max}$
Experiment 1	74.19%	1.08	1.13
Experiment 2	77.42%	1.05	1.11
Experiment 3	83.87%	1.11	1.12
Experiment 4	67.74%	1.10	1.19

Table 6 Experimental results.

Experiment	*accuracy*	*speedup*	$\mathit{speedup}_{\max}$
Experiment 1a	64.52%	1.05	1.13
Experiment 2a	67.74%	1.03	1.11
Experiment 3a	83.87%	1.11	1.12
Experiment 4a	64.52%	1.10	1.19

Table 7 Results without *quality measures*.

There is a high accuracy in the range from 74% to 84% for binary classifications (78% on average) and 68% for the three class case. This is signifficantly higher than the a priori probability for selecting the larger class (55% for Experiments 1 and 2, 58% for Experiment 3 and 42% for Experiment 4). More important, the predictions actually improve the total execution time. On average, 63% of the highest speedup possible is reached with the predictions of our approach. These results not only are of practical relevance, i.e., the prediction of the system best suited. They also confirm the hypothesis that there is a strong relationship between source-code characteristics and runtime behaviour.

Looking at Experiments 1, 2 and 3, we investigate the impact of the individual metrics used. Our motivation is not to learn more about individual metrics, but to see if there are important categories (cf. Table 2) and less important ones. The C5.0 implementation used can assess which attributes occur most frequently close to the root of the decision tree. Such an analysis of all trees generated reveals that the following five attributes are the most important ones, in decreasing order: *exit_count*, *notice_22_type*, *freq_sum*, *edges* and *notice_27_percent*. Therefore, attributes from all categories in Table 2 turn out to be important to predict runtime behaviour. As we did not expect a high impact of the *quality measures* (i.e., *notice_22_type* and *notice_27_percent*), we run our experiments again, but without using the *quality measures*. Table 7 contains the results.

The experiments show that the *quality measures* have an influence in Experiments 1, 2 and 4 where the accuracy decreases, but are not relevant in Experiment 3 where the same results are obtained. This behaviour can be explained as follows: *Quality measures* are rarely used in the decision trees in Experiment 3, and other metrics are more significant in this experiment. As one example of the *quality measures*, the metric *notice_27_percent* indicates a high number of function-return points (`return`-statements). Even if intended as a code-quality measure, this measure also quantifies the complexity of a function. This explains why this metric indeed contributes to the classification results.

6 Discussion

The approach investigated here has turned out to be useful for a fast identification of the best-suited architecture for a given application, in terms of runtime. Even for an expert it would be difficult to impossible to determine a good architecture by only looking at the source code. Up to now, traditional approaches such as benchmarking are used to this end. But compared to our approach, benchmarking has two disadvantages: First, it requires access to the systems in question, to actually run the benchmark programs. Our approach in turn only requires execution times (and no further runtime characteristics) for the benchmark programs used for learning – but not for the program for which the best architecture shall be predicted. Further, as long as one relies on standard benchmarks such as the SPEC benchmark used here, the runtimes on various systems are available 'for free', e.g., in an online database. This information can therefore be used to build the classification model. When applying the model, no runtime-related information is needed. Second, benchmarking requires time, ranging from a few minutes up to several hours. Our approach, in contrast, requires only a few seconds to determine the architecture best suited.

Since we have used only programs from the SPEC benchmark suites, our prediction accuracy is actually better than it looks at first sight, as we now explain. Applications can be roughly categorised into three classes: I/O-intensive, memory-intensive, and compute-intensive. Most of the SPEC CPU benchmark programs used here are compute-intensive, and none of the SPEC CPU 2006 programs used (cf. Table 1) shows any significant I/O-activity [17]. Each benchmark program has a memory footprint of less than 1 GB [4], which is smaller than the main memory of the systems we used. In addition, the platforms used for our experiments are relatively similar. Each of them uses the same x86 instruction-set architecture and only differs in the implementation, e.g., the pipeline of the Pentium 4 670 in System 2 is longer than the one in the AMD Opteron 256 of System 5. So we expect the prediction accuracy to increase when not only compute-intensive benchmarks are used, but also I/O-intensive or memory-intensive ones. It is also likely that our prediction accuracy increases when using a broader variety of systems, e.g., systems with a different instruction-set architecture like Itanium or PowerPC processors.

One potential way of improving our results further would probably be to make use of the different degrees of importance of functions. A rule of thumb says that 10% of all functions are responsible for 90% of the workload. As we have explicitly decided not to consider any runtime information in this study, our approach gives the same importance to every function, even to functions which are never called during an actual execution. In our current research – and in contrast to the main hypothesis of this current study – we investigate whether (and by how much) the utilisation of function-call frequencies can improve prediction quality.

7 Conclusions and Outlook

The question which platform yields the best performance for a certain application is a fundamental issue in the computer industry. Traditional approaches to deal with this issue make use of simulations, analytical models or experimental executions. This means that either a simulation model, an analytical model or an existing system must be available. Furthermore, these approaches are rather time-consuming. The approach studied in this current paper in turn deploys data-mining methods in order to do the prediction as follows: We generate metrics by analysing the source code of the application in question. We use off-the-shelf benchmarks to generate training data for a classifier, i.e., we extract those metrics for the benchmark programs. A classifier then determines the best suited computer architecture based on a given set of characteristics. A distinctive feature of our classification approach is that it works with fine-grained software metrics on the function level, while it derives predictions for entire programs. Its classification accuracy in our experiments has been 78% on average. The approach can predict the runtime behaviour of benchmark programs with previously unknown runtime behaviour on the target platforms, allowing to choose the best platform.

The work described here is part of a larger effort aiming at the deployment of data-mining techniques for system design and computer-architecture problems. On one hand, we are currently trying to increase the classification accuracy further. For example, we are currently investigating the usage of program-dependence-graph [12] metrics. Such graphs can be derived from static source code and include data dependencies, in addition to control dependencies as in control-flow graphs. We reckon that such dependencies are relevant for execution performance, as data dependencies affect pipelining and register usage, and the metrics used so far might not sufficiently cover these aspects. Further, we are examining the impact of function-call frequencies on prediction quality, as described in Section 6. On the other hand, future investigations will try to reveal dependencies between source-code properties and computer-architecture characteristics with our approach. Up to now, our objective has been to predict the fastest platform for a given application. We plan to investigate how to correlate the source-code related metrics with micro-architectural details, e.g., the cache architecture, and how to generate respective predictions. From a computer-architecture point of view, this would be of enormous help when designing processors for specific applications.

Acknowledgements

We are indebted to Nikolay Iakovlev, Markus Korte and Stephan Schosser for their valuable contributions, suggestions and discussions.

References

1. Allen, F.E.: Control Flow Analysis. In: Proc. of a Symposium on Compiler Optimization, SIGPLAN Notices, pp. 1–19 (1970)
2. Castillo, P.A., Mora, A.M., Guervós, J.J.M., Laredo, J.L.J., Moretó, M., Cazorla, F.J., Valero, M., McKee, S.A.: Architecture Performance Prediction Using Evolutionary Artificial Neural Networks. In: Proc. of the European Workshop on Bio-Inspired Heuristics for Design Automation (EvoHOT) (2008)
3. Eichinger, F., Böhm, K.: Selecting Computer Architectures by Means of Control-Flow-Graph Mining. In: Proc. of the Int. Symposium on Intelligent Data Analysis (IDA) (2009)
4. Henning, J.L.: SPEC CPU 2006 Memory Footprint. SIGARCH Comput. Archit. News **35**(1), 84–89 (2007)
5. Hoste, K., Phansalkar, A., Eeckhout, L., Georges, A., John, L.K., Bosschere, K.D.: Performance Prediction Based on Inherent Program Similarity. In: Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques (PACT) (2006)
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), 1–34 (2008)
7. Jones, C.: Applied Software Measurement. McGraw-Hill (2008)
8. Joshi, A., Phansalkar, A., Eeckhout, L., John, L.: Measuring Benchmark Similarity Using Inherent Program Characteristics. IEEE Trans. Computers **55**(6), 769–782 (2006)
9. Kühnemann, M., Rauber, T., Runger, G.: A Source Code Analyzer for Performance Prediction. In: Proc. of the Int. Parallel and Distributed Processing Symposium (IPDPS) (2004)
10. Kuncheva, L.I.: Combining Pattern Classifiers: Methods and Algorithms. Wiley (2004)
11. McCabe, T.: A Complexity Measure. IEEE Trans. Software Eng. **2**(4), 308–320 (1976)
12. Ottenstein, K.J., Ottenstein, L.M.: The Program Dependence Graph in a Software Development Environment. SIGSOFT Softw. Eng. Notes **9**(3), 177–184 (1984)
13. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
14. Saavedra, R.H., Smith, A.J.: Analysis of Benchmark Characteristics and Benchmark Performance Prediction. ACM Trans. Comput. Syst. **14**(4), 344–384 (1996)
15. Shepperd, M.: A Critique of Cyclomatic Complexity as a Software Metric. Software Engineering Journal **3**(2), 30–36 (1988)
16. Singh, K., İpek, E., McKee, S.A., de Supinski, B.R., Schulz, M., Caruana, R.: Predicting Parallel Application Performance via Machine Learning Approaches. Concurrency and Computation: Practice and Experience **19**(17), 2219–2235 (2007)
17. Ye, D., Ray, J., Kaeli, D.: Characterization of File I/O Activity for SPEC CPU 2006. SIGARCH Comput. Archit. News **35**(1), 112–117 (2007)