Automatic Estimation of Performance Requirements for Software Tasks of Mobile Devices

Simon Schwarzer, Patrick Peschlow, Lukas Pustina, Peter Martini
Institute of Computer Science 4, University of Bonn
Römerstrasse 164
D-53117 Bonn, Germany
{schwarzer,peschlow,pustina,martini}@cs.uni-bonn.de

ABSTRACT

This paper introduces a new method to predict performance requirements of mobile devices' software tasks using system models describing the hardware and software. With the help of clustering algorithms and linear regression, behavioral models of software tasks are generated automatically. These models are used to project the runtime of representative parts of the software tasks. The runtime of representative execution parts is determined with instruction-accurate simulations which are not feasible for whole executions. The inputs for the projection task a model of the hardware platform and input data parameters, especially the data size. A major advantage of this approach is that the developers do not have to estimate the performance requirements themselves. In this way the method helps to seamlessly integrate the performance analysis process into the development process. The paper introduces the ideas in detail and presents an evaluation of the proposed method for typical software tasks of mobile devices.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*

General Terms

Experimentation, Performance

1. INTRODUCTION

In the area of embedded and mobile devices there is a strong need to predict the performance of future systems without building hardware prototypes. Due to short product cycles, cost pressure during development, and a rapid growth in this area with additional applications of the devices, e.g., multimedia smart phones, it is desirable to perform performance analyses without hardware prototypes. In the literature a number of performance engineering methods are introduced. The Software Performance Engineering

(SPE) process is probably the best known [26]. The basis of all these methods is the specification of resource requirements for all software tasks and the specification of hardware components' capabilities. In early development stages of completely new systems, this information has to be estimated, because no predecessor system is available for measurements. The development of mobile devices is usually not started from scratch, but developers can often reuse already existing implementations of certain software tasks [4]. For the performance analysis this has the advantage that performance requirements for these tasks can be better determined. The general recommendation to obtain meaningful figures for resource requirements is to use measurements, because the accuracy of the requirement specifications has a direct influence on the accuracy of the overall performance analysis.

In performance analysis a typically required information is the amount of CPU time per software task. These performance values are usually scaled with a processor speedup rate to adjust them to the modeled hardware. We show that this approach can lead to undesirable results in the area of mobile devices. Especially in the area of mobile systems this can result in wrong design decisions because hardware and software have to be carefully matched. A subsequent modification of the hardware is not as easy as for desktop or server systems because the upgrade capabilities are very limited. Since ARM processors are widely deployed in mobile systems, this paper focus on ARM processors [2, 3, 28].

In the area of mobile devices with configurable hardware platforms and a wide variety of hardware modification possibilities, it is a challenging task for developers to estimate the performance impact of hardware modifications on the runtime of algorithms. Moreover, software developers think in units of the input data when they specify workloads for a scenario, but the performance requirement annotations need to be specified in CPU time. For example, consider a use case containing a software task of compressing a photo with the *jpeg* algorithm. If the performance scenario is changed and a photo with a higher resolution and more pixel is used as input for the use case, the performance requirements increase and have to adapted by the developer. When the hardware platform is changed, the required CPU time has to be updated, too. Developers have to be enabled to vary input data parameters in an easy way. The determination of performance requirements is an additional effort, which inhibits the smooth integration of performance analysis in the development process. At this point it could be desirable if resource requirements would be determined automatically

based on the input data parameters and the modeled hardware. Therefore, a method is required that predicts the performance requirements with the help of the indication of the program, the input data parameters, and a model of the hardware platform. The goal of our method is to automatically generate performance requirement estimations of software tasks for the specified hardware platform. These predicted requirements do not need to be scaled with a processor rate because the tool already took the hardware settings into account.

The rest of the paper is structured as follows. Section 2 presents shortcomings of the usage of simple processor speed-up rates and motivates our method. Section 3 gives a short introduction into the modeling of hardware platforms with UML. Section 4 describes the properties of program executions which form the basis for our ideas of behavioral model creation. The creation of the behavioral models is described in section 5. Afterwards an evaluation of the method is presented in section 6. Section 7 concludes the paper. Related work is introduced in the respective sections where needed.

2. MOTIVATION FOR BEHAVIORAL WORKLOAD MODELS

Common performance engineering methods require the annotation of software tasks with the expected resource requirements. The SPT profile and its successor, the MARTE profile, introduce and define these annotations and their properties. For example, the MARTE profile specifies to include the required demand on the executing CPU in time units in the annotation of a software task [21]. Figure 1 presents an example adapted from the official MARTE profile document. The tasks are annotated with a ≪PaStep≫ annotation and the requirements on the executing processor are specified in the `hostDemand` tag in time units, i.e., 0.5 ms and 2.5 ms, respectively.
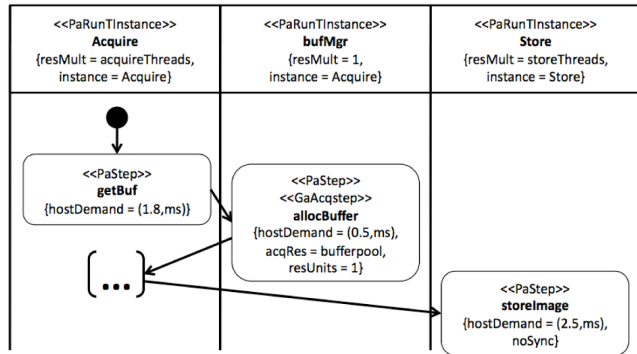

Figure 1: Annotation example adapted from the official MARTE profile specification.

In the same way as software tasks are equipped with annotations specifying their non-functional properties, hardware components are also equipped with annotations. The executing component, i.e., the CPU, is annotated with a property specifying the relative speedup compared to a reference processor. The processor rate is used to adapt the specified software requirements to the modeled hardware. These performance requirement specifications are the basis of the performance analysis and the performance predictions of the system under development.
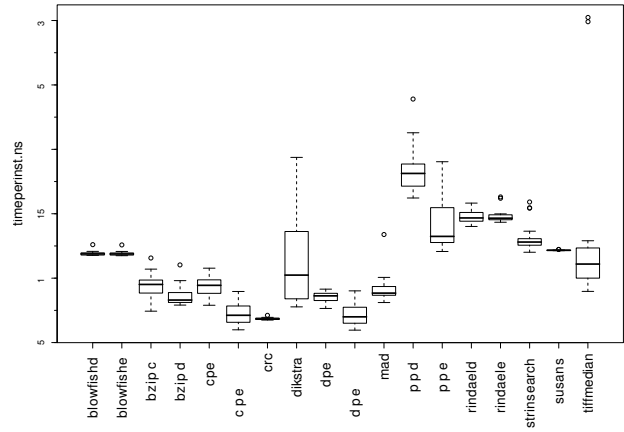

Figure 2: Box plots of the average time per instruction for program input pairs on an ARM9 with an instruction cache of 16 KB and a data cache of 8 KB.

2.1 Performance Engineering and Mobile Systems

In the area of mobile devices, developers have a bigger influence on hardware details than in the development of desktop and server software systems. Therefore, the application of this approach raises the following questions: How does the processor rate change when an additional cache level is introduced? How does the processor rate change when a slower memory module is used? Are simple processor rates sufficient at all to reflect different platform configurations that are available for mobile systems?

2.2 Shortcomings of Processor Rates

The standard approach from the hardware developers' point of view is to simulate standard benchmarks on hardware models of the different configurations and determine an average speedup. This speedup value would be used as processor rate to adjust the performance requirements. We simulate executions of typical algorithms for mobile systems with several inputs on different platforms to answer the question whether it is a valid approach to use a unique average speedup value to adjust the performance requirements of all software tasks. The programs are taken from the MiBench suite [16] and the MediaBench II suite [14]. The first suite is specialized on embedded and mobile devices and the latter suite focuses on multimedia algorithms. Input data for the simulation runs are taken from the MiDataSet collection [15].

In order to be able to compare runtimes of different programs and executions of the same program with different input data sizes, we calculate the average time required for the execution of one instruction for each execution. Figure 2 shows a box plot of this average time per instruction for several executions with different inputs of typical mobile device algorithms simulated on a standard OMAP board with an ARM processor and an instruction cache of 16 KB and a data cache of 8 KB [28].

It can be seen that the average time per instruction differs significantly for the programs. For example, the average time per instruction in case of the *pgp decryption (pgp_d)* program is several times higher as the values for the *jpeg*
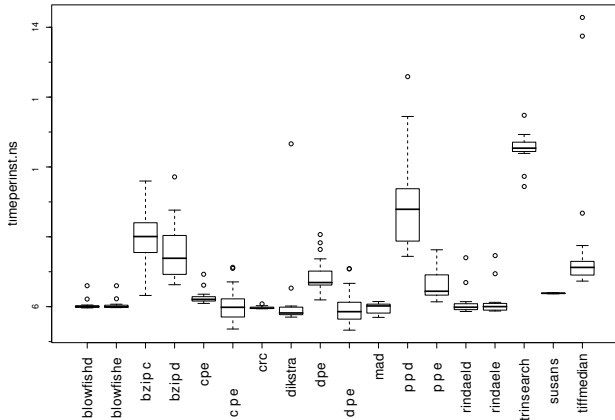
Figure 3: Box plots of the average time per instruction for program input pairs on an ARM9 with an instruction and data cache of 32 KB and a unified 2nd level cache of 128 KB.

algorithms. Outliers, which are drawn as circles in the box plot, occur nearly for all algorithms. These are executions with very small input files, so that the initialization phase is the dominant part in the execution which influences the average time per instruction value.

Figure 3 shows the average time per instruction for the same algorithms, but on a platform with a different cache hierarchy, i.e., 32 KB instruction and data cache and a unified second level cache of 128 KB. In general, it can be seen that the average time per instruction decreases, but the time per instruction does not decrease by the same factor for all algorithms. For example, on the first platform configuration, the average time per instruction for *jpeg compressions (cjpeg)* is higher than for *jpeg decompressions (djpeg)*. But on the platform with larger caches the situation for these algorithms is vice versa. The same can be observed for *dijkstra* and *djpeg*.

2.3 Program-specific Processor Rates

These analyses show that it is not sufficient to determine one average processor rate to reflect the performance influence of platform changes like cache and memory modifications. A solution might be to determine processor rates for single programs or for groups of programs. This would be a valid approach if the executions could be accurately captured by mean values. However, during the execution of programs particular parts reoccur, e.g., loop bodies or functions. If only one program part reoccurs and characteristics like instruction mix and memory access patterns are similar, an average processor rate per program may be sufficient. However, usually several program parts reoccur during a program's execution. These parts can have completely different characteristics, so that the performance influence of platform modifications can be different. Figure 4(a) shows average time per instruction during an execution of the *pgp decryption* program. The average time per instruction is determined for fixed intervals of 100,000 instructions. These intervals are plotted on the x-axis and the corresponding average time per instruction is denoted on the y-axis. It can be seen that phases with different times per instruction ex-

ist. Figure 4(b) shows the average time per instruction for intervals of a *jpeg compression* execution. The plot has a different structure, but once again it can be seen, that different program sections with different performance demands are executed. Therefore, it is not sufficient to determine one average processor speedup even on the granularity of single programs, because platform modifications can have different performance impacts on program parts and the occurrence of program parts can depend on the input data size.
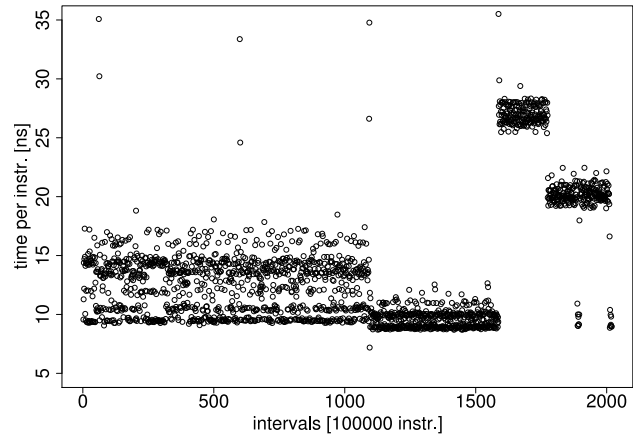
(a) *pgp decryption*
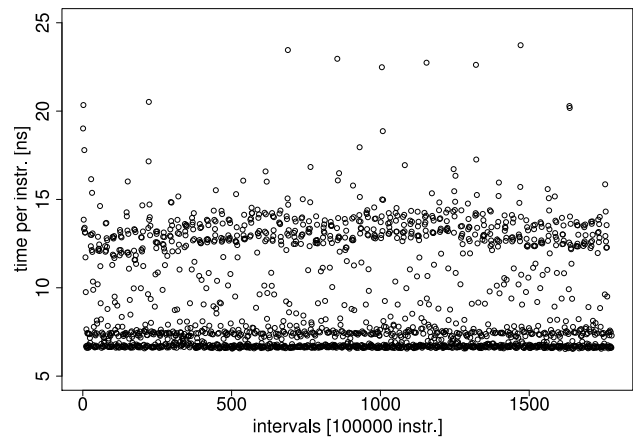
(b) *jpeg compression*

Figure 4: Time per instruction values for executions of *pgp decryption* and *jpeg compression*. The time per instruction is calculated for intervals of 100,000 instructions.

These results show that specifying meaningful values for performance requirements and resources is not an easy task. Especially in the context of mobile system design, where developers modify the hardware of the system, it is a challenging task to determine good performance estimations.

2.4 A Performance Engineering Process for Mobile Systems

A possibility to increase the accuracy of requirement estimations for software tasks that are already implemented are instruction-accurate simulations of selected executions. These simulations would have to be integrated in the performance engineering process. Figure 5 presents a schematic
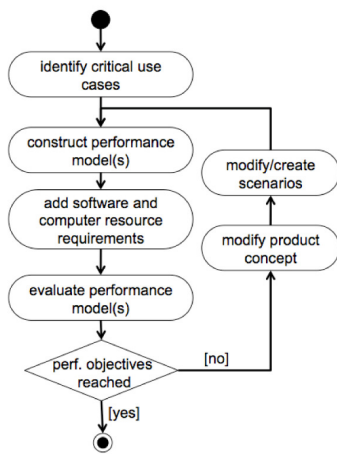
Figure 5: Standard performance analysis cycle.

Figure 6: Optimized performance analysis cycle with behavioral models in the analysis loop.

overview of the performance analysis cycle used in standard software performance engineering methods, e.g., [26].

Since the selection of the hardware platform is also part of the system design process in mobile systems development, the simulation of the software tasks on detailed hardware models would have to be integrated into the performance analysis loop in the step *"add software and hardware resource requirements"*. In this adapted approach the detailed hardware simulation would become part of the performance analysis loop. Unfortunately, these simulations are very time consuming, i.e., runtimes of days or even weeks. Therefore, it is not practical to have these detailed simulations in the loop to find the optimal software/hardware settings. Moreover, these simulations are based on real executions of programs requiring input data files. Our goal is to enable developers to specify the input parameters and the size of the input data only. Therefore, the time-consuming simulations have to be removed from the loop and more abstract workload models are required. Figure 6 presents an adapted performance engineering process which addresses the aforementioned requirements. The detailed hardware simulation of existing algorithms is replaced with a detailed hardware simulation that takes behavioral models as input. These simulations are intended to be orders of magnitudes faster than instruction-accurate simulations of complete program executions and base on input parameters instead of input data. The behavioral models have to be created in a prerequisite step which has to be independent of the hardware configuration. In this way, the behavioral models are valid for all modeled hardware platforms in the analysis loop. Of course, the usage of the behavioral models leads to less accurate performance estimates than instruction-accurate simulation of complete program executions. Nevertheless, the advantages of the behavioral models are that no real input data is required during the analysis cycle and resource requirements for any input data size can be easily predicted within a reasonable amount of time. These advantages outweigh the disadvantage of losing some accuracy.

The method frees developers from the necessity to annotate software tasks and hardware resources with estimated performance requirements and processor rates. Instead, developers are enabled to simply specify the crucial input data parameters of the software tasks and model the hardware in
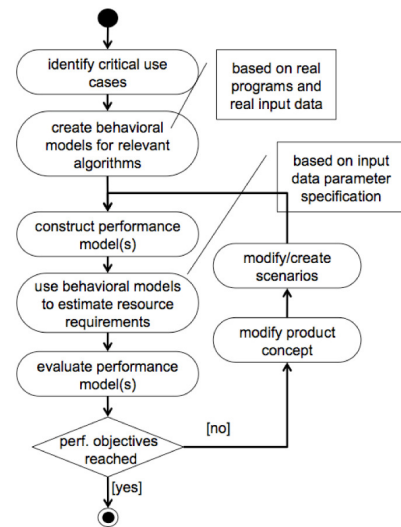
detail, so that the performance demands can be automatically predicted. The method takes the program, crucial input parameters, and the hardware model as input to predict the required CPU time. From the developers' view, the tool is a black box as depicted in figure 7.
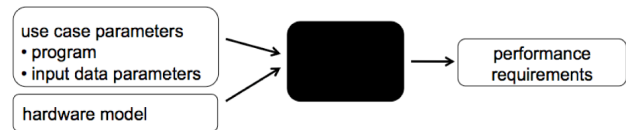

Figure 7: The developers' view of the method is a black box which takes the software task, input data parameters, and a hardware model as input.

3. DETAILED PLATFORM MODELING WITH UML/SYSML

Our method requires a detailed hardware modeling to simulate requirements of software tasks' executions in detail. Since the de facto standard for modeling in the area of software and performance engineering is UML, the hardware modeling should also be enabled with UML and its profiles MARTE and SysML. In [23] we introduced a modeling approach for embedded and mobile device hardware, especially ARM processor based platforms. In the following, we present the most important ideas and aspects of the modeling approach. Due to the decomposition and refinement capabilities of UML, it is possible to seamlessly integrate a detailed hardware specification in a model primarily focused on software aspects. The components of the platform are modeled with *basic block diagrams* and *internal block diagrams* of SysML which are similar to UML's *class diagrams* and *composite structure diagrams*, respectively. Instances of hardware components which are defined as *blocks*, are used as *parts* in internal block diagrams. Figure 8 depicts an internal block diagram specifying the architecture of an ARM9 system. The cache hierarchy and connections

between the caches, the bus, and the memory system are modeled. Stereotypes defined in the MARTE profile are applied to enhance the model with semantics. Since the MARTE profile does not provide detailed enough annotations to model the processor pipeline in the required level of detail, we introduced new stereotypes to model the processor internals in UML [23]. These UML diagrams can be used to automatically configure the instruction-accurate platform simulator. In this way the detailed hardware modeling is seamlessly integrated in the system model and can be automatically used for the performance predictions.
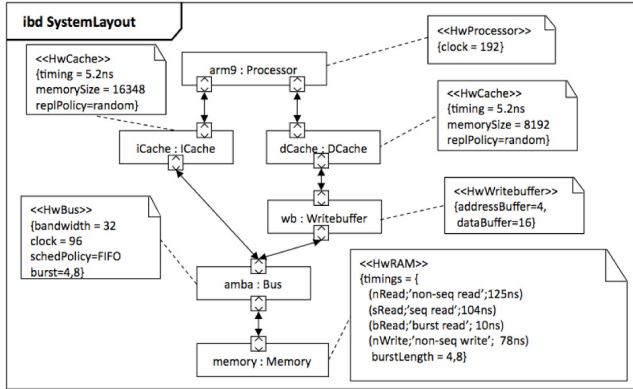

Figure 8: Internal block diagram specifying an ARM9 example system.

4. KEY CONCEPTS OF THE METHOD

This section introduces the key concepts of our method. The basis of the method are the following properties of program executions. In the following, these properties are listed and are examined in detail afterwards.

1. Programs have one or more *input data classes*. An input data class is characterized by the code which is executed by the program while processing input data. Members of the same input class have similar executed program code parts.

2. Single program executions have recurring phase behavior. The same or similar parts of the program code are executed at different times during execution.

3. The recurring phases are very similar for program executions of inputs from the same input class.

4. The length and occurrence of recurring phases depend on the size of the input data.

The first observation, stating that programs can have input classes, is already reviewed in literature of different areas. For example, in the context of compiler optimizations, representative runs are used as a basis to identify hot paths in programs [6]. Eeckhout et al. examine the input variability of programs to enhance the design of benchmark workloads and to reduce the number of program executions required to cover the workload space [9, 13]. Wall used matchings between two executions of a program on the level of functions to determine the difference in executions of the same program [29]. We adapted these matchings

to be usable on the level of basic blocks. A basic block is a section of code with only one entry and one exit point. Basis blocks have a much finer granularity than functions. The original matchings are based on the frequency of function calls, we adopted the metric and use the *coverage* of basic blocks so that not only the frequency is considered but also the size of a basic block. The coverage of a basic block bb for a given program execution is defined as: $\text{coverage}_{bb} = \frac{\text{frequency of bb} * \text{size of bb in instructions}}{\text{overall number of instructions}}$. Figure 9 presents a box plot of frequency matching scores for several executions of *pgp encryption*. The *n frequency matching score* between two profiles, A and B, is defined as the sum of coverages in profile B of the n basic blocks with the highest coverage in profile A divided by the total of the sum of coverages of the top n basic blocks of profile B. Values close to 1 indicate a high level of similarity between two executions. For each execution, the box plot shows the matching scores with all other executions. It can be seen that some executions differ and have lower average matching scores. In the case of *pgp encryption* these are executions with small inputs for which the start and end phase dominate the execution. The other executions, however, show a high similarity and belong to the same input class. In case of *jpeg* algorithms and *tiffmedian*, images stored in gray colorspace lead to a different execution profile than full color images (no figure shown).
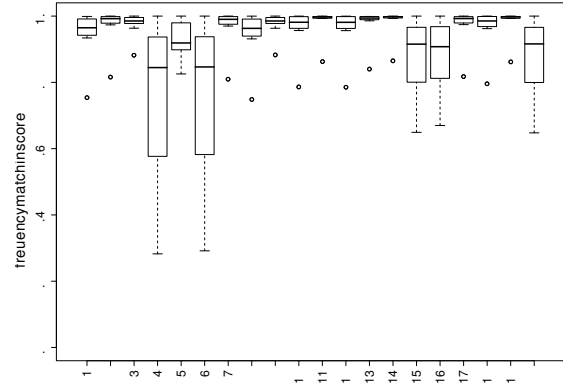

Figure 9: Basic block frequency matching score for executions of *pgp encryption*.

Next, we address property 2, which states that single program runs have recurring phase behavior. Figure 10 depicts a *similarity matrix*. A similarity matrix compares objects with each other, and for each pair of objects a dot is plotted. The color of a dot corresponds to the level of similarity between the corresponding objects. For the matrix in figure 10 an execution of the *pgp decryption* algorithm is split into fix length, non-overlapping intervals. These intervals are compared with each other and the more similar two intervals are, the darker the corresponding dot is drawn. Since we want to demonstrate that a program exhibits recurring phases of similar code regions, we determine *basic block vectors* for each interval and calculate the similarity between these vectors. A basic block vector is a vector containing the frequencies of each basic block during the observed time interval. Similar investigations for SPEC benchmarks can be found in [24, 25]. As distance or similarity metric, respectively, we choose the city block distance. The vectors

are normalized so that the city block distance between any pair of vectors is in the interval $[0, 2]$.

In figure 10, intervals with a length of 100,000 instructions are compared with each other. It can be seen that phase behavior exists. Dark dots indicate a high similarity of the corresponding intervals. If one takes a point on the diagonal line of the coordinate system and follows the line parallel to the x-axis, all dark dots on this line indicate a similar and thus, recurring interval.
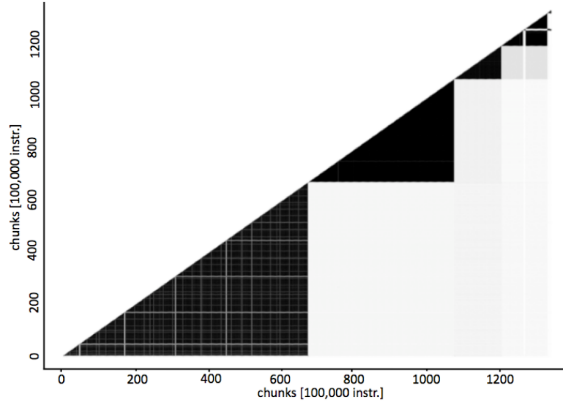

Figure 10: Similarity matrix of basic block vectors of a *pgp decryption* execution with an interval size of 100,000 instructions.

Next, we show an example for the third property, which states that recurring phases are similar not only within one program execution, but are also similar to phases of other program executions for inputs from the same input class. For this reason we compare basic block vectors of two different program executions with each other. Figure 11 depicts the similarity matrix for two different executions of *pgp decryption*. The similarity matrix is not quadratic in this case, because the executions are of different length. Again, recurring patterns can be seen, which demonstrates that similar intervals exist. Since the comparison is based on the occurrence of basic blocks, this shows that the same program code areas are executed.
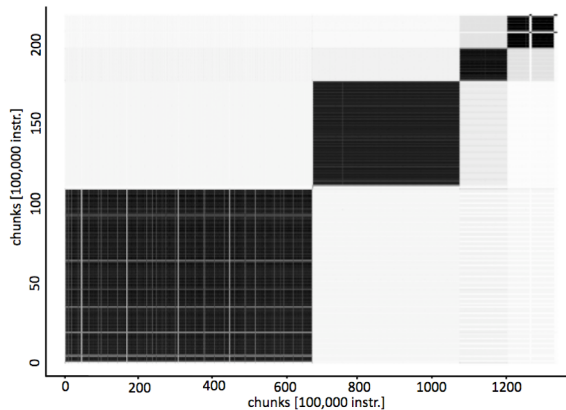

Figure 11: Similarity matrix for intervals from different executions of *pgp decryption*.

Finally, we address the fourth property, which states that the lengths or the occurrence of recurring phases depend on the size of the input data. Figure 11 also underlines this assumption, because the rectangles are not quadratic. The dark parts are horizontally sustained and the intervals of the execution with the larger input are drawn on the x-axis. This demonstrates that from each recurring phase more intervals are executed for the larger input file. For a phase whose occurrence is independent from the size of the input, the rectangle would be quadratic.

The presented properties of program executions are the basis for our method of behavioral model creation. In the following section the method is described in detail.

5. BEHAVIORAL MODELS CREATION

This section describes the key ideas and the algorithm for our behavioral workload model creation. The goal is to develop a method that generates behavioral models for software tasks of mobile devices. The behavioral models shall take input parameters of a software task as input and predict the runtime on a modeled hardware platform. The key idea of the method is to find similar parts in a software task's executions for inputs from the same input class and to mathematically describe the relationship between the input data sizes and the occurrence of recurring and similar parts. In the following we assume that only executions belonging to the same input data class are considered. The input data classes can be found by analyzing several executions of a program with different inputs and utilize basic block matchings as presented in section 4. At this point in the method some domain knowledge or manual interaction is required to identify the input data parameters which are responsible for different execution paths, e.g., identifying the color palette of images as an input data class parameter.

Figure 12 depicts the main steps of the method. First we create processor instruction traces of a software task's executions for different inputs. The traces are split into fix length intervals. These intervals are compared with each other and similar intervals are grouped together. The detected groups are program phases of the software task. Then the number of intervals for each phase and execution is determined. These numbers in combination with the sizes of the input data are used to determine a mathematical function describing the relationship of input data size and number of intervals per phase. Since an appropriate interval size is unknown a priori, we have to analyze a set of different interval sizes. This aspect is discussed in more detail in section 5.4. Details and challenges for each step are presented in the following.

5.1 Recording of Instruction Traces

In the first step, processor instruction traces for executions of software tasks are recorded. Instruction traces are captured with the help of a modified version of the popular processor emulator Qemu [7]. This modified version records all executed instructions, accessed registers, and memory addresses. These data are written into a buffer of the host memory during execution. When the buffer is full, the emulation is paused and the whole buffer is written onto hard disk. In this way the effects of the emulation's slowdown due to the recording are minimized.

It is also possible to use our method with execution-driven tools instead of trace-driven tools. The advantage of execution-driven generation of interval characteristics and sim-
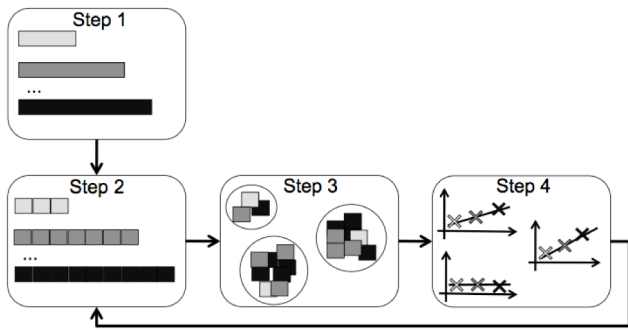
Figure 12: Overview of the steps of the behavioral model creation.

ulations is that no hard-disk storage has to be provided for instruction traces. Nevertheless, we apply the trace-driven approach due to the following reasons. An advantage of the trace-driven approach is that reproducibility can be easily guaranteed because once a trace is recorded it can be used as input without unintended changes due to effects of the execution environment. The disadvantage of huge storage requirements is not as incisive for mobile device software tasks as for SPEC benchmark traces because much less instructions are executed. The lack of freely available tools supporting ARM processors is another reason, since the implementation of a trace-driven simulator is less time consuming than the implementation of the whole processor logic. A possible candidate was the popular SimpleScalar simulator [5], but the required SimpleScalar ARM compiler is out of date and moreover, not all system calls are implemented in case of ARM binaries. Therefore, we use our trace-driven ARM processor based platform simulator [23] and Qemu.

5.2 Interval Characteristics

In the second step, the instruction traces are split into fixed length intervals and characteristics for these intervals are calculated. The behavioral models have to describe the program behavior as a function of the input parameters. Therefore, appropriate characteristics have to be found. In the literature, different types of characteristics which describe the execution of a program are proposed [20, 12, 11, 13, 10, 18, 19, 22, 24, 25]. These characteristics can be classified into *architecture-dependent* and *architecture-independent* characteristics.

The architecture-dependent characteristics measure properties of the hardware during the execution of a program. For example, the instruction and data cache miss rates or the average required time per instruction are characteristics of this class. The disadvantage is that these characteristics can vary when the program is executed on different hardware as already shown in section 2 in the average time per instruction figures for different platforms. Therefore, this characteristic class is not appropriate for our analysis because the behavioral models have to be usable for all hardware platforms.

Architecture-independent characteristics can be divided into code-based properties and properties of the instruction sequences, register, and memory accesses. Properties of the latter type are for example the instruction mix, the live time of registers, the reuse distance of memory addresses,

etc. In contrast, code-based characteristics describe the execution with elements of the source code. The number of function calls, loop iterations, or basic block occurrence are examples for this class of characteristics. Code-based characteristics have the disadvantage that only executions of the same binary can be compared with each other, because the functions and basic blocks are unique for a binary. In contrast, non-code-based characteristics can be used to compare executions of different programs, e.g., the instruction mix or memory access properties can be compared between two completely independent programs. A disadvantage of the non-code-based characteristics is that different functions or different basic blocks can have the same or very similar non-code-based characteristics, e.g., a similar instruction mix. This introduces fuzziness into the characterization when the behavior has to be analyzed. Since we need characteristics to analyze the program behavior for executions of the same program binary but with different inputs, the restrictions of the code-based characteristics can be ignored. Therefore, code-based profiles are best suited for our purpose and we characterize the intervals with the help of basic block vectors.

5.3 Identification of Common Phases

In the third step, we want to identify intervals which are similar and group them together. This is a typical task for cluster algorithms from the data mining area [27]. Different cluster algorithms exist having their own pros and cons. Some algorithms have been already utilized in related work. Eeckhout et al. analyze the composition of benchmarks and determine properties of programs of a benchmark suite to analyze which parts of the workload space are covered by the benchmark programs. They apply clustering algorithms on vectors of microarchitecture-independent characteristics like the instruction mix and register usage to detect programs in benchmark suites which have similar hardware requirements. Since the number of elements they have to cluster equals the number of program-input pairs in the benchmark suites under study, they apply *hierarchical clustering algorithms*. Hierarchical clustering algorithms are usually more stable but also more time consuming than non-hierarchical algorithms [27].

All cluster algorithms require a proximity or distance measure, respectively, to be able to compare objects with each other. In our case the objects are intervals from the instruction trace for which basic block vectors have been determined. In principle any distance metric is possible, but the city block distance has been successfully applied in similar contexts and has the advantage that differences in the same dimensions get a higher weight than with the Euclidean distance [25, 17]. In this way, a difference in the occurrence of one basic block can not be compensated by small or no differences of other basic blocks.

Since the number of executed unique basic blocks can easily reach thousands, the dimension of the basic block vectors become large. Unfortunately, a large elements' dimensionality hinders the clustering process. In case of a k-means algorithm, the distances between the elements and already identified groups have to be calculated several times. Moreover, the memory consumption of the basic block vectors becomes a significant challenge and identifying similar vectors becomes harder. This is known as the *curse of dimensionality*, and so called feature selection methods have to

be applied to reduce the number of dimensions [27]. Eeck-hout et al. use *principal component analysis (PCA)* to transform architecture-independent characteristic vectors of different programs into vectors of much smaller dimensionality, thereby only losing a small amount of overall variance of the data set. PCA is a mathematical method to find new dimensions that better capture the variability of the data. In an iterative process, a new set of variables is determined. These variables are linear combinations of the original variables and usually only a small number of these new variables is required to capture a sufficient amount of the overall variance. The authors of Simpoint [17] use *random projection* to transform interval basic block vectors into a space with fewer dimensions. They empirically determine a value of 15 dimensions to be sufficient in finding the same number of groups as with vectors of full dimensionality.

The first approach of using PCA has the drawback that this analysis has to be executed on all basic block vectors with full dimensionality. In our experiments it was not possible to apply PCA in all cases, because the amount of data was too large. We apply some kind of irrelevant feature reduction and reduce the basic block vectors to the number of dimensions necessary to cover a certain amount of the program's executions. The coverage of a basic block is defined as the number of executed instructions belonging to the particular basic block divided by the number of overall executed instructions. In this way, basic blocks that are small and seldomly executed are not taken into account during the analysis. Figure 13 presents the cumulative coverage of basic blocks for several executions of the *pgp decryption* program. The y-axis depicts the cumulative coverage of the basic blocks and the x-axis lists the basic blocks sorted by their coverage in a decreasing manner. The order of the basic blocks is determined by first analyzing all executions. For the sake of readability, the x-axis does not depict all basic blocks but is cut off after a coverage of 99%. The outlier curves belong to the two smallest executions.

Table 1 gives an overview of the number of basic blocks needed to reach a particular coverage level for different programs from the Mibench suite and inputs from the MiDataSets collection. A coverage level of 90% does already significantly reduce the dimension of the basic block vectors.
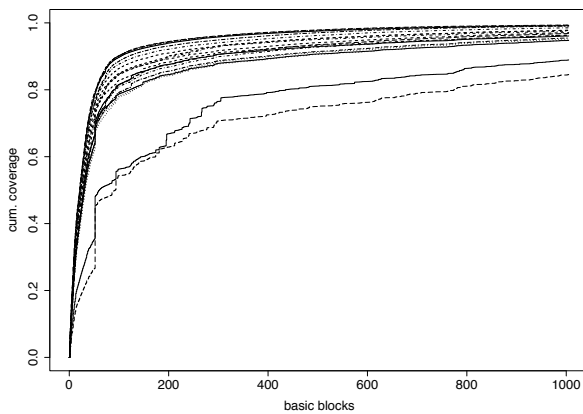

Figure 13: Cumulative coverage of basic blocks for executions of *pgp decryption*.

Table 1: Number of basic blocks for different coverage level.

program	85%	90%	95%	99%	100%
cjpeg	26	35	63	409	4661
djpeg	27	36	58	161	4832
pgp_d	72	111	273	1005	5184
pgp_e	138	272	552	1376	5061
dijkstra	5	6	8	80	3413
rijndael_d	62	80	111	297	3871
rijndael_e	65	86	119	278	3743
tiffmedian	29	76	332	1271	5128
bzip2comp	133	161	249	545	5533
bzip2decomp	82	115	185	520	5209

5.4 Finding Appropriate Program Phases

After reducing the basic block vectors for each interval to the most frequently occurring basic blocks, these vectors are used as input for the k-means algorithm which groups similar interval vectors. The k-means algorithm requires that the number of groups has to be specified a priori. Unfortunately, the number of program phases of the software task under study is unknown and has to be found automatically. Moreover, different software tasks exhibit phase behavior on different granularity levels, so that the number of program phases can be influenced by the current interval size, too. Therefore, we iterate over several numbers of groups and choose the clustering with the best properties for our behavioral model creation. In determining an appropriate number of groups, a tradeoff exists between a fine grained classification with many phases and a coarse classification with only a few large phases. The two extreme cases intuitively demonstrate this tradeoff. Assigning each interval its own phase results in a very fine grained classification with as much phases as intervals. This makes generation of mathematical models basing on the number of phase members imfeasible. On the other hand, putting all intervals into one phase ignores the phase behavior of the program executions. Since the goal is to capture the phase behavior of programs in order to be able to construct models for the relationship of phase occurrence and input data size, the goodnees of the behavioral models of the current clustering is estimated. For each combination of interval size and number of program phases, a score (cf. next section) is calculated to determine a combination which is well suited for model creation. The combination of interval size and number of phases, that achieves the best score is used for the final behavioral model.

5.5 Linear Regression Models

In the fourth step, mathematical functions are determined describing the relationship between input data sizes and number of interval members per program phase and execution, i.e., for each program phase a separate model describing the relationship of input data size and number of intervals of this phase is determined. The behavioral model for the software task under study consists of these determined functions. Linear regression is used to determine mathematical functions describing the relationship of input data size and interval members. Linear regression can not only be applied to linear functions, e.g., $y = \beta_1 \cdot x + \beta_0$, as per-

haps the name suggests. The term *linear* refers to the linear combination of the individual terms β_i so that any mathematical function can be incorporated into the model, e.g., a quadratic function $y = \beta_2 \cdot x^2 + \beta_1 \cdot x + \beta_0$. The model can be non-linear in the regressor variable but is linear in the β_i parameter. We assume that the type of formula such as the order of the polynomial, the *e*-function, etc., is provided as an input parameter to the algorithm. This is no general restriction because several functions can be tested and the best fit can be chosen automatically. Moreover, the runtime class of the software task should be known by the performance analyst. For each combination of interval size and number of program phases these models are determined to calculate a score. This score is the weighted sum of the model residuals for each program phase. The weights are calculated as the ratios between the number of instructions belonging to the corresponding program phase divided by the number of overall instructions. A smaller value of the score indicates a better model.

Algorithm 1 presents a high-level description of the algorithm which performs all aforementioned tasks. Since it is not possible to iterate over all possible interval sizes, a discrete set of interval sizes has to be provided. In the literature, large interval sizes of one to ten millions of instructions are recommended, but the programs analyzed in the literature are usually part of the SPEC benchmark suites and the aim is not to generate behavioral models. The SPEC benchmark programs are very untypical for mobile devices, e.g., gcc, ocean simulations, and extensive floating point calculations. Therefore, we apply smaller interval lengths of 100,000 or 1,000,000 instructions.

Algorithm 1 High level description of the behavioral model creation algorithm

---

Input: program binary, input data files, interval sizes, list of program phase numbers
Output: representatives for program phases, behavioral model (input size → runtime)

for all input data **do**
    create trace
    extract input data size
    store basic block sequence
for all interval_size in interval_sizes **do**
    for all basic block sequences **do**
        create interval basic block vectors
        reduce basic block vector dimension
    for all k in program_phase_numbers **do**
        perform k-means clustering algorithm
        for all program_phase in program_phases **do**
            linear_regression(input sizes, #members)
        calculate weighted score
determine best score
return behavioral model for best score

5.6 Runtime Predictions

After the combination of interval size and program phase with the lowest weighted residuals has been found, the determined functions can be used to predict the numbers of program phase members for arbitrary input data sizes. To predict the runtime for an arbitrary input data size for a given

hardware platform, the numbers of program phase members have to be multiplied with representative runtimes of the corresponding program phase. The representative runtime for one program phase is determined by simulating one or several intervals of the program phase on the modeled hardware. The performance analyst can choose between two strategies to determine a representative runtime. In the first approach, only one interval is simulated. For this purpose, an interval which is representative for the program phase is selected. This is done by selecting the interval which is closest to the center of the determined group during the clustering process of step three. In the second approach, a configurable amount of randomly selected intervals of each program phase, e.g., 20 or 30, are simulated on the hardware model and the average runtime is calculated. Both approaches have pros and cons. The advantage of the first approach is that only one interval has to be simulated so that the required simulation time is minimal. The advantage of the second approach is that the analyst obtains additional information about the variability of the runtime of the program phases. Moreover, with this approach it is possible to estimate a distribution of the runtimes.

5.7 Summary

This section presented key ideas of our method for the generation of behavioral models. The single steps of the method were described in detail and their individual challenges were discussed. Program executions are split into intervals which are characterized with the help of basic block vectors. The dimension of these vectors is reduced and a clustering algorithm is executed to find similar parts in all executions. Finally, the determined program phases and the input data sizes of the original executions are used to construct mathematical functions that describe the relationship of input data size and number of program phase members. Representative runtimes for each program phase are determined by simulations of one or several intervals.

6. EVALUATION

This section presents results of the application of the introduced method for typical software tasks of mobile devices. The whole process of recording the execution traces with Qemu, creating the interval basic block vectors, identifying input classes, and generating the behavioral models for each input class was performed. The evaluation focuses on a subset of the MiBench suite [16]: *bzip2*, *dijkstra*, *jpeg*, *pgp*, *rijndael*, and *tiffmedian*.

bzip2 is a lossless data compression algorithm. The *dijkstra* algorithm calculates the shortest path between nodes in a given graph. *jpeg* can be used for lossy image compression and decompression. *pgp* is the Pretty Good Privacy public key encryption and decryption algorithm. *rijndael* is the new Advanced Encryption Standard (AES) algorithm, and *tiffmedian* is an image processing algorithm that reduces the color palette of images. The input data for the algorithms are taken from the MiDataSet collection [15]. Some inputs have been omitted due to their size which prevented them to be loaded into the device's memory.

6.1 Example Analysis of pgp decryption

In the following we use the *pgp decryption* algorithm as an example to present intermediate results for the steps of the method. Afterwards, results of runtime predictions for the

aforementioned software tasks on different hardware platforms are presented.

The input data class analysis for the *pgp* algorithms does not reveal any special behavior depending on the type of input, e.g., text or binary data. However, executions for small input files lead to different basic block profiles which was already presented in figure 9. Therefore, we restrict the analysis to executions with at least 10 million instructions. Next, the algorithm for finding a good combination of interval length and number of program phases is executed. Since the executions have lengths between approx. 10 million and 200 million instructions, we choose interval lengths starting with 100,000 instructions and ending at 1,000,000 instructions. For the number of possible program phases we consider 1 to 10 phases.

Figure 14 shows a clustering of all interval basic block vectors of the *pgp decryption* executions under study for an interval length of 1,000,000 instructions. Since the dimension of the basic block vectors is too large to be displayed in a two- or three-dimensional diagram, we applied PCA and depict only the two variables with the highest variance (cf. section 5). Both principal components cover nearly 80% of the overall variance. It can be seen that groups of similar vectors exist.
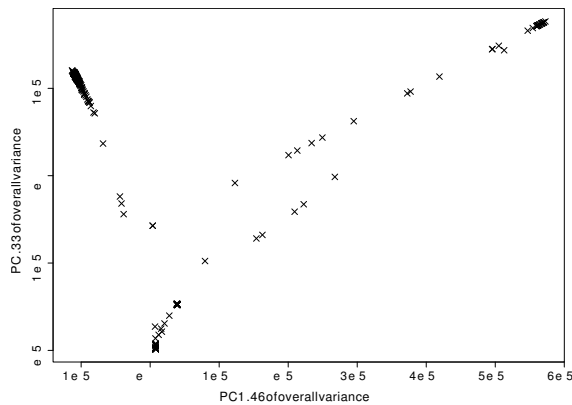

Figure 14: Clustering of basic block vectors of *pgp decryption* with a length of 1,000,000 instructions.

In addition to the evaluation of the clustering with the help of a standard clustering scoring, i.e., sum of squared errors, an additional scoring basing on the residuals of the linear regression is calculated. We set the regression function to be a polynomial of first degree, i.e., $y = \beta_1 \cdot x + \beta_0$. Figure 15 depicts the determined functions for each of the detected program phases from the preceding clustering for an instruction length of 1,000,000 instructions and four program phases. The x-axis depicts the input data size and the y-axis shows the number of cluster members per program phase. The solid lines show the prediction function and the dotted lines depict the 95% confidence interval of the prediction.

In order to determine the best combination of interval size and number of program phases, a score is calculated. This score is the weighted sum of residuals for each of the program phase regression models. The weight is determined as the ratio of each program phase. In the same way, the scores for all combinations of interval sizes and number of program
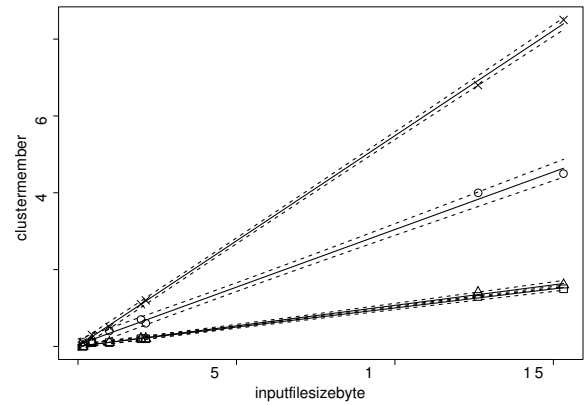

Figure 15: Regression lines for the analysis of *pgp decryption* with an interval length of 1,000,000 instructions and four program phases.

phases are determined. Figure 16 depicts these scores for all interval sizes and number of program phases considered. On the x-axis the interval sizes are drawn, the number of program phases are denoted on the y-axis, and the z-axis shows the score for the respective combination. A small score indicates a better applicability of the combination to be used as basis for behavioral model creation. In case of *pgp decryption*, interval lengths of 500,000 or 1,000,000 instructions have good scores. Finally, the combination with the best score is selected for the behavioral model creation.
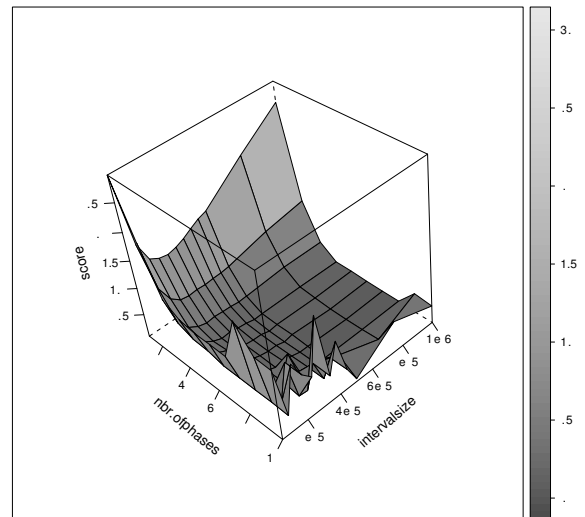

Figure 16: Visualization of the score that is used to find the best combination of interval size and number of phases in case of *pgp decryption* executions.

6.2 Runtime Predictions

To evaluate the prediction accuracy of our method, we simulate executions of software tasks on different platform

models and compare the simulation results with the predicted runtimes. The platforms are modifications of OMAP boards with an ARM9 processor and an ARM11 processor, respectively. The ARM9 processor is modeled with a clock frequency of 192 MHz, and the instruction and data cache can be accessed in one clock cycle. An AMBA system bus [1] connects the processor, caches, and the main memory (cf. figure 8). In case of the ARM11 processor, a clock frequency of 210 MHz is configured and a cache hierarchy with a second level cache is used.

For each software task we perform a leave one out analysis with the executions under study and compare the predicted runtimes to the runtimes determined by simulations of the complete instruction traces. For estimation of representative runtimes of the detected program phases, the developers can choose between the simulation of just one representative interval or an average runtime determined by simulations of several intervals (cf. section 5). In the following, we present results for the representative strategy and the sample strategy with 30 intervals. For the evaluation of the method, we assume a perfect warmup of the hardware, e.g., caches.

Table 2 presents evaluation results of the prediction accuracy of our method. The table compares the two strategies of determining representative runtimes with each other, i.e., representative vs. random samples. The modeled hardware is an ARM9 processor with an instruction cache of 16 KB and a data cache of 8 KB. The presented relative errors are averages of all leave one out runs for the respective algorithm. The standard deviation of these values is presented as a precentage of the respective relative error. It can be seen that our method predicts the runtimes with a feasible accuracy. In general, the representative strategy performs as good as the sample strategy, but in case of *pgp encryption* the prediction is not usable. In this case, the interval closest to the cluster center is not a feasible representative. Therefore, the recommendation is to use the sample strategy, because outliers are of no consequence if the number of samples is large.

Table 2: Mean relative errors for two prediction strategies for different software tasks.

	ARM9 i16/d8 random 30 samples		ARM9 i16/d8 cluster representative	
program	rel.err.[%]	std.[%]	rel.err.[%]	std.[%]
bzip2_d	24.46	28.28	22.42	27.82
cjpeg	11.94	14.13	21.53	16.45
djpeg	7.27	5.39	12.78	6.21
dijkstra	36.07	47.30	51.21	38.04
pgp_d	10.31	7.84	13.62	10.16
pgp_e	20.10	16.81	97.59	33.40
rijndael_d	4.87	3.07	8.05	3.88
tiffmedian	12.80	10.44	16.77	10.55

Table 3 presents accuracy results for two different hardware platforms while using the sample strategy with 30 samples. The first platform consists of an ARM9 processor with an instruction and data cache of 32 KB and a unified 2nd level cache of 128 KB. The second platform is equipped with the same cache hierarchy but a faster ARM11 processor. Variations in the accuracy of the predictions exist, but the average relative errors are in a range that is suitable for performance estimations. Though a completely accurate prediction of the true runtimes is not to be expected, because the individual executions, despite having similar basic block profiles, exhibit slightly different time per instruction values. This can be seen in figures 2 and 3, where for example the *pgp* and *tiffmedian* boxes are stretched.

In case of *dijkstra*, the executed basic blocks are very similar (cf. table 1), but the algorithm works on the whole input data and does not process it block by block. Therefore, data cache effects become important for large inputs and intervals of executions of small inputs are not necessarily representative for executions of large inputs although the same areas of code are executed. With the help of reuse distances of memory accesses it is possible to detect this effect [8].

Table 3: Mean relative errors for the prediction with *sample* strategy for different software tasks on different hardware platforms.

	ARM9 i32/d32/u128 random 30 samples		ARM11 i32/d32/u128 random 30 samples	
program	rel.err.[%]	std.[%]	rel.err.[%]	std.[%]
bzip2_d	30.79	31.78	30.71	32.32
cjpeg	10.46	10.93	10.43	10.93
djpeg	8.44	5.66	7.98	5.48
dijkstra	16.34	19.35	16,57	20.09
pgp_d	8.30	7.15	8.21	7.10
pgp_e	10.51	5.88	10.79	6.39
rijndael_d	2.59	1.86	2.61	1.83
tiffmedian	10.18	8.76	11.19	9.53

7. CONCLUSION

This paper presented a method to automatically generate performance requirement estimations for software tasks. System developers are enabled to just specify input data parameters and the size of the input data instead of resource requirement estimations. The performance requirements of the software tasks for the modeled hardware are automatically predicted by our method. These predictions base on behavioral models generated in a prerequisite step which is not part of the performance analysis loop to find an optimal hardware/software configuration. The application of the behavioral models is orders of magnitudes faster than instruction accurate platform simulations and required CPU times for arbitrary input data parameters can be predicted.

Future work will focus on the evaluation of further algorithms from the MiBench and MediaBench II suites and on possibilities to estimate the accuracy of the predictions. Furthermore, we will evaluate the usage of data address characteristics in addition to basic block vectors in order to increase the accuracy for algorithms such as *dijkstra*. Moreover, a detailed analysis of the required level of basic block coverage would be interesting for reducing the runtime of the clustering algorithm. Furthermore, a combination of our basic block coverage approach and random projection is also promising. Our application of linear regression to determine the relationship between the input data size and the number of phase members could be enhanced by using genetic programming. Interesting approaches to increase the accuracy of the predictions are the application of variable length intervals to determine similar parts in executions, and a special treatment of the start and end phases.

8. REFERENCES

[1] ARM. *AMBA Specification (Rev 2.0)*, May 1999.

[2] ARM Limited. *ARM926EJ-S Technical Reference Manual*, 2003.

[3] ARM Limited. *ARM11 MPCore Processor Technical Reference Manual r1p0*, Feb 2008.

[4] Arnold S. Berger. *Embedded Systems Design: An Introduction To Processes, Tools, And Techniques*. CMP Books, 2001.

[5] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[6] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[8] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not.*, 38(5):245–257, 2003.

[9] L. Eeckhout. Measuring Benchmark Similarity Using Inherent Program Characteristics. *IEEE Trans. Comput.*, 55(6):769–782, 2006. Student Member-Ajay Joshi and Student Member-Aashish Phansalkar and Senior Member-Lizy Kurian John.

[10] L. Eeckhout, J. Sampson, and B. Calder. Exploiting Program Microarchitecture Independent Characteristics and Phase Behavior for Reduced Benchmark Suite Simulation. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization*, pages 2–12, Austin, TX, USA, 10 2005. IEEE.

[11] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Designing Computer Architecture Research Workloads. *Computer*, 36(2):65–71, 2003.

[12] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Workload Design: Selecting Representative Program-Input Pairs. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 83–94, Charlottesville, VA, USA, 9 2002. IEEE Computer Society.

[13] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the Impact of Input Data Sets on Program Behavior and its Applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2 2003.

[14] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf. MediaBench II video: Expediting the next generation of video systems research, 2009.

[15] G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam. MiDataSets: Creating The Conditions For A More Realistic Evaluation of Iterative Optimization. In *International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2007)*, January 2007.

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[17] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7, Sep 2005.

[18] K. Hoste and L. Eeckhout. Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics. *IEEE Workload Characterization Symposium*, 0:83–92, 2006.

[19] K. Hoste and L. Eeckhout. Characterizing the Unique and Diverse Behaviors in Existing and Emerging General-Purpose and Domain-Specific Benchmark Suites. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 157–168, Austin, TX, USA, 4 2008. IEEE.

[20] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: a preliminary application to the data stream. In *Workload characterization of emerging computer applications*, pages 145–163. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[21] OMG. *A UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)*. Object Management Group, 2009.

[22] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2005)*, pages 10–20, Austin, TX, 3 2005. IEEE.

[23] L. Pustina, S. Schwarzer, and P. Martini. A Methodology for Performance Predictions of Future ARM Systems Modelled in UML. In *Proceedings of the 2nd Annual IEEE International Systems Conference Syscon 2008*, 2008.

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57, 2002.

[25] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and Exploiting Program Phases. *IEEE Micro*, 23(6):84–93, 2003.

[26] C. U. Smith and L. G. Williams. *Performance solutions: a practical guide to creating responsive, scalable software*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.

[27] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.

[28] Texas Instruments. *OMAP5912 Applications Processor Data Manual*, Dec 2003.

[29] D. W. Wall. Predicting program behavior using real or estimated profiles. *SIGPLAN Not.*, 26(6):59–70, 1991.