A Data Analysis Method for Software Performance Prediction

Gianluca Bontempi*
IMEC
Leuven, Belgium
gbonte@imec.be

Wido Kruijtzer
Philips Research Laboratories
Eindhoven, The Netherlands
wido.kruijtzer@philips.com

Abstract

This paper explores the role of data analysis methods to support system-level designers in characterising the performance of embedded applications. In particular, we address the performance modelling of software applications running on an embedded microprocessor. We propose a data analysis method, which, on the basis of a parameterisation of the software functionality and the hardware architecture, is able to predict the number of execution cycles on an embedded processor. Experiments with standard computational code (sorting, mathematical computation) and with MPEG variable length decoding are presented to support this claim.

1 Introduction

Performance analysis is crucial in many phases of system design, like system optimisation, design validation and IP authoring [4]. A particular instance of performance modelling is represented by high-level software estimation which addresses the problem of predicting the performance of an embedded software application using a high level characterisation of the functionality and the architecture. The high-level approach allows faster simulation with respect to cycle-true processor simulation techniques and is a key enabler for fast embedded designs, where the role of software is incessantly growing. In literature there are many works available on software estimation, both for large software systems and embedded applications. Software estimation techniques can be divided into static [13] and dynamic techniques [12][3]. Static techniques analyse a software specification without executing it and are often employed for worst-case analysis. Dynamic techniques, on the other hand, address the analysis of run-time behaviours and are preferred when the system performance is heavily data-dependent. For this reason we will restrict our focus to dynamic estimation problems only. Many of the dynamic approaches in literature adopt the *annotation technique* in order to define a performance model. This means that the functional code is sequentially annotated (e.g., line-by-line or block-by-block) with performance expressions, and then, once the functional code is executed, the overall performance estimation is computed. In this paper we propose a performance modelling technique which aims to go beyond sequential performance annotation, with the aim of taking advantage of more sophisticated data analysis techniques [7] than the ones currently used for software performance modelling. In fact, the current annotation practice merges together two steps of performance modelling that are (i) the definition of the *signature*, i.e. the set of features that are relevant for predicting the performance and (ii) the *model* estimation, i.e. the calibration of the model predicting the performance. As a consequence the existing annotation methods are limited to consider only linear performance modelling where the contribution of each annotation expression is linearly combined. Our proposal intends to separate these two steps, stressing how the definition of the signature is independent of the estimation of the model and proving that by making these two steps independent we can improve the prediction accuracy of the performance model. The closest work to the approach presented in this paper is discussed in [8]. Some of the ideas of the authors of [8], like the use of linear regression models for software estimation, are present also in this paper with the main difference that here a more general application is suggested and a different experimental setting is reported. Also, to the best of our knowledge, we are the first to consider dynamic performance models, which are (i) nonlinear and (ii), based not only on functional characteristics (e.g., number of instructions) but also on hardware features (e.g., memory delay).

2 A definition of performance model

There is a large amount of scientific work related to the notion of performance but a compact definition of performance model appears to be missing. We aim to fill this

* The research work was carried out during the stay of Gianluca Bontempi at Philips Research Laboratories in Eindhoven, The Netherlands

gap by proposing an abstract definition that can be easily instantiated and applied to practical problems. Before introducing the definition, some preliminary terminology is required. For that purpose, we will refer extensively to the terminology based on the *Y-chart representation* [1][9]. A Y-chart representation decomposes the design process of an embedded application into several steps: the specification of the functional model, the specification of the candidate architecture, the mapping of the functional tasks to the architectural components and the assessment of the resulting design. The *functional model* is a description of the behaviour of the application (e.g., a MPEG decoder) [5]. The *architecture* is the set of architectural components (e.g., microprocessor, ASIP, bus, memory, operating system) which, once combined according to some architectural criterion (e.g., bus centric platform), are expected to implement the functional behaviour. Once the functionality and the architecture are defined, the following step is to map the functional processes to the architectural components (e.g., the designer partitions the MPEG functionality by implementing a portion of it as software running on a microprocessor and the remaining part as custom hardware). At this stage, in order to have an insight on the performance of the application, some annotation with a performance model is required. Once this is done, the performance of the functional-architecture mapping can be evaluated by system-level simulation. The resulting performance characterisation may inspire the designer to improve the design, e.g., by refining the functional description, by introducing alternative architectural components or by changing the mapping.

According to our definition, *a performance model P* is a relation $P: F \times A \rightarrow C$ where F is the set of functional models, A is the set of possible architectures and $C \subset \Re^{+}$ is the set of values of a generic performance metric (e.g., the number of execution cycles, the power dissipation, and so on). In more practical terms, the couple {functional model, architectural description} can be described by a set of parameters S, hereafter denoted as the *signature* of the application. A performance model can then be rewritten in the form $P: S \rightarrow C$ where $S \in \Re^{m}$ is the set of signatures and m denotes the number of signature parameters which describe the couple {functional model, architectural description}. A signature is then composed of functional terms (like the number of executed instructions) and architectural terms (like the number of wait cycles per memory access).

3 Data analysis for software estimation

Once defined the performance model as an input/output relation, we can adopt well-known techniques in data analysis [7] to address independently the two steps of performance modelling: the definition of the signature and the estimation of the signature/performance relation. In practice we propose a procedure made of the following steps (represented by boldface numbers in Figure 1):

1) Definition of relevant application benchmarks. The first step of the procedure aims to select a representative family of applications to be used for collecting measurements. This family should cover a large spectrum of alternative functionalities (e.g., different streaming algorithms) and architectures (e.g., different platforms) if we require a high degree of generalisation from our performance model.

2) *Definition of the performance metric*. The designer must choose what is or what are the most relevant performance criteria (e.g., latency, power dissipation) for assessing the application.

3) *Definition of the signature*. We assume that the designer is able to define a set of parameters that determines the performance of the system. If this set were too large, then hindering the required accuracy, feature selection methods [12][19] could be used to reduce the dimensionality.

4) *Choice of the signature extractor*. We take into consideration only dynamic extractors. An example of dynamic extractor will be presented in Section 4.

5) *Collection of measurements from the reference model and from the signature extractor*. This step is generally time expensive but the intention is indeed to perform this measurement procedure only once and use, henceforth, the estimated model as predictor of the reference model.

6) *Modelling of the input/output relation on the basis of the data collected in step 5).* We propose the extension of the modelling techniques from simple linear models to non-linear models. This appears to be innovative in the performance analysis literature where no work has been devoted to investigate more complex performance models.

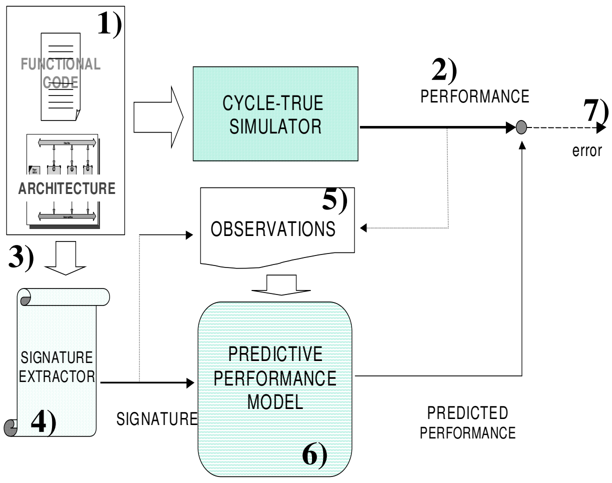

Figure 1. The estimation procedure of the software prediction model.

So far in performance modelling linear relations $c = \sum_{j=1}^{m} d_j s_j$ between the signature and the performance measure have been postulated [8][3]. While this assumption appears to be intuitive in the case of functional signature for a non-pipelined processor, it is probably too conservative in the case of complex functional and architectural signatures and maybe totally inadequate for non-timing performance metrics (e.g., power dissipation). To overcome this limitation, we propose to investigate also non-linear models $c = P(s)$ which, although less intuitive, could bring strong improvements in terms of accuracy. Among the large amount of results in statistical non-linear regression and machine learning we propose a local modelling technique, called *lazy learning* [2]. In local modelling the value of an unknown mapping P is estimated focusing on the region surrounding the point $s=q$ where the estimation itself is required. The procedure essentially consists of three steps (Figure 2): i) for each query point $s=q$, select a set of neighbors and weight their relevance according to some relevance criterion (e.g., the distance) ii) choose a local regression function f in a restricted family of parametric functions iii) compute the regression value $P(q)=f(q)$. So doing the approach requires keeping in memory the set of observations for each prediction, instead of discarding it as in a global modeling approach (e.g., linear regression). At the same time, local modeling requires only simple approximators (e.g., constant and/or linear) to model the dataset in a neighborhood of the query point. Moreover, the method is intrinsically adaptive, since the availability of new measurements requires simply the updating of the observation set. This feature is particularly relevant in performance modeling since new benchmarks and architecture can be integrated in the model at reduced cost

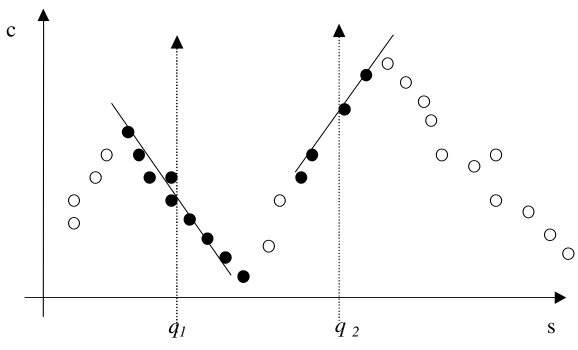

Figure 2 The values of c for the two queries s=q_1 and s=q_2 are returned by two local linear models (solid lines) which fit the samples (filled dots) in a neighborhood of the queries.

.

Lazy learning is a particular instance of local modelling which provides an automatic way of selecting the optimal number of neighbours for a given set of data. For more details on lazy learning, we refer the reader to [2].

7) *Validation*. Once the model is estimated, it is mandatory to evaluate how the performance prediction deteriorates by changing the scenario, or in other terms how the calibrated model is able to *generalise* in front of new scenarios.

8) *Performance model delivery*. Once the performance model has been estimated and validated, we can use it in order to document the software component. The performance description can be either in the form of a mathematical relation or in the form of a black box function, which, once entered with the signature, returns an evaluation of the performance. Note that the performance model could be employed at different granularity levels, depending on the rate at which the functional signature is measured. For example we could imagine to measure the functional signature after each basic block execution (low granularity) or at specific control signals (high-level granularity).

4 The experimental setting

We have instantiated the methodology introduced in the previous section in a real experiment.

1) *Benchmarks definition*: we consider two sets of functional benchmarks differing both for the size of the code and the generality of the application. The first is a *general-purpose benchmark set (B1),* made of 6 portions of software ranging from sorting algorithms (20 lines), mathematical computation (45 lines) to inverse discrete cosine transform (IDCT) (370 lines). The second is a pure *multimedia benchmark set (B2)* consisting of the variable length decoder (vld) algorithm a well-known component of the MPEG2 decoder which typically should be performed in real-time. This piece of code has a greater size (about 5000 lines) than the benchmarks taken into consideration in B1 and is intended to be a more realistic setting for testing the modelling capability of the method.

We consider a simple reference architecture made by one microprocessor (5 pipeline stage MIPS3000 with no operating system), one bus and one memory. We analyse a set of architectures obtained by sweeping the initial reference over two parameters:

– *number W of wait cycles* of the memory in the range from 0 to 4.
– the ratio *R=CPU_clock/bus_clock* in the range [1..5]: note that the CPU clock is kept fixed and that the larger this ratio the slower is the bus speed.

2) *Definition of the performance metric*: the performance measure to be predicted is the number of execution cycles. The prediction error is measured by the *NMS* (Normalised Mean Squared Error) and the *PE* (Percentage Error) criteria.

3-4) *Signature definition and extraction*: The functional signature of the software benchmarks is extracted by using *IPROF,* a highly portable instruction level profiler running on the Sparc processor, which performs complexity analysis of programs [11]. Its instruction set is made of $m=42$ instructions. Each signature extraction implies the generation of a vector s of m integers where the j-th element ($j=1,\dots,m$) denotes the number of times that the j-th instruction was executed during the emulation of the software.

5) *Data collection*: Every component of the B1 set is executed *15* times, each time with a different input configuration. As a result, we collect *90* measures for this benchmark. The code B2 is run for 2 different MPEG2 input files: *popplen.m2v* (7 pictures) and *tennis.m2v* (9 pictures). For both runs, a measure of the number of execution cycles (in the order of 10^8) is taken at the end of each picture. For this benchmark we collect a total of *16* measurements.

The reference performance numbers are extracted by using the Philips internal TSS tool [10]. TSS (Tool for System Simulation) is a cycle-true simulation framework which allows for a description of complex hardware architectures in the C programming language.

After running for N times both the signature extractor and the TSS reference model, we obtain a dataset $D=\{s_i,c_i\}$ $i=1,\dots,N$, of N samples where s_i is a vector of non-negative integers of dimensionality m.

6) *Model estimation.* On the basis of the dataset D we estimate a prediction model $c=P(s)$ of the number of cycles c. Both linear and non-linear models are taken into consideration. In particular we adopt the *multiple linear regression* technique [6] to identify the linear model and the *lazy learning* algorithm [2] as an example of non-linear approach based on the locally weighted regression technique.

7) *Model Validation.* This step aims to assess the accuracy of the performance model in a statistically significant way. To this aim, we adopt a conventional *training-and-test* procedure [7]. This means that the original data set, made of N samples, is decomposed t times in two non overlapping sets namely (i) the *training set*, made of N_{tr} samples, used to train the prediction model, and (ii) the *test set*, composed of $N_{ts}=N-N_{tr}$ samples, used to validate the prediction model according to the *PE* and *NMS* error criteria. After having trained and tested the prediction model for t times, the generalisation performance of the predictor is evaluated by averaging the error criteria over the t test sets. Note that the particular case where $t=N$, $N_{ts}=1$ and $N_{tr}=N-1$ is generally denoted as the *leave-one-out* (LOO) validation in the statistical literature.

4.1 Experimental results

We performed two prediction experiments.

1. Prediction accuracy in front of new functional applications: in this experiment we keep the architecture as fixed and we assess the robustness of the estimation method in front of new runs of the same application and in front of new functional applications. We consider (i) a *leave-one-sample-out* (LOO) approach, where each time a sample is put aside and the remaining ones are used to predict its value and (ii) a *partitioning* (PART) approach where the set of benchmark applications is split in a training-and-test way so that a test set contains only the applications which are not contained in the training set.

		NMS		PE	
		LIN	NLIN	LIN	NLIN
B1	LOO	6.6e-3	0.29	0.6%	4.4%
	PART	0.03	0.05	19.3%	8.8%
B2	LOO	1.4e-6	1.1e-5	0.06%	0.15%
	PART	2.2e-5	0.15	0.24%	17%

Table 1 Prediction error in front of new functionalities.

Note that while the LOO experiment returns the average accuracy of the prediction in front of new runs of the same functionality, the partitioning experiment assesses the average accuracy of the performance model in front of new functionalities.

The *NMS* and *PE* prediction errors of the linear model (LIN) and nonlinear model (NLIN) are summarised in Table 1.

2. Prediction accuracy in front of new architectures: the experiment aims to demonstrate that the method is able to generalise also over a number of hardware architectures, differing both in terms of memory and bus characteristics. The functionality is given by the *vld* algorithm decoding the tennis.m2v streaming file: we decompose the functionality in the decoding of *8* distinct pictures. The number of architectures taken into consideration is *23* since we consider all the combinations of the values of W and R (Section 4) except two of them ($\{W=3, R=5\}$ and $\{W=4,R=5\}$) because of excessive simulation time. Therefore the total number of experiments is equal to the product $N=8*23=184$.

We adopt again a training-and-test approach, where each time we keep the samples related to a specific architectural parameterisation and a specific picture aside as a test set and we use the remaining data to predict the number of cycles for the test set.

In particular we explore three different signatures of the performance model: (i) *functional signature only* (FUNC), i.e. the model is $c = P(s_f)$ where s_f contains only functional terms, (ii) *architecture signature only* (ARCH), i.e. the model is $c = P(s_a)$ with s_a containing only architectural terms and (iii) *both functional and* *architecture signature* (F&A), i.e. the model is $c = P(s_f, s_a)$. Table 2 reports the average results of the linear and nonlinear model, once the different components of the signature are taken into consideration. From this table we see that the contributions of the functional signature or the architectural signature alone are not sufficient for having a good prediction and that the non-linear model outperforms the linear one in the F&A case.

SIGNAT.	NMS		PE	
	LIN	NLIN	LIN	NLIN
FUNC	1.08	1.41	70.2%	55.2%
ARCH	0.68	0.76	41%	41%
F & A	0.21	0.05	30.7%	4.4%

Table 2 Prediction error in front of new architectures for different signatures.

Figure 3, Figure 4 and Figure 5 illustrate the measured number of cycles vs. the predictions returned by the three nonlinear predictors, each featuring a different signature. In the figures[1] the architecture is kept constant during *8* consequent decoding experiments (experiments among two vertical grid lines). Note that the variation of the measured number of cycles is both due to different functionalities (inside a cluster of *8* decoded pictures) and to different architecture parameterisations (among clusters of *8* consequent decoded pictures).

As shown in Table 2 and depicted in Figure 3 and Figure 4, the contributions of the functional signature or the architectural signature alone are not sufficient for having a good prediction.

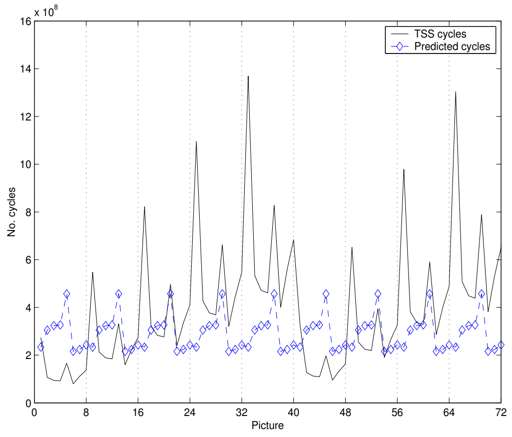

Figure 3. Real (TSS) vs. predicted number of cycles: nonlinear model having as input the functional signature only.

[1] For the sake of readability only a subset of the architectures is reported in the illustrations.

In particular, Figure 3 shows that the nonlinear prediction model $c = P(s_f)$ is able to account for the variation due to the functionality but is not able to account for the variability due to the architecture. The opposite happens in Figure 4, where the prediction model $c = P(s_a)$ can account for the effect due to the change of architectures but is not accurate in predicting the changes due to the functionality. Finally as shown in Figure 5, joining together the functional and the architectural signature in the non-linear model $c = P(s_f, s_a)$ we can have a very good generalisation also over different architectural parameterisations.

4.2 Discussion

In general terms, we observe that the data analysis approach returns good prediction accuracy. This result is still more remarkable if we consider that a TSS simulation takes many hours to return a performance number that a trained performance model can approximate in less than one second. About the linear vs. nonlinear model comparison, some considerations are required. On the basis of the benchmark B2 in Table 1 we could prematurely conclude that the linear model is more robust and reliable than the nonlinear approach.

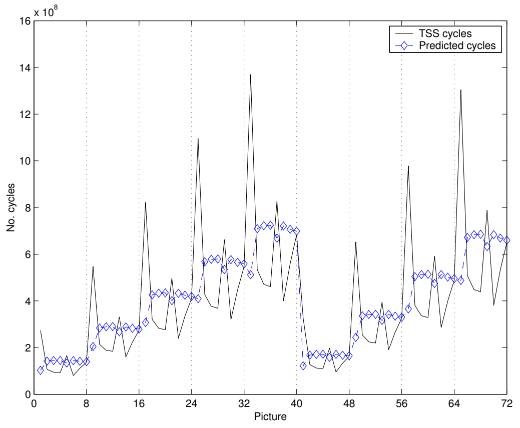

Figure 4. Real (TSS) vs. predicted number of cycles: nonlinear model having as input the architectural signature only.

However, the modest behaviour of the non-linear model in this case is due to two main reasons: (i) few data for the huge dimensionality of the problem and (ii) the fact that we are considering functional signatures only, making then realistic the assumption of linearity. Once these limitations do not hold any more, nonlinear models can return a very good prediction performance. This is shown by the results with the benchmark B1 in Table 1 and especially by the remarkable results in Table 2. There, by increasing the number of observations and by taking into consideration also architectural signatures, nonlinear models outperform linear representations.

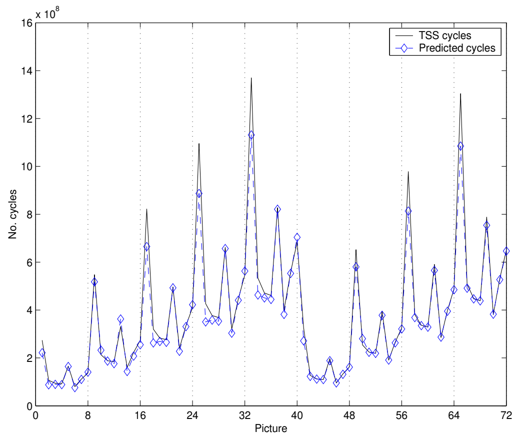

Figure 5. Real (TSS) vs. predicted number of cycles: nonlinear model having as input both the functional and the architectural signature.

In very general terms, the experimental setting of the paper provides some evidence that an accurate data-driven estimation procedure is possible and that the resulting prediction model can be robust enough to generalise to different functional applications and different architectures. Future work should extend this work by considering more complex functionalities and architectures (e.g., varying instruction sets, number of computational units, number and types of processors, partitioning and so on).

A final consideration is worthy to be done here. In this paper we limited on purpose our discussion to a data driven modelling approach which by definition trades accuracy for readability. The proposed approach is therefore effective in situations where accurate prediction and robustness are assumed to be more valuable than the easiness of interpretation. Future work should address configurations where a readable model is required, for example by combining this approach with more easy to interpret techniques.

References

[1] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, B. Tabbara (1997) *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press.

[2] Bontempi G., Birattari M., Bersini H. (2000) *A model selection approach for local learning.* Artificial Intelligence Communications, 13, 1, 41-48.

[3] C. Brandolese, W. Fornaciari, F. Salice , D. Sciuto (2001*) Source-Level Execution Time Estimation of C Programs.* Proceedings of CODES '01, Copenaghen, Denmark.

[4] J-Y. Brunel, A. Sangiovanni-Vincentelli, R. Kress and W. Kruijtzer (1998) *COSY: A methodology for system design based on reusable hardware & software IP's*. Technologies for the information Society, J-Y.Roger ed. IOS press, 709-716

[5] E.A. de Kock, G. Essink, W. Smits, P. van der Wolf, J-Y. Brunel, W.Kruijtzer, P. Lieverse, and K. Vissers (2000) *YAPI: Application modeling for signal processing systems* Design Automation Conference 2000.

[6] R. Draper, H. Smith (1981) *Applied Regression Analysis*. New York, John Wiley and Sons.

[7] U. Fayyad, G. Piatesky-Shapiro, P. Smyth (1996*) The KDD Process for Extracting Useful Knowledge from Volumes of Data*. Communications of the ACM, 39(11), 27-34.

[8] P. Giusto, G. Martin, E. Hancourt (2001) *Reliable estimation of execution time of embedded software.* Proceedings of DATE 2001, 580-587.

[9] B.Kienhuis, E. Depretteree, K. Vissers, and P. van der Wolf (1997) *An approach for quantitative analysis of application-specific dataflow architectures*. In ASAP'97.

[10] W. Kruijtzer (1997) *TSS: Tool for System Simulation*, Philips-IST Newsletter, 17, 5-7.

[11] P. M. Kuhn, W. Stechele (1998) *Complexity analysis of the emerging MPEG-4 standard as a basis for VLSI implementation,* SPIE 3309 Visual Commun. Image Processing, 24-30.

[12] M. Lazarescu, M. Lajolo, J. Bammi, E. Hancourt, L. Lavagno. (2000*) Compilation-based software performance estimation for system-level design.* Proceedings of Int. Workshop on Hardware/Software Codesign.

[13] Y. Li, S. Malik, A. Wolfe, (1995) *Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software*. Proceedings of the IEEE Real-Time Systems Symposium, 298 - 307.