Performance Prediction for Families of Data-Intensive Software Applications

Jacques Verriet
TNO-ESI
Eindhoven, The Netherlands
jacques.verriet@tno.nl

Reinier Dankers
Océ Technologies B.V.
Venlo, The Netherlands
reinier.dankers@oce.com

Lou Somers
Océ Technologies B.V.
Venlo, The Netherlands
lou.somers@oce.com

ABSTRACT

Performance is a critical system property of any system, in particular of data-intensive systems, such as image processing systems. We describe a performance engineering method for families of data-intensive systems that is both simple and accurate; the performance of new family members is predicted using models of existing family members. The predictive models are calibrated using static code analysis and regression. Code analysis is used to extract performance profiles, which are used in combination with regression to derive predictive performance models. A case study presents the application for an industrial image processing case, which revealed as benefits the easy application and identification of code performance optimization points.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; • **Mathematics of computing** → *Regression analysis*; • **Theory of computation** → Program analysis; • **Computing methodologies** → Modeling methodologies;

KEYWORDS

Software performance engineering; loop analysis; product families; data-intensive systems

ACM Reference Format:
Jacques Verriet, Reinier Dankers, and Lou Somers. 2018. Performance Prediction for Families of Data-Intensive Software Applications. In *Proceedings of ACM/SPEC International Conference on Performance Engineering (ICPE'18 Companion).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3185768.3186405

1 INTRODUCTION

Many companies make not a single product, but a family of products that share components. Examples are printer manufacturers, that use the same image processing software for a family of printers, and car manufacturers, that develop different cars that share the same chassis. The product family members' similarities allow development cost to be divided over several products. To allow software reuse within a family of products, the software needs to capture the commonalities of the family members; the differences are typically captured by the software's configuration options.

It is essential to have an accurate estimation of a system's performance during the early phases of development. Encountering performance issues in the later stages of development typically involves high correction costs, whereas correction in the early stages have much lower cost [21].

This paper describes a method to predict the performance of families of data-intensive systems. Examples of such systems are image processing pipelines, which can be found in printing systems, health care systems, advanced driver assistance systems and electron microscopes. The method allows performance prediction for future product family members from data collected from existing family members.

The method consists of three steps: (1) static code analysis, (2) model fitting, and (3) performance prediction. Our work builds on the work of Hendriks et al. [14], who couple model-based performance engineering to the V-model development process [9]. The regression used by Hendriks et al. [14] is based on the knowledge of the performance engineer: the validity of fitted performance models is checked using the engineer's knowledge of the software. In case of complex (legacy) software, the mental model of the engineer need not match the actual code organization. To consider the software structure, we add static code analysis for model calibration. This analysis is used to extract code structure information, which is used for fitting performance models that relate a function's (configuration and input) parameters and its execution time.

The paper is organized as follows. Section 2 provides an overview of related work. Sections 3 and 4 describe the method's static code analysis and model calibration, respectively. In Section 5, the method is applied to an industrial image processing case. Section 6 concludes the paper.

2 RELATED WORK

Performance engineering has been applied to many types of software systems. An overview of software performance engineering literature can be found in the survey by Balsamo et al. [1]. The method described in this paper uses two methods that are commonly used for performance engineering, i.e. static code analysis and regression, but that, to our knowledge, have not been used in combination. The combination allows more informed model calibration, as structural software knowledge is used for model fitting.

Model Extraction The model structure is extracted from the code and interactively refined to obtain a performance profile used for regression. Ferrari et al. [8] and Van Gemund [19] use a similar, but manual, approach; they create and refine code structure models to predict a system's performance.

There are also examples of automated model extraction for software systems. Unlike our approach, these typically view the system as a black box. For instance, Krogmann et al. [17] extract performance models from Java bytecode using genetic search. For each operation, they derive a formula describing the operation's performance. Using a benchmark of the underlying hardware platform, they are able to estimate the bytecode's performance. Brosig et al. [3] reconstruct an architectural performance model by observing a system at run time. They extract a system's control flow and its relevant parameters to create and calibrate a performance model for component-based systems.

Static Code Analysis Our model extraction is based on static code analysis, i.e. code analysis that does not rely on the execution of the code. It has frequently been used for WCET analysis [22], i.e. analyzing a program's worst-case execution time. Using control flow analysis, the possible control flows of a piece of software are extracted. For data-intensive systems, it is important to estimate the execution time of nested loops. This involves an estimation of the execution time of the loop body and an estimation of loop bounds. Often annotations of the software developer are used [16], but fully automated loop bound estimation is also possible [11, 12].

A drawback of the code analysis for WCET analysis is its complexity: users need to know the analysis tools' internals to obtain tight WCET estimations [22]. We use static code analysis with a different goal: we extract nested loop structures from code. This means that we can use simpler code analysis techniques. We rely on parsers that produce Abstract Syntax Trees (AST), which are traversed to obtain the information needed to create a model. ASTs have been used for program transformations [20], such as software rejuvenation [4]. Besides WCET analysis, we have not found applications of ASTs for performance engineering.

Regression Regression is commonly used for performance engineering. We apply informed regression: knowledge about the software is used to calibrate performance models. In the literature, however, regression is typically done without looking at the internal structure of the system being modeled. The most common form of regression is (multiple) linear regression [18]; it is also used for performance engineering [2, 7]. Huang et al. [15] use sparse polynomial regression for performance prediction. Instead of looking at the structure of the code, they introduce feature weights to search for the most dominant system features in to fit a polynomial performance model. Other regression techniques used for performance engineering include robust regression [5] and regression splines [6].

```
int main() {
  int pop[R][C];
  for (int j=1;j<R-1;j++) {
    for (int i=1;i<C-1;i++)
      pop[j][i]=rand()%2;
  }
  for (int x=0;x<G;x++) {
    cout<<"Gen "<<x<<":"<<endl;
    print(pop);
    life(pop);
  };
  return 0;
}
```

```
void life(int old[R][C]) {
  int tmp[R][C];
  copy(old, new);
  for (int j=1;j<R-1;j++) {
    for (int i=1;i<C-1;i++) {
      int cnt=old[j-1][i]+old[j-1][i-1]+
        old[j][i-1]+old[j+1][i-1]+
        old[j+1][i]+old[j+1][i+1]+
        old[j][i+1]+old[j-1][i+1];
      if (cnt<2||cnt>3) tmp[j][i]=0;
      if (cnt==2) tmp[j][i]=old[j][i];
      if (cnt==3) tmp[j][i]=1;
    }
  }
  copy(new, old);
}
```

Figure 1: C++ code for Game of Life

```
FUNCTION int main()
  SUM
    FOR(int j=1;j++;j<R-1)
      FOR(int i=1;i++;i<C-1)
        FUNCTION rand
    FOR(int x=0;x++;x<G)
      SUM
        FUNCTION print
        FUNCTION life
```

```
FUNCTION void life(int pop[R][C])
  SUM
    FUNCTION copy
    FOR(int j=1;j++;j<R-1)
      FOR(int i=1;i++;i<C-1)
        CONSTANT
    FUNCTION copy
```

Figure 2: Filtered ASTs of Game of Life functions

3 STATIC CODE ANALYSIS

This section describes how functions' execution profiles are extracted from source code. The profiles are translated into performance profiles that relate a function's (configuration and input) parameters and its execution time.

We aim for an easy-to-use manner to extract execution profiles from source code. The code analysis consists of five steps: (1) Parsing, (2) Tree filtering, (3) Function replacement, (4) Tree simplification, and (5) Loop bound analysis.

Parsing For the first step, we use a parser that creates an Abstract Syntax Tree (AST) for each source code file. Such parsers are available for all common programming languages.

To demonstrate our method, we use the Game of Life [10] as a running example. Two of its functions are shown in Figure 1. To parse the code, we use a C++ parser of Rufino.[1]

Tree Filtering In the second step, the ASTs are traversed and *loop structure trees* are created for each function definition. A tree's root consists of a function's name, parameters, and return type. Its body consists of a subset of the function bodies in the AST; it contains the loop structures and the function calls. Other AST elements are removed, as we assume that these do not significantly contribute to the execution time of a data-intensive system.

For the Game of Life, the AST is traversed and a loop structure tree is created for each function declaration. The trees of functions `main` and `life` are shown in Figure 2.

Function Replacement The third step is interactive: a user needs to specify a function that is to be analyzed. The corresponding loop structure tree contains loop constructs and function calls. The function replacement step selects a function call and retrieves the trees of all functions with matching name and number of arguments. Next, the user

[1] http://github.com/ricardojlrufino/eclipse-cdt-standalone-astparser

```
FUNCTION int main()
  SUM
    FOR(int j=1;j++;j<R-1)
      FOR(int i=1;i++;i<C-1)
        CONSTANT
    FOR(int x=0;x++;x<G)
      SUM
        FOR(int j=1;j++;j<R-1)
          FOR(int i=1;i++;i<C-1)
            CONSTANT
        FOR(int j=0;j++;j<R)
          FOR(int i=0;i++;i<C)
            CONSTANT
        FOR(int j=1;j++;j<R-1)
          FOR(int i=1;i++;i<C-1)
            CONSTANT
        FOR(int j= 0;j++;j<R)
          FOR(int i=0;i++;i<C)
            CONSTANT
```

```
FUNCTION int main()
  SUM
    FOR rows
      FOR columns
        CONSTANT
    FOR generations
      SUM
        FOR rows
          FOR columns
            CONSTANT
        FOR rows
          FOR columns
            CONSTANT
        FOR rows
          FOR columns
            CONSTANT
        FOR rows
          FOR columns
            CONSTANT
```

Figure 3: Function replacement and tree simplification of Game of Life function main

selects one of these functions, after which the function call gets replaced with the corresponding tree.

The tree of Game of Life's function `main` contains function calls of `rand`, `print`, and `life`. After the function `life` has been selected, a new tree is created containing code from both functions. In subsequent steps, the other functions are replaced; the resulting tree is shown left in Figure 3.

Tree Simplification Each time a function call gets replaced, the width and depth of the function's loop structure tree increases. To reduce its complexity, the tree is simplified after each replacement. A loop structure tree is simplified by traversing the tree in a depth-first order and replacing certain patterns by simpler, equivalent patterns. For instance, sequential occurrences of empty loop structure trees are replaced by a single empty loop structure tree and nested structures with only one child by this single child. Function replacement and tree simplification are continued until all function calls have been replaced.

The Game of Life example does not require any tree simplification; the final loop structure tree of function `main` is the one shown left in Figure 3.

Loop Bound Analysis At this point, we have a loop structure tree with all nested loops with the loop bounds as they can be found in the source code. For data-intensive systems, these bounds are typically variables which depend on the system's configuration or input. We use developer knowledge, similar to the annotations used for WCET analysis [16], to identify these variables.

The loop bound analysis step strongly depends on the application domain. For instance, for an image processing application, one may find a nested loop structure in which the outer loop iterates over all lines of an image and the inner loop over all pixels of the line. This kind of domain knowledge is to be provided by the developer. The result of the last step is a loop structure tree in which all loop bound variables have been replaced by their functional equivalents. We will call this a function's *performance profile*. It is input for the model calibration described in Section 4.

For the Game of Life, it is simple to replace the loop bounds by their functional equivalents: R and C equal the population dimensions, and G the number of generations. These loop bounds are seen as the essential parameters to express the performance of the function `main`. After replacing the actual loop bounds by their interpretation, we obtain the loop structure tree shown right in Figure 3.

We assume that all (innermost) loop bodies take constant time. This means that we can combine subsequent loops with (nearly) identical loop bounds. This simplification leads to the end result, which contains two sequential nested loops; an initialization loop and a main loop. The resulting loop structure tree has identified three parameters that influence the performance of function `main`: the number of rows and columns of a population, and the number of generations.

4 MODEL CALIBRATION

In Section 3, we have described how performance profiles are derived from code. These profiles are used to create analytic performance models. We aim for an approach that can be applied by software developers in industry. The proposed approach involves five steps: (1) Code instrumentation, (2) Data collection, (3) Model fitting, (4) Performance prediction, and (5) Model validation. Like in Section 3, we use the Game of Life as a running example.

Code Instrumentation A common way to enable performance measurements is to instrument the code with measurement functionality. For instance, one can instrument the start and end of functions with instructions to measure their execution times.

For the Game of Life, we are interested in the execution time of function `main`. Only this function needs to be instrumented; its code is instrumented by measuring the time at entry and exit. The difference between these times provides a measurement of the function's execution time.

Data Collection The second step involves collecting performance data from the instrumented code. The performance data should cover the parameters identified by the code analysis described in Section 3. Data can either be collected using systematic measurements or by mining available (performance) log files.

For the Game of Life, measurements have been done by systematically varying the number of rows (R), columns (C), and generations (G). We have performed 8,000 experiments by varying each parameter from 5 to 100 with steps of 5. Based on the code analysis, one would expect that the execution time of `main` depends on the product $P = R \cdot C \cdot G$ and the population size $S = R \cdot C$. Scatter plots of the execution time T plotted against R, C, G, S, and P are shown in Figure 4.

Model Fitting The model fitting step has two inputs: the identified performance profiles and the collected measurement data. A performance profile describes the nested loops of a function and as such describes how parameters influence the performance of this function. We use regression for model fitting; we let the regression be guided by the performance profiles to avoid finding relations that cannot be explained by a system's internals.

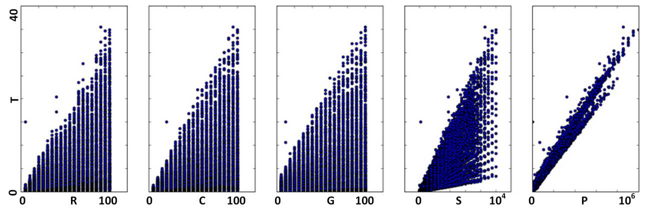

Figure 4: Game of Life execution times

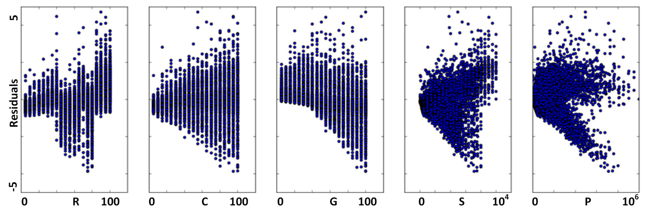

Figure 5: Residual plots

Regression is a statistical method to analyze the relation between variables [18]. The most common form is (multiple) linear regression, which is well suited for the relations as expressed by the performance profiles. It fits a linear function relating a function's input parameters and its performance, as well as an evaluation of the significance of the fitted function. As such, regression also provides an evaluation of the performance profiles. Finding an appropriate performance function typically involves several regression steps, which are guided by the performance profiles and the evaluation of the earlier steps.

The scatter plots in Figure 4 suggest a linear relation between $P = R \cdot C \cdot G$ and T. This is investigated further using linear regression [18]: the fitted model indicates that T can be estimated as $0.1107 + 3.47 \cdot 10^{-5} \cdot P$. Regression's t-tests indicate that the it is unlikely that the actual intercept and slope values deviate significantly from the fitted values 0.1107 and $3.47 \cdot 10^{-5}$. Another quality attribute is the coefficient of determination (R^2), which equals the amount of execution time variation explained by variable P. For this fit, R^2 equals 0.978, leaving about 2 percent of unaccounted variation.

Residual analysis can be used to further assess the validity of a fitted function [18]. Residuals are the differences between observed values and estimated values. Scatter plots of the residuals of the fitted function are shown in Figure 5. The residual plots do not show a relation of R, C, G, S, and P with T. This suggests that all execution time variability is captured by P.

However, the left plot in Figure 6 shows that small Game of Life instances typically have execution times that are much smaller than the fitted intercept of 0.1107. This suggests that the intercept should be zero, despite its significance according to the t-tests.

A second regression analysis was therefore performed without an intercept: T can be estimated as $3.51 \cdot 10^{-5} \cdot P$. The removal of the intercept had little influence on the slope and

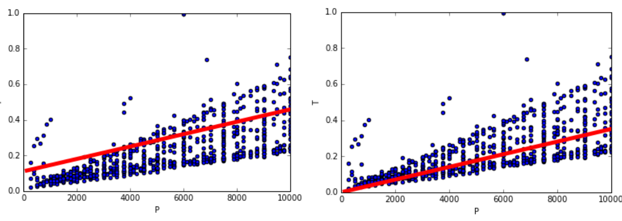

Figure 6: First and second regression model

the coefficient of determination. This means minor differences between the first and second fit for large instances, but a significant difference for small instances. The right plot in Figure 6 shows that the second fit is much better for small instances. Hence, we will use the second fit.

Performance Prediction A fitted performance model is used to predict the performance of future product family members by applying the fitted function to the (input and configuration) parameters of the new family member.

For the Game of Life, performance prediction can be done by varying parameters R, C, and G. We have considered instances with at most 100 rows, columns, and generations. Suppose we want to predict the execution time of an instance with 200 rows, columns, and generations. The fitted model predicts an execution time of $3.51 \cdot 10^{-5} \cdot 200^3 = 280.84$.

Model Validation The last step is a validation and consolidation step. The first four steps are typically applied during the early stage of development when prototypes of the new product family member are not yet available. As soon as prototypes are available, they can be used to validate the fitted performance model. This involves comparing measurements and predictions. Differences are to be explained and used to update and consolidate the performance model. The consolidated model can be used to predict the performance of a new future product family member by applying the steps described in this paper.

For the Game of Life, no new product is to be developed. To validate the earlier prediction, we did 20 measurements with 200 rows, columns, and generations. This gave an average execution time of 270.27. The difference with the prediction is less than 4 percent. Hence the fitted model accurately predicts the execution time of this larger instance.

5 CASE STUDY

The method described in Sections 3 and 4 has been applied to an industrial case involving a family of wide-format printers. These printers produce images of widths and heights of several meters. These images are printed in bands called swaths: a carriage containing print heads moves over the medium while their nozzles jet ink onto it.

The focus of the case is the printers' embedded image processing, which is called the *data path*. The data path takes a rasterized bitmap and applies image processing functions to calculate the firing sequences of the printer's nozzles. Each of these functions is applied for the subsequent swaths of the input bitmap. The data path functionality includes common image processing steps such as copying, resampling and masking, but it also includes application-specific steps. An example is called nozzle failure compensation (NFC). Nozzles may be temporarily unavailable due to various disturbances. To guarantee high image quality, the drops of the unavailable nozzles are jetted by available nozzles. NFC is the computation of the replacement nozzles.

The data path was originally developed for one wide-format printer. Later, it has been adapted to serve a family of printers. In early phases of development of a new wide-format printer, we want to quickly and accurately predict the execution time of its data path, i.e. without developing physical prototypes. For instance, we would like to predict the data path performance of printers with different dimensions and ones with different numbers of nozzles.

Static Code Analysis The data path is part of the printer's embedded software, which comprises circa 6,500 files with a total of circa 2,000,000 lines of code describing circa 90,000 functions. For all functions, loop structure trees were created. The function calls in the data path functions' loop structure trees were iteratively replaced by the loop structure trees of the called functions. The time needed to replace all function calls strongly depends on the function's call tree width and depth. For a simple image processing function, function call replacement takes less than a minute. For a more complex function, like NFC, this operation requires more time. In total, the static code analysis took a few hours.

The loop bounds of the final structure trees were analyzed to obtain input for the calibration of a performance model. For most of the data path's image processing steps, the analysis revealed a nested loop that iterates over the height and width of a swath. Besides these swath dimensions, NFC's performance profile also includes the number of defect nozzles.

Model Calibration To create a data path performance model, the image processing functions were instrumented. Function entry and exit times were measured on a test setup, and the differences were logged as execution times. Static code analysis revealed three parameters that influence the execution time of the performance of the embedded data path: the width of the input bitmap, the number of nozzles in the printer's print head, and the number of nozzles that require compensation. This was confirmed using experiments.

Regression was applied to the collected data using the identified performance profiles as guidelines. For most data path functions, regression confirmed the expected linear dependency between the swath size and the function's execution time and negligible execution times for small swaths. However, model calibration also revealed that a masking function required a significant initialization time, whereas software developers had expected this to be negligible. The performance of the masking step was analyzed in more detail by inspecting the code. This showed that the masking step includes the preparation of the swath data structures. The scatter plots in Figure 7 show the execution times of this preparation step. It shows that the preparation of the data structures is independent of the number of nozzles, the number of nozzle defects, the swath width, and the swath size. After separating

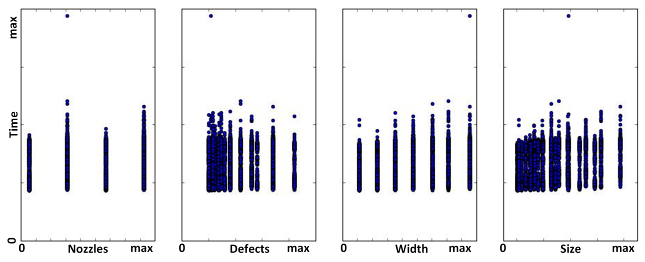

Figure 7: Swath data structure initialization times

preparation and masking, the masking does not require a large initialization time anymore.

For NFC, the expected performance model is also different from most image processing steps. Regression confirmed that NFC's execution times are determined by the number of defect nozzles and the swath dimensions.

This means that all parameters that were expected to influence the image processing times also appeared in the performance models. However, not all loops in a function's call hierarchy have a significant influence: some fitted performance functions were simpler than the found performance profiles.

Performance Analysis The fitted models were used to predict the performance of different printer configurations without actually constructing them. It is possible to assess the influence of printer carriage dimensions, bitmap widths, and numbers of defect nozzles on the data path performance. If future wide-format printers allow wider images or have more nozzles than existing ones, then the models can be used to predict the corresponding data path execution times. The performance models can also be used to vary the sequence of image processing steps. The latter is relevant if future printers require different image processing.

Benefits The case has shown benefits besides the performance prediction. One benefit is the identification of unexpected performance behavior. An example is the masking step that involved a large initialization time. Similarly, the model can be used to identify performance bottlenecks. The models identify the most expensive steps; this allows data path designers to focus their attention on optimizing the functions that contribute most to the execution time.

The possibility to adapt the sequence of image processing steps also provides a way to optimize data path performance. A typical data path may contain resampling steps, which change the image resolution. To optimize performance, the image processing should be applied to as few data as possible. As the models allow the analysis of different sequences, the optimal sequence can be selected without implementing and testing all of them.

Reflection We have explained the application of code analysis and regression to predict the performance of configurations of a printer's data path. The time needed for the code analysis step strongly depends on the size of a function's call tree. The value of code analysis is limited for functions with a small call tree; these can be analyzed by examining the code.

However, we observed that it is not possible to manually create call trees of the more complex data path functions like NFC. In other words, an important value of code analysis step is keeping overview of functions being called.

The data path that we analyzed is an application that does not share its computational resources with other functionality. This means that the execution times can directly be predicted using the performance models. The models can, however, also be used for applications that share computational resources. This requires scaling a function's execution speed with the amount of resources assigned to it. A convenient way to achieve this is the blueprint of Hendriks et al. [13].

The performance prediction (implicitly) assumes that new product family members will run on the same hardware as existing family members. A new family member may also be equipped with different hardware. To obtain accurate performance predictions, the differences between the platforms should be considered. This is described by Hendriks et al. [14]; they use an on-line performance benchmark to predict an application's execution times on one or more cores of an unknown processor from the application's execution times on a known processor.

6 CONCLUSION

We have described a software performance engineering method for families of data-intensive (embedded) systems. The method involves two steps: (1) a static code analysis step that creates function performance profiles by parsing the code and adding developer knowledge, and (2) a model calibration step that combines the performance profiles and system measurements to create predictive performance models. The method has been applied to the image processing of a family of wide-format printers. It allows the prediction of the performance of future printers with different dimensions. Moreover, analysis using the method revealed information about the execution times of existing printers that was not known before. Because the method is implemented using simple means, it is usable for software developers in industry.

In the future, we plan to improve the applicability of our approach by combining it with benchmarking [14], multi-processor estimation [14] and the system-level performance modeling blueprint of Hendriks et al. [13]. This combination allows the performance prediction of a data-intensive software application on different computational platforms, which it shares with other applications.

ACKNOWLEDGMENTS

The research is carried out as part of the Octo+ program under the responsibility of ESI, part of TNO, with Océ Technologies B.V. as the carrying industrial partner. The Octo+ research is supported by the Netherlands Organisation for Applied Scientific Research TNO.

REFERENCES

[1] S. Balsamo, A. di Marco, P. Inverardi, and M. Simeoni. 2004. Model-based performance prediction in software development: A survey. *IEEE Trans. on Softw. Eng.* 30 (2004), 295–310.

[2] G. Bontempi and W. Kruijtzer. 2002. A Data Analysis Method for Software Performance Prediction. In *Conference on Design, Automation and Test in Europe (DATE'02)*. 971–976.

[3] F. Brosig, N. Huber, and S. Kounev. 2011. Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 183–192.

[4] J. L. Cánovas Izquierdo and J. García Molina. 2014. Extracting models from source code in software modernization. *Softw. & Syst. Model.* 13 (2014), 713–734.

[5] G. Casale, P. Cremonesi, and R. Turrin. 2008. Robust Workload Estimation in Queueing Network Performance Models. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*. 183–187.

[6] M. Courtois and M. Woodside. 2000. Using Regression Splines for Software Performance Analysis. In *2nd International Workshop on Software and Performance (WOSP'00)*. 105–114.

[7] E. Eskenazi, A. Fioukov, and D. Hammer. 2004. Performance Prediction for Component Compositions. In *7th International Symposium on Component-Based Software Engineering (CBSE 2004)*. LNCS, Vol. 3054. 280–293.

[8] D. Ferrari, G. Serazzi, and A. Zeigner. 1983. *Measurement and Tuning of Computer Systems*. Prentice Hall, Upper Saddle River, NJ.

[9] K. Forsberg and H. Mooz. 1991. The relationship of system engineering to the project cycle. 1 (1991), 57–65.

[10] M. Gardner. 1970. Mathematical Games – The fantastic combinations of John Conway's new solitaire game "life". *Sci. Am.* 223 (1970), 120–123.

[11] J. Gustafsson, B. Lisper, C. Sandberg, and N. Bermudo. 2003. A Tool for Automatic Flow Analysis of C-programs for WCET Calculation. In *8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*. 106–112.

[12] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. 1998. Bounding Loop Iterations for Timing Analysis. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)*.

[13] M. Hendriks, T. Basten, J. Verriet, M. Brassé, and L. Somers. 2014. A blueprint for system-level performance modeling of software-intensive embedded systems. *Int. J. Softw. Tools for Technol. Transf.* 18 (2014), 21–40.

[14] M. Hendriks, J. Verriet, T. Basten, M. Brassé, R. Dankers, R. Laan, A. Lint, H. Moneva, L. Somers, and M. Willekens. 2015. Performance Engineering for Industrial Embedded Data-Processing Systems. In *1st International Workshop on Processes, Methods, and Tools for Engineering Embedded Systems (PROMOTE2015)*. LNCS, Vol. 9459. 399–414.

[15] L. Huang, J. Jia, B. Yu, B. Chun, P. Maniatis, and M. Naik. 2010. Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression. *Adv. Neural Inf. Process. Syst.* 23 (2010), 883–891.

[16] R. Kirner, J. Knoop, A. Prantl, M. Schordan, and A. Kadlec. 2011. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Softw. & Syst. Model.* 10 (2011), 411–437.

[17] K. Krogmann, M. Kuperberg, and R. Reussner. 2010. Using Genetic Search for Reverse Engineering of Parametric Behavior Models for Performance Prediction. *IEEE Trans. Softw. Eng.* 36 (2010), 865–877.

[18] D. C. Montgomery, E. A. Peck, and G. G. Vining. 2001. *Introduction to linear regression analysis* (third ed.). John Wiley & Sons, Inc., New York.

[19] A. J. C. van Gemund. 2003. Symbolic performance modeling of parallel systems. *IEEE Trans. Parallel and Distrib. Syst.* 14 (2003), 154–165.

[20] E. Visser. 2005. A survey of strategies in rule-based program transformation systems. *J. Symbol. Comput.* 40 (2005), 831–873.

[21] J. C. Westland. 2002. The cost of errors in software development: evidence from industry. *J. Syst. and Softw.* 62 (2002), 1–9.

[22] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. 2008. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.* 7 (2008), Article 36.