Computer Science • 17 (3) 2016 http://dx.doi.org/10.7494/csci.2016.17.3.407

Agnieszka Kamińska
Włodzimierz Bielecki

STATISTICAL MODELS TO ACCELERATE SOFTWARE DEVELOPMENT BY MEANS OF ITERATIVE COMPILATION

Abstract *Minimization of data-processing time and reduction of software-development time are important practical problems to be tackled by modern computer science.*

This paper presents the authors' proposal of a family of statistical models for the estimation of program execution time, which is an approach focused on both of the above problems at the same time. The family consists of a general model and specific models and has been elaborated based on empirical data collected for pattern-program loops representing some arbitrarily selected features related to the program structure and the specificity of a program-execution environment.

The paper presents steps to elaborate the aforementioned family as well as the results of the carried-out experimental research. The paper demonstrates how the elaborated models can be applied in iterative compilation for optimization purposes, allowing us to reduce the time of software development and produce code with minimal execution time.

Keywords program execution time, optimizing compilation, iterative compilation, statistical models, processor cache

Citation Computer Science 17 (3) 2016: 407–435

1. Introduction

The reduction of software development time is an important practical problem to be tackled by modern computer science. Resolving this problem is an object of research carried out both in the scientific and industrial centers.

Special attention is paid to compilation. During compilation, a computer program written in a programming language comprehensible to a human is converted into an executable form comprehensible to a computer. Applying the appropriately selected transformations at a compilation stage, one can transform a program, written in a given programming language and for a given computer platform, to various yet semantically equivalent executables which however differ in execution times. Within the compilation known as optimizing, one tries to select transformations allowing for the shortest execution time of a resultant executable in the target environment.

In view of the great complexity of the organization of modern computers, applying methods used in optimizing compilation it is impossible to undoubtedly indicate which of possible versions of the source code of a given program will have the shortest execution time in a given target environment. Iterative compilation is one of the possible ways to produce such a code.

Within iterative compilation, all considered and semantically equivalent source codes of a given program are executed in a target environment; their execution times are compared, and the source code with the shortest execution time is selected for final use [10, 13].

In the case of programs intended for solving complex problems for large data sets, it may take several hours or even days to complete a single iteration of iterative compilation. Such a situation takes place, for example, for real-life problems that, in view of the necessity to be quickly solved, are subjected to being solved by means of parallel computing.

For the sake of its potentially being very time consuming, iterative compilation can be costly in practical applications, especially in the case of commercial software development. Therefore, a potential improvement in iterative compilation is to use a mathematical model in order to select from possible source-code variants of a given program those with the shortest expected execution times.

Modern computer architectures are so complex that it is not possible to undoubtedly indicate – without executing all considered source code variants in the target hardware environment – the source-code variant with the shortest execution time in the target hardware environment. Therefore, the mathematical model would be used for identifying, among the considered source-code variants, several variants with shortest *expected* execution times and iterative compilation would be performed only on the so-selected variants instead of on the entire set of all the considered variants. This would result in the shortening of iterative compilation time with no deterioration of its results.

Potential practical advantages related to such an improvement in iterative compilation and the scientific gap in this area have become an inspiration for the authors' solution presented in this paper and involving the elaboration of a family of iterative compilation-oriented statistical models for the estimation of program execution time. The authors' solution is based on statistical models allowing for taking into account a large number of factors influencing program execution time and the complexity of their mutual relations.

Because most of time-consuming operations – calculations made within computer programs – are executed in loop nests, the scope of the applicability of an elaborated family of statistical models is limited to a class of coarse-grained parallel loop nests, represented in the OpenMP C/C++ standard.

Coarse-grained granulation [12] takes place when the time of the execution of data-processing-related operations in a program is longer than the total time of initializing these operations and transferring the data needed for the execution of these operations. This type of granulation corresponds with the nested-loop structure in which the outermost loop of the nest is parallel. Coarse-grained granulation is typically used in the parallelization of programs executed by currently very popular multiprocessor machines with shared memory.

OpenMP [17] is currently a very popular standard for representing parallelism of applications written in C and C++ and intended for execution on multiprocessor machines with shared memory.

The contribution of this paper over the related work is as follows:

- A general model for the estimation of parallel coarse-grained program execution time,
- Two specific models derived from the general model,
- The demonstration of practical advantages of using the presented specific models in iterative compilation.

The rest of the paper is organized as follows. Section 2 presents the idea and basic assumptions of a family of statistical models for the estimation of program execution time. Section 3 outlines a general model. Section 4 describes how to estimate the values of parameters of the general model and use these estimates to derive specific models from the general model. Section 5 discusses the quality of estimations made according to the obtained models. Section 6 describes how the elaborated specific models can be applied in iterative compilation. Section 7 presents the results of experimental research focused on examining practical advantages of using specific models in iterative compilation. Section 8 discusses related work; conclusions are drawn in Section 9.

2. Basic assumptions of statistical models for the estimation of program execution time

A family of statistical models for the estimation of program execution time is based on a general model that makes it possible to estimate the execution time of coarse-grained program loop nests presented in the OpenMP C/C++ standard.

Program execution time has been assumed as the dependent variable of a general model. We have assumed that quantitative variables reflecting the factors that significantly influence program execution time should be the independent variables of a general model. Apart from dependent and independent variables, a general model comprises parameters whose values are unknown a priori.

We have decided that the values of these parameters should be determined for a specific computer environment by means of regression analysis carried out for the empirical data collected in this environment.

Although there are also various evolutionary methods that are used for modeling purposes, we have decided not to involve any of them in our research. Contrary to statistical methods, evolutionary methods are highly unpredictable in terms of the cost of using them and the quality of results they produce [7] – and this unpredictability makes evolutionary methods unfit for being considered as a potential way of carrying out the proposed improvement of iterative compilation.

In order to collect the empirical data necessary for determining the values of model parameters, we have used programs prepared specially for this purpose. These programs are hereafter referred to as pattern programs.

Because of a significant disproportion between the processor speed and memory-access time of today's computers, cache memory is used in processors; it is a bridge in communication between a processor and the main memory. For this reason, we have decided to reflect in pattern programs typical situations of taking advantage of cache memory. These situations are characterized by means of data reuse and cache interference; hence, each pattern program represents an arbitrarily assumed combination of two characteristics: data reuse and cache interference. These two characteristics are sufficient to cover and describe the whole intended scope of the applicability of our family of statistical models.

Because we use only a very small number of characteristics with a very limited number of value variants as the basis for pattern programs, a proposed approach is highly general and requires only a small number of pattern programs to be prepared. As a consequence, the time to elaborate the resultant specific models is much shorter than it would be in the situation when a greater number of characteristics with a greater number of value variants was used as the basis for pattern programs.

There are two types of data reuse: temporal and spatial. Temporal data reuse takes place when the data fetched from a specific memory location are many times reused in the program. Spatial data reuse takes place when the data adjacent, within a given cache line, to the data fetched from a specific memory location are used in the program [1, 20]. Both types of data reuse can be easily identified from a source code, applying the approach proposed in [20].

Cache interference takes place when a cache line containing data, which can be reused in a program, is overwritten with new data, despite the fact that there is sufficient unoccupied space in the cache where to the new data could well be fetched – however, because of cache organization, a specific and already-occupied cache line has to be overwritten instead [1, 6, 8, 19]. The influence of cache interference on data reuse can be assessed based on the source code of a program, applying the approach proposed in [14].

In order to elaborate a family of statistical models, we have elaborated and used pattern programs that reflect two typical situations of taking advantage of cache memory:

- reusing data stored in the cache with no cache interference (this situation is represented by pattern program *nonInterf*),
- reusing data stored in the cache with cache interference (this situation is represented by pattern program *matmul*).

The source codes of the both pattern programs are presented in Table 1.

Table 1

Pattern programs.

Assumptions	**Realization**
Pattern program 1 Data reuse with no cache interference	**Loop nest *nonInterf*** int ma[N][N], mb[N][N], mc[N][N], md[N][N], me[N][N]; int i, j, N; for $(i = 0; i \le N - 1; i++)\{$ for $(j = 0; j \le N - 1; j++)\{$ ma[i][j] = 1; mb[i][j] = mc[i][j] + md[i][j]*me[i][j]; } }
Pattern program 2 Data reuse with cache interference	**Loop nest *matmul*** int ma[N][N], mb[N][N], mc[N][N]; int i, j, k, r, N; for $(i = 0; i \le N - 1; i++)\{$ for $(k = 0; k \le N - 1; k++)\{$ r = ma[i][k]; for $(j = 0; j \le N - 1; j++)\{$ mc[i][j] = mc[i][j] + r*mb[k][j]; } } }

After substituting the parameters of a general model with values determined by means of regression analysis, the general model becomes a specific one. A specific model defines a general model for a particular situation by assigning relevant values to the parameters of the general model.

Each specific model is derived from a general model for a particular pattern program. A specific model can be applied not only to a pattern program but also to other programs with the same data-reuse type, as in the case of the pattern program. We call such programs “non-pattern”.

In order to avoid the extrapolation of a specific model beyond the data range for which the model is constructed, we have elaborated assumptions regarding the scope of the applicability of a specific model to non-pattern programs. For the aforementioned purpose, we have introduced assumptions limiting:

- the total size of data processed in a program,
- the maximum number of iterations in a single chunk of iterations assigned to be executed by a program thread,
- program execution time.

To assess whether it is possible to estimate with sufficient accuracy the execution time of non-pattern programs by applying a specific model, we have elaborated a method of assessing the quality of estimates generated by a specific model. This quality-assessment method relates the achieved estimates to real values determined empirically in a target environment.

Specific models can be used in iterative compilation to estimate the execution time of various source-code variants of a given program. For each source-code variant of a given program, one calculates the estimated execution time as per a relevant specific model. Then, several source-code variants with best estimates (i.e., shortest expected execution times in the target hardware environment) are subjected to iterative compilation. From these source-code variants, the source-code variant for final use is selected based on the results of the carried-out iterative compilation.

3. General model

The execution time of a program is the resultant of the interaction of a great number of various factors. Because of the number and heterogeneity of these factors, it is not possible to identify and quantify them all so that all of them could be included in our model for the estimation of program execution time. Therefore, in order to elaborate a model, we have decided to act in the following way: select some factors that potentially influence program execution time, empirically prove that the selected factors indeed influence program execution time, and quantify their influence as the independent variables of the model. Intuitively, the execution time of a given program depends on factors related to the environment of program execution, the structure of an executed program, and the way in which the program is executed. Taking into account the expected area of the application of our model for the estimation of program execution time, these intuitively selected factors are equivalent to:

a) the structure of a parallel program and the type of parallelism exposed by this program,
b) the specificity of a problem to be resolved in parallel,
c) parameters of a hardware environment in which a parallel program is to be executed.

In a model, we have quantified the influence of factors a), b), and c) on the program execution time in the following way.

a) A parallel program and the type of parallelism exposed by this program

In the OpenMP C/C++ standard, programs are executed by multiple threads. The time of execution of a parallel program depends on the number of invoked OpenMP threads – therefore, the number of OpenMP threads executing the program has been adopted as an independent variable ($X4$) of a general model.

While executing a program loop nest, each of the invoked OpenMP threads is assigned to execute a certain number of iterations of the loop nest. Depending on the adopted way of assigning loop-nest iterations to OpenMP threads, particular threads may be assigned to execute either identical or different numbers of loop-nest iterations.

Loop-nest iterations to be executed are assigned to OpenMP threads in portions called chunks; depending on the settings made chunks may be of identical or different sizes. As all the invoked threads simultaneously start executing their assigned iterations, the time of execution of a program loop nest is determined by the execution time of the thread that *last* finishes executing its assigned iterations. This essentially will be the thread that has been assigned to execute the greatest number of iterations, and the time in which this thread executes its assigned iterations is determined by the size of the largest chunk of iterations assigned to this thread. Therefore, we have adopted as an independent variable ($X3$) of a general model the maximum number of iterations in a single chunk of iterations assigned to be executed by an OpenMP thread.

b) The specificity of a problem to be resolved in parallel

From a low-level perspective, the specificity and variety of problems to be solved are reflected in the number and type of arithmetic operations to be executed by the processor. A simple yet effective way of expressing this observation quantitatively is to assign different weights to different types of arithmetic operations. Weights should be selected based on the analysis of the execution times of instructions in a given processor. With this approach, it is guaranteed that different types of arithmetic operations (e.g., addition and multiplication) are comparable. Therefore, the total weighted number of arithmetic operations per single program thread has been adopted as an independent variable ($X2$) of a general model.

c) Parameters of a hardware environment in which a parallel program is to be executed

Ideally, all of the data needed by a processor during program execution should be available in the processor cache at the moment when they are requested, instead of being just then fetched from main memory into processor cache.

On the other hand, the capacity of cache memory and its replacement policy (associativity) determine what fraction of the data processed in a program will be available in the cache right at the moment they are requested.

This means that the time of program execution depends on the following factors:

1. The actual capacity of cache memory in a given computer system and its replacement policy (associativity).

2. The minimum capacity of direct-mapped cache, which is necessary in order to contain all of the data processed in a program, assuming full temporal and spatial reuse of the data stored in cache memory. The minimum data-storage capacity in question can be estimated by means of data footprint [14, 20]. In order to calculate the data footprint for a given program, it is sufficient to know its source code; there is no need to execute this program. Calculation of the data footprint can be carried out automatically and, at the same time, statically – by parsing the source code of the program and using the parse results to obtain the data reuse factors; based on this, the value of the data footprint is easy to determine (according to the methods presented in [14] and [20]).
3. The relationship between factors 1 and 2.

In connection with the discussion above, a relationship between factors 1 and 2 has been adopted as an independent variable ($X1$) of a general model.

Thus, the final list of potential independent variables of a model comprises the following variables: $X1$, $X2$, $X3$, and $X4$. To empirically prove which of these potential independent variables indeed influence the program execution time, we have used regression analysis.

The defined independent variables take into account and reflect many different aspects of the parallel execution of a program; see Table 2. All of its remaining aspects that are not covered by the independent variables are indirectly reflected in the parameters of the model.

Table 2
Determiners of independent variables.

$\boldsymbol{X1}$	$\boldsymbol{X2}$
• Cache size (L1 and L2) • Cache organization • Data reuse in the program • Type of data reuse in the program • Cache interference • Program structure	• Number of arithmetic operations executed in a program • Type of arithmetic operations executed in a program • Time of execution of particular types of processor arithmetic operations
$\boldsymbol{X3}$	$\boldsymbol{X4}$
• Structure of a parallel program • Type of parallelism exposed by a program • Way of assigning tasks to particular threads executing a program	• Number of OpenMP threads

With such a list of independent variables of a model to be formulated and assuming that the dependent variable is Yt that estimates CPU time of the execution of a program loop nest by all program threads (Yt is expressed by the number of CPU clock cycles), we have undertaken regression analysis. The object of regression analysis was the empirical data collected for two pattern programs (*nonInterf* and *matmul*) prepared especially for this purpose. The selected method of regression analysis was linear regression based on the classical method of least squares.

According to the assumptions of linear regression, a dependency between the observed values of dependent variable Y and the values of independent variables $X1$, $X2$, ..., and Xp is expressed by the following equation:

$$Y_i = a_0 + a_1 X1_i + a_2 X2_i + ... + a_p Xp_i + \varepsilon_i = Yt_i + \varepsilon_i \tag{1}$$

where:

i is the identifier of observations ($i = 1, \dots, n$),

$a_0, \dots, a_p$ are unknown parameters; the values of these parameters are estimated by means of the classical method of least squares,

$X1_i, \dots, Xp_i$ are known values of independent variables $X1$, $X2$, ..., Xp, corresponding to the value of variable Y for the i^{th} observation,

Y_i is the value of dependent variable Y for the i^{th} observation,

Yt_i is the theoretical (estimated) value of dependent variable Y for the i^{th} observation,

ε_i is the statistical error (disturbance, noise) for the i^{th} observation.

Equation (1) is typically applied when there is a linear dependency between the dependent variable and independent variables. However, regression analysis assumes that equation (1) may also be applied if there is a nonlinear yet linearly transformable (by use of appropriate transformations, e.g. logarithms) dependency between the dependent variable and independent variables. Such nonlinear yet linearly transformable dependencies are: power, exponential, logarithmic, or hyperbolic. As the actual type of dependency between independent variables $X1$, $X2$, $X3$, $X4$ and dependent variable Yt is unknown a priori, a general model (which is a linear regression model derived by means of the classical method of least squares) could take one of the following forms:

- A linear form, expressed by the following equation:
$$Yt = a_1 \times X1 + a_2 \times X2 + a_3 \times X3 + a_4 \times X4 \tag{2}$$
- A power form, presented by the following equation:
$$Yt = X1^{a1} \times X2^{a2} \times X3^{a3} \times X4^{a4} \tag{3}$$
- An exponential form, expressed by the following equation:
$$Yt = a1^{X1} \times a2^{X2} \times a3^{X3} \times a4^{X4} \tag{4}$$
- A logarithmic form, presented by the following equation:
$$Yt = a1 \times \log X1 + a2 \times \log X2 + a3 \times logX3 + a4 \times logX4 \tag{5}$$
- A hyperbolic form, expressed by the following equation:

$$Yt = a1 \times \frac{1}{X1} + a2 \times \frac{1}{X2} + a3 \times \frac{1}{X3} + a4 \times \frac{1}{X4} \tag{6}$$

Note: Parameter $a0$ is not taken into account in equations (2) $\div$ (6) because it has no practical sense for the modeled phenomenon.

To determine the ultimate form of a general model, we have used:

- coefficient of determination R^2 (in order to determine the character of a dependency between the dependent variable and particular independent variables of a model),
- adjusted R^2 (in order to form an ultimate list of independent variables of a model).

Taking into account the nature of variables $X1$, $X2$, $X3$, $X4$, Yt and their mutual relations, we could assume that a dependency between all of these variables takes the power form expressed by equation (3).

This assumption has been verified by examining the value of the coefficient of determination (R^2) calculated for:

- variable Yt and all of the independent variables considered altogether (case 1/),
- variable Yt and a particular independent variable considered individually (cases 2/ $\div$ 5/).

The values of the coefficient of determination obtained for both pattern programs are presented in Tables 3 and 4. For case 1/ and both programs, the greatest value of R^2 (which indicates the best-fitted model of all of the considered models) has been obtained for power model (3). Moreover, for both programs, the power model is very well-fitted for cases 2/ $\div$ 5/ as well. According to the rules of regression analysis, this proves that a power function is best fitted to analytically describe a dependency between the dependent variable and each of the considered potential independent variables of the model.

Table 3

Values of the coefficient of determination for various possible forms of the general model – for the *nonInterf* program.

Form of the model	1/ $R^2_{Yt.X1,X2,X3,X4}$	2/ $R^2_{Yt.X1}$	3/ $R^2_{Yt.X2}$	4/ $R^2_{Yt.X3}$	5/ $R^2_{Yt.X4}$
linear	0.9738484	0.0602237	0.9239709	0.6125842	0.6390095
power	**0.9999580**	**0.8968516**	**0.9957804**	**0.9653380**	**0.9203893**
exponential	0.9845407	0.3399016	0.7284349	0.8848293	0.9194891
logarithmic	0.9557310	0.4977595	0.7366662	0.6611192	0.6387514
hyperbolic	0.9458243	0.9239709	0.0602237	0.5872795	0.5997874

The carried-out analysis indicates that variables $X1$, $X2$, $X3$, and $X4$ significantly influence program execution time – hence, they were defined as the potential independent variables of a general model. Whether all or only some of these variables should be chosen, we can decide only after analysis of the empirical data. To make such a decision, we have calculated the value of the adjusted coefficient of determination for power model (3) and all possible combinations of the potential independent variables (i.e., $X1$, $X2$, $X3$, $X4$).

Table 4
Values of the coefficient of determination for various possible forms of the general model – for the *matmul* program.

Form of the model	1/ $R^2_{Yt.X1,X2,X3,X4}$	2/ $R^2_{Yt.X1}$	3/ $R^2_{Yt.X2}$	4/ $R^2_{Yt.X3}$	5/ $R^2_{Yt.X4}$
linear	0.9506216	0.0002301	0.9286567	0.3616036	0.4771490
power	**0.9999514**	**0.6540767**	**0.9982205**	**0.9119271**	**0.9183616**
exponential	0.9645971	0.1066810	0.5703056	0.4310208	0.9170599
logarithmic	0.8230448	0.8095558	0.5858303	0.5074892	0.4774223
hyperbolic	0.8098927	0.7669016	0.0014395	0.3219836	0.4602693

If the degree to which a model explains the changes of values of a dependent variable indeed increases once a particular independent variable is added to the model, then the value of the adjusted coefficient of determination is greater than that for the case when an independent variable in question is not included in the model. Therefore, in a regression model, we should include those variables (as independent ones) whose combination exposes the greatest value of the adjusted coefficient of determination.

The values of the adjusted coefficient of determination obtained for both pattern programs are presented in Tables 5 and 6. For both pattern programs, the greatest value of adjusted R^2 has been obtained when we take into account all of the potential independent variables in power model (3); i.e., variables: $X1$, $X2$, $X3$, and $X4$.

Based on the obtained values of R^2 and adjusted R^2, we have adopted the following general model:

$$Yt = X1^{a1} \times X2^{a2} \times X3^{a3} \times X4^{a4} \tag{7}$$

where:

Yt is the estimated CPU time for the execution of the program loop nest by all program threads, expressed by the number of CPU clock cycles,

$X1$ states for a value expressing the relationship between the total size of cache L1 and L2 per single OpenMP thread and data footprint per single OpenMP thread,

$X2$ is the total weighted number of arithmetic operations per single OpenMP thread,

$X3$ is the maximum number of iterations in a single chunk of iterations assigned to be executed by an OpenMP thread for a given assignment of iterations to OpenMP threads,

$X4$ is the number of OpenMP threads executing the program,

$a1$, $a2$, $a3$, and $a4$ are parameters whose values are determined by means of regression analysis on the empirical data collected in a target software-hardware environment for a specially prepared sample.

Table 5

Values of the adjusted coefficient of determination for various possible combinations of potential independent variables – for the *nonInterf* program and power model (3).

Variables of the model				R^2	Adjusted R^2
$X1$				0.8968516	0.8919398
	$X2$			0.9957804	0.9955795
		$X3$		0.9653380	0.9636874
			$X4$	0.9203893	0.9165983
$X1$	$X2$			0.9982909	0.9981200
$X1$		$X3$		0.9663456	0.9629802
$X1$			$X4$	0.9300614	0.9230676
	$X2$	$X3$		0.9966362	0.9962998
	$X2$		$X4$	0.9959525	0.9955477
		$X3$	$X4$	0.9702187	0.9672406
$X1$	$X2$	$X3$		0.9983220	0.9980570
$X1$	$X2$		$X4$	0.9999401	0.9999306
$X1$		$X3$	$X4$	0.9747436	0.9707558
	$X2$	$X3$	$X4$	0.9970075	0.9965350
$\boldsymbol{X1}$	$\boldsymbol{X2}$	$\boldsymbol{X3}$	$\boldsymbol{X4}$	**0.9999580**	**0.9999486**

Table 6

Values of the adjusted coefficient of determination for various possible combinations of potential independent variables – for the *matmul* program and power model (3).

Variables of the model				R^2	Adjusted R^2
$X1$				0.6540767	0.6458404
	$X2$			0.9982205	0.9981782
		$X3$		0.9119271	0.9098301
			$X4$	0.9183616	0.9164178
$X1$	$X2$			0.9994628	0.9994366
$X1$		$X3$		0.9383487	0.9353413
$X1$			$X4$	0.9655228	0.9638410
	$X2$	$X3$		0.9982451	0.9981595
	$X2$		$X4$	0.9982402	0.9981543
		$X3$	$X4$	0.9549970	0.9528018
$X1$	$X2$	$X3$		0.9994630	0.9994227
$X1$	$X2$		$X4$	0.9999501	0.9999463
$X1$		$X3$	$X4$	0.9796433	0.9781166
	$X2$	$X3$	$X4$	0.9982646	0.9981345
$\boldsymbol{X1}$	$\boldsymbol{X2}$	$\boldsymbol{X3}$	$\boldsymbol{X4}$	**0.9999514**	**0.9999464**

4. Estimation of parameter values for specific models

Our goal is to determine the values of parameters for specific models for a computer environment and a pattern program in such a way that these models could also be valid for non-pattern programs executed in this computer environment. Therefore, we have decided to determine the values of parameters $a1$, $a2$, $a3$, and $a4$ for a given environment by means of the statistical analysis of the empirical data collected in this environment.

To determine the values of parameters $a1$, $a2$, $a3$, $a4$, we have used two pattern programs: *nonInterf* and *matmul*. Each of the pattern programs represents an arbitrarily assumed combination of data reuse and cache interference.

Taking into account data reuse and cache interference, programs can be classified as follows:

- programs with no data reuse – in practice, very rarely used and therefore not considered in an elaborated model
- programs with data reuse:
  - without cache interference – sample pattern program: *nonInterf*
  - with cache interference – sample pattern program: *matmul*

The source codes of the *nonInterf* and *matmul* programs are presented in Table 1.

Empirical data collected for a pattern program are the basis for determining the values of parameters $a1$, $a2$, $a3$, and $a4$ of a specific model referring to all such programs that represent the same combination of data reuse and cache interference as a pattern program. In this paper, a program, which is not a pattern program, but represents the same combination of data reuse and cache interference as a pattern program, is referred to as a non-pattern program.

It should be stressed here that pattern programs *nonInterf* and *matmul* are *exemplary* pattern programs with the characteristics presented in Table 1. These programs have been adopted simply in order to determine exemplary specific models on the basis of general model (7). This realization of the pattern programs (i.e., *nonInterf* and *matmul*) is one of *many possible* realizations. Assuming some other realization of pattern programs, one could derive specific models with domains different from the domains of specific models derived from pattern programs *nonInterf* and *matmul*. This, in turn, means that the proposed approach is highly universal, as it provides the possibility of changing the domain of a specific model simply by modifying a pattern program.

In view of the complexity of contemporary hardware, it is essential to define some limits for the empirical data collected in the hardware environment and used for determining the values of parameters $a1$, $a2$, $a3$, and $a4$ so that there are rules clearly stating what data are representative for an environment under analysis. For this purpose, it has been assumed that for each pattern program:

1. The total size of the data processed in a loop nest does not exceed the size of the L2 cache available for a single processor.

Assumption 1 is expressed by the following formula:

$$\lambda = \frac{total\_matrix\_size(N)}{L2\_per\_processor} \leq 1 \tag{8}$$

where:
total_matrix_size(N) is the total size *(in bytes)* of the data occupied by the array variables available in a loop nest, with the upper bounds of loop indices dependent on N,
L2_per_processor is the size *(in bytes)* of L2 cache memory available for a single processor.

2. The relative difference between the mean and maximum number of iteration chunks per single OpenMP thread for a given assignment of iterations to OpenMP threads does not exceed 50 % (the value assumed a priori).

   Assumption 2 is expressed by the following formula:

$$\theta = \frac{no\_chunks_{max} - no\_chunks_{average}}{no\_chunks_{average}} \leq 0.5 \tag{9}$$

where:
no_chunks_{max} is the maximum number of iteration chunks per single OpenMP thread for a given assignment of iterations to OpenMP threads,
$no\_chunks_{average}$ is the mean number of iteration chunks per single OpenMP thread for a given assignment of iterations to OpenMP threads.

We used the following environment to carry out all experiments discussed in this paper: processor: Intel Core 2 Quad Q6600; number of processor cores: 4; number of processor threads: 4; L1 data cache: 4 x 32 KB *(8-way set associative, 64-byte line size)*; L2 cache: 2 x 4096 KB (*16-way set associative, 64-byte line size)*; operating system: Linux Slax 6.1.2; compiler: gcc 4.2.4; version of OpenMP: 2.5; compilation level optimization: turned off; compilation with the option: -O0.

For assumptions 1 and 2, the exemplary pattern programs, and the computer system environment as above, we have derived the following specific models:

- for the *nonInterf* pattern program:
$$Yt = X1^{-0.325431} \times X2^{0.675172} \times X3^{-0.082602} \times X4^{0.981967} \tag{10}$$
- for the *matmul* pattern program:
$$Yt = X1^{-0.298695} \times X2^{0.623738} \times X3^{0.014426} \times X4^{0.962976} \tag{11}$$

A resultant regression model should not be extrapolated outside the data range for which the regression model has been constructed because the character of a dependency between the values of independent and dependent variables is unknown outside the data range in question.

To avoid the risk of such an extrapolation while applying specific models to non-pattern programs, we have formulated the following detailed assumptions regarding the scope of applicability of the specific models:

1. The value of λ , calculated for a non-pattern program by means of equation (8), should not exceed the minimum/maximum value of λ calculated for a corresponding pattern program. This assumption is expressed by the following inequalities:

$$\lambda_{min}(referenceLoop) \leq \lambda \leq \lambda_{max}(referenceLoop) \tag{12}$$

   where:
   λ holds the value of λ calculated for a non-pattern program,
   $\lambda_{min}(referenceLoop)$ represents the minimum value of λ for a corresponding pattern program,
   $\lambda_{max}(referenceLoop)$ states for the maximum value of λ for a corresponding pattern program.

2. The value of θ, calculated for a non-pattern program as per equation (9), cannot exceed 0.5.

3. The actual time of the execution of a non-pattern program in a target environment should be of the same order of magnitude as the time of the execution of a corresponding pattern program. This assumption is expressed by the following inequalities:

$$\gamma_{min}(referenceLoop) \leq \gamma \leq \gamma_{max}(referenceLoop) \tag{13}$$

   where:
   γ is the actual CPU time for the execution of a program by all program threads, expressed by the number of CPU clock cycles,
   $\gamma_{min}(referenceLoop)$ is the shortest actual CPU time for the execution of a program loop nest by all program threads, expressed by the number of CPU clock cycles,
   $\gamma_{max}(referenceLoop)$ is the longest actual CPU time for the execution of a program by all program threads, expressed by the number of CPU clock cycles.

The assumption expressed by inequalities (13) has been introduced because there can be such programs for which assumptions 1 and 2 are met, however, despite the similarity between these programs and the corresponding pattern programs in respect to data reuse and cache interference, the programs may differ so much from corresponding pattern programs in other respects as to have execution times of a completely different order of magnitude than that of corresponding pattern programs. This situation is not a problem, though, as by changing the number and type of arithmetic operations executed in pattern programs, one can easily change execution times of pattern programs and, consequently, tailor them to various orders of magnitude – so that they can be used as pattern programs for real-life programs with very different execution times.

5. Verification of the quality of estimations

Verification of the quality of estimations made according to the proposed general model is equivalent to the assessment of the quality of specific models derived from the general model.

The quality of specific models has been assessed in a qualitative aspect and a quantitative aspect.

The qualitative quality assessment of specific models has been focused on recognizing whether one can satisfactorily use estimated execution times obtained by applying specific models to non-pattern programs in order to select, from the considered source-code variants of a given program, a subset *certainly* containing the variant with the shortest actual execution time in a given target hardware environment.

In practice, this means the following: for a given size of a problem, one should check whether the trend of changes in measured execution times per program thread of particular variants of a given program matches the trend of changes in corresponding estimates per program thread calculated according to the elaborated models.

The quantitative quality assessment of the specific models in question has been focused on determining the estimation errors that one can expect to obtain while using the models. A relative estimation error has been calculated as follows:

$\delta_{Y(per\_thread)}$ is the relative estimation error for $Yt(per\_thread)$, calculated according to the following formulae:

$$\delta_{Y(per\_thread)} = \left| \frac{Yt(per\_thread) - \gamma(per\_thread)}{\gamma(per\_thread)} \right| \times 100\% \tag{14}$$

$$Yt(per\_thread) = \frac{Yt}{X4^{a4}} \tag{15}$$

$$\gamma(per\_thread) = \frac{\gamma}{X4^{a4}} \tag{16}$$

where:

Yt is the estimated CPU time for the execution of a program loop nest by all program threads, calculated according to a relevant specific model and expressed by the number of CPU clock cycles,

$Yt(per\_thread)$ is Yt per thread,

γ is the actual (i.e., empirically measured) CPU time spent on executing a program loop nest by all program threads, expressed by the number of CPU clock cycles,

$\gamma(per\_thread)$ is γ per thread,

$X4$ is the number of OpenMP threads executing the program,

$a4$ is parameter $a4$ of a relevant specific model.

It should be stressed here that because the main goal of the model application is iterative compilation, the qualitative quality assessment and its results are much more important than the quantitative quality assessment and its results. Within the trend-matching verification carried out in the qualitative quality assessment, various source-code variants of a given program are sorted in descending order by their estimated execution times per program thread as calculated according to the model. The resultant sequence of source-code variants is then compared with a sequence of the same source-code variants yet sorted in descending order by their measured execution times per program thread. The trend-matching verification allows us to find out whether, by applying only a model, one can select – from all of the considered source-code variants of a given program – a small subset of source-code variants where in the source-code variant with the minimal actual execution time in a hardware environment is for certain included. Then, iterative compilation is carried out only for the source-code variants from the selected subset. Therefore, if it is indeed possible to select the subset in question, the estimation errors obtained within the quantitative quality assessment are of minor importance.

6. Application of specific models in iterative compilation

The objective of iterative compilation is to find, among the semantically equivalent source-code variants of a given program, the variant with the shortest execution time in a target environment. Typically, iterative compilation is carried out as follows. In a target hardware environment, one executes the considered, semantically equivalent source-code variants of a given program, registers their measured execution times per program thread, and selects the source-code variant with the shortest-measured execution time per program thread for final use. This means that, if t semantically equivalent source-code variants of a given program are under consideration, all of these source-code variants have to be executed in the target environment to find within iterative compilation the variant intended for final use. The time cost of such iterative compilation is equal to the total time of execution of the t source-code variants of the program in the target environment.

Our goal is to obtain the same result (i.e., source-code variant) as the one obtained within the typical iterative compilation described above, but with a lower time cost involved in comparison to the typical iterative compilation. We have decided to achieve this goal by decreasing the number of source-code variants to be executed in the target environment from t to k ($k < t$) in order to find the variant with the minimal execution time. As nothing is known beforehand about the actual execution times of the considered t source-code variants in the target environment, we have decided to limit an empirical search to those source-code variants with the k shortest estimated execution times as per our respective specific model. The fundamental problem here is what value of k guarantees that the source-code variant with the shortest execution time in the target environment is selected for final use.

Let k_{min} be the minimum value of k.

Let an "empirical code sequence" and an "empirical time sequence" be, respectively, a sequence of semantically equivalent source-code variants of a given program, sorted in descending order by their measured execution times per program thread and a sequence of the corresponding measured execution times. Knowing the empirical code sequence for a given program is sufficient for finding its source-code variant with the shortest execution time in the target environment. By the very definition of an empirical code sequence, such a source-code variant is its last element. The definition also implies that it is necessary to carry out iterative compilation in order to form an empirical code sequence.

Let a "theoretical code sequence" and a "theoretical time sequence" be, respectively, a sequence of semantically equivalent source-code variants of a given program, sorted in descending order by their estimated execution times per program thread calculated in accordance with our relevant specific model and a sequence of the corresponding estimated execution times. To form a theoretical code sequence, we apply a relevant specific model.

Because of the specificity of regression analysis, the estimates obtained by using our specific models differ from the real values measured in the target environment – hence, the theoretical time sequence differs from the empirical time sequence. This implies that for the considered source-code variants of the program under analysis, the empirical code sequence may be different from the theoretical code sequence.

However, in view of the assumed sorting criteria, both the empirical time sequence and theoretical time sequence for a given program are monotonically decreasing. Taking this fact as well as the relationships between the discussed time and code sequences into account, the value of k_{min} for a given program can be determined as follows. Let E_{last} be the last code variant in an empirical code sequence. In the corresponding theoretical code sequence, we have to find such code variant s that $s = E_{last}$. Then, the position of s in the theoretical code sequence taken in reverse order defines the value of k_{min}.

The above-proposed way of supporting iterative compilation by using specific models is illustrated by the example of the UA_diffuse_3 benchmark from the NAS Parallel Benchmarks (NPB) suite [11, 16] (upper bounds of loop indices are dependent on parameter N; in the example, $N = 30$).

The source code of the parallel UA_diffuse_3 loop nest is presented in Table 7.

Adopting different values of variables NUM_THREADS and CHUNK_SIZE, one can obtain various (but semantically equivalent) variants of the source code of the UA_diffuse_3 benchmark. The values of variables NUM_THREADS and CHUNK_SIZE should be selected so that the execution time of a given program is as short as possible.

We have created nine different (yet semantically equivalent) source-code variants for the UA_diffuse_3 benchmark, by adopting various values of variables NUM_THREADS and CHUNK_SIZE for particular source-code variants, as presented in Table 8.

For each of these nine source-code variants, we **estimated** execution time per thread in the target environment as specified in Section 4. To make estimations, we

Table 7

Source code of the parallel UA_diffuse_3 loop nest.

```
int N = 30;
int NUM_THREADS = ?;
int CHUNK_SIZE = ?;          //Possible values:
                             • value defined by the developer,
                             • default value (equal to N / NUM_THREADS),
    resulting from the specificity of the OpenMP standard.
int tm1[N][N][N], u[N][N][N], wdtdr[N][N];
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for private(iz, k, j, i) schedule(static, CHUNK_SIZE)
for (iz = 0; iz < N; iz++) {
    for (k = 0; k < N; k++) {
        for (j = 0; j < N; j++) {
            for (i = 0; i < N; i++) {
                tm1[iz][j][i] = tm1[iz][j][i] + wdtdr[k][i]*u[iz][j][k];
            }
        }
    }
}
```

Table 8

Semantically equivalent source-code variants of the UA_diffuse_3 program.

source-code variant	NUM_THREADS	CHUNK_SIZE
1	2	default
2	2	5
3	2	3
4	3	3
5	3	default
6	3	5
7	4	5
8	4	3
9	4	default

used the specific model derived from a pattern program representing the same combination of data reuse and cache interference as in the UA_diffuse_3 benchmark. Next, the nine source-code variants were sorted in descending order by their aforementioned estimated execution times, thus forming a theoretical code sequence (see Figure 1).

Then, in the target environment as specified in Section 4, we executed all nine source-code variants and registered their execution times per thread as measured empirically. The source-code variants were sorted in descending order by their aforementioned measured execution times, thus forming an empirical code sequence (see Figure 1).

Next, we used the obtained theoretical code sequence and empirical code sequence to determine the value of k_{min} for the considered example. In the considered example,

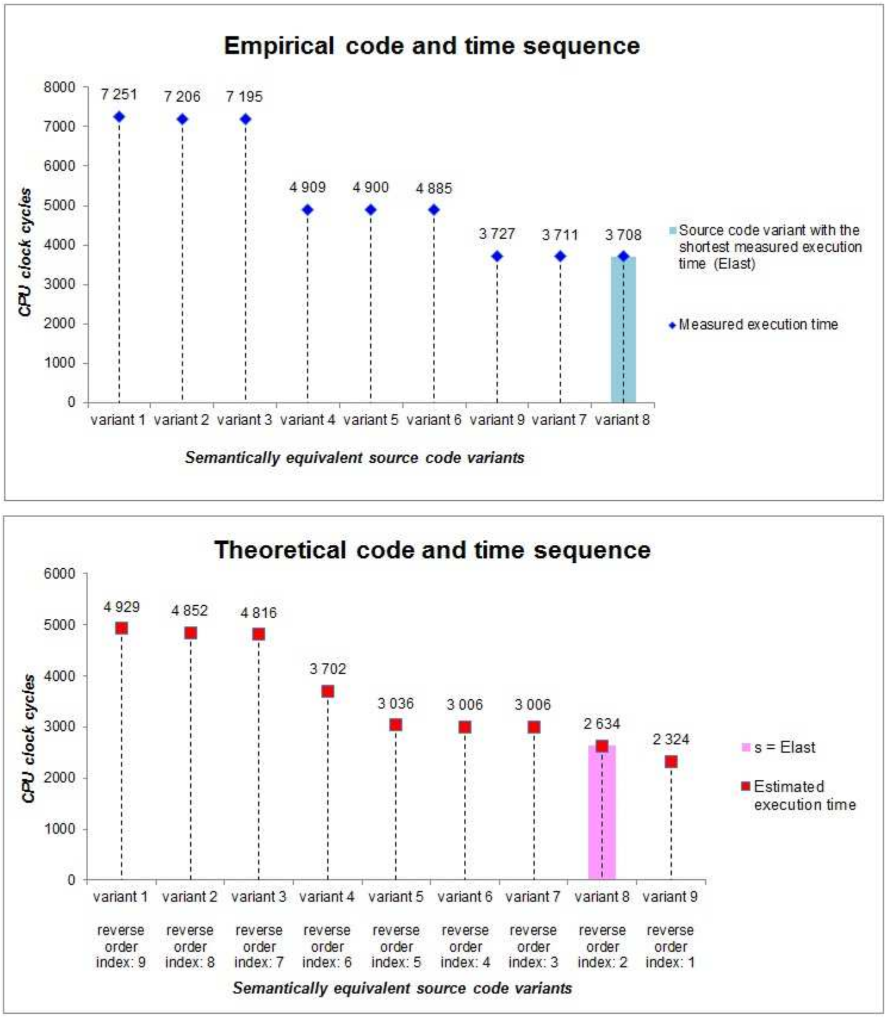

Figure 1. Empirical and theoretical code and time sequences for the UA_diffuse_3 ($N = 30$) program.

the number of source-code variants is 9; i.e., $t = 9$. As shown in Figure 1, the E_{last} code variant from the empirical code sequence is variant 8; i.e., E_{last} = variant 8. Compared with the theoretical code sequence, the E_{last} code variant is equivalent to the eighth element of the theoretical code sequence. Hence, s = variant 8. When the theoretical code sequence is sorted in reverse order, the position of s in the reversed sequence is 2. This means that, for the considered example, $k_{min} = 2$.

Figure 2 presents the time of iterative compilation for various assumed values of k. For $k = k_{min} = 2$, the time of iterative compilation is about 7 500 CPU clock cycles. For $k = t = 9$, the time in question is about 47 500; i.e., more than six-times longer.

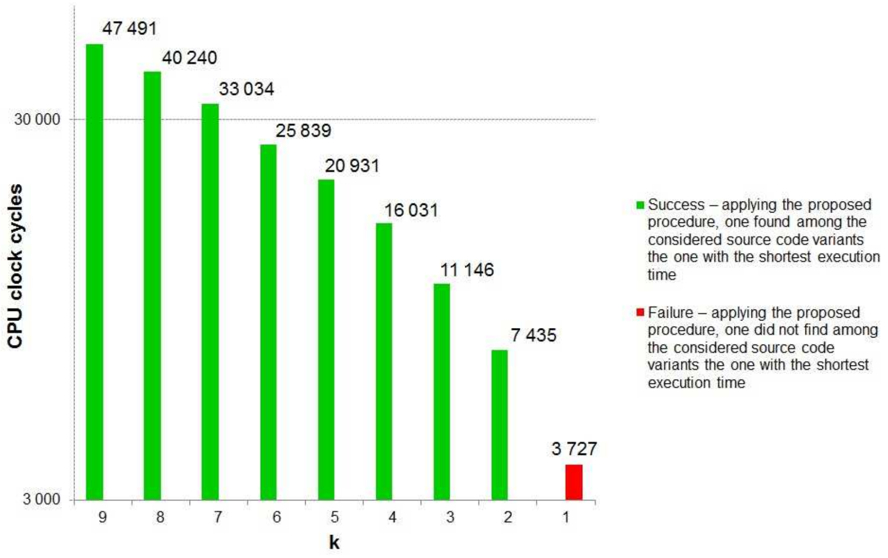

Figure 2. Time of the iterative compilation of the UA_diffuse_3 ($N = 30$) program carried out in accordance with the proposed procedure, for various possible values of k.

Generalizing the above-presented example, and assuming that k is such that $k_{min} \leq k < t$, we have derived the following procedure of how to apply iterative compilation supported by our models to find among t semantically equivalent source-code variants the one with the shortest execution time in a given environment:

1. Based on the data reuse and cache interference criteria (see Sections 2 $\div$ 4), select the specific model applicable to the provided input source code.
2. Generate/provide t semantically equivalent source-code variants of the provided input source code.
3. For each of the t semantically equivalent source-code variants, estimate the execution time per program thread according to the selected specific model.
4. Form the theoretical code sequence by sorting the t semantically equivalent source-code variants in descending order by their estimated execution times per program thread, calculated in accordance with the selected specific model.
5. Execute in the target environment the last k source codes from the theoretical code sequence, and select for final use the one with the shortest-measured execution time per program thread.

7. Results of experiments

In order to demonstrate that the obtained models are indeed useful in iterative compilation, we have used the NAS Parallel Benchmarks (NPB) suite [11, 16]. NPB is a test suite dedicated to the performance assessment of parallel computers and consists of a great number of very various loop nests.

Ten NPB programs were selected for our experiments. The selected benchmarks are different from the pattern programs, however, according to the analysis of the source codes of the benchmarks (which we carried out according to the technique presented in [20] and [14]), the benchmarks represent the same combination of data reuse and cache interference as the pattern programs. By means of the exemplary specific models, we estimated execution times for various source-code variants of the ten selected programs (from 6 to 9 for each benchmark). In total, we estimated the execution times for 241 various source codes.

For each of the selected NPB benchmarks, the trend of changes in the measured execution times per program thread of particular variants of a given loop nest is matched by the trend of changes in the corresponding estimations per program thread calculated according to a relevant specific model.

The mean and maximum relative estimation errors calculated in relation to execution times measured empirically for all source-code variants adopted for a given loop nest and the size of a problem (which we comprehend as the product of differences between upper and lower bounds of particular loop indices) do not exceed 55 and 65 percentage points (detailed results are presented in Tables 9 and 10, respectively).

For each of the selected NPB benchmarks, we have also estimated the reduction of iterative compilation time that could be achieved by applying specific models in accordance with our procedure on how to support iterative compilation with such models. The meaning of the variable names used is as follows:

t is the number of all various input source-code variants for a given loop nest,

k $(0 < k \leq t)$ is the assumed number of source-code variants with the shortest estimated execution times for a given loop,

k_{min} represents the minimum value of k that guarantees that one selects for final use the source-code variant with the shortest execution time measured in a target environment.

For each of the selected NPB benchmarks, k_{min} was determined as described in Section 6.

The achieved results are presented in Tables 11 and 12.

Summing up the results presented in Tables 11 and 12 for the selected NPB loop nests, we have obtained the following reduction of iterative compilation time by decreasing the number of iterations from t to k_{min}:

- minimum reduction: 3.53 times
- maximum reduction: 12.82 times.

Table 9

Quality assessment of estimates calculated according to specific model (10).

Loop nest	Size of a problem, S	Number of various source-code variants subjected to the estimation of execution time	Resultant mean for $\delta_{Y\,t(per\_thread)}$ [%]	Resultant maximum for $\delta_{Y\,t(per\_thread)}$ [%]
CG3	75,000	8	14.27	24.97
CG3	118,000	8	13.73	24.01
CG3	160,000	8	12.48	21.46
CG4	100,000	8	10.48	20.43
CG4	215,000	8	11.66	24.43
CG4	330,000	8	13.66	27.06
FT2	27,000	8	53.34	60.43
FT2	54,872	9	51.54	60.75
FT2	91,125	8	53.05	60.43
L11	39,800	8	16.42	32.04
L11	69,960	8	16.47	32.85
L11	108,570	8	14.84	28.76
MG3	26,000	6	25.96	34.41
MG3	57,444	6	29.21	38.35
MG3	88,888	6	31.04	40.51
UA2	80,000	6	6.80	15.29
UA2	173,333	6	5.12	13.61
UA2	266,666	6	6.44	17.01

where the loop nests are denoted as follows:
CG3 – CG_cg_3
CG4 – CG_cg_4
FT2 – FT_auxfnct_2
L11 – LU_HP_pintgr_11
MG3 – MG_mg_3
UA2 – UA_diffuse_2

The empirical results presented in Tables 11 and 12 also indicate that, for the selected NPB loop nests, it is quite safe to assume without actually finding k_{min} that, if $t \geq 6$, then $k_{min} \leq \text{floor}(t/2)$. In our future research, we plan to formulate more general conclusions regarding the upper limits for the value of k_{min}.

The experimental research has been focused on demonstrating the usefulness of our proposed procedure of supporting iterative compilation with specific models when applied to small benchmark codes. The achieved, positive results indicate that it is worth examining whether the proposed procedure is also useful for real-life programs. Tailoring pattern programs for real-life programs and verifying the usefulness of our proposed procedure for real-life programs are the intended directions of our future research.

Table 10

Quality assessment of estimates calculated according to specific model (11).

Loop nest	Size of a problem, S	Number of various source-code variants subjected to the estimation of execution time	Resultant mean for $\delta_{Yt(per\_thread)}$ [%]	Resultant maximum for $\delta_{Yt(per\_thread)}$ [%]
UA3	810,000	9	31.60	38.46
UA3	6,250,000	9	16.88	24.02
UA3	25,411,681	9	27.84	49.59
UA4	810,000	9	28.55	35.80
UA4	6,250,000	9	12.70	20.18
UA4	25,411,681	9	26.49	45.39
U11	980,000	9	10.49	20.44
U11	18,891,585	9	11.30	29.81
U11	80,807,759	9	14.86	37.89
U16	970,200	9	12.27	23.60
U16	18,820,830	9	11.10	29.05
U16	80,621,136	9	14.86	34.97

where the loop nests are denoted as follows:
UA3 – UA_diffuse_3
UA4 – UA_diffuse_4
U11 – UA_transfer_11
U16 – UA_transfer_16

8. Related work

Optimizing compilation, iterative compilation, and program execution-time estimation are objects of scientific research carried out in many centers. Various solutions have been proposed: methods for forecasting program execution time [4], estimating program execution time [5, 9, 15], optimizing program execution time [2, 3] or selecting the program source code with the shortest-expected execution time [18].

A method for elaborating models intended for forecasting execution times of particular parallel and distributed programs is presented in [4]. The proposed method is based on linear regression. It assumes that a dedicated model for forecasting program execution time should be formed for each program in a target computer environment. Models elaborated in such a way are very well-fitted to the empirical data and, as such, are a valuable tool for forecasting program execution time in the considered domains of independent variables. However, elaborating a model in accordance with the proposed method is time consuming (for each program, one has to elaborate a separate model).

A proposal of estimating program execution time by using atoms (i.e., elementary components of a program source code) is presented in [5]. Atoms are equivalent to terminal symbols of the grammar of a given programming language. The estimated

Table 11

Reduction of iterative compilation time after applying specific model (10).

Loop nest	Size of a problem, S	t	k_{min}	Iterative compilation time (T) for t source-code variants	Iterative compilation time (T) for k_{min} source-code variants	Reduction of iterative compilation time = (Tt / Tk_{min})
CG3	75,000	8	2	1,957.38	341.16	5.74
CG3	118,000	8	2	3,007.58	516.88	5.82
CG3	160,000	8	2	4,086.69	698.61	5.85
CG4	100,000	8	2	2,053.32	354.45	5.79
CG4	215,000	8	1	4,334.33	369.65	11.73
CG4	330,000	8	1	6,630.62	560.69	11.83
FT2	27,000	8	1	1,970.33	170.83	11.53
FT2	54,872	9	3	4,262.63	1,115.40	3.82
FT2	91,125	8	3	6,449.17	1,827.18	3.53
L11	39,800	8	2	2,411.02	415.78	5.80
L11	69,960	8	1	4,189.57	357.66	11.71
L11	108,570	8	1	6,463.89	547.95	11.80
MG3	26,000	6	1	1,415.69	168.43	8.41
MG3	57,444	6	2	3,077.12	723.66	4.25
MG3	88,888	6	2	4,721.80	1,105.69	4.27
UA2	80,000	6	1	1,507.84	178.29	8.46
UA2	173,333	6	2	3,209.56	753.97	4.26
UA2	266,666	6	2	4,952.80	1,160.27	4.27

where the loop nests are denoted as follows:
CG3 – CG_cg_3
CG4 – CG_cg_4
FT2 – FT_auxfnct_2
L11 – LU_HP_pintgr_11
MG3 – MG_mg_3
UA2 – UA_diffuse_2

time of program execution depends on: the number of occurrences of particular types of atoms Ai; data sets Dj on which atoms operate and statistical deviation δ. The value of δ depends, in turn, on various complex factors (e.g., a source-code structure, characteristics of a compiler, a processor architecture, etc.). This approach does not take into account how memory hierarchy influences program execution time; moreover, the assumed way of estimating the value of δ is too simplified.

A method for estimating and minimizing the worst-case execution time (WCET) of a program is presented in [15]. The method is intended for use in optimizing compilers, and the main purpose of its application is to assess whether applying a given compiler level optimization results in a shorter execution time of a program as compared to that obtained without applying an optimization in question. It has been assumed that a separate, dedicated model should be created for each possible optimization. The proposed method focuses on the estimation of the program execution time for

Table 12

Reduction of iterative compilation time after applying specific model (11).

Loop nest	Size of a problem, S	t	k_{min}	Iterative compilation time (T) for t source-code variants, $x10^3$	Iterative compilation time (T) for k_{min} source-code variants, $x10^3$	Reduction of iterative compilation time = (Tt / Tk_{min})
UA3	810,000	9	2	47.49	7.43	6.39
UA3	6,250,000	9	1	367.22	28.66	12.81
UA3	25,411,681	9	3	1,553.45	440.50	3.53
UA4	810,000	9	1	45.46	3.55	12.80
UA4	6,250,000	9	1	349.38	27.29	12.80
UA4	25,411,681	9	3	1,436.87	406.68	3.53
U11	980,000	9	1	39.44	3.08	12.82
U11	18,891,585	9	1	754.40	58.90	12.81
U11	80,807,759	9	1	3,322.82	259.52	12.80
U16	970,200	9	2	39.22	6.14	6.39
U16	18,820,830	9	1	753.42	58.80	12.81
U16	80,621,136	9	1	3,219.80	251.49	12.80

where the loop nests are denoted as follows:
UA3 – UA_diffuse_3
UA4 – UA_diffuse_4
U11 – UA_transfer_11
U16 – UA_transfer_16

a given compiler level optimization and involves carrying out a time-consuming learning process separately for each optimization.

A tournament predictor is presented in [18]. It is a model that for given input data – performance characteristics of a program and two different sequences of compiler level optimizations – indicates a sequence of optimizations, which once applied, results in a shorter program execution time as compared to the other sequence. The independent variables of the model proposed in [18] are dynamic characteristics of the program (i.e., they are collected and calculated at run time) – in practice, this means that program profiling has to be carried out whenever the model is to be used for a new program.

A random search-strategy algorithm is proposed in [9]. By applying this algorithm, it is possible to reduce the time of iterative compilation. The algorithm makes use of a method for finding the minimum execution time of a program. A method in question lets one determine what program execution time is if no cache misses occurred during the execution of the program. However, it is not guaranteed that applying the random search-strategy algorithm during iterative compilation of a given program will help find such a source-code variant of the program whose execution time will be approximately equal to the minimum execution time of this program.

The idea of optimizing source codes of programs by applying iterative compilation with kernel decomposition is presented in [3] and [2]. The proposed approach comprises several steps; it separates the optimization of memory use (focused on data locality) from the optimization of processor operations (focused on instruction level parallelism). However, for the approach presented in [3] and [2] to be effective in practice, it is essential to use a good compiler, i.e. such a compiler that is capable of generating kernels with very good performance characteristics.

In view of the above-discussed limitations of the approaches presented in [2, 3, 4, 5, 9, 15], and [18], the approaches in question are not adequate for carrying out the proposed improvement of iterative compilation, which involves an analytical selection from semantically equivalent source-code variants of a given program the ones with several shortest expected execution times in order to limit the empirical selection of the best source code thereto.

The solution we present in this paper is free from the limitations spoken about in the aforementioned approaches. Applying this solution, it is possible to quickly elaborate models for the estimation of program execution time that are adequate for both pattern programs for which models have been derived and for completely different (non-pattern) programs that have only the presence of data reuse and cache interference in common with a corresponding pattern program. Therefore, our solution is adequate for carrying out the proposed improvement of iterative compilation.

9. Conclusion

This paper presents the authors' family of statistical models for the estimation of program execution time. The family consists of a general model as well as specific models. The family has been elaborated based on the empirical data collected for pattern-program loops representing some arbitrarily selected features related to the program structure and the specificity of a program-execution environment.

Exemplary specific models belonging to the family have been used to estimate execution times of non-pattern programs. The accuracy of estimations is satisfactory.

We have also estimated the reduction of iterative compilation time (and, as a consequence, the related software development time) that could be achieved by applying the proposed procedure of supporting iterative compilation with our specific models.

For this purpose, we have applied the proposed procedure to altogether 241 source-code variants of altogether 10 different programs coming from the NPB benchmark suite. As a result thereof, the time of iterative compilation for the particular programs has been reduced from approximately 3 to 13 times (detailed results are presented in Tables 11 and 12).

The large number of programs and source-code variants used in our experimental research indicates that the achieved, positive results cannot be regarded as accidental.

The achieved results show that the authors' solution presented in the paper is adequate for use in iterative compilation for optimization purposes and, at the same time, gives the possibility of reducing the time of software development.

References

[1] Aho A., Lam M., Sethi R., Ullman J.: *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2 ed., 2006.

[2] Barthou D., Donadio S., Carribault P., Duchateau A., Jalby W.: Loop optimization using hierarchical compilation and kernel decomposition. *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 170–184, 2007.

[3] Barthou D., Donadio S., Duchateau A., Jalby W., Courtois E.: Iterative compilation with kernel exploration. *Languages and Compilers for Parallel Computing*, pp. 173–189, 2007.

[4] Berlińska J.: *Methods of creating statistical models characterizing parallel and distributed applications (in Polish)*. Politechnika Szczecińska, 2005.

[5] Brandolese C., Fornaciari W., Salice F., Sciuto D.: Source-level execution time estimation of C programs. *Proceedings of the ninth international symposium on Hardware/software codesign*, pp. 98–103, 2001.

[6] Coleman S., McKinley K.: Tile Size Selection Using Cache Organization and Data Layout. *ACM SIGPLAN Notices*, vol. 30, pp. 279–290, 1995.

[7] Eiben A.E., Michalewicz Z., Schoenauer M., Smith J.E.: Parameter control in evolutionary algorithms. *Parameter setting in evolutionary algorithms*, pp. 19–46, 2007.

[8] Esseghir K.: *Improving data locality for caches*. Rice University, 1993.

[9] Fursin G.: *Iterative Compilation and Performance Prediction for Numerical Applications*. University of Edinburg, 2004.

[10] Fursin G., O'Boyle M., Knijnenburg P.: Evaluating iterative compilation. *Languages and Compilers for Parallel Computing*, pp. 362–376, 2005.

[11] Haoqiang J., Frumkin M., Yan J.: *The OpenMP implementation of NAS parallel benchmarks and its performance*. NASA Ames Research Center, 1999.

[12] Ishizaka K., Obata M., Kasahara H.: Coarse grain task parallel processing with cache optimization on shared memory multiprocessor. *Languages and Compilers for Parallel Computing*, pp. 352–365, 2003.

[13] Knijnenburg P., Kisuki T., O'Boyle M.: Iterative compilation. *Embedded processor design challenges*, pp. 171–187, 2002.

[14] Lam M., Rothberg E., Wolf M.: The Cache Performance and Optimization of Blocked Algorithms. *ACM SIGARCH Computer Architecture News*, vol. 19, pp. 63–74, 1991.

[15] Lokuciejewski P., Stolpe M., Morik K., Marwedel P.: Automatic Selection of Machine Learning Models for WCET-aware Compiler Heuristic Generation. *Proceedings of the 4th Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART)*, pp. 3–17, 2010.

[16] NASA Advanced Supercomputing Division: *NAS Parallel Benchmarks*. `http://www.nas.nasa.gov/publications/npb.html` Accessed 28 July 2015.

[17] OpenMP: *The OpenMP API specification for parallel programming.* `http://www.openmp.org/`. Accessed 28 July 2015.

[18] Park E., Kulkarni S., Cavazos J.: An Evaluation of Different Modelling Techniques for Iterative Compilation. *Proceedings of the 14th international conference on compilers, architectures and synthesis for embedded systems*, pp. 65–74, 2011.

[19] Temam O., Fricker C., Jalby W.: Cache interference phenomena. *ACM SIGMETRICS Performance Evaluation Review*, vol. 22, pp. 261–271, 1994.

[20] Wolfe M.: *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

Affiliations

Agnieszka Kamińska
West Pomeranian University of Technology, Faculty of Computer Science and Information Technology, Szczecin, Poland, `agnieszka_kaminska@wp.pl`

Włodzimierz Bielecki
West Pomeranian University of Technology, Faculty of Computer Science and Information Technology, Szczecin, Poland, `wbielecki@wi.zut.edu.pl`

Received: 25.11.2015
Revised: 5.05.2016
Accepted: 5.05.2016