Journal of Statistics Applications & Probability
An International Journal

http://dx.doi.org/10.18576/jsap/050101

Application of Statistical Models to Select Tile Size Minimizing the Execution Time of Parallelized Tiled Loop Nests

Agnieszka Kamińska[*] *and Włodzimierz Bielecki*

Faculty of Computer Science and Information Technology, West Pomeranian University of Technology, ul. Żołnierska 49, 71-210 Szczecin, Poland

Received: 14 Oct. 2015, Revised: 14 Dec. 2015, Accepted: 16 Dec. 2015
Published online: 1 Mar. 2016

Abstract: The reduction of software development time is an important practical problem to be dealt with by contemporary computer science. Resolving this problem is an object of research carried out both in scientific and industrial centres. One of the main areas of this research is compilation. Within compilation, a computer program written in a programming language comprehensible for a man is converted into an executable form comprehensible for a computer. Applying appropriately selected transformations (tiling also known as blocking) during compilation, one can transform a program, written in a given programming language and for a given hardware platform, to various yet semantically equivalent executables which however differ in execution times.
The paper presents a statistical model which allows for selecting from semantically equivalent, tiled source code variants of a given program the variants with best anticipated execution times. The paper also demonstrates how the elaborated model can be applied in iterative compilation for shortening software development time.

Keywords: program execution time, tiling (blocking), iterative compilation, statistical models

1 Introduction

Minimization of data processing time and reduction of software development time are important practical problems to be dealt with by contemporary computer science. These problems are of particular importance in practical applications of computers - as, irrespective of the application area, computer users expect the machines to solve problems and execute tasks as quickly as possible and at the lowest possible cost. In order to meet these expectations, many different concepts regarding the operation of computer hardware and software have been elaborated and implemented.

Particular attention should be paid here to parallel computing and tiling (also known as blocking).

The essence of parallel computing is division of the task to be executed into smaller tasks which are then executed simultaneously (or, in other words, in parallel); this in turn results in shortening of the time of execution of the initial task. In view of the multitude and variety of modern programming languages and software development methodologies, there is great diversity in the structure of modern computer programs. Despite this diversity, most of time consuming operations - i.e. calculations made within programs - is executed in program loops, so as sequences of operations repeated until the stop condition has been fulfilled (a single repetition of the sequence of operations defined in the loop is called ”iteration”). Therefore, loops and in particular nested loops (which are loops executed within other loops; the entire structure comprising all the related loops is called ”loop nest”) are the primary areas for parallelization in parallel programs.

To shorten the execution time of loop nests, one often transforms their source codes to semantically equivalent forms, by applying the transformation known as tiling (blocking). Within tiling, the iteration space of a given loop nest is divided into smaller fragments (tiles/blocks) [1,27]; in consequence thereof iterations are executed not in the original order but in the order imposed by tiling. The purpose of tiling is to increase the reuse of data already loaded into the processor cache and decrease the number of data transfers from the main memory (RAM) to the processor cache. As a result thereof, the

[*] Corresponding author e-mail: agnieszka_kaminska@wp.pl

program execution time is shortened. The shortening of program execution time is observed when the program is executed sequentially [26, 28] - which makes it reasonable to suppose that tiling may also be effective when the program is executed in parallel.

One of the most significant problems related to tiling is how many iterations of a tiled loop should be executed within a single tile, or, in other words, what tile size to adopt for a given iteration space [7, 20]. The tile size should be such that the reuse of data already loaded into the processor cache is greatest possible.

Immanent characteristics of parallel computing and tiling involve the following, significant consequences:

- Parallel computing may be applied only to these problems and tasks which are divisible into ”parts” that can be processed in parallel.
- Similarly to sequential processing combined with tiling, the organization of processor cache should be taken into account in parallel processing combined with tiling.
- If parallel computing may be applied to a given problem or task, then there is *at least* one way of dividing the problem/task into ”parts” intended for parallel processing.

If there exists more than one possible division of the initial problem/task into ”parts” intended for parallel processing, the inevitable question is: *Which of the possible divisions is the best one, i.e. results in the shortest time of solving/executing a parallelized problem/task in the target hardware environment?*

In view of the complexity of contemporary hardware, the proper answer to this question cannot be found in a purely theoretical way; it can only be found in an empirical way - through iterative compilation. Within iterative compilation, all considered and semantically equivalent source code variants of a given program are executed in a target environment; their execution times are compared and the source code variant with the shortest execution time is selected for final use. The considered source code variants are executed in particular iterations of iterative compilation, in one source code variant per iteration manner.

In case of programs solving complex problems for large data sets, it may take several hours or even days to execute a single iteration of iterative compilation. Such a situation takes place e.g. for real life problems which, 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 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 the ones with the shortest anticipated (estimated) execution times and limit the empirical selection of the source code intended for final use to the so reduced set.

Potential practical advantages related to the proposed improvement in iterative compilation and the scientific gap in this area have become an inspiration for our solution presented in this paper and involving the selection of the optimum - from the program execution time point of view - tile size for parallelized programs by means of applying iterative compilation oriented statistical models for the estimation of program execution time. We have decided to base our approach on statistical models because of a great number of factors influencing program execution time and the complexity of their mutual relations.

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

- using pattern programs to elaborate a general model for the estimation of program execution time,
- using the general model and empirical data collected in the target hardware environment to elaborate a regression specific model enabling one to estimate execution times for programs having the same, arbitrarily assumed characteristics as pattern programs,
- applying the regression specific model to select the tile size minimizing the execution time of a parallelized program in the target hardware environment.

The rest of the paper is organized as follows. Section 2 explains the role of memory hierarchy and how to increase benefits resulting thereof by tiling. Section 3 describes the idea and basic assumptions of a general and specific model for the estimation of program execution time. Section 4 outlines a general model and its equation. Section 5 describes how we have estimated the values of parameters of an exemplary specific model derived from the general model. Section 6 discusses how we have verified the quality of estimations made according to the obtained specific model. Section 7 presents the results of our experimental research. Section 8 discusses related work; conclusions are drawn in Section 9.

2 Tiling to increase benefits from memory hierarchy

A disproportion between the achievable speed of processing arithmetic/logical instructions by processors and the time of accessing the memory subsystem is one of fundamental sources of performance problems with data processing in a computer system. It takes several times longer to fetch data from some given locations in the RAM memory than to

execute basic arithmetic or logical operations on these data. In consequence, memory is a bottleneck for data processing in the entire computer system [2,3,15,16,25,29,30].

To reduce the impact of this disproportion on program execution time, hierarchical memory organization has been introduced. It involves division of the entire memory available in a computer system into levels differing in their position (i.e. the physical distance) in relation to the processor and in consequence, their access time [10,15,22,24]. Data which are required to be provided within program execution are first attempted to be fetched from the memory hierarchy level closest to the processor; if they are not found therein they are searched for on the other (i.e. of a greater access time) memory hierarchy levels.

The list of particular memory hierarchy levels, sorted in the ascending order by their physical distance from the processor and access time, is as follows:

–processor registers,
–processor cache L1,
–higher level processor caches (L2 and L3; L3 is mainly used for highly specific purposes),
–main memory (RAM),
–hard drives,
–offline memory.

Hierarchical memory organization results first and foremost in differences between data localities of particular levels of the memory hierarchy. Data locality of a given memory hierarchy level is a degree to which the data already loaded therein are sufficient to meet new data requests related to program execution without fetching any new data from higher memory hierarchy levels (i.e. the levels for which their physical distance from the processor is greater than that of the given memory hierarchy level). Increasing the data locality of the given memory hierarchy level results in shortening of program execution time.

Tiling is a software technique popularly used for increasing data locality. Tiling can be applied at various memory hierarchy levels; most often it is used at the processor cache level.

The most general case of tiling is that of a loop nest containing p nested loops which is transformed into a new, semantically equivalent loop nest containing $2p$ nested loops so that an appropriately selected number of iterations is executed within the p innermost loops thereof [1,26]. As a result of such a transformation, the original, continuous iteration space of the loop nest operating on table variables (matrices) is divided into smaller, continuous subspaces known as tiles or blocks; each tile operates on some continuous fragments of the table variables (matrices) from the original iteration space.

The idea of tiling can be explained in a simple way by the example of a source code for matrix multiplication. The source code for matrix multiplication without tiling is presented in Table 1. The source code for matrix multiplication with tiling is presented in Table 2.

Table 1: Matrix multiplication without tiling

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      D[i, j] = D[i, j] + E[i, k] * F[k, j];
```

Table 2: Matrix multiplication with tiling

```
for (ii = 0; ii < N; ii + B)
  for (jj = 0; jj < N; jj + B)
    for (kk = 0; kk < N; kk + B)
      for (i = ii; i < min(ii + B, N); ii++)
        for (j = jj; j < min(jj + B, N); jj++)
          for (k = kk; k < min(kk + B, N); kk++)
            D[i, j] = D[i, j] + E[i, k] * F[k, j];

Note: min(p, q) is a smaller of two values: p, q
```

One of the most significant problems related to tiling is the selection of size of the tiles into which the iteration space of a loop nest shall be divided. The tile size is optimal if the corresponding program execution time is minimal. Despite the extensive research in this area, the problem of selection of the optimum tile size on a non-empirical basis has not been solved yet [21]. In practice, this means that only empirical selection of the tile size guarantees that the optimum tile size is found - which, if there are many possible variants to choose from, makes this approach very time consuming and hence impractical.

The alternative to time consuming, empirical selection of the optimum tile size in tiling is to formulate and then use an algorithm which allows for finding a suboptimal tile size in tiling [11]. A proposal of such an algorithm for tiling at the processor cache level is presented in [14].

In [14] it has been demonstrated that selecting the size of a square tile for the processor cache level tiling of loop nests one should take into account the organization of processor cache and size of table variables (matrices) processed in the loops to be tiled. For a given loop nest, these 2 factors determine data reuse and interference and, in consequence, the loop (hence, the entire program) execution time.

There are 2 types of data reuse: temporal data reuse and spatial data reuse. Temporal data reuse takes place when the data fetched from a given 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 given memory location are used in the program [1,28].

Interference takes place when a cache line containing data that can be reused during execution of the program is overwritten with new data despite the fact that there is sufficient unoccupied space in the processor cache whereto the new data could well be stored instead - however, because of the cache organization, a specific and already occupied cache line has to be overwritten [1,5,6,23].

There are 2 types of interference: self interference and cross interference. Self interference takes place when the data that are already in the cache memory and the data that are to replace them are the elements of one and the same table. Cross interference takes place when the data that are already in the cache memory and the data that are to replace them are not stored under the same scalar variable/table [5,14].

Because of the specificity of cache memory operation, data reuse and interference are related to one another.

Namely, if for a given program (program loop) there is no temporal data reuse and the declared in the source code total size of data to be processed in the program does not exceed the size of the L2 processor cache, the probability that interference occurs is negligibly low. When spatial data reuse is the only type of data reuse in the program, the data fetched to the processor cache memory are fully used by the processor in a very short time since fetching them, hence the low probability in question. Thus, because of the very specificity of spatial data reuse, the probability of replacing the data fetched to the cache memory and still to be used during execution of the program, with other data, is negligibly low. In consequence, we have assumed that non-occurrence of temporal data reuse for a given program is equivalent to non-occurrence of interference for that program.

If for a given program (program loop) there is temporal data reuse, then the higher the temporal data reuse of the data already fetched to the processor cache memory, the higher the probability of replacing with other data the data already fetched to the cache memory and still to be used during execution of the program. This is a consequence of the specificity of temporal data reuse which involves reusing the data fetched to the processor cache memory for a long time since the moment of fetching them. Therefore, we have assumed that occurrence of temporal data reuse for a given program is equivalent to occurrence of interference for that program.

From the practical point of view, tiling is only profitable when there is temporal data reuse in the tiled loop [5] - which, in view of the discussion presented above, is equivalent to occurrence of interference. However, on the other hand, according to [14] the side size of a square tile should be such as to prevent occurrence of self interference (yet, occurrence of cross interference is acceptable) since occurrence of self interference results in a much longer program execution time than that in the case when there is no self interference. Occurrence of cross interference does not impact program execution time as strongly as occurrence of self interference.

Applying the approach proposed in [14], one can calculate for a square tile its boundary side size B (i.e. the largest possible side size that can be applied when tiling a square matrix of side size N with square tiles) which induces no self interference.

The relationship between B, the actually used side size ($BLOCK\_SIZE$) of a square tile, the side size (N) of a square matrix under tiling and the resultant self interference is as follows:

If $0 < BLOCK\_SIZE \leq B$ then there is no self interference.

If $B < BLOCK\_SIZE < N$ then there is self interference.

If $BLOCK\_SIZE \geq N$ then tiling is not profitable from the practical point of view.

To quantitatively estimate the impact of cache level tiling on data locality, one can use data footprint. Data footprint is an estimate calculated for a given program based on its source code.

Data footprint indicates the estimated minimum capacity of direct mapped cache memory which is essential for simultaneously storing all the data processed in the program, assuming that the data stored in the cache are fully reused

(both in the temporal and spatial aspect). Thus, data footprint indicates the minimum estimated volume of data which will be fetched from the main memory to the cache memory during execution of the program.

If a given program can be expressed by means of various but semantically equivalent source codes, then the source code variant which requires the smallest total volume of data to be fetched to the cache memory during execution of the program involves the best data reuse and, in consequence, the shortest execution time.

3 Statistical model for the estimation of program execution time

Because most of time consuming operations - calculations made within computer programs - are executed in loop nests, we have decided to limit the scope of the applicability of a statistical model to be elaborated to a class of loop nests which are often used in practice, namely: coarse grained parallel loop nests, represented in the OpenMP C/C++ standard. We have tiled the loop nests in question to shorten their execution time in the target hardware environment.

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 loop nest structure in which the outermost loop of the nest is parallelized. Coarse grained granulation is typically used in parallelization of programs executed by currently very popular multiprocessor machines with shared memory [13].

OpenMP [18] 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.

Our statistical model for the estimation of program execution time is based on a general model, i.e. a general equation of a function to estimate the execution time of tiled coarse grained program loop nests presented in the OpenMP C/C++ standard.

Program execution time has been assumed as the dependent variable of the general model. We have assumed that quantitative variables reflecting factors which significantly influence program execution time should be the independent variables of the general model. Apart from dependent and independent variables, the general model comprises parameters the values of which 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 empirical data collected in this environment. Regression analysis has been selected for this purpose as it is a very well developed and commonly used method of identification and description of dependencies found in sets of empirical data; moreover applying regression analysis it is possible to obtain simple analytical equations (regression models) which express the dependencies in question with very good accuracy.

In order to collect the required empirical data, we have used a program prepared specially for this purpose and representing some arbitrarily selected features which are specific for tiling. This program is hereafter referred to as the pattern program. Taking into account the conclusions presented in section 2, the assumed exemplary pattern program *matmul* takes account of data reuse and interference. The source code of this program is presented in Table 3.

Table 3: Pattern program *matmul*

```
int ma[N][N], mb[N][N], mc[N][N];
int i, j, k, r, N;

for (i = 0; i < N; i++) {
   for (k = 0; k < N; k++) {
      r = ma[i][k];
      for (j = 0; j < N; 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. The specific model defines the general model for a particular situation by assigning relevant values to the parameters of the general model.

Each specific model is derived from the general model for a particular pattern program. Since for each program loop it is possible to clearly state whether the loop exposes data reuse and interference and, more importantly, what program

behaviour results thereof, these 2 criteria can be used to divide loops (hence, programs) into groups. Then, specific models for a given hardware environment could be elaborated only for representatives of particular groups (i.e. pattern programs) yet be used by entire groups. Thus, a specific model derived for a particular pattern program can be applied both to the pattern program and to other programs with the same data reuse type and interference as in case of the pattern program (such programs we hereafter call non pattern).

Although the influence of data reuse and interference on program execution time is clear, the same cannot be said about the influence of tiling and it is so because the problem of determining a dependency between the tile size in tiled code and the related program execution time has still not been solved for a general case. For this reason we have decided not to tile pattern programs, as the dependencies empirically grasped from tiled pattern programs would be specific for these programs and could hardly generalize to non pattern programs. Still, as the data reuse type and interference are the common characteristics of both untiled pattern programs and tiled non pattern programs, specific models derived for untiled pattern programs can be supposed to generalize sufficiently well to non pattern tiled programs. We demonstrate in section 7, based on the experimental results obtained for an exemplary specific model derived for the untiled *matmul* pattern program, that the specific model in question can satisfactorily be used for tiled non pattern programs.

In order to prevent the extrapolation of a specific model beyond the data range for which the model is constructed, we have clearly defined what the scope of the applicability of a specific model to non pattern programs is, by introducing limitations on:

–the total size of data processed by a program,
–the maximum number of iterations in a single chunk of iterations assigned to be executed by an OpenMP program thread,
–program execution time,
–the maximum tile side size for a square tile.

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

Specific models can be used in iterative compilation to estimate the execution times of various source code variants of a given program. Based on resultant estimates, one selects for execution in a target environment the source code variants with several shortest anticipated execution times. From the so reduced set of source code variants, one selects for final use the source code variant with the shortest execution time measured in the target hardware environment. As a result of applying the specific models in such a way, the total time of carrying out iterative compilation for non pattern programs, which meet the limitations related to application of specific models, will be shortened - since, instead of executing in the target hardware environment all the source code variants, one would only execute the source code variants from a reduced set of code variants.

4 General model

The execution time of a program is the resultant of the interaction of a great number of various heteregenous factors. So, it is not possible to identify and quantify them all so that all of them could be included in a model for the estimation of program execution time. Therefore, in order to elaborate the model, we have decided to act in the following way: select some factors which 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 a 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 the hardware environment in which a parallel program is to be executed.

To derive a model, we have quantified the influence of factors a), b) and c) on 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, parallelism is realized by multiple threads. The time of the 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 a potential independent variable ($X4$) of the general model.

If a task to be executed is carried out within a program loop nest, each of invoked OpenMP threads is assigned to execute a certain number of iterations of the loop nest. Depending on an 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.

The time of the execution of a program loop nest is determined by the execution time of the thread which has been assigned to execute the greatest number of iterations, and in particular by the size of the largest chunk of iterations assigned to this thread.

Therefore, we have adopted as an independent variable ($X3$) of the 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 resolved in programs are reflected in the number and type of arithmetic operations to be executed by a 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 execution times of instructions for 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 the general model.

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

Because of a significant disproportion between the processor speed and memory access time of today's computers, it is the memory - and especially the quickly accessible processor cache memory - that is one of the hardware elements that determine the program execution time.

Ideally, all the data needed by the 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 the main memory into the processor cache.

On the other hand, the capacity of the cache memory and its replacement policy (associativity) determine what fraction of data processed in the 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 processor cache memory in a given computer system and its replacement policy (associativity).

2. The minimum data storage capacity of direct mapped processor cache, which is necessary in order to contain all the data processed in a program, assuming that the data stored in the cache memory are fully reused (both in the temporal and spatial aspect) and that for tiling one used a square tile of the tile side size ($BLOCK\_SIZE$) which induces no self interference.

The minimum data storage capacity in question can be estimated by means of data footprint (according to the methods presented in [14] and [28]). 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. When tiling has been applied, calculating data footprint one should take into account the resultant tiles into which the iteration space of a tiled loop nest has been divided. In practice, this means that instead of adopting as the data footprint for the tiled iteration space a data footprint calculated for the entire iteration space, one should sum data footprints of the resultant tiles and take this sum as the data footprint for the tiled iteration space.

To illustrate the above, let us consider an exemplary iteration space set by the 2 loops: loop j and loop i (for simplicity, it is assumed that for both the loops the number of iterations is identical and equal to $MATRIX\_SIZE$, i.e. $N_i = N_j = MATRIX\_SIZE$), which is tiled with a square tile with the $BLOCK\_SIZE$ side size as presented in Figure 1. For this exemplary iteration space, the data footprint is equal to the sum of data footprints of the below listed resultant tiles:

–tiles for which: $j = i = BLOCK\_SIZE$
–tiles for which: $j = BLOCK\_SIZE$ and $i = N_i \bmod BLOCK\_SIZE = MATRIX\_SIZE \bmod BLOCK\_SIZE$
–tiles for which: $j = N_j \bmod BLOCK\_SIZE = MATRIX\_SIZE \bmod BLOCK\_SIZE$ and $i = BLOCK\_SIZE$
–tiles for which: $j = N_j \bmod BLOCK\_SIZE = MATRIX\_SIZE \bmod BLOCK\_SIZE$ and $i = N_i \bmod BLOCK\_SIZE = MATRIX\_SIZE \bmod BLOCK\_SIZE$

and not to the data footprint calculated for the entire (i, j) iteration space.

3. A relation between 1. and 2.

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

Thus, the final list of the potential independent variables of our model comprises the following variables: $X1$, $X2$, $X3$, $X4$.

With such a list of the potential independent variables of the 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, expressed by the

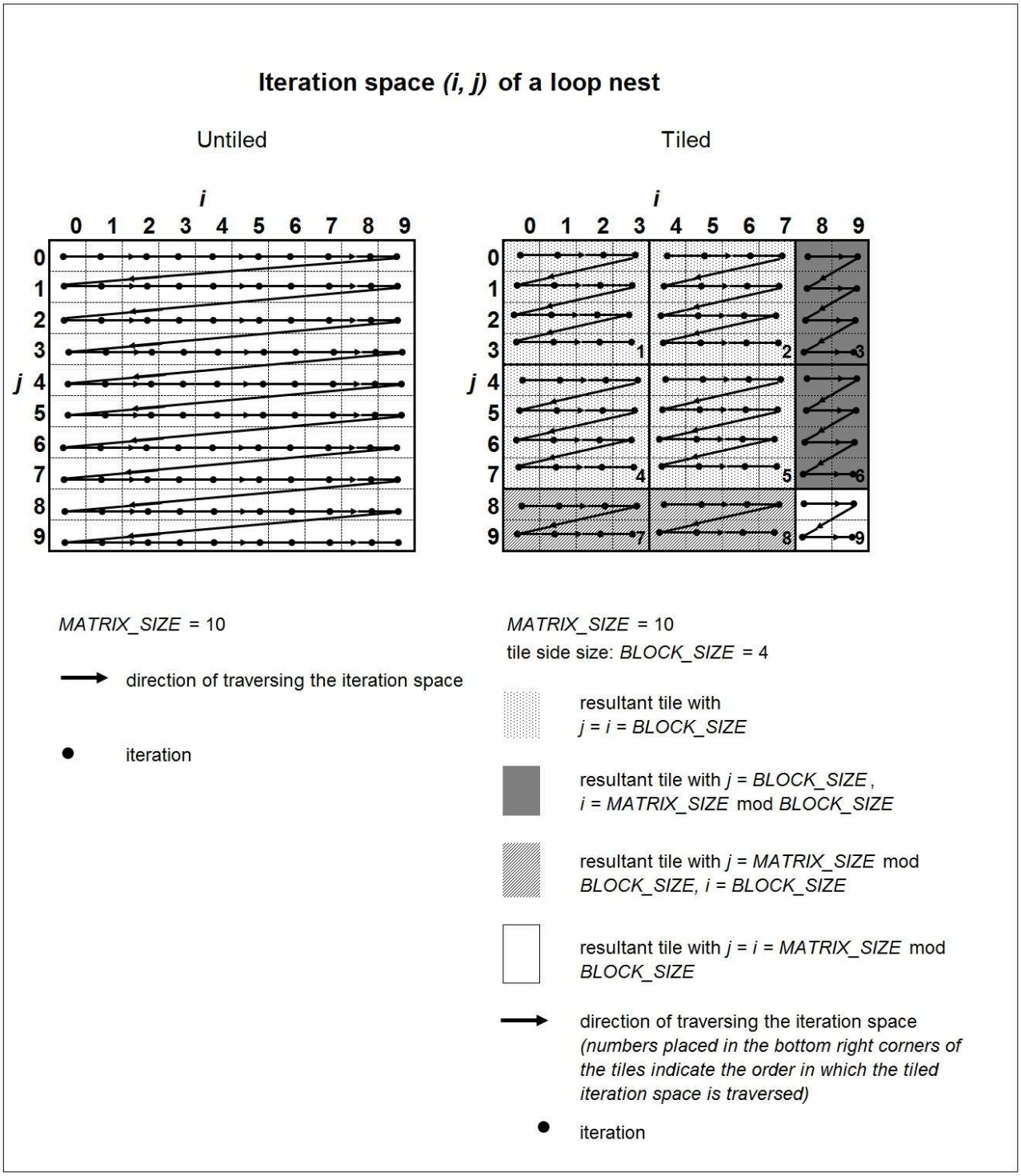

Figure 1: Untiled vs tiled iteration space of a loop nest

number of CPU clock cycles, we have undertaken regression analysis. The object of the regression analysis was empirical data collected for the *matmul* pattern program prepared specially for that 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 corresponding values of independent variables $X1, X2, \ldots, Xp$ is expressed by equation (1):

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

where:

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

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

$X1_i, \ldots, Xp_i$ are known values of independent variables $X1, X2, \ldots, Xp$, 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) can be applied when a dependency between empirically found values of the dependent variable and independent variables is either linear or linearly transformable nonlinear (i.e. power, exponential, logarithmic, or hyperbolic).

Therefore, for independent variables: $X1, X2, X3, X4$ and dependent variable Yt, the 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 equation (2):

$$Yt = a_1 \times X1 + a_2 \times X2 + a_3 \times X3 + a_4 \times X4 \tag{2}$$

–a power form, expressed by equation (3):

$$Yt = X1^{a1} \times X2^{a2} \times X3^{a3} \times X4^{a4} \tag{3}$$

–an exponential form, expressed by equation (4):

$$Yt = a1^{X1} \times a2^{X2} \times a3^{X3} \times a4^{X4} \tag{4}$$

–a logarithmic form, expressed by equation (5):

$$Yt = a1 \times \log X1 + a2 \times \log X2 + a3 \times logX3 + a4 \times logX4 \tag{5}$$

–a hyperbolic form, expressed by equation (6):

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

Note: Parameter a_0 is not taken into account in equations (2) ÷ (6) because it has no practical sense for the modelled 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 select the ultimate list of independent variables from the potential 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 these variables is a power one.

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

–variable Yt and all the independent variables considered altogether (case 1/),

–variable Yt and particular independent variables considered individually (cases 2/ ÷ 5/).

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

The values of the coefficient of determination obtained for the *matmul* pattern program are presented in Table 4. The greatest value of R^2 for case 1/ has been obtained for power model (3). Moreover, the aforementioned power model is very well fitted for cases 2/ ÷ 5/ as well. This proves that there exists a power dependency between the dependent variable and each of the considered potential independent variables of the model.

Variables $X1$, $X2$, $X3$, $X4$ have been proposed as potential independent variables of the general model. The reason for defining these variables as potential is that the carried out analysis of R^2 calculated for Yt and Xi (where Xi denotes a proposed independent variable; $i = 1,2,3,4$) indicates that all these variables influence program execution time. However, only after calculating the adjusted coefficient of determination for each possible subset of variables $X1$, $X2$, $X3$, $X4$ one can say which subset of the variables has the greatest value of the adjusted coefficient of determination and in consequence, the strongest influence on program execution time.

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

Table 5: 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
$X1$	**$X2$**	**$X3$**	**$X4$**	**0.9999514**	**0.9999464**

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 relation 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,

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

5 Estimation of parameter values for a specific model

Our goal is to determine the values of parameters for a specific model for a given computer environment by means of applying such a method that could be reapplied for any computer environment. Therefore, we have decided to determine the values of parameters $a1$, $a2$, $a3$, $a4$ for a given environment by means of the statistical analysis of empirical data collected in this environment.

To determine the values of parameters $a1$, $a2$, $a3$, $a4$, we have used the *matmul* pattern program. The source code of the *matmul* program is presented in Table 3.

Empirical data collected for a pattern program are the basis for determining the values of parameters $a1$, $a2$, $a3$, $a4$ of a specific model referring to all such programs which 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 the pattern program, is referred to as a non pattern program.

It should be stressed here that the *matmul* pattern program is an *exemplary* pattern program adopted simply in order to determine an exemplary specific model on the basis of general model (7). This realization of the pattern program (i.e. by adopting the *matmul* program) is one of *many possible* realizations. Assuming some other realization of a pattern program, one could derive a specific model with domains different from the domains of the specific model derived from the *matmul* pattern program. 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 order to obtain empirical data that are representative for an environment under analysis, it has been assumed that for the adopted pattern program:

1. The total size of the data processed in a loop nest does not exceed the size of 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 processed in a loop nest, with 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.

3. The side size of a square tile in tiling should not be greater than the boundary size B inducing no self interference and calculated in accordance with the approach proposed in [14].

For assumptions 1. $\div$ 3., the assumed pattern program (*matmul*) and hardware environment of empirical research (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), we have derived the following specific model:

$$Yt = X1^{-0.298695} \times X2^{0.623738} \times X3^{0.014426} \times X4^{0.962976} \tag{10}$$

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 a specific model to non pattern programs, we have formulated the following detailed assumptions regarding the scope of the applicability of specific models:

1. The value of λ, calculated for a non pattern program as per 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:

$$\lambda_{min}(referenceLoop) \leq \lambda \leq \lambda_{max}(referenceLoop) \tag{11}$$

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,
$\lambda_{max}(referenceLoop)$ represents 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:

$$\gamma_{min}(referenceLoop) \leq \gamma \leq \gamma_{max}(referenceLoop) \tag{12}$$

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 (for a given sample of measured CPU times) 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 (for a given sample of measured CPU times) 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 (12) 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 corresponding pattern programs in respect of data reuse and cache interference, in other respects the programs may differ so much from corresponding pattern programs 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.

4. The side size of a square tile in tiling should not be greater than the boundary size B inducing no self interference and calculated in accordance with the approach proposed in [14].

6 Verification of the quality of estimations

The verification of the quality of estimations made according to the proposed general model is equivalent to the assessment of the quality of an exemplary specific model derived from the general model for the *matmul* pattern program.

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

The qualitative quality assessment of the specific model has been focused on recognizing whether applying the specific model one can estimate with satisfactory accuracy the execution time of non pattern programs meeting the assumptions regarding the scope of the applicability of specific models.

In practice, this means that one should check whether, for a given size of the problem solved in a program (program loop nest), the trend of changes in measured execution times per OpenMP thread of particular variants of a given program

matches the trend of changes in corresponding estimated execution times per OpenMP thread calculated according to the elaborated model.

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

$\delta_{Yt(per\_thread)}$ is the relative estimation error for $Yt(per\_thread)$, calculated according to formula (13)

$$\delta_{Y(per\_thread)} = 100\% \times \left| \frac{Yt(per\_thread) - \gamma(per\_thread)}{\gamma(per\_thread)} \right| \tag{13}$$

$$Yt(per\_thread) = \frac{Yt}{X4^{a4}} \tag{14}$$

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

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 OpenMP 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, a sequence of various source code variants of a given program sorted in the descending order by their estimated execution times per OpenMP thread calculated according to the model is compared with a sequence of the same source code variants sorted in the descending order by their measured execution times per OpenMP thread. The trend matching verification allows us to find out whether, applying only a model, one can properly select from all considered source code variants of a given program a small subset of source code variants including the source code variant with the minimal actual execution time in a hardware environment. Then, iterative compilation is carried out only for the source code variants from the selected subset. Therefore, if the subset in question is properly selected, the estimation errors obtained within the quantitative quality assessment are of minor importance.

7 Results of experimental research

In order to demonstrate that the obtained model is indeed useful in iterative compilation, we have used the NAS Parallel Benchmarks (NPB) suite [9,17]. We have chosen NPB because it is a test suite dedicated for the assessment of the performance of parallel computers and consists of a great number of very various loop nests.

4 loop nests (benchmarks) selected from the NPB test suite were the object of the experimental research. Selected loop nests consist of 3 to 4 loops. For our experiments, each of the selected loop nests was transformed to several semantically equivalent forms, by parallelizing its outermost loop (so as to expose coarse grained parallelism) and tiling its two innermost loops. There are no dependencies in the parallelized loops - therefore, these loops were parallelized and tiled manually. According to the conclusions presented in [14], in case of tiling a sequentially processed loop nest the tile of the boundary side size B (i.e. the largest possible side size inducing no self interference that could be applied when tiling a square matrix of side size N) is optimal from the execution time perspective. Therefore, in our experimental research carried out for loop nests processed in parallel, it was assumed that the tile side size could not be greater than the boundary side size B; the value of B was calculated by applying the approach proposed in [14].

The benchmarks selected for our experimental research are different from the pattern program, but they represent the same combination of data reuse and cache interference as the pattern program. By means of the exemplary specific model derived for the *matmul* pattern program, we estimated execution times for various source code variants of the 4 selected benchmarks. In total, we estimated execution times for 54 various, tiled source codes.

For each selected benchmark, the experimental research was carried out in the following way:

For each (semantically equivalent) source code variant of a given benchmark, we **estimated** its execution time per OpenMP thread in the target hardware environment, by applying a specific model derived for the *matmul* pattern program.

The estimates were sorted in the descending order, forming a sequence. Then, source codes corresponding to particular elements of the sequence were executed in the target environment and their empirically measured execution times per OpenMP thread were registered.

Let:

t be the number of all various input source code variants for a given benchmark,

k $(0 < k < t)$ be the value given by the user and denoting the assumed number of source code variants with shortest estimated execution times for a given loop nest,

k_B represent the minimum value of k which guarantees that one selects for final use from semantically equivalent source code variants the source code variant with the shortest execution time measured in the target environment and that the set of source code variants used for selection purposes will include this source code variant which

(i)has been tiled with the square tile of the maximum side size not inducing self interference and, at the same time,
(ii)has the shortest measured execution time of all source code variants tiled with this tile side size.

k_{min} represent the minimum value of k which guarantees that one selects for final use from semantically equivalent source code variants the source code variant with the shortest execution time measured in the target environment.

For each benchmark under analysis, we compared estimated and measured execution times of its source code variants having the k shortest estimated execution times. For each benchmark, we selected for final use from the k source code variants the variant with the shortest **measured** execution time.

The fundamental problem here is what value of k should be adopted and whether, for a given benchmark and its source code variants, $k_{min} = k_B$.

If $k = t$, then our approach does not reduce iterative compilation time.

If $k = 1$, it is not certain whether the source code variant with the shortest measured execution time will be selected for final use because the specific model estimates execution time with errors. It should be noted here that the larger the value of variable k, the longer the time of iterative compilation. At the same time, the increase in k increases the probability of selecting for final use the source code variant with the shortest execution time from all input variants. So, what is the minimum value of k (i.e. k_{min}) which guarantees that the source code variant with the shortest measured execution time will be selected for final use is less than t iterations? The value of k_{min} and k_B can be determined by comparing the measured and estimated execution times (per OpenMP thread) of t source code variants. Let M be the source code variant with the shortest measured execution time per OpenMP thread. Let t source codes be sorted in the descending order by their estimated execution times per OpenMP thread, forming a sequence C and let s be such a source code variant in sequence C that $s = M$. Then, k_{min} is the position of s in sequence C taken in the reverse order. The value of k_B is equal to the position, in sequence C taken in the reverse order, of this source code variant which has been tiled with the tile of the maximum side size not inducing self interference (calculated as per [14]) and which at the same time has the shortest estimated execution time of all source code variants tiled with the tile side size in question.

For each of the selected benchmarks, we have assessed the quality of estimates obtained by applying the specific model. For each of the benchmarks, the trend of changes in the measured execution times per OpenMP thread of particular variants of a given benchmark is matched by the trend of changes in corresponding estimations per OpenMP thread calculated according to a relevant specific model. The conclusion about the matching trends has been formulated based on the comparison of the respective linear trends.

The mean and maximum relative estimation errors calculated in relation to execution times measured empirically for all source code variants adopted for a given benchmark and the size of a problem, solved within the benchmark, do not exceed 45 and 60 per cent points (detailed results are presented in Table 6), respectively.

For each of the selected NPB benchmarks, we have also estimated the reduction of iterative compilation time (hence, software development time) that could be achieved by selecting the side size of a square tile for parallelized benchmarks by means of applying the specific model derived for the *matmul* pattern program. The so obtained estimates have been compared to the estimated shortening of iterative compilation time that could be achieved by applying the maximum side size (of a square tile) inducing no self interference and calculated according to [14].

The results are presented in Tables 7 ÷ 8.

The carried out empirical research indicates that the maximum tile side size (of a square tile) inducing no self interference and calculated according to [14] does not always optimize program execution time (see Table 7). In case of benchmarks UA_diffuse_3 and UA_diffuse_4, the empirically found tile side size optimizing program execution time is smaller than the maximum tile side size (of a square tile) inducing no self interference and calculated according to [14].

When selecting the side size of a square tile for a parallelized source code based on our model and the procedure described in this section, the source code variant selected for final use from semantically equivalent source code variants is the one tiled with the tile of the size optimizing program execution time. This source code variant is found within k_{min} iterations of iterative compilation. The results presented in Table 8 indicate that for the analysed benchmarks, finding the optimum source code variants among t semantically equivalent source code variants within k_{min} instead of t iterations of

Table 6: Quality assessment of estimates calculated according to specific model (10)

	Loop nest (benchmark), problem size S						
	UA3, 50	UA4, 50	UA4, 66	UA11, 100	UA11, 267	UA16, 100	UA16, 267
[1]	9	9	6	9	6	9	6
[2]	34.59	27.41	26.46	41.76	39.94	40.90	37.38
[3]	40.19	35.65	29.30	56.95	52.75	54.73	50.22

[1] Number of various, semantically equivalent source code variants subjected to the estimation of execution time
[2] Resultant mean for $\delta_{Yt(per\_thread)}$ [%]
[3] Resultant maximum for $\delta_{Yt(per\_thread)}$ [%]

The loop nests are denoted as follows:
UA3 - UA_diffuse_3 (tiled)
UA4 - UA_diffuse_4 (tiled)
UA11 - UA_transfer_11 (tiled)
UA16 - UA_transfer_16 (tiled)

Table 7: Selection of the side size for a square tile in tiling

	Loop nest (benchmark), problem size S						
	UA3, 50	UA4, 50	UA4, 66	UA11, 100	UA11, 267	UA16, 100	UA16, 267
[1]	41	41	31	41	24	41	24
[2]	26	32	24	41	24	41	24
[3]	26	32	24	41	24	41	24

[1] Maximum side size (of a square tile) inducing no self interference and calculated according to [14]
[2] Tile side size optimizing program execution time - as per the results of our experimental research
[3] Tile side size for k_{min}

The loop nests are denoted as follows:
UA3 - UA_diffuse_3 (tiled)
UA4 - UA_diffuse_4 (tiled)
UA11 - UA_transfer_11 (tiled)
UA16 - UA_transfer_16 (tiled)

iterative compilation results in reducing the iterative compilation time from approximately 4 to approximately 14 times (detailed results are presented in Table 8).

If, according to [14], one assumes that the maximum tile side size (of a square tile) inducing no self interference is the tile side size optimizing program execution time - then, applying our procedure of searching among t semantically equivalent source code variants for the source code variant with optimum execution time, the source code variant with optimum execution time is found within k_B iterations of iterative compilation. For benchmarks UA_diffuse_3 and UA_diffuse_4, $k_{min} < k_B$ - and for these benchmarks finding among t semantically equivalent source code variants the source code variant with optimum execution time within k_B instead of t iterations of iterative compilation takes $2 \div 3$ times longer than in the situation when the optimum source code variant is found within k_{min} iterations of iterative compilation. For benchmarks UA_transfer_11 and UA_transfer_16, $k_{min} = k_B$ - hence, for these benchmarks, finding among t semantically equivalent source code variants the source code variant with optimum execution time within k_B instead of t iterations of iterative compilation takes the same time as in the situation when the optimum source code variant is found within k_{min} iterations of iterative compilation.

The experimental research has been focused on demonstrating the usefulness of our proposed approach when applied to small benchmark codes. In view of the achieved, positive results we plan to fully implement our approach, integrate it with some parallelizing source to source compiler and examine using a greater number of various benchmark codes the effectiveness of the so enhanced compiler in generating source codes optimized for execution time.

Table 8: Reduction of iterative compilation time after applying specific model (10), derived for the *matmul* pattern program

	Loop nest (benchmark), problem size S						
	UA3, 50	UA4, 50	UA4, 66	UA11, 100	UA11, 267	UA16, 100	UA16, 267
[1]	9	9	6	9	6	9	6
[2]	1	1	1	1	2	1	2
[3]	3	2	2	1	2	1	1
[4]	372 407	355 263	585 501	50 545	791 499	51 597	787 679
[5]	29 061	27 649	83 959	3 620	219 195	3 727	219 197
[6]	87 324	55 458	168 056	3 620	219 195	3 727	219 197
[7]	4.26	6.41	3.48	13.96	3.61	13.84	3.59
[8]	12.81	12.85	6.97	13.96	3.61	13.84	3.59

[1] t
[2] k_{min}
[3] k_B
[4] Iterative compilation time for t source code variants - Tt
[5] Iterative compilation time for k_{min} source code variants - Tk_{min}
[6] Iterative compilation time for k_B source code variants - Tk_B
[7] Reduction of iterative compilation time $\frac{Tt}{Tk_B}$
[8] Reduction of iterative compilation time $\frac{Tt}{Tk_{min}}$

The loop nests are denoted as follows:
UA3 - UA_diffuse_3 (tiled)
UA4 - UA_diffuse_4 (tiled)
UA11 - UA_transfer_11 (tiled)
UA16 - UA_transfer_16 (tiled)

8 Related work

Program execution time, iterative compilation and selection of the optimum tile size in tiling are objects of scientific research carried out in many centres. Within this research, various solutions have been proposed, namely methods for: selecting the optimum tile size in tiling [14,5,31], forecasting program execution time [4], estimating program execution time [8] or selecting the program source code variant with the shortest anticipated execution time [19].

An algorithm for finding the maximum tile side size which, when applied in order to tile square matrices with a square tile, induces no self interference, is presented in [14]. By applying this algorithm it is possible to very quickly determine the side size of the searched tile however - as shown by our experimental research described herein - in case of parallelized source codes, applying the tile side size as found by the algorithm does not always result in the optimum program execution time.

An algorithm similar in the assumptions made to algorithm [14] yet allowing for finding rectangular tiles is presented in [5].

A proposal of using ”synthetic” tiled programs which expose 6 features related to temporal and spatial data reuse in order to determine the tile size optimizing program execution time is presented in [31]. The presence of the above features can be easily detected based only on the source code of a program. Synthetic programs are used for generation, by means of neural networks, of the model enabling one to select the optimum tile size. The model is created for a given hardware architecture and compiler. Preparation of the model is highly time consuming, because a lot of training data has to be collected for teaching the neural network.

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 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 random search strategy algorithm is proposed in [8]. 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. The method in question lets one determine what the 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, one will find such a source code variant of the program whose execution time will be approximately equal to the minimum execution time of this program.

A tournament predictor is presented in [19]. It is a model which 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 in comparison with the other sequence. The independent variables of the model proposed in [19] 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.

In view of the above discussed limitations of the approaches presented in [14,5,31,4,8,19] the approaches in question are not adequate for carrying out the proposed improvement of iterative compilation, which involves an analytical selection from semantically equivalent tiled source code variants of a given program the ones with shortest anticipated execution times in order to limit the empirical selection of the best source code variant to the so reduced set.

The solution we present in this paper is free from the limitations mentioned in the aforementioned approaches. To our best knowledge, it is the first attempt to use statistical models for selection of the tile side size (of a square tile) minimizing program execution time.

Applying our solution, it is possible to quickly elaborate models for the estimation of program execution time, which are adequate both for pattern programs from which models have been derived and for completely different (non pattern) programs which 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 our statistical model for the estimation of program execution time taking into account tiling. The model has been elaborated based on empirical data collected for the *matmul* pattern program representing some arbitrarily selected features related to the program structure and the specificity of program execution environment.

The elaborated specific model has 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 in consequence, the related software development time) which could be achieved by applying specific models in accordance with the proposed procedure of supporting iterative compilation with such models. For the exemplary programs presented in the paper and coming from the NPB benchmark suite, we have selected the tile side size minimizing program execution time within iterative compilation supported with the elaborated specific model in the way as described in section 7. By applying the specific model for this purpose, the time of iterative compilation has been shortened from approximately 4 to approximately 14 times (depending on the benchmark) in relation to the time which would be needed for carrying out the tile side size selection if iterative compilation was not supported with the specific model. Detailed results are presented in Table 8.

The achieved results indicate that our solution presented in the paper is adequate for use in iterative compilation for selection of the tile side size (of a square tile) optimizing program execution time and hence, gives a possibility of reducing the time of software development.

References

[1] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman. Compilers: Principles, Techniques, and Tools (2nd Edition), Addison Wesley, 2006.
[2] K. Asanovic et al. The landscape of parallel computing research: A view from Berkeley, EECS Department, University of California, Berkeley, 2006.
[3] G. Bell, R. Sites, W. Dally, D. Ditzel, Y. Patt. Architects Look to Processors of Future, Microprocessor Report, **10**, 1-7 (1996).
[4] J. Berlińska, Methods of creating statistical models characterizing parallel and distributed applications. PhD thesis (in Polish), Politechnika Szczecińska, 2005.
[5] S. Coleman, K.S. McKinley. Tile Size Selection Using Cache Organization and Data Layout, ACM SIGPLAN Notices, **30**, 279-290 (1995).
[6] K. Esseghir, Improving data locality for caches. Master's thesis, Rice University, 1993.
[7] B.B. Fraguela, M.G. Carmueja, D. Andrade. Optimal tile size selection guided by analytical models, Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005, 565-572 (2005).
[8] G. Fursin, Iterative Compilation and Performance Prediction for Numerical Applications. PhD thesis, University of Edinburgh, 2004.
[9] J. Haoqiang, M. Frumkin, J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance, NASA Ames Research Center, 1999.

[10] J.L. Hennessy, D.A. Patterson. Computer architecture: a quantitative approach, Elsevier, 2007.
[11] C. Hsu, U. Kremer. A quantitative analysis of tile size selection algorithms, The Journal of Supercomputing, **27**, 279-294 (2004).
[12] K. Ishizaka, M. Obata, H. Kasahara. Coarse grain task parallel processing with cache optimization on shared memory multiprocessor, Languages and Compilers for Parallel Computing, 352-365 (2003).
[13] I.H. Kazi, D.J. Lilja. Coarse-grained Speculative Execution in Shared-memory Multiprocessors, Proceedings of the 1998 International Conference on Supercomputing - ICS '98, Melbourne, Australia, 93-100 (1998).
[14] M.S. Lam, E.E. Rothberg, M.E. Wolf. The Cache Performance and Optimization of Blocked Algorithms, ACM SIGARCH Computer Architecture News, **19**, 63-74 (1991).
[15] N.R. Mahapatra, V. Balakrishna. The processor-memory bottleneck: problems and solutions, Crossroads, **5** (1999).
[16] S.A. McKee, Reflections on the memory wall, Proceedings of the 1st conference on Computing frontiers, 162-167 (2004).
[17] NAS Parallel Benchmarks (2015), http://www.nas.nasa.gov/publications/npb.html
[18] The OpenMP API specification for parallel programming (2015), http://www.openmp.org/
[19] E. Park, S. Kulkarni, J. Cavazos. An Evaluation of Different Modeling Techniques for Iterative Compilation, Proceedings of the 14th international conference on compilers, architectures and synthesis for embedded systems, 65-74 (2011).
[20] J. Ramanujam, P. Sadayappan. Tiling of Iteration Spaces for Multicomputers, ICPP, **2**, 179-189 (1990).
[21] J. Shirako et al. Analytical bounds for optimal tile size selection, Compiler Construction, 101-121 (2012).
[22] W. Stallings, Computer Organization and Architecture: Designing for Performance (5th Edition), Prentice Hall, 2000.
[23] O. Temam, C. Fricker, W. Jalby. Cache interference phenomena, ACM SIGMETRICS Performance Evaluation Review, **22**, 261-271 (1994)
[24] R. Van Der Pas, Memory hierarchy in cache-based systems, Sun Blueprints, 2002.
[25] G.V. Wilson, Practical parallel programming, The MIT Press, Cambridge, MA, 1995.
[26] M.E. Wolf, M.S. Lam. A Data Locality Optimizing Algorithm, ACM Sigplan Notices, **26**, 30-44 (1991).
[27] M. Wolfe, More iteration space tiling, Proceedings of the 1989 ACM/IEEE conference on Supercomputing, 655-664 (1989).
[28] M. Wolfe, High Performance Compilers for Parallel Computing, Addison-Wesley, 1996.
[29] W.A. Wulf, S.A. McKee. Hitting the memory wall: implications of the obvious, ACM SIGARCH computer architecture news, **23**, 20-24 (1995).
[30] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, F. Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs, Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, 93-104 (2007).
[31] T. Yuki et al. Automatic creation of tile size selection models, Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization, 190-199 (2010).

Agnieszka Kamińska received the MSc degree in computer science from the West Pomeranian University of Technology in Szczecin in 2009. In 2014, she received the PhD degree in computer science from the West Pomeranian University of Technology in Szczecin. Her research interests include parallel computing, optimization of compilation and application of statistical methods in software engineering.

Włodzimierz Bielecki received the MSc degree in computer science from the Kiev International University of Civil Aviation in 1975, the Candidate of Sciences (PhD) degree in computer science from the Kiev Institute of Electrodynamics of the Ukrainian Academy of Sciences in 1980, the Doctor of Sciences degree in computer science from the Kiev Institute of Simulation Problems in Power Engineering of the Ukrainian Academy of Sciences in 1989 and in 1996 - the Professor title from the Ukrainian Education Ministry. Since 1994, he is with the West Pomeranian University of Technology in Szczecin where he works as the head of the Department of Software Engineering. His research interests include parallel and distributed computing, optimizing compilers and compilation techniques.