Dynamic Tiling Optimization for Polly Compiler

Dominik Adamski, Michał Szydłowski, Grzegorz Jabłoński, and Jakub Lasoń

***Abstract*—This article presents dynamic tiling optimization for Polly compiler. It describes heuristic technique which can be applied to increase efficiency of tiling optimization. Proposed solution is based on open-source tools (LLVM and Polly compiler) and it proves that dynamic tiling optimization can be achieved by extraction code of tiled loop into seperate function. The compiler can generate multiple versions of the optimzed functions. Each of them is optimized by different tile size. The runtime decides during program's execution which optimized version of the given function is the most appropriate.**

***Index Terms*—Tiling optimization, compiler optimization, Polly compiler, LLVM**

I. Introduction

FROM the beginning of computer science there exists a problem in speed differences between processors and memories. Processors have usually higher frequencies than memories containing data necessary for processor to perform calculations and in result processors spend much time being idle [1]. That is why they have really fast cache memory at their disposal, however because of the cost of such memory is pretty high, its amount is not sufficient. This problem is more widely known as data locality problem and it is crucial for high performance computing, especially during execution of loops through significant amount of data. Nowadays this problem is connected to phenomenon called cache memory miss, which occurs when processor asks for further data and it is not in cache memory. Then computer needs to retrieve data from RAM memory, which is slower than processor speed so from here we have this idle time of processors when they are wasting time [2]. Obvious solution for that is to decrease number of cache misses, so processor can operate without obstacles and data loading from RAM will take place during time when processor will be performing other tasks.

Many solutions and optimizations were proposed to minimize impact of data locality by decreasing cache misses, one of such methods is called tiling. It derived from strip mining transformation, which was invented in times of vector processors. It takes an original loop from program and divides it into smaller ones, called stripes, what on vector processors allowed for vectorization of smaller loops but nowadays it hold almost no improvement for execution speed of programs [3]. Tiling is utilizing the same idea but it is more suitable for modern processors as it enables more possibilities for other improvements and giving some gains on its own. Tiling usually works on loop nests, transforming it into even larger loop nest by adding additional loops to the inside of the nest, increasing loops number twice [4]. However, overall number of iterations remain the same, they are just grouped differently, what already can provide deacrease in cache memory misses [5]. Let's take a look at simple loop nest:

```
for(int i = 0; i < n; i++){
   for(int j = 0; j < n; j++){
      Stmt(i,j);
   }
}
```

This loop nest is transformed by tiling into more complex form:

```
for(int Ti = 0; Ti < n; Ti += 64){
   for(int Tj = 0; Tj < m; Tj += 64){
      for(int i = Ti; i < min(Ti+63, n); i++){
         for(int j = Tj; j < min(Tj+63,m); j++){
            Stmt(i,j);
         }
      }
   }
}
```

As it can be seen the number of iteration is exactly the same but they happen in parts instead of iterating through whole `j` loop and then starting next `i` loop iteration. Number `64` in these loops is called tile size and is very important for efficiency of this optimization as it tries to limit amount of data loaded to cache memory from RAM, so for one iteration processor would have all data it needs to complete calculation, without a need to load additional data. Unfortunately this number have to be optimized for each computer, because many processors are different from each other and have different cache memory configurations.

II. State of the Art

Selection of optimal tile size is a complicated task. There are many factors which should be taken into account. Optimal tile size is dependent not only on target hardware (number of cores, memory organization etc.) but also on source code [6]. It means that almost each loop under compilation should be analysed separately. Moreover it is highly unlikely that there is one optimal tile size of given loop for various target platform. The aim of this section is to present state of the art for optimal tile selection techniques.

A. Static analysis

This approach allows to find optimal tile size selection on the basis of input source code analysis. Compiler tries to find the most appropriate tile size during compilation. There are several approaches. Some approaches tries to find the

D. Adamski, M. Szydłowski, G. Jabłoński and J. Lasoń were with the Department of Microelectronics and Computer Computer Science, Lodz University of Technology, Poland, Wólczańska 221/223, 90-924 Lodz, Poland (e-mail: adamski@dmcs.pl)

most appropriate tile size on the basis of simplified cache architecture [7]. Such solution can be successful for old processors for which there is only one cache level. Unfortunately, the new processors are equipped with multiple levels of cache memory and their influence on overall performance cannot be neglected. Another drawback of this approach is the trade-off between compilation time and model accuracy. The more complicated model the more time is needed for code optimization [8].

Recent research has shown that analytical approach can be used to limit the number of possible tile sizes which can be further examined by empirical tests [6].

B. Dynamic methods

Dynamic search methods require special runtime which can be used for optimal tile selection. Optimized program is run with special runtime which tries to find the best tile size on the basis of the previous execution results. This method does not require any compilation time analysis. The weakness of this approach lies in the large search space. Some heuristics have been proposed to improve search speed, but their effectiveness is limited and they can cause that the obtained result is suboptimal [9].

C. Other approaches

Most projects are focused on rectangular shape of tile. However, there are also some research which were focused on different approaches to tiling optimization. It was proved, hexagonal tile shaping can be beneficial [10].

D. Polly compiler

Polly compiler finds the regions of the code for which the result of code execution is not dependent on the order of executed instructions. Such regions of the code are defined as Static Control Parts (SCoP). Code inside SCoP region can be freely rearrange to maximize data locality. Possibility of free rearrangement of executed statements enables tiling optimization without any data hazard. Polly compiler can perform tiling optimization, but the tile size should be given before start of compilation process [11].

Such approach is not fully optimal. Optimal tile size selection is dependent not only on target processor architecture but also on data accesses inside the SCoPs. Polly compiler does not support parametric tiling. It only allows user to change the default tile size before compilation process. Such approach does not lead always to the best performance hence optimal tile selection is dependent on target architecture and memory calls inside the loops.

E. LLVM infrastructure

Proposed optimizer is working on LLVM IR code. LLVM IR code is an intermediate internal language, which is used for source code analysis and optimization. Its features, like single static assignment rule, greatly expose data dependency and program flow. Thanks to these features the code analysis and optimization is easier than for source code written in high level language [12], [13]. Polly compiler uses LLVM analysis and transformation passes as the framework for its own analysis and optimizations.

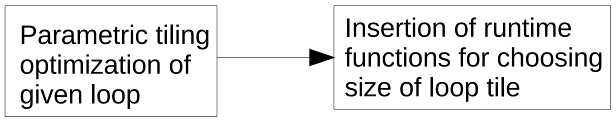
Parametric tiling optimization of given loop
Insertion of runtime functions for choosing size of loop tile

Fig. 1. Generation of function optimized with different tile size

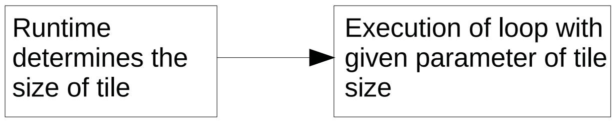
Runtime determines the size of tile
Execution of loop with given parameter of tile size

Fig. 2. Execution of parametrically optimized functions

III. Proposed Approach

In authors' opinion it is worth to focus on dynamic approaches of tiling optimization. This approach is more versatile than static code analysis. The research was focused on polyhedral code analysis. The main aim of the study was to enhance the ability of Polly compiler by adding an option of dynamic tile size selection.

A. Parametric tiling of the loop

Two approaches were examined. The first approach was based on dynamic changes of tiled loop iteration ranges during program execution. This approach requires that loops can be parametrically tiled. The compiler needs to insert callbacks to runtime before every SCoP region. The main aim of runtime callbacks is to determine the execution of the proper tile size during program. This approach allows to search the most optimal tile size between wide range of possible values of tile sizes. Process of insertion and adjusting tile size can be done as one of optimization passes. Figure 1 illustrates this idea. Compiler can parametrically tile loops which are ready for this type of optimization and in the next step it inserts callbacks to runtime. The executable program needs to be linked with the runtime library. Figure 2 shows how tile size is chosen during program execution. The main problem of this approach lies in difficulty of parametric tiling. The research has shown that Polly compiler does not support parametric tiling. The lack of this feature has caused the second approach has been chosen as a proof of concept of the thesis that parametric tiling can have positive impact on code optimization.

B. Separately optimized functions

The second approach tries to omit the problem of modification of Polly compiler. It allows to quickly evaluate if more sophisticated tiling optimization is beneficial. The main simplification is that the optimized function is placed in separate source file. All eligible for tiling loops in such function will be optimized in the same way. Proposed approach generates multiple versions of the optimized functions with

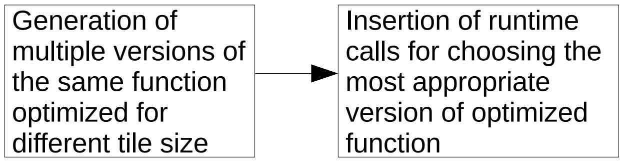
Generation of multiple versions of the same function optimized for different tile size
Insertion of runtime calls for choosing the most appropriate version of optimized function

Fig. 3. Parametric loop tiling compilation process

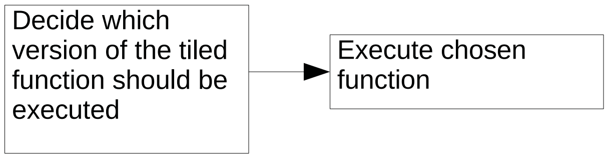
Decide which version of the tiled function should be executed
Execute chosen function

Fig. 4. Parametrically tiled loop execution

different tile size. Compiler inserts special callbacks to the runtime. Runtime routines decides on the basis of historical data which optimized version of the code should be executed. Figures 3 and 4 present the steps of optmization.

The drawback of this method is the limited range where proposed approach can be applied. The functions which should be optimized need to be placed in separate compilation unit. Each unit should be optimized separately. The process of optimization should be performed several times. There is trade-off between number of optimized versions of the functions and the size of binary code. Large number of optimized with different tile size versions of the code allows to choose more appropriate tile size. Unfortunately the large number of available versions of the code prolongs the compilation time and it negatively influences the size of binary code.

IV. Detailed Description

This section provides the detailed description of the implemented solution. It presents the major technologies used in proposed optimizer. The strong and weak points of chosen technologies are presented.

A. Data base

Statistical approach of finding the most appropriate tile size requires a tool for storing results of previous optimization attempts. The knowledge about the past optimization results is required for choosing actual tile size. There is strong probability that the tile size, for which the loop execution was the fastest in the past, can be the most beneficial for consecutive loop execution.

It was decided that the MySQL database should be used for storing the results of previous optimization attempts. This choice ensures high flexibility and it allows to store history of optimization attempts connected with information about the executed loop. This database can be used in further research and verification of the most successful analytical model for the best tile size prediction.

The runtime functions invoke SQL commands to save information about performance of optimized SCoP. Runtime will ask database for previous optimization results. It will need it for making a decision for the most promising tile size.

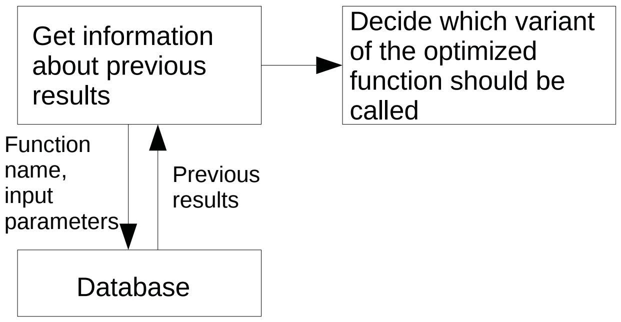
Get information about previous results
Decide which variant of the optimized function should be called
Function name, input parameters
Previous results
Database

Fig. 5. Runtime decision model

B. Insertion of runtime callbacks

New LLVM pass was implemented. The main role of this pass was insertion of runtime callbacks into LLVM IR code. This pass is launched after SCoP detection pass, because it is dependent on the results of the SCoP analysis. New pass operates on the whole translation unit. It inserts the runtime callback after the last instruction before SCoP region and it adds runtime callback before the first instruction after the SCoP region.

C. Runtime execution

Runtime is responsible for recording time of tiled SCoP execution. It uses PAPI library [14] for accurate measurement of time execution. Runtime manages database and it chooses the most appropriate tile size for upcoming SCoP. Current version of runtime is only a prototype and it should be treated as the base for further development. The runtime is entirely written in C and it can be easily replaced by new one. Figure 5 presents high-level model of runtime execution.

V. Test Environment

The proposed solution has been tested with Polybench 4.2.1 beta benchmark. This test suite consists of 30 benchmarks which represent calculations typical for scientific calculations [15]. Tests were performed on Ryzen 5 1600 CPU equipped with 16GB DDRAM4 memory. Its frequency was equal to 2400MHz and the delay equals 14 CL. All benchmarks were compiled by Clang + modified Polly compiler (base version version 3.6). PAPI library was chosen for time measurements. The time of runtime execution is not taken into account, because current runtime is only proof of concept and it should be heavily optimized.

VI. Results

The efficiency of the proposed solution has been measured by compiling and running benchmarks available in Polybench. Firstly benchmarks were tested on small data set. The best tile size was determined for each test case. Later, it was checked if the most beneficial tile sizes for the small data set are also beneficial for large data set. The results are summarized in tables I and II. The final tile size column in tables I denotes the tile size on which runtime algorithm has settled after N

= 40 executions of the test program. Every time result is an average value over those measurements. Relative gain column denotes the relative difference between number of clock cycles needed for polybench item execution optimized with the best and the worst tile size. The last column in table I denotes the relative gain in execution speedup. The last column in table II indicates measured relative slowdown between time of execution of benchmark item optimized with tile chosen for large data set and the tile size, which is the most beneficial for small data set.

VII. Conclusions

Proposed solution proves that adjusting tile size can be profitable. The optimization gain is strongly dependent on the quality of optimization. If the runtime wrongly predict the tile size then the optimization can strongly deteriorate the final performance.

Proposed method of dynamic tiling requires initial set of predefined tile sizes for which it is possible to choose the most profitable tile size. The number of examined tiles should be as wide as possible, because the large number of examined tile sizes increases the chances of choosing the tile size which is the closest to the ideal tile size. On the other hand it is impossible to infinitely increase spectrum of examined tile sizes, because it will cause that binary size of the target application to be infinitely large. Currently there is no implemented method of selection of the initial set of examined tile sizes.

The same tile size was the most beneficial for the small and the large data set for 17 of 28 (61%) test cases. It shows that memory access patterns have more influence on the tile size prediction in comparison to size of the iteration space. Analysis of memory access patterns can be helpful for determination of the most beneficial tile size, but it does not allow to predict the optimization gain. The relative differences between execution times for the worst and the best tile sizes can be different for the large and for the small data sets.

Assumption, that tile sizes for small data sets are also the most beneficial for large data sets, provides wrong results for 9 test cases (32% of all test cases). This group of test cases is not homogeneous.There are minor performance differences for 4 test cases (`correlation`, `doitgen`, `syr2k` and `trisolv`) for small data set. The tile size chosen for these items was slightly better then the next tile size. Unfortunately, this correlation is not valid for the large data set. These benchmarks show the need of improvement of tile selection algorithm. The simple algorithm, which does not analyse access patterns, might propose suboptimal tile size for increased iteration domain.

There is relative small group of benchmark items for which the tile size selection does not provide significant speedup. For 2 out of 28 (`2mm` and `nussinov`) test cases the optimization performance is independent of tile size prediction and the problem size.

TABLE I
Average clock cycles for Polybench benchmarks with small dataset

name	tile size 32 [CLOCK CYCLES]	tile size 16 [CLOCK CYCLES]	tile size 8 [CLOCK CYCLES]	final tile size	relative time gain [%]
2mm	881525	589053	844692	16	49.65
3mm	1504316	1560705	1551890	32	3.75
adi	11934455	10964840	10177789	8	17.26
atax	119318	117850	87604	8	36.20
bicg	99426	94854	87474	8	13.66
cholesky	862194	869092	872902	32	1.24
correlation	988328	881112	884135	16	12.17
covariance	783918	747161	1006149	16	34.66
deriche	935146	933036	931494	8	0.39
doitgen	1234980	1169767	1372282	16	17.31
durbin	44802	44669	44613	8	0.42
gemm	1030774	858323	590559	8	74.54
gemver	128544	138801	106560	8	30.26
gesummv	65189	84920	64148	8	32.38
gram-schmidt	989425	967511	1120346	16	15.80
heat-3d	4029962	4088840	4452858	32	10.49
jacobi-1d	45121	46507	43949	8	5.82
jacobi-2d	2306831	2417012	2678807	32	16.12
lu	1817723	1768361	1939395	16	9.67
ludcmp	1682192	1608699	1616065	16	4.57
mvt	115774	91873	82132	8	40.96
nussinov	3788688	3813246	3759383	8	0.78
seidel	19325449	17464029	13003843	8	48.61
symm	531234	531781	524385	8	1.41
syr2k	755692	742227	843606	16	13.66
syrk	699197	326774	579456	16	113.97
trisolv	35095	36591	35535	32	4.26
trmm	413051	419431	372811	8	12.50

TABLE II
Average clock cycles for Polybench benchmarks with large dataset

name	tile size 32 [CLOCK CYCLES x 1000]	tile size 16 [CLOCK CYCLES x 1000]	tile size 8 [CLOCK CYCLES x 1000]	experi-mental tile size	slow down for small data tile [%]
2mm	5576752	4239220	4826810	16	0.00
3mm	9452857	9518995	9307992	8	1.56
adi	52577094	55136048	51750594	8	0.00
atax	35031	31083	36554	16	17.60
bicg	20659	16371	13467	8	0.00
cholesky	5073177	5040956	5058721	32	0.00
correlation	3404845	2557451	1830694	8	39.70
covariance	3381781	2144597	1796382	8	19.38
deriche	456036	434686	423838	8	0.00
doitgen	1563917	1862553	1636740	32	19.10
durbin	9365	9212	9204	8	0.00
gemm	4383883	3464819	2989667	8	0.00
gemver	66070	44697	43757	8	0.00
gesummv	15473	13716	8825	8	0.00
gram-schmidt	6635160	6630227	6572911	8	0.87
heat-3d	14804893	15051326	15774077	32	0.00
jacobi-1d	8808	8955	8385	8	0.00
jacobi-2d	5919404	6649292	7729609	32	0.00
ludcmp	12467337	12196215	12255536	16	0.00
lu	20410968	19940526	22373147	16	0.00
mvt	36449129	21663824	26985736	16	24.57
nussinov	14170793	14263003	14185980	32	0.11
seidel	70900022	64095754	46675213	8	0.00
symm	5649252	5680972	5564798	8	0.00
syr2k	2582038	3281677	2711786	32	27.10
syrk	2356186	1725509	866231	8	99.20
trisolv	8940	7122	8520	16	25.53
trmm	2092912	2058561	2044500	8	0.00

VIII. Further Improvements

The runtime algorithm of the tile selection should be improved. It should take into account more factors in comparison to currently implemented. Polly compiler enables static code analysis. The results of this analysis should be taken into account during selection of the tile size. It is thought that more input data should improve the quality of tile size prediction. It should be especially profitable for the cases for which simple mean algorithm does not provide valid results for the same access patterns but different iteration domain.

Currently runtime overhead is not taken into account because the main aim of this research was to check new opportunities of the tiling optimization. The next stages of project development should include minimization of runtime overhead. The database calls should be replaced by more effective mechanism of asking for information about past loop execution.

Limitations, which are connected with trade-off between binary size of application and number of examined tile sizes, can be eliminated by implementation of parametric tiling in Polly. Polly should divide iteration space into multiple parametric tiles. The concrete size of the tile will be determined by the runtime callbacks, which will be automatically inserted by optimization passes. As a result, runtime could adjust tile size during program execution instead of choosing the most profitable tile size from predefined set. This approach allows runtime to find the most optimal tile size from wide array of promising solutions without increasing binary size of the application.

References

[1] C. Carvahlo, "The gap between processor and memory speeds," in *Proceedings of the 3rd Internal Conference on Computer Architecture (ICCA'02)*. Department of Informatics, Minho University, 2002, pp. 27–34.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[3] Y. N. Srikant and P. Shankar, Eds., *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. CRC Press, 2007.

[4] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '89. New York, NY, USA: ACM, 1989, pp. 655–664. [Online]. Available: http://doi.acm.org/10.1145/76263.76337

[5] A. LaMarca and R. E. Ladner, "The influence of caches on the performance of sorting," *J. Algorithms*, vol. 31, no. 1, pp. 66–104, Apr. 1999. [Online]. Available: http://dx.doi.org/10.1006/jagm.1998.0985

[6] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar, "Analytical bounds for optimal tile size selection," in *ETAPS International Conference on Compiler Construction (CC'12)*. Tallinn, Estonia: Springer Verlag, Mar. 2012.

[7] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, ser. FOCS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 285–. [Online]. Available: http://dl.acm.org/citation.cfm?id=795665.796479

[8] A. M. Malik, "Optimal tile size selection problem using machine learning," in *Proceedings of the 2012 11th International Conference on Machine Learning and Applications - Volume 02*, ser. ICMLA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 275–280. [Online]. Available: http://dx.doi.org/10.1109/ICMLA.2012.214

[9] M. Rahman, L.-N. Pouchet, and P. Sadayappan, "Neural network assisted tile size selection," in *International Workshop on Automatic Performance Tuning (IWAPT'2010)*. Berkeley, CA: Springer Verlag, Jun. 2010.

[10] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege, "Hybrid hexagonal/classical tiling for gpus," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 66:66–66:75. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544160

[11] T. Grosser, A. Größlinger, and C. Lengauer, "Polly - performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 4, 2012. [Online]. Available: https://doi.org/10.1142/S0129626412500107

[12] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[13] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002, *See* `http://llvm.cs.uiuc.edu`.

[14] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[15] L.-N. Pouchet, "Polybench/C - the Polyhedral Benchmark suite," http://www.cse.ohio-state.edu/ pouchet/software/polybench/, accessed: 2017-09-14.

Dominik Adamski received BSc degree in Telecommunications and Computer Science in 2012 at International Faculty of Engineering, Lodz University of Technology, Poland. As for the master studies in 2013 (also TUL), he focused on compiler construction and source-to-source code transformation. Currently he continues his research on PhD studies in Department of Microelectronics and Computer Science in TUL. He is working on effective code optimization techniques.

Michał Szydłowski obtained his BSc degree in Telecommunications and Computer Science at International Faculty of Engineering, Lodz University of Technology, Poland in 2014. For his master studies, he has been working on the project 'Automatic Adaptive Tile-Size Tuning for LLVM-Polly Compiler' and received MSc title in 2016. Since his Bachelor degree, has also been working as a software developer building scalable web services.

Grzegorz Jabłoński was born in 1970. He received MSc and PhD degrees in electrical engineering from Lodz University of Technology in 1994 and 1999 respectively. He is currently an Assistant Professor in the Department of Microelectronics and Computer Science Lodz University of Technology. His research interests include compiler construction, microelectronics, simulation of electronic circuits and semiconductor devices, thermal problems in electronics, digital electronics, embedded systems and programmable devices.