ADVANCES IN CYBER-PHYSICAL SYSTEMS

Vol. 4, No. 2, 2019

SIMPLE UNIVERSAL TRANSLATOR AS AN ALTERNATIVE COMPILER-COMPILER

Anatoliy Melnyk, Nazar Kozak

Lviv Polytechnic National University, 12, S. Bandery Str., Lviv, 79013, Ukraine.
Authors' e-mail: aomelnyk@lp.edu.ua, nazar.kozak@mail.com

Submitted on 28.11.2019

Abstract: This article deals with ways of how to implement a simple universal translator. Such universal translator may be an alternative to the compiler-compiler.

Index Terms: compiler-compiler, universal translator, Backus–Naur form.

I. INTRODUCTION

When working on automated code generators for graphics accelerators, there is often a need to translate one presentation of data or program code into another. Accordingly, there is a need to develop software tools for these purposes. In most cases, it was only necessary to map the constructions of one high-level language into another, since all parallelization problems are solved separately. Therefore, to simplify the solution of such problems, we decided to develop the universal translator.

Such universal translator is an analogy of the compiler-compiler and actually implements its functionality. The advantage of such translator is its prescription for translating the language constructions of one high-level language into another without the compilation of additional means.

II. OVERVIEW OF EXISTING DECISIONS

Together with the theory of compiler creation [1], the concept of compiler-compiler creation [2, 3] emerged. In practice, compiler-compilers are not as commonly used as compilers themselves. This is primarily due to the fact that a small set of programming languages is used in each specific period of history, so developing automation tools to create compiler development tools seems like an inappropriate task. At the same time, the compiler-compiler implementation tools still exist and can be considered.

Among the existing solutions that can serve as tools for implementing the compiler-compilers there are several software libraries.

First solution can be Sprit [4] from the Boost library. This tool is particularly well suited for writing simple parsers. The scope of this solution is to write text information analysis systems.

Lex [5] is another tool for this purpose. It is well suited for the writing lexical analyzers. It is often used with Yacc [6], which is designed to perform parsing. GNU Bizon is also used for parsing.

All these tools have partial functionality and are often shared.

Completely different class of such tools can be attributed to LLVM (Low Level Virtual Machine) [7]. This solution, unlike the previous ones, has a complete set of tools. Many modern compilers are built on the basis of LLVM. The disadvantage of LLVM is that this solution is tied to its internally represented IR (intermediate representation).

An alternative to such means is further considered universal translator aimed at conversion of one high-level code to another. Such translation will be sufficient, since this translator will serve to display the calculations obtained as a result of high-level synthesis [8]. Such a concept is close to [9, 10, 11, 12] and resembles multifaceted optimization [13]. Such translation can also be used to optimize the code itself [14, 15, 16] or to optimize code execution for a graphical accelerator [17, 18]. Although there are other optimization approaches [19, 20, 21, 22] for which even machine learning is applied [23], such approaches do not use code parallelization.

III. UNIVERSAL TRANSLATOR

Fig. 1 shows the principle of compiler-compiler and universal translator using. Both solutions are used to solve the same problem, but differ in principle. The compiler-compiler allows to generate a compiler program that needs to be compiled further. During generation, configuration files will be used to describe the target language. In the case of the universal translator, this approach is not applied, since the universal translator itself will be the target compiler. The configuration files, which are used in conjunction with the source code (Fig. 1(b)) will be an input for it.

Although compiler-compiler has more optimization capabilities, the universal translator can be used as a single monolithic module of the system. It fits in well with the concept of its use for CUDA automatic code generation systems.

The proposed universal translator is used to translate one high-level representation into another, but can also be used to generate assembler code (Fig. 2). All of the resulting components are integrated with the suite of different computer system programming tools.

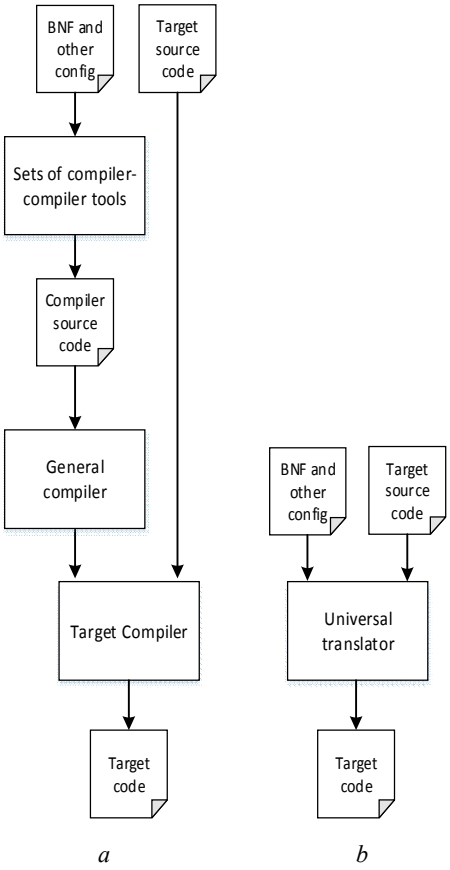

Fig. 1. Compiler-compiler (a) and universal translator (b)

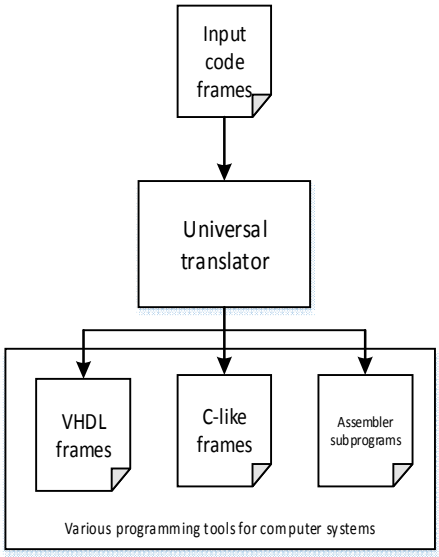

Fig. 2. Components of programming environment

BNF (Backus – Naur form) is often used to express the input language syntax. A BNF specification is a set of derivation rules (Listing 1).

Listing 1

<body_for_true> ::= 'THEN' <statement> ';'	Input-language

As with the compiler-compiler concept, the input code can be specified with BNF notations. The universal translator discussed here has a feature that code generation rules can also be specified using BNF (Listing 2). For this application, two additional attributes are specified indicating the sequence of the rule application.

Listing 2

<body_for_true>.<PRE_RODUCTION> ::= "{" <body_for_true>.<POST_RODUCTION> ::= "}"	C-like
<body_for_true>.<PRE_RODUCTION> ::= <body_for_true>.<POST_RODUCTION> ::= "jmp label%d_point_end;\r\nlabel%d_body_for_false:\r\n"	Assembly

Fig. 3 shows the general structure of the proposed translator. Several intermediate views are used to store the translator's output of each previous step. The peculiarity of the system is using of the same tables for storing input language notation and corresponding notation for output code generation. Since the syntax of the proposed universal translator is based on recursive descent, the stack in explicit form is not used here.

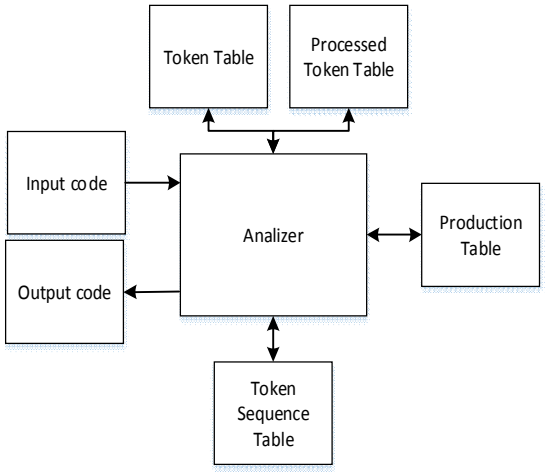

Fig. 3. Structure of the universal translator

At the beginning, the token table initializes the processed token table. Then, when processing the input code, a table of consecutive tokens is filled. Each time one accesses the table, all the identifiers in the tables are indexed starting from the next. This achieves the simplicity of presentation of all syntactic constructions of the input language.

When generating source code, the production table is analyzed. The rules in this table can be of two types. The first type is applied before processing the corresponding syntax, and the second – after.

The recursion depth is provided. The greater the depth of recursion is set, the more one compiler passage is analyzed for language constructions.

IV. UNIVERSAL TRANSLATOR IMPLEMENTING

The structure listed in Listing 3 is used to store notations. This structure stores the name of the tokens, the token attribute, and the set of notation tokens, which also include the attributes.

Listing 3

```
struct SourceNotations{
          char token[MAX_TOKEN_LENGTH_T];
          unsigned int atribute;
          struct {
                    char token[MAX_TOKEN_LENGTH_T];
                    unsigned int atribute;
          } notationTokens[MAX_TOKENS4TOKEN_COUNT_T];
          unsigned int assignationType;
};
```

The following notations may have the attributes of either a sequential rule or rules to select one of the valid values:

- SEQUENCE_PATERN_TYPE_TA
- VARIATIVE_PATERN_TYPE_TA

Notations for code generation also include attributes that indicate the sequence of application of the rules:

- PRE_RODUCTION_FOR_ABSTRACT_TOKEN_TYPE
- POST_RODUCTION_FOR_ABSTRACT_TOKEN_TYPE

Listing 4 gives an example of defining such a notation to describe input grammar.

Listing 4

```
{ "<body_for_true>", ABSTRACT_TOKEN_TYPE |
SEQUENCE_PATERN_TYPE_TA, { { "~~", 0 }, { "THEN",
KEYWORD_TOKEN_TYPE }, { "<statement>", ABSTRACT_TOKEN_TYPE |
REPEAT_TOKEN_ATRIBUTE_TYPE_TA | MANDATORY_TOKEN_ATRIBUTE_TYPE_TA
}, { ";", KEYWORD_TOKEN_TYPE }, { "", 0 } } }
```

Listing 5 shows a similar notation for code generation.

Listing 5

```
{ "<body_for_true>", ABSTRACT_TOKEN_TYPE |
SEQUENCE_PATERN_TYPE_TA | POST_RODUCTION_FOR_ABSTRACT_TOKEN_TYPE,
{ { "~~", 0 }, { "    jmp label%d_point_end;\r\n
label%d_body_for_false:\r\n", 0 }, { "", 0 } } }
```

The Token Table structure (Listing 6) is used to describe the scanned tokens.

Listing 6

```
struct TokensTable {
          struct {
                    unsigned int tokenId;
                    unsigned int atribute;
          } notationTokens[MAX_TOKENS4TOKEN_COUNT_T];
          char * tokenStr;
          unsigned int recordType;
          void * tokenValue;
          unsigned int elementarPoint;
          unsigned int assignationType;
} *tokensTable, *processedTokensTable, *productionsTable;
```

This data structure will be applied three times:

- Token Table array – for input tokens;
- Processed Token Table array – for current processed tokens;
- Production Table array – for tokens to be used for processed data.

Listing 7 shows a structure that will store the sequence of tokens that will display the source code. This table together with the token description table is obtained after the lexical analysis.

Listing 7

```
struct TokensSequenceTable{
          unsigned int tokenId;
          unsigned int row;
          unsigned int column;
          unsigned int scaner_marker;
          unsigned int lexer_marker;
          unsigned int syntaxer_marker;
          unsigned int semantixer_marker;
          unsigned int pragmatixer_marker;
          unsigned int synthesizer_marker;
          unsigned int flags;
          unsigned int error;
          struct {
                    unsigned int tokenId;
                    char * tokenStr;
          } assignedAbstractToken;
} *tokensSequenceTable;
```

As parser work is versatile for parsing and code generation, these records can be stored in a single array.

The production itself is implemented with the function of the prototype listed in Listing 8.

Listing 8

```
void makeProductionOut(char ** productionOut,
struct TokensTable * tokensTable,
          unsigned int tokenId4Patern,
          unsigned int unroll_deep_downcounter,
          unsigned int upStageProductionTokenId,
          unsigned int * endPointFilters);
```

Header generation is implemented separately. Such headers will be different for different target code. Therefore, in general, the code generation process consists of a sequential call for two functions (Listing 9).

Listing 9

```
makeProductionHeaderOut(&productionOutPtr,
                    processedTokensTable,
                    processedProgramPatternId,
                    MAX_CAPTURE_DEEP,
                    0, endPointFilters);
makeProductionOut(&productionOutPtr,
                    processedTokensTable,
                    processedProgramPatternId,
                    MAX_CAPTURE_DEEP,
                    0, endPointFilters);
```

V. APPLICATION

Designed translator is used to present intermediate data in an acceptable form for their use in the system of automatic code parallelization on different links of such system. One of the tasks of such application is using parallel representation of CUDA cores in one implementation and VHDL code in implementation for FPGA.

This universal translator can also be used to generate code that can be used with one of the modern programming languages. The MASM32 assembler was selected as the target code to run such functionality. The script for his work is listed in Listing 10.

Listing 10

```
@echo off
setlocal
..\..\portable_masm32\masm32p\masm32\bin\ML /c /coff %1.asm
if errorlevel 1 goto terminate
rem link /SUBSYSTEM:CONSOLE
/LIBPATH:..\..\portable_masm32\masm32p\masm32\lib %1.obj
..\..\portable_masm32\masm32p\masm32\bin\link
/SUBSYSTEM:CONSOLE
..\..\portable_masm32\masm32p\masm32\lib\msvcrt.lib %1.obj
if errorLevel 1 goto terminate
echo OK
:terminate
endlocal
EXIT %errorlevel%
```

Such application implementation involves the use of the generated code as an add-on software module. This will allow to implement the programming interface for different types of programmable systems. There are various options for integrating such systems based on MASM32. For this purpose, the environment itself has been expanded by the large number of modern programming languages (Fig. 4).

Fig. 4. Set of language tools

For additional functionality, *.bat files that contained the necessary commands to compile the code into the appropriate programming languages were used. See Listing 11 for an example.

Translator has showed correct work for the selected input data (Fig. 5). All stages of the broadcast are worked correctly.

Listing 11

```
setlocal
SET PROJECT_DIR=%CD%
SET BASE_DIR_OFFSET=..\..
SET PATH=%PROJECT_DIR%;%CD%\%BASE_DIR_OFFSET%\ext;%PATH%
SET PATH=%CD%\%BASE_DIR_OFFSET%\MinGW\bin;%PATH%
SET PATH=%CD%\%BASE_DIR_OFFSET%\python\Python27;%PATH%
SET PATH=%CD%\%BASE_DIR_OFFSET%\node\node-v8.9.4-win-x86;%PATH%
SET PYTHON=%CD%\%BASE_DIR_OFFSET%\python\Python27;
xcopy .\* %CD%\%BASE_DIR_OFFSET%\node\node-v8.9.4-win-x86 /y /q
cd %CD%\%BASE_DIR_OFFSET%\node\node-v8.9.4-win-x86
CALL node-gyp rebuild
cd %PROJECT_DIR%
rd node_modules /s /q
rd build /s /q
xcopy %CD%\%BASE_DIR_OFFSET%\node\node-v8.9.4-win-
x86\node_modules\bindings node_modules\bindings\ /s /e /y /q
xcopy %CD%\%BASE_DIR_OFFSET%\node\node-v8.9.4-win-x86\build
build\ /s /e /y /q
pause
endlocal
```

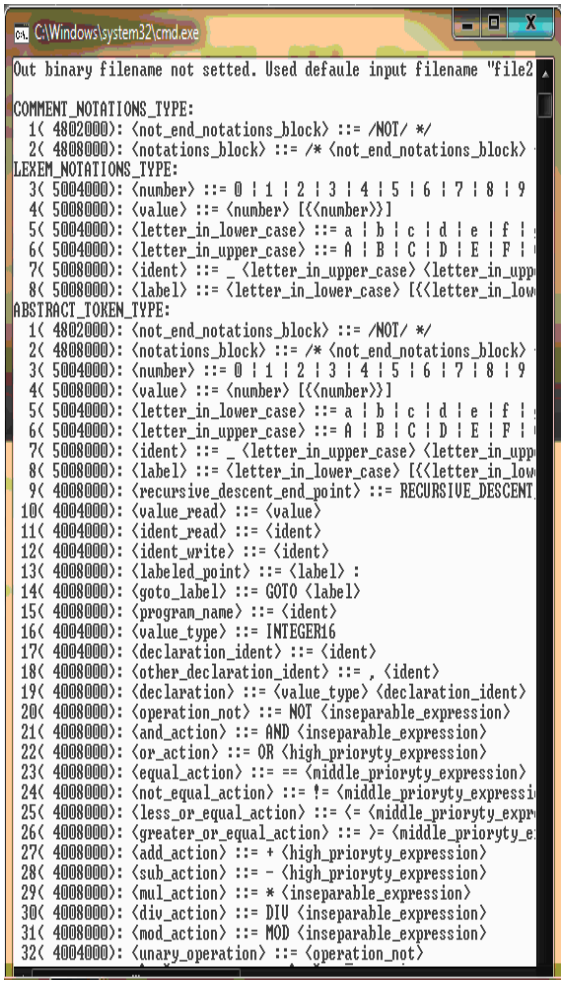
C:\Windows\system32\cmd.exe
Out binary filename not setted. Used defaule input filename "file2
COMMENT_NOTATIONS_TYPE:
1(4802000): <not_end_notations_block> ::= /NOT/ */
2(4808000): <notations_block> ::= /* <not_end_notations_block>
LEXEM_NOTATIONS_TYPE:
3(5004000): <number> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
4(5008000): <value> ::= <number> [{<number>}]
5(5004000): <letter_in_lower_case> ::= a | b | c | d | e | f |
6(5004000): <letter_in_upper_case> ::= A | B | C | D | E | F |
7(5008000): <ident> ::= _ <letter_in_upper_case> <letter_in_upp
8(5008000): <label> ::= <letter_in_lower_case> [{<letter_in_low
ABSTRACT_TOKEN_TYPE:
1(4802000): <not_end_notations_block> ::= /NOT/ */
2(4808000): <notations_block> ::= /* <not_end_notations_block>
3(5004000): <number> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
4(5008000): <value> ::= <number> [{<number>}]
5(5004000): <letter_in_lower_case> ::= a | b | c | d | e | f |
6(5004000): <letter_in_upper_case> ::= A | B | C | D | E | F |
7(5008000): <ident> ::= _ <letter_in_upper_case> <letter_in_upp
8(5008000): <label> ::= <letter_in_lower_case> [{<letter_in_low
9(4008000): <recursive_descent_end_point> ::= RECURSIVE_DESCENT
10(4004000): <value_read> ::= <value>
11(4004000): <ident_read> ::= <ident>
12(4004000): <ident_write> ::= <ident>
13(4008000): <labeled_point> ::= <label> :
14(4008000): <goto_label> ::= GOTO <label>
15(4008000): <program_name> ::= <ident>
16(4004000): <value_type> ::= INTEGER16
17(4004000): <declaration_ident> ::= <ident>
18(4008000): <other_declaration_ident> ::= , <ident>
19(4008000): <declaration> ::= <value_type> <declaration_ident>
20(4008000): <operation_not> ::= NOT <inseparable_expression>
21(4008000): <and_action> ::= AND <inseparable_expression>
22(4008000): <or_action> ::= OR <high_prioryty_expression>
23(4008000): <equal_action> ::= == <middle_prioryty_expression>
24(4008000): <not_equal_action> ::= != <middle_prioryty_expressi
25(4008000): <less_or_equal_action> ::= <= <middle_prioryty_expr
26(4008000): <greater_or_equal_action> ::= >= <middle_prioryty_e
27(4008000): <add_action> ::= + <high_prioryty_expression>
28(4008000): <sub_action> ::= - <high_prioryty_expression>
29(4008000): <mul_action> ::= * <inseparable_expression>
30(4008000): <div_action> ::= DIV <inseparable_expression>
31(4008000): <mod_action> ::= MOD <inseparable_expression>
32(4004000): <unary_operation> ::= <operation_not>
```

*Fig. 5. Processing input code*

The possibility of the parallel code using with other programming languages is considered. It allows its integration into almost any programming system.

## VI. CONCLUSION

During the research, a universal translator was developed. Such translator with the help of appropriate notations allows to realize arbitrary transformation of the intermediate representation.
```

In general, we can distinguish the following properties:

- the input language can be set using BNF;
- code generation can also be set using BNF;
- it is possible to generate an intermediate data;
- it is possible to re-optimize the elements.

REFERENCES

[1] Aho, Sethi, Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986. ISBN 0-201-10088-6.

[2] Peter Mosses. SIS: A Compiler-Generator System Using Denotational Semantics, Report 78-4-3, Dept. of Computer Science, University of Aarhus, Denmark, June 1978.

[3] C. Stephen Carr, David A. Luther, Sherian Erdmann, The TREE-META Compiler-Compiler System: A Meta Compiler System for the Univac 1108 and General Electric 645, University of Utah Technical Report RADC-TR-69-83.

[4] Spirit 2.5.5. https://www.boost.org/doc/libs/1_67_0/libs/spirit/doc/html/index.html 12.12.2019.

[5] Lesk, M.E.; Schmidt, E. "Lex – A Lexical Analyzer Generator". Retrieved August 16, 2010.

[6] Levine, John R.; Mason, Tony; Brown, Doug (1992). lex & yacc (2 ed.). O'Reilly. pp. 1–2. ISBN 1-56592-000-7.

[7] The LLVM Compiler Infrastructure Project. Retrieved March 11, 2016.

[8] Melnyk, A., Salo, A., Klymenko, V., Tsyhylyk, L. Chameleon – system for specialized processors high-level synthesis, Scientific-technical magazine of National Aerospace University "KhAI", Kharkiv, 2009. No. 5, P. 189–195.

[9] S.-I. Lee, T. Johnson, and R. Eigenmann. Cetus – an extensible compiler infrastructure for source-to-source transformation. In Proc. Workshops on Languages and Compilers for Parallel Computing, 2003.

[10] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2009.

[11] Y. Liu, E. Z. Zhang, amd X. Shen. A Cross-Input Adaptive Framework for GPU Program Optimization. In Proc. IEEE International Parallel & Distributed Processing Symposium, 2009.

[12] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories. In Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008.

[13] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral mode: part I, on dimensional time. In Proc. International Symposium on Code Generation and Optimization, 2007.

[14] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, in Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 2008, p. 4.

[15] J. Ansel, Y. L. W. ans Cy Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms, in The International Symposium on Code Generation and Optimization, ser. CGO '11, 2011.

[16] J. Kurzak, H. Anzt, M. Gates, and J. Dongarra. Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs, IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 7, 2016.

[17] A. Magni, C. Dubach, and M. O'Boyle. Automatic optimization of thread-coarsening for graphics processors, in Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, ser. PACT '14, 2014, pp. 455–466.

[18] S. Unkule, C. Shaltz, and A. Qasem. Automatic restructuring of GPU kernels for exploiting inter-thread data locality, in Proceedings of the 21st International Conference on Compiler Construction, ser. CC'12, 2012, pp. 21–40.

[19] Y. Yang, P. Xiang, J. Kong, M. Mantor, and H. Zhou. A unified optimizing compiler framework for different gpgpu architectures, ACM Trans. Archit. Code Optim., vol. 9, no. 2, pp. 9:1–9:33, 2012.

[20] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation, in Proceedings of the International Symposium on Code Generation and Optimization: Feedback directed and Runtime Optimization, ser. CGO '04, 2004.

[21] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space, in Workshop on Profile and Feedback-Directed Compilation, 1998.

[22] P. M. Knijnenburg, T. Kisuki, and M. F. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation, The Journal of Supercomputing, vol. 24, no. 1, pp. 43–67, 2003.

[23] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus the iterative optimization, in Proceedings of the International Symposium on Code Generation and Optimization, ser. CGO '06, 2006, pp. 295–305.

Anatoliy Melnyk has been a Head of Computer Engineering Department at Lviv Polytechnic National University since 1994. He graduated from Lviv Polytechnic Institute with the Engineer Degree in Computer Engineering in 1978. In 1985 he obtained his Ph.D in Computer Systems at Moscow Power Engineering Institute. In 1992, he received his D.Sc. degree at the Institute of Modelling Problems in Power Engineering of the National Academy of Science of Ukraine. He was recognized for his outstanding contributions into high-performance computer systems design as a Fellow Scientific Researcher in 1988. He became a Professor of Computer Engineering in 1996. From 1982 to 1994 he was a Head of Department of Signal Processing Systems at Lviv Radio Engineering Research Institute. From 1994 to 2008 he was a Scientific Director of the Institute of Measurement and Computer Technique at Lviv Polytechnic National University. From 1999 to 2009 he was a Dean of the Department of Computer and Information Technologies at the Institute of Business and Perspective Technologies, Lviv, Ukraine. Since 2000 he has served as a President and CEO of Intron ltd. He has also been a professor at Kielce University of Technology, University of Information Technology and Management, Rzeszow, University of Bielsko-Biala, John Paul II Catholic University of Lublin.

Nazar Kozak was born in 1985 in Ukraine. He received the B.S. and the M.S. degrees in computer engineering at Lviv Polytechnic National University in 2007 and 2008. He has been doing scientific and research work since 2008. His work resulted in 13 publications. Currently, he is an assistant proessor at the Computer Engieering Deparment, Lviv Polytechnic National University.