AUTOREFERAT (załącznik w języku angielskim)

1. Name, surname:

Marek Pałkowski

2. Diplomas and scientific degrees:

6th July 2004 – MSc Eng. at the Faculty of Computer Science of the Technical University in Szczecin. Thesis entitled *Methods of distributed applications development*,

20th January 2009 – *PhD* at the Faculty of Computer Science of the West-Pomeranian University of Technology in Szczecin. Thesis entitled *Algorithms for increasing the parallelism extraction in program loops*,

3. Employment:

From 1st April 2009 to 31st December 2009 – *research assistant* at the Department of Software Engineering, Faculty of Computer Science of the West-Pomeranian University of Technology in Szczecin.

Since 1st January 2010 – *assistant professor* at the Department of Software Engineering, Faculty of Computer Science of the West-Pomeranian University of Technology in Szczecin.

Since 1st October 2014 *head* of Programming Techniques Section.

4. Series of publications: **„Techniques based on the transitive closure of the dependence graph for creating compilers automatically optimizing source code"**

Papers for consideration (in chronological order)

1. Anna Beletska, Włodzimierz Bielecki, Albert Cohen, Marek Pałkowski, Krzysztof Siedlecki, *Coarse-grained loop parallelization: Iteration Space Slicing vs affine transformations,* 2011, Parallel Computing, 37(8), pp. 479-497. **IF:** 1.311, **Ministerial Points:** 30, **Own contributions:** 30%.

Palli

2. Włodzimierz Bielecki, Marek Pałkowski, Tomasz Klimek, *Free scheduling for statement instances of parameterized arbitrarily nested affine loops,* 2012, Parallel Computing 38(9), pp. 518-532. **IF:** 1.214, **Ministerial Points:** 30, **Own contributions:** 33.3%.
3. Marek Pałkowski, Włodzimierz Bielecki *TRACO: Source-to-Source Parallelizing Compiler*, 2016, Computing and Informatics 35(6), pp. 1277-1306. **IF:** 0.488, **Ministerial Points:** 15, **Own contributions:** 70%.
4. Włodzimierz Bielecki, Marek Pałkowski, *Tiling arbitrarily nested loops by means of the transitive closure of dependence graphs*, 2016, International Journal of Applied Mathematics and Computer Science 26(4), pp. 919-939. **IF:** 1.420, **Ministerial Points:** 25, **Own contributions:** 50%.
5. Tomasz Klimek, Marek Pałkowski, Włodzimierz Bielecki, 2016, *Synchronization-Free Automatic Parallelization for Arbitrarily Nested Affine Loops*, wydawca: IEEE, International Symposium: Computer Architecture and High Performance Computing Workshops (SBAC-PADW), Los Angeles, CA, USA. **Ministerial Points:** 15 (Web of Science), **Own contributions:** 33.3%.
6. Marek Pałkowski, Włodzimierz Bielecki, 2016, *An Iteration Space Visualizer for Polyhedral Loop Transformations in Numerical Programming,* Annals of Computer Science and Information Systems, Vol. 8, str. 705-708, wydawca: IEEE, (FedCSiS 2016), Gdańsk. **Ministerial Points:** 15 (Web of Science), **Own contributions:** 70%.
7. Marek Pałkowski, Włodzimierz Bielecki *Parallel tiled code generation with loop permutation within tiles,* 2017, Computing and Informatics 36(6), pp. 1261-1282. **IF:** 0.488, **Ministerial Points:** 15, **Own contributions:** 60%.
8. Marek Pałkowski, Włodzimierz Bielecki, *Parallel tiled Nussinov RNA folding loop nest generated using both dependence graph transitive closure and loop skewing*, 2017, BMC Bioinformatics 18:290, pp. 1-10 (open-access). **IF:** 2.435, **Ministerial Points:** 35, **Own contributions:** 70%.
9. Włodzimierz Bielecki, Marek Pałkowski, Piotr Skotnicki *Generation of parallel synchronization-free tiled code*, 2017, Computing, Springer Vienna, ISSN=1436-5057, DOI: 10.1007/s00607-017-0576-3. **IF:** 1.589, **Ministerial Points:** 25, **Own contributions:** 33.3%.

Pałk

10. Marek Pałkowski, Włodzimierz Bielecki, *Tuning Iteration Space Slicing based tiled multi-core code implementing Nussinov's RNA folding*, 2018, BMC Bioinformatics 19:12 (open-access) **IF:** 2.435, **Ministerial Points:** 35, **Own contributions:** 70%.

Almost two decades ago, accelerating the speed of computer processors has been significantly limited by physical constraints such as exponential increase of the number of transistors, heat emission and miniaturization problems. Hence, the hardware manufacturers have concentrated on multi-core platform development. Shortly thereafter, parallel computers such as graphics cards for general purpose computing and multi-core coprocessors have appeared. Numerous computational centers dealing with parallel processing for various computer simulations were appeared, for example the Polish research centers in Krakow (Cyfronet), Poznań (PSNC) or Gdańsk (CI Task)[1]. As a result, the interest of researchers and engineers was focused on constructing optimal programming code using available machine power. The concept of code optimization is understood as parallelization and locality improvement. It is a special task to build compilers automatically translating the original code into a faster and semantically correct form without a significant developers' involvement.

Most of calculations are in iterative constructs, so optimizing compilers implement many efficient algorithms for program loop transformations. Automatic loop transformations are not trivial tasks due to the variety of program codes. Various mathematical tools are used in these techniques, and their effectiveness is rated with program loop applicability, speed-up (computing time reduction), scalability (speed-up growth with increasing the number of threads or problem size) and code locality (increasing cache usage).

The most popular and very effective solution is the affine transformations framework (ATF) used for program loop optimization. ATF is currently the most advanced parallel extraction technique [Feautrier92_1, Feautrier92_2, Lim94, Bondhugula08, Verdoolaege13, Bondhugula16]. It combines a large class of transformations such as interleaving, inversion, fusion, permutation, and finds parallelism for shared-memory and distributed systems. The main idea of ATF is to extract and apply an affine transformation of program loops in such a manner that dependent statement instances belong to the same space partition or belong to different time partitions containing independent statement instances. An important advantage of ATF is therefore the application of one mathematical method to many transformations.

However, classical ATF algorithms do not allow for extraction of all existing parallelism and/or improvement of code locality in the general case of the program loop nest. Limitations can be especially observed for loops with the irregular dependence graph[1, 8, Mullapudi14, Wonnacott13][2].

The polyhedral model is a dominant technique of optimizing compilers development. The iteration space, loop boundaries, table references, and conditional expressions are described by affine expressions in this model. The polyhedral model allows for implementing three steps:

---

[1] The national centers are listed in the top two hundred of the top500.org.

[2] References from the publication cycle are numbered, while references from the bibliography (placed at the end of this document) are marked with a first author surname and a year.

Pałk

- loop dependence analysis,
- loop transformations,
- code generation.

The authorial discussed series of publications presents a number of authorial algorithms for program loop transformations by means of the dependence graph transitive closure without finding any affine functions. These algorithms are an extension of the theory of iteration space slicing; they extend the existing scope of application and have been implemented in the authorial TRACO compiler. This tool implements all three components of the polyhedral model. Dependence analysis and code generation are performed using the same tools as in related approaches. Hence, the originality and innovations of discussed authorial results consist in proposed techniques of transformation of program loops. The main characteristic and combined feature of all authorial algorithms is the calculation and usage of the transitive closure of relations describing all dependences in program code. The primary purpose of the research is to derive code transformations and its acceleration capability on modern multi-core platforms when related solutions, including ATF techniques, fail to optimize code or generate code with worse quality. The presented series of publications is a continuation and a strong development of research started in my dissertation on algorithms parallelizing the program code. The presented series of publications is a continuation and a strong extension of the research started in my Ph.D dissertation on algorithms parallelizing program code.

The significant authorial contribution in the discipline of computer science includes the extension of the theory of developing optimizing compilers using the transitive closure of dependence graphs allowing us to increase the applicability of optimizing compilers; the implementation of proposed solutions in the open source TRACO compiler and demonstrating the effectiveness of implemented algorithms for real-life applications and benchmarks that perform numerical calculations and simulations in real-life areas as dynamic programming, bioinformatics, computational fluid mechanics, or linear algebra.

Representation of loop dependences and the transitive closure of dependence graph

Presented algorithms are based on Presburger arithmetic, which is an axiomatic system of natural numbers with addition, also known as Peano arithmetic, without multiplication. The arithmetic language contains binary values of 0 and 1, and adding as the binary function + [Fisch74].

In the discussed series of publications, loop dependences are represented with relations [Pugh93] whose constraints are built of Presburger formulas. Relations combine two sets including loop statement instances and constraints by means of which the first set is mapped to the second and the $\rightarrow$ symbol. A set can represent the whole iteration space, a subset of statement instances, or tiles. Such a representation is concise of brief and independent of a target architecture or programming language. Furthermore, relations are used to represent parameterized dependence graphs (related approaches use the synonymous matrix form).

In order to analyze and transform programs, sets and relations are used whose linear constraints over integer variables are joined by the logical negation ($\neg$), conjunction ($\wedge$), alternate ($\vee$), and exists ($\exists$) operators. In the various stages of proposed algorithms,

operations are performed on sets and relations such as intersection ($\cap$), union ($\cup$), difference (-), domain, range, composition ($\circ$), relation application on set. The fundamental operation is the transitive closure of relations representing dependences in program loops. Transitive closure is an operation derived from the graph theory and represents all transitive paths between vertices in a directed graph. A relation is a mathematical representation of the directed graph, so the relation transitive closure describes the graph transitive closure and vice versa. The exact transitive closure of the relation is defined as the infinite number of the unions of the power k relations [Kelly96]. The power k of the relation is the k-fold composition of this relation [Verdoolaege10]. In general, we have to calculate transitive closure for a parameterized graph. In practice this means that parametric variables are constant (for example its boundaries) but they are not known at compile time.

William Pugh et al. from the University of Maryland first proposed an algorithm for calculating the transitive closure of parameterized relations representing dependence graphs [Kelly96]. Although, there is still no universal algorithm for calculating exact transitive graph closures for the parameterized relation, current approaches [Verdoolaege10, Bielecki10, Bielecki14] allow us to find the exact closure of the dependence graph for a wide range of program loops, and consequently to apply transformations without applying affine transformations. Additionally, studies on the improvement of the transitive closure design or its approximation are a current topic of research [Verdoolaege11, Feautrier12, Feautrier15]. The algorithms aimed at calculation of transitive closure are out of the scope of this publication series.

Limitations of Affine Transformations

ATF techniques are now recognized as the most effective methods for automatic code optimization and are implemented in such state-of-the-art and related compilers, e.g. Pluto [Bondhugula08], Pluto+ [Bondhugula16], PTile [Baskaran10], POCC [Park11], PPCG [Verdoolaege13]. In those compilers, the same mathematical tool is used to extract parallelism and improve code locality by tiling and permutation of program loops. The ATF approach is implemented as a complex sequence of transforms that change the execution order of statement instances of program loops. ATF techniques guarantee correct code execution by honoring all dependences, i.e., each dependence source is executed before the corresponding dependence destination. The result of the affine mapping function is the information about the assigned processor number, logical time, and new iteration for all instances of the statement, while the affine function is the input to a code generator.

In the general case, the limitation of ATF is the lack of solutions for the affine set of qualities and inequalities, which are required to find proper affine transformations. Factors limiting this method are mainly non-uniform dependences, the occurrence of negative coefficients in distance vectors, or cycles in the graph representing inter-tile dependences. The disadvantages of classical ATF algorithms are described in [Feautrier92_1, Feautrier92_2, Bondhugula16, Verdoolaege13, Wonacott13] and in this series of publications.

Paper [1] demonstrates that ATF fails to extract slices available in particular loop nests, ATF is not able to extract all slices available in a loop and ATF does not allow for the extraction of synchronization-free slices with multiple ultimate dependence sources.

Pallli

Paper [2] explains that parallelism extracted with ATF transforms contains more synchronization points than required, i.e. no whole parallelism can be extracted (not all independent statement instances are available at a given logical time).

Paper [4] shows examples that cannot be tiled by means of ATF algorithms. Whereas, paper [8] shows dynamic programming applications (DP) for which the innermost loop nests cannot be tiled with ATF; such a possibility is a key to achieve satisfactory acceleration of computation. These constraints are involved by an irregular graph of dependences with inter-tile dependences cycles.

Comparisons of ATF with proposed algorithms are presented in theoretical and practical ways, i.e., by analyzing generated code by means of the author's compiler and related tools based on ATF.

Parallelism extraction based on the transitive closure of the dependence graph

Iteration Space Slicing (ISS) was introduced by Pugh and Rosser [Pugh97] as an extension of the program slicing proposed by Weiser [Weiser84]. Coarse-grained parallelism is represented with synchronization-free slices or with slices requiring occasional synchronization.

Pugh and Rosser applied iteration space slicing to optimize inter-process communication. They showed, in particular, how extraction of slices allows for program loop fusion, tolerance of communication delays and message merging. However, the authors did not present automatic and tiled iteration space slicing by means of the transitive closure of dependence graphs. The dependence analysis proposed by Pugh and Wonnacott, and implemented in the Petit tool [Pugh93], is used to implement the algorithms presented in this publication series. Dependences, presented by means of relations, are input data for author's iterative space slicing algorithms and allow us to derive loop transformations that guarantee respecting all dependences in target code.

A key operation is to compute the transitive closure of the union of all dependences. If it is not possible to find an exact closure, it is still possible to approximate it. This means that calculated transitive closure will include false (not existing) paths (dependences). Generated code will be still valid but less optimal [Verdoolaege10].

A slice is the maximal sub-graph in the dependence graph, which does not have any undirected path with another slice. Because the dependence graph is a directed graph, all dependence relations should be enlarged with their inversions; in this way we get an undirected dependence graph [Pugh97].

The topology of a slice can be a chain, tree, or general graph (multi-income edge graph). Pugh and Rosser did not explain how to automatically find sets of representative (lexicographically minimal) statement instances in slices and their transitive dependent statement instances, and how to generate code based on these sets. However, their team has proposed many useful models for determining transformation and code optimization by means of Presburger arithmetic and the Omega library [Kelly95].

In paper [1], we show how to extract slices and generate synchronization-free parallel code using operations on sets and relations.

If a loop dependence graph represents only one independent slice, then compilers apply the scheduling of the statement instances in a time by means of ATF. Uday Bounghdula

from the Ohio State University has developed the state-of-the-art Pluto compiler which is able to parallelize a wide spectrum of program loops by means of classical ATF transformations, like pipelining or loop skewing. However, techniques implemented in Pluto [Allen01, Wolfe95, Bacon93, Banerjee93, Feautrier92_1, Feautrier92_2] fail to find free schedules [Darte00], i.e., fail to extract maximal fine-grained parallelism in the general case [2].

Wonnacott discovered that there is a lack of maximum parallelism in Pluto tiled code[Wonnacott13]. Recently, Pluto has been extended with tiling techniques to maximize parallelism for stencil computations [Bondhugula17].

Paper [Bielecki03] presents a wide range of program loops which can be parallelized by means of free-scheduling and Presburger arithmetic, even if there is no possibility to apply affine mappings. However, the approach is limited to only unparameterized loops.

In paper [2] of this series, a free scheduling method is proposed for parameterized loops with regular and irregular dependences. Using transitive closure and the power k of relations, it is shown that it is possible to find maximal parallelism (simultaneous execution of parallel statement instances as soon as all their operands are available) for program loops that cannot be parallelized using classical transformations.

Code locality improvement by means of loop tiling

Loop tiling or blocking groups statement instances into smaller blocks to increase reuse of cache memory. This is one of the basic transformations that improves the locality of the program code. In parallel processing, blocks are indivisible macro-instructions and they coarse the granularity of code. The early work on tiling [Irigoin88, Wolf91] and new advanced methods [Bondhugula08, Griebl04, Wonnacott13] are based on classical affine transformations techniques implemented in Pluto and other tools (sparse [Strout04], nonlinear polyhedral [Kim09] and iteration space models [Pugh97]). The most advanced classical methods are based on affine transformations and implemented in the state-of-the-art Pluto compiler. In the Pluto+ tool [Bondhugula16], ATF was combined with index set splitting techniques [Griebl00] to extend tiling for periodic iteration spaces.

However, classical affine transformations for loop tiling are limited for some loop classes. In [Mullapudi14, Wonnacott15], it has been shown that ATF does not tile loops when there are cycles in the inter-tile dependence graph (such cycles prevent finding the scheduling of blocks [Feautrier92_1, Feautrier92_2]) or produces code with limited scalability (no parallelism or a small degree of parallelism at start-up). The second limitation was solved by changing the shape of the tile from rectangular to diamond or trapezoidal for stencil computations [Grosser14]. These programs or benchmarks, such as Gauss-Seidel or Jacobi's algorithms of linear algebra [Pouchet15], require only a constant number of adjacent cells for computing a data cell at a given time. However, these solutions are useless when there is irregularity in the dependence graph, for example, when a whole row and a whole column of previously calculated cell values of the same array are required to calculate a given cell. It leads to non-uniform dependences and numerous inter-tile dependence cycles in the graph that limit such techniques as ATF-based diamond tiling [Bondhugula17]. Examples of such dependence patterns are present in applications of dynamic programming algorithms [8].

Paper [4] presents a novel approach to tile program loops by means of the transitive closure of dependence graphs. If it is possible to compute an exact or approximated transitive closure of the union of all dependence relations, the compiler generates code without cycles in

the inter-tile dependence graph. In order to eliminate cycles, statement instances within a tile, which originate cycles are moved to a lexicographically greater tile. In paper [4], the correctness of this approach has been proved, whereas the paper [Palkowski15] presents how to parallelized tiled code using techniques based on iteration space slicing and free-scheduling [1,2] and other classical techniques such as unimodular or affine transformations [6].

Additionally, the effectiveness of the method presented in paper [4] is demonstrated for loop cases when affine transformations fail to tile loops or to tile all loops in band (e.g., in case of dynamic programming applications), affine mapping is not able to form diamond, trapezoidal or hexagonal tiles, and loop permutation cannot be applied to improve code locality. The presented technique combines the features of the polyhedral model and the iteration space slicing framework. Automation of tiling algorithms was performed in the implementation of the author's compiler [3].

Technical contributions

Algorithm implementation was performed in the author's TRACO compiler (acronym for TRAnsitive ClOsure) [Traco2017]. An analysis of source code and generation of optimal code are not possible without an automatic tool, even for small programs. Automation of transformations is therefore necessary due to the high degree of complexity of parameterized arbitrary nested program loops.

TRACO is a source-to-source compiler under the GPL license. Hence, the compiler contains appropriate modules for a structural analysis of program code, i.e., pre-processing of source code and post-processing of output code to a compilable form. Between these modules are placed compilation modules defined by the stages realized within the polyhedral model, i.e., dependence analysis, loop transformations, and code generation. This modular design can be found in related compilers, such as Pluto and Pluto+.

The Petit [Pugh93] and Pet [Verdoolaege12] tools are used for carrying out dependence analysis. The modern and constantly being developed Integer Set Library (ISL) by Sven Verdoolaege [Verdoolaege10] is used to perform operations on sets and relations within the scope of Presburger arithmetic. The ISL and Cedric Bastoul's tool, Cloog [Bastoul04], are adopted to generate code. The same libraries are used in related solutions such as Pluto, Pluto+, and PCGG. The original element of TRACO is therefore a transformation module based on transitive closure.

The TRACO sources contain about 100 thousands lines of code (without third-party components and examples). TRACO is written in C / C ++ and Python mostly by me. TRACO is dedicated to Linux systems. The structure and construction of the compiler is explained in detail in [3]. TRACO is an open source tool (traco.sourceforge.net).

The most comparable tool for TRACO, in terms of code optimization, is the Pluto compiler, which implements ATF techniques in a transformation module. Other tools like Cetus and Par4All do not offer code locality improvements like loop tiling. Among commercial compilers, the Intel tool, ICC, is most popular whose loop transformation capability is less than that of Pluto and TRACO. However, it generates fast binary code with hardware optimization and vectorization and can be used to compile source codes generated by TRACO [3].

TRACO transforms program loops within the polyhedral model and C/C ++ syntax. The compiler has been practiced in many programs in the following areas: physics simulation,

Palkowski

computational fluid dynamics, linear algebra, image and signal processing, dynamic programming (combinatorial optimization), and others. In the initial phase of the algorithm design, artificial codes were used with the target dependence patterns. In the main part of experimental study, real-life programs and benchmarks (static control flow is embedded in many real-life codes) were practiced.

Results within the ACM classification

In the actual ACM classification, the results obtained belong to the following fields and subdivisions[3]:

- *Software and its engineering* → *Compilers* (High)
- *Software and its engineering* → *Parallel programming languages* (High)
- *Software and its engineering* → *Automatic programming* (High)
- *Computing methodologies* → *Parallel computing methodologies* (High)
- *Theory of computation* → *Parallel computing models* (High)
- *Theory of computation* → *Scheduling algorithms* (High)
- *Theory of computation* → *Parallel algorithms* (High)
- *Mathematics of computing* → *Graph theory* (High)
- *Theory of computation* → *Dynamic programming* (High)
- *Applied computing* → *Bioinformatics;* (High)
- *Applied computing* → *Computational Biology* (High)
- *Computing methodologies* → *Massively parallel and high-performance simulations* (Medium)
- *Software and its engineering* → *Multithreading* (Medium)
- *Software and its engineering* → *Massively parallel systems* (Medium)
- *Computing methodologies* → *Linear algebra algorithms* (Medium)
- *Computing methodologies* → *Modeling and simulation* (Medium)
- *Hardware* → *Digital signal processing* (Medium)

Conclusions

The main achievements presented in the series of publication are as follows:

- extension of the iteration space slicing theory [1,2,4,5,7-10],
- formalizing an algorithm for synchronization-free slices extraction with the arbitrary topology of the graph dependence and parallel code generation [1,5,9],
- applying the power k of dependence relations to determine free scheduling and apply it to parallel code generation [2],
- an algorithm for loop tiling based on the transitive closure of dependence graphs [4,7-10],
- demonstrating the greater potential of the proposed techniques for optimizing program loops in dynamic programming applications [8,10],

[3] The appropriate relevance ("High", "Medium", or "Low") is given in brackets.

- an approach of automatic optimization of Nussinov code for RNA folding allowing for higher performance in comparison to related solutions [8,10],
- a novel combination of blocking algorithms based on transitive closure with classical ATF algorithms for scheduling [4,7,8,10],
- demonstration of affine transformations limitations and presentation of possible solutions based on the theory of parameterized graphs [1-10],
- development of the TRACO compiler [1-10].

In this series of publication, the topics have been analyzed that had so far been rarely exploited by the scientific community of automatic programming and iteration space slicing.

The issue of code optimization based on transitive closure has been started in the co-operation with researchers from the Inria Institute (France) and the Milan Polytechnic. I hope to further interest in the international scientific community with the latest topics of automatic program loop optimization with irregular graphs of dependences [4, 9], such as those found in dynamic programming applications [8] and models with tile size selection [10]. The papers of this series have been cited by well-known experts in the field of optimizing compilers, e.g., Michelle Mills Strout (Univ. of Arizona), David Wonnacott (Havefort College, Philadelphia), Paul Kelly (Imperial College London), Sven Verdoolaege (KU Leuven, Belgium), Albert Cohen (Inria), Louis-Noel Pouchet (Univ. of California) and Sanjay Rajopadhye (Colorado State University).

The presented series includes innovative solutions and applications in the field of computer science, more specifically from compiler theory (automatic code optimization and code generation), parallel processing, programming techniques and graph theory.

The publication cycle completes the single-subject list presented in points II A) and E) of the science achievement list (43 publications together with the publication cycle). The total ministerial score is 537, of which the own contribution of the author of the summary is 282.3. The list includes primarily conference papers and one monograph [Bielecki11].

The scientific and technical contributions presented in individual papers of the publication series are summarized in the following part of this document.

1. *Coarse-grained loop parallelization: Iteration Space Slicing vs affine transformations*

This paper presents algorithms for the extraction of independent code fragments, i.e., synchronization-free and coarse-grained parallelism. Generated parallel code is effective on multicore and distributed architectures with the high costs of synchronization and communication, respectively.

The algorithms for iteration space slicing extract the slices from dependence graph, D, as its weakly connected components. In other words, a slice is the maximum subgraph of graph D which is not connected to any another one with any undirected or directed path.

The ultimate dependence sources of dependences are statement instances that are the sources of the dependences and not the destinations of the dependences.

The representative (source) of the slice is its lexicographically minimal source, i.e., the lexicographically minimal statement instance of all the statement instances belonging to this slice.

Synchronization-free slices extracting requires a union of all dependence relations in a preprocessed form for arbitrary nested loops.

Pallli

To extract coarse-grained parallelism represented with synchronization-free slices, we need to carry out the following steps:

- find a set of representative sources of slices;
- reconstruct slices from their representatives and generate code scanning these slices.

The paper presents the method of distinguishing the topology of a graph: a chain, tree, or general graph. All ultimate sources are representatives for chains and trees. In the general graph, first a relation connecting ultimate sources of a single slice should be formed. Then such a relation is used to extract a single slice representative, which is the lexicographically smallest ultimate dependence source of a slice. It is also possible to find slice representatives by means of the non-directed dependence graph and the transitive closure of the original and inverse dependence relations [5].

Finding all statement instances of a slice is performed by means of applying a graph closure to slice representatives. Code generation includes two steps. First, a parallel loop scanning representative sources is generated. Next, its tuples are combined with sequential inner loops scanning all statement instances of slices. Two algorithms are given in this paper allowing for defining all slice statement instances in an affine form using a counter *for* loop and in a non-affine form using a conditional *while* loop.

The paper describes a detailed comparison of iteration space slicing and affine transformations.

For carrying out an experimental study, real-life codes were used from the NASA Parallel Benchmark suite (computational fluid dynamics) [NPB15] and the UTDSP benchmark suite (digital signal processing applications) [Sean99].

Execution times, speed-up, and efficiency of optimized codes were analyzed for a platform with eight quad-core processors and a 96-core graphics card. The efficiencies of codes scanning slices by the counter and condition loops were also compared.

The paper was written in collaboration with the Inria Research Institute (authors Cohen and Beletska). Papers [4, 5, 9, Palkowski15] present a combination of the described technique for synchronization-free parallelism extraction with loop tiling.

2. *Free scheduling for statement instances of parameterized arbitrarily nested affine loops*

Paper [2] presents an algorithm for free-scheduling of statement instances of affine program loops. Free scheduling allows for execution of statement instances as soon as all their operands are available and is the fastest partitioning of statement instances to execute parallel code (guarantees the minimal number of synchronization points). This allows for the extraction of maximal fine-grained parallelism.

The related approaches [Feautrier92_1, Feautrier92_2, Darte94, Darte96] and solutions implemented in the Pluto compiler are based on linear and affine schedules, which do not guarantee forming free scheduling in the general case of program loops.

This approach is based on the calculation of the k-th power of relation R, R^k, representing all dependences in the program loop nest. Relation R^k is applied to a set of ultimate dependence sources. Parameter k represents a given number of time partition. Ultimate dependence sources are performed in the first time partition, $k = 0$. By subsequent compositions of dependence relations on sets, subsequent sets of statement instances are defined and executed for $k = 1,2,3$ ....

Palli

In the dependence graph, all statement instances are connected with ultimate sources with paths of length k or more. When a statement instance is connected with an ultimate source with several paths of different lengths, the paper presents a formula based on relation R^k and transitive closure, which allows us to calculate the longest path defining the time when this instance should be executed under the free schedule.

The output of the algorithm is a set whose first tuple element represents logical time while the reminding elements state for statement instances to be executed at this time. This set is used to generate code whose outermost loop scans serially time partitions, while inner loops enumerate statement instances in parallel for a given logical time.

The exact power of the dependence relation is a condition to generate correct free-scheduling. In this paper, R^k is computed by means of the author's implementation based on the Omega library. It is worth adding that the calculation of exact relation R^k guarantees the calculation of the exact transitive closure R^+ [Verdoolaege11].

When it is not possible to calculate exact R^k, two alternative solutions are presented: applying an approximation, which results in some scheduling (non-free) or repeating the procedure for the iteration space of internal loops (external loops are executed sequentially).

In the experimental section, the time of the free-schedule calculation time and the other compilation steps are presented for the NASA Parallel Benchmark suite. Program performance time, speed-up, and efficiency were evaluated by means of a 96-core graphics card using the CUDA programming interface. It has been shown that the time of the transfer of data between CPU operating memory and GPU memory is acceptable in practice.

Paper [Bielecki15] discusses a combination of this technique for free-scheduling with loop tiling [4].

3. *TRACO: Source-to-Source Parallelizing Compiler*

This publication describes the implementation of algorithms enhanced with additional program loop transformations. The TRACO compiler has replaced the previously developed the Iteration Space Slicing Framework (ISSF) library. Like in the state-of-the-art implementation of the Pluto compiler, in TRACO many automation mechanisms have been introduced.

First, algorithm implementations were transferred from the undeveloped Omega library [Kelly95] to the modern and up-to-date ISL library. ISL supports all Presburger arithmetic operations including more efficient algorithms for calculating transitive closure and the power operation of relations [Verdoolaege10].

Second, the implementation is a source-to-source tool, i.e., the input and output is a code or its part containing program loops satisfying the C/C ++ syntax. This allowed for processing complex codes represented within the polyhedral model, accelerating experimental studies, and reducing the risk of mistakes. The compiler converts polyhedral code produced with a code generator into compilable code according to the OpenMP standard (for multicore and co-processors) [OpenMP17] and the OpenACC standard for GPUs [OpenACC17].

Algorithms are implemented in Python within the islpy [Klöckner15] library, which allows for the usage of the ISL library. Prototyping of new algorithms with Python is faster than C/C++. TRACO is also equipped with the Petit dependence analyzer, Pet analyzer, Clan code analyzer [Bastoul08] and code generators: cloog [Bastoul04] and ISL AST [Verdoolaege12].

Pallli

The algorithms of the extraction of independent code fragments and parallelism with synchronization were enhanced by the transformations of variable privatization and parallel reduction. This expanded the compiler's usability and reduced the number of dependence relations on input.

The experimental part shows the capabilities of the compiler for the NASA Parallel Benchmark and PolyBench [Pouchet15] suites. Experiments were performed on multicore processors and coprocessors, and Tesla graphics cards. The results of experiments show the practical compiler usage due to short code execution time resulting in higher speed-up and efficiency, and smaller communication cost.

This publication contains a comparative analysis of results with those obtained by means of other related tools, such as the Intel C++ Compiler (ICC), Pluto, Par4All, Cetus, and PIT. The experimental part compares the performance of generated code by the TRACO compiler with those generated with the above tools.

TRACO is a multi-module platform. It is open-source software with publicly available sources (traco.sourceforge.net) for researchers, students, and programmers. The compiler allows for automatic parallelization of serial programs in C with improved code locality. The platform also allowed us to analyze in later studies other transformations, which are difficult under a manual analysis. TRACO allowed us to combine previous approaches with new sophisticated optimizations, loop blocking in particular, to improve code locality due to increasing cache usage.

4. *Tiling arbitrarily nested loops by means of the transitive closure of dependence graphs*

The publication presents an innovative approach to tiling arbitrary nested program loops by means of the transitive closure of the dependence graph. It is a combination of the polyhedral model and iterative space slicing and does not require finding affine functions for tiling. The solution does not depend also on the full permutability of the loop band.

The iteration space is initially divided into rectangular tiles. Then, statement instances being dependence destinations in a tile are moved to some lexicographically greater tile, which includes the corresponding sources. Statement instances are moved to the lexicographically maximal tile, if the statement instance is the destination of many dependences belonging to different tiles. The formal proof of this approach is presented in this paper. It is worth noting that the target shape of the tile can be changed. Such a modification is necessary if negative coefficients are present in distance dependence vectors. In extreme cases, code can become sequential. Statement instances are tiled, if they are within at least two program loops.

The efficiency of the algorithm determines the possibility of calculating the exact transitive closure of the dependence graph. However, an approximation of transitive closure (not existing paths are added to the graph representing transitive closure) also guarantees generation of valid code but with decreased parallelism and locality.

Generated code contains double number of loops compared to the original loop nest. The first group of loops scans tile identifiers, while the second group of loops scans inner-tile statement instances sequentially. To generate parallel code, inter-tile dependences are only significant. It is worth noting that inter-tile dependence graphs generated under the discussed approach do not include cycles. In order to parallelize tiled code, any technique, including unimodular, affine [7,8], those based on transitive closure [9, Palkowski15], or the *k*-power of

Palki

dependence relations can be applied [Bielecki15, Palkowski15]. The paper discusses how to extract parallelism from generated tiled code. In such a case, target tiles and inter-tile dependences are input of well-known transformations instead of statement instances and original dependences.

The paper presents the preprocessing of sets, dependence relations, and tile identifiers for arbitrary nested loops to achieve the same dimensions of tuples that is required for execution of operations on sets and relations. The applicability of the technique is demonstrated with the *k6* benchmark of the Livermore Loops suite that resolves the general linear recurrence. The benchmark is tiled with the proposed approach, while ATF implemented in such tools like Pluto, fails to tile this code using affine functions. Additionally, a manual transformation was performed to parallelize this benchmark and accelerate it on a multiprocessor machine (the parallel reduction allows us to remove many dependences).

Experiments were carried on a 48-cores machine for codes from the NASA Parallel Benchmark and the PolyBench Benchmark suites. Speed-up and scalability of codes generated with TRACO are reported.

5. *Synchronization-Free Automatic Parallelization for Arbitrarily Nested Affine Loops*

The paper describes the extraction of independent code fragments in arbitrarily nested program loops. Compared to the approach presented in paper [1], this technique does not require calculation of the transitive closure of the union of all dependence relations for the loop nest. Instead, the algorithm is based on the SCC graph[4] and calculates the transitive closure of self-dependences for each loop statement and the transitive closure for each pair of loop statements. This approach requires an undirected graph of dependences.

For this purpose, a set of dependence relations is extended with their inverted forms, i.e., relations calculated with applying the relation inverse operator, where a relation domain becomes a relation range and vice versa. The method of calculating representative points is as follows: for the *i*-th nest, the domain of relations whose are the dependence statement instances sources are located in the *i*-th nest is subtracted from the range of relations whose dependence statement instances destinations are in the *i*-th nest.

The algorithm uses the iterative transitive closure calculation presented in the paper [Bielecki14], which is implemented in the TRACO compiler. Transitive closure is calculated by means of the modified Floyd-Warshal algorithm. An undirected dependence graph is used to extract representative statement instances of slices and connected with them transitive dependent statement instances. A code generation process has been simplified to a single stage in comparison to the solution presented in paper [1]. A set of representatives is input of a code generator. This technique simplifies the extraction of code fragments especially for arbitrarily nested loops with many statements within the loop nest.

The paper presents examples of simulation applications from the NAS Parallel Benchmark suite and explains steps of presented algorithms extracting slices available in the loop nest and considers limitations of ATF by means of the state-of-the-art Pluto compiler.

---

[4] A strongly connected component of the dependency graph (SCC) is a graph in which nodes represent loop statements and edges indicate dependencies between instance instances.

Experimental studies show a significant acceleration for examined codes on Intel Xeon Phi coprocessors. Furthermore, execution times of transitive closure calculation with the Omega and ISL libraries have been compared with time obtained by TRACO, which is much shorter for all examined programs from the NAS Parallel Benchmark suite.

In summary, the presented approach demonstrates the possibility of using transitive closure to extract independent code fragments when the exact closure of the union of dependence relations is not possible to calculate at short time. The paper extends the applicability of the approach presented in paper [1] by means of using the undirected dependence graph and the iterative algorithm of transitive closure computing.

6. *An Iteration Space Visualizer for Polyhedral Loop Transformations in Numerical Programming*

The paper describes a tool for visualizing iteration spaces and particular sub-spaces of program loops. This module is integrated with the TRACO compiler. It visualizes dependences for loop nests of depth two and three, i.e., in two- and tree-dimensional iteration spaces. It retrieves information from TRACO about available parallelism and block shapes in a given iteration space. Various functions are available like rotation, zooming, coloring, filtering, block transparency. The tool is useful for design and validation of new algorithms of program loop transformation.

The visualization module is written in Python and based on the matplotlib library. Sets and relations are retrieved from the islpy library. A unified development environment greatly facilitated the development of this tool.

Visualization is integrated with algorithms that use the transitive closure of the dependence graph, slices extraction and free scheduling at the statement instance and tile levels. The paper presents examples of program loops with their various methods of visualization. Valuable options are loop nest separation into subspaces of blocks together with dependences and displaying parallelism at the tile level. The paper presents also visualization for the polyhedral program from the Livermore Benchmark suite, which calculates the Planck distribution.

Related tools are considered. The closest solution is islplo whose limited 3D capabilities have prompted us to write own tool. Additionally, there are other modules for graphical presentation, however, they are integrated only with affine transformation approaches. Author's visualization retrieves data from the TRACO compiler in the form of relations and sets. It allows us to graphically present results of particular steps of algorithms implemented in TRACO.

7. *Parallel tiled code generation with loop permutation within tiles*

This paper discusses the extension of loop tiling based on the transitive closure of the dependence graph [4]. Performance is enhanced by permutation of loops in tiled code.

Our experimental study shows that it is worth combining loop tiling with loop interchange. Changing loop order leads for changing reading of array columns into reading of array rows, which is significant for programs written in C/C ++ performed on shared memory computers, and improves locality of target code.

If there is no dependence in the program loop, any loop interchange is correct. Loop permutation is a generalization of loop interchange and can be applied to each loop band, not just inner ones. Index variables of inner loops should be the last indexes of arrays used in statements. The order of program loops can be also determined by input variables established by the expert (e.g. an experienced programmer).

Transitive closure is also used to find a proper loop permutation. The loop tiling approach [4] is applied to sub-tiles. The paper presents the proof of the correctness of such a solution. It is worth noting that this is not the classic interchange, but a solution very close to it. In order to generate sub-tiles, sets of the previous and following sub-tiles are defined according to the tiling algorithm presented in [4]. This is only necessary if there are negative components in distance vectors. In the case of non-negative elements of distance vectors, we obtain a solution identical to the classic permutation of program loops.

Many related solutions, including Irigoin and Triolet [Irigoin88] and Xue [Xue97, Xue12], can only be used for non-negative dependence vectors (between instruction instances or tiles) on input. Mullaupadi and Bondhugula [Mullapudi14] pointed out the excessive ``conservatism'' of these limitations. Building static tiling schemes, although not easy, leads to higher performance.

The experimental part of the paper presents the implementation of the approach within the TRACO compiler. Results are illustrated by real-life codes from the NASA Parallel Benchmark 3.3 suite. The study was performed on 32 threads. Synchronization-free slice extraction algorithms [1] have been used for code parallelization, although any other parallelization transformations can be applied to interchanged and tiled code. The study presents a comparison of execution times [4] of tiled codes with those obtained with permutation enabled and disabled. The paper describes loop nest types for which the interchange of tiled loop bands does not change code performance and those, for which interchange allows for significant code performance increasing including programs, for which super-linear speed-up is achieved.

8. *Parallel tiled Nussinov RNA folding loop nest generated using both dependence graph transitive closure and loop skewing*

The paper presents an automatic parallelization and loop tiling of the Nussinov RNA folding program loop nest. This computational task belongs to bioinformatics and predicts RNA structures by maximizing base pairs of sequences.

Transitive closure is used to tile code, while parallelism is extracted by means of the classic loop skewing transformation. This is the first attempt to generate static parallel code that blocks all three program loops of the Nussinov algorithm.

The prediction of RNA structures is a computationally complex task and one of the most important tasks of computational biology. Codes that implement algorithms such as Nussinov's folding can be represented within the polyhedral model, where all expressions in the program loops are affine. However, classic transformations do not allow for efficient code generation.

These limitations are due to the fact that these tasks belong to the complex nonserial polyadic dynamic programming (NPDP) class where the dependence graph is full of non-

Palle

uniform dependences (the dependence vector is not represented only by constants) and cycles of inter-tile dependences.

Mullaupadi and Bondhugula [Mullaupadi14] have described the limitations of the ATF methods implemented in the Pluto compiler and proposed dynamic scheduling of tiles with reduction chains. As a result of their research, however, there is no static code for NPDP loops. Futhermore, only the first two loops from the three ones of the Nussinov algorithm are tiled. Wonnacott's team described tiling all three loops by dividing the iteration space into two sub-spaces such that the first one can be tiled while the second one reminds untiled. However, they did not propose how to parallelize code under their approach. The Pochoir compiler [Tang11] allows for a diamond-shaped tiling for the Gotoh RNA folding, but it does not tile all program loops. The popular manual and parallel implementation of RNA prediction, GTFold (gtfold.sourceforge.net) does not take into account techniques for improving the locality of code.

Those limitations of the related solutions and the possibility to calculate the exact closure of the Nussinov loop nest have become the motivation of this paper. Applying the transitive closure of the union of all dependence relations, it is possible to tile all program loops. The statement instances of tiles are divided according to the approach presented in paper [4] to valid and invalid (dependence destinations which sources belong to lexicographically later tiles). Invalid statement instances are automatically moved to lexicographically greater tiles in order to make them valid. Such a tiling is a novel solution and does not occur in related approaches and tools. The paper explains how to automatically generate parallel code and includes the proof of the correctness of tiled code after applying loop skewing.

In the experimental part, the prediction was performed on sequences with 2200 (mean) and 5000 (longest) length for homo sapiens mRNA taken from the NCBI database [NCBI17]. However, the acceleration is independent of the RNA sequence content, it depends only on its length) and similar results are obtained for random RNA strengths.

Considerable computational acceleration for 64 threads that execute on two modern Intel Xeon E5-2699 v3 multi-core processors and 244 threads on four Intel Xeon Phi 7120P co-processors cores was achieved. The study demonstrated computational scalability and improved code locality compared to codes based on two-dimensional tiling generated by the Pluto+ compiler.

The paper presents the possibility of combining tiling methods based on the transitive closure of the dependence graph with affine transformations. Furthermore, a specific class of NPDP real-life programs has been demonstrated, for which the algorithms implemented in the TRACO compiler are more efficient than the popular Pluto compiler methods and other related work. Such tasks include other bioinformatics algorithms, e.g., DNA sequencing using Smith-Waterman and Neddleman-Wunch techniques. The Zuker's RNA prediction loop nest can be tiled in the same manner as Nussiov's folding [Palkowski17] while the Pluto algorithms fails to tile it. Additionally, there are other classic NPDP optimization problems of dynamic programming such as optimal polygon triangulation, matrix multiply chain, binary tree search, and longest string matching [Cormen09], which dependence pattern is the same as the pattern of the Nussinov loop nest.

The power of classical ATF methods has been demonstrated on numerous polyhedral benchmarks, stencil tasks in particular, where array cells are periodically calculated using neighboring cells values. In NPDP tasks, each cell needs scanning all predecessors in a given

row and column (where the task complexity is usually $O(n^3)$ or $O(n^4)$, and memory $O(n^2)$). This is due to dynamic programming (DP) nature which is a tabular method of solving problems instead of inefficient recursion checking the same sub-problems. Filling the cost table leads to irregularities in the DP code dependence graph, which limits ATF efficiency and requires a graph analysis based on transitive closure in order to achieve better code optimization.

9. *Generation of parallel synchronization-free tiled code*

This paper presents a solution of synchronization-free tiled parallelism based on the transitive closure of the dependence graph for arbitrary nested program loops. Loop tiling is based on the technique presented in paper [4]. Parallel threads are presented as independent slices scanning tiles or sub-tiles. It has been shown that parallelism at the statement instance level contains more threads and that the level of independent tiles calculated by the approach [4] does not allow for the maximum number of threads for the tiled code.

The technique consists of the following steps: extraction of slices [1] at statement instance level, loop tiling with the valid lexicographical order of target tiles, application of common parts of slices and tiles (which differs from previous solutions [4, Palkowski15] forming the slices from tiles), and parallel code generation.

Target tiles are divided into the following categories: without representative points, with one point, and with multiple representative points. In the latter case, a tile can be divided again into parts which contain at least one representative point. The algorithm also allows experts to form own relations mapping representative points inside a single tile to increase parallelism degree and/or code locality.

The implementation of the algorithms was performed in the TC compiler [TC2017], which is a newer branch of the TRACO compiler, which transforms program loops using the transitive closure of the dependence graph. This tool is based entirely on the ISL library and was written only in C ++.

In the experimental study, synchronization-free tiled codes of eight applications from the PolyBench 4.1 [Pouchet15] suite were generated. Analyzing the acceleration of the target codes was performed for tiles and sub-tiles. For two applications, super-linear speed-up has been achieved, which exceeds the number of threads several times. It has been shown that the acceleration for other codes is comparable or higher than that of codes generated by the Pluto compiler based on ATF.

10. *Tuning Iteration Space Slicing based tiled multi-core code implementing Nussinov's RNA folding*

The paper deals with the problem of choosing the proper sizes of tiles to increase the locality of parallel code generated by the TRACO compiler. In order to find proper sizes, the TSS (tile size selection) technique is used. It is based on the model of parametric tiled code (parameters determine sizes). The paper presents a method of reproducing such a code on the basis of unparametric codes. Fixed sizes are replaced by parametric expressions, which are not necessarily have to be affine. Using parameterized code, it is possible to automatically scan the search space and determine good tile sizes.

This technique is applied to parallel tiled code of the Nussinov algorithm discussed in [8]. For NPDP problems like Nussinov's folding, affine techniques fail to tile all internal loops. Unfortunately, the familiar tools for parametric loop tiling are based only on ATF. The paper shows that tiling all loops allows for a wider search for the good tile sizes in a three-dimensional space.

In order to determine good tile sizes experimentally, a three dimensional space of 20 sizes for each one dimension, i.e. $20^3 = 8000$ combinations, was scanned over short RNA sequences (2200 nucleotides). The following conclusions were made. First, tiling the outermost loop is not efficient because 3D tiles usually do not fit in the cache memory or most of the tiles are not rectangular after tile correction. Second, it is more important to tile the second and third loops in such a manner that preserving the original rectangular shape of the tiles is the more common than tiles correction. Hence, the size of the second tile dimension must be a row greater than the third one, because too large sizes of the third dimension decrease code locality. Experimentally found good tile sizes B = [1, b2, b3], where b2> b3, can be termed the "golden mean" and are also valid for longer RNA sequences.

The experimental study compares code generated with TRACO with that obtained by means of the Pluto compiler, which has much worse locality, as already demonstrated in [8]. A comparison has been made also with Chang's [Chang10] and Li [Li14] manual generated codes. Chang manually modified Nussinov's recursion to improve the locality of code by calculating the diagonal cells filled in the sequence pairing table. Li improved this modification by using the lower part of the array (it is not used in Nussinov's recursion) and inserting there copies of the cell. In this way, he replaced the time-consuming reading of the cells in columns by reading them in rows of the unused part of the Nussinov array. Li's results overcome the results of the tiled code generated by means of the Pluto compiler.

Tilling the second and third innermost loops with good tile sizes enables improved code performance, which is higher than the Li's implementation for RNA sequences longer than 2500. For sequential tiled code (1 thread), acceleration was 3.7, while for 32 threads, a super-linear speed-up is observed (112.9). Time benefits have been improved by about 30 ~ 40% in comparison to the results from [8]. The use of the code quality estimation model for different tile sizes is definitely better than the empirical and manual search of good tile sizes because automatic search of a space provides more information from a large number of time trials and allows for achieving higher performance of examined NDPD codes.

References

[Allen01] R. Allen, K. Kennedy, *Optimizing compilers for modern architectures: A Dependence-based Approach*, Morgan Kaufmann Publishers, Inc., 2001.

[Bacon93] D. F. Bacon, S. L. Graham, O. J. Sharp, *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys 26, 1993.

[Banerjee93] U. Banerjee. *Loop Transformations for Restructuring Compilers*. pages 328, Kluwer Academic, 1993.

[Baskaran10] M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, P. Sadayappan: *Parameterized tiling revisited*. In:Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization. CGO '10, pp. 200–209. ACM,New York, NY, USA, 2010.

[Bastoul04] C. Bastoul, *Code generation in the polyhedral model is easier than you think*, PACT'13, IEEE International Conference on Parallel Architecture and Compilation Techniques, Juan-les-Pins, France, pp. 7–16, 2004.

[Bastoul08] C. Bastoul. *Extracting polyhedral representation from high level programs*. Technical Report, LRI, Paris-Sud University, 2008. Related to the Clan tool.
[Beletskyy03] V. Beletskyy, K. Siedlecki, *Finding free schedules for non-uniform loops*, in: Euro-Par 2003 Parallel Processing, Lecture Notes in Computer Science, vol. 2790/2003,, pp. 297–302., 2003.
[Bielecki10] W. Bielecki, T. Klimek, M. Pałkowski, A. Beletska, 2010, *An iterative algorithm of Computing the Transitive Closure of a Union of Parameterized Affine Integer Tuple Relations*, Lecture Notes in Computer Science, Springer, Volume 6508/2010 str. 1611-3349
[Bielecki11] W. Bielecki, M. Pałkowski, 2011, *Ekstrakcja drobno- i gruboziarnistej równoległości w pętlach programowych*, Wydawnictwo ZUT Szczecin, ISBN 9788376630977, liczba stron 260, (monografia).
[Bielecki14] W. Bielecki K. Kraska, T. Klimek, *Using basis dependence distance vectors to calculate the transitive closure of dependence relations by means of the Floyd-Warshall algorithm*. Journal of Combinatorial Optimization, 30(2), s. 253-275, 2014.
[Bielecki15] Włodzimierz Bielecki, Marek Pałkowski, Tomasz Klimek, 2015, Free Scheduling of Tiles Based on the Transitive Closure of Dependence Graphs., Lecture Notes in Computer Science, Springer, Vol. 9574, 11th International Conference on Parallel Processing and Applied Mathematics (PPAM'15), Kraków, str. 133-142.
[Bondhugula08] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan, *A practical automatic polyhedral parallelizer and locality optimizer,* in: Conference on Programming Language Design and Implementation, ACM, pp. 101–113, 2008.
[Bondhugula16] U. Bondhugula, A. Acharya, A. Cohen *The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests*, ACM Transactions on Programming Languages and Systems (TOPLAS), vol 38, issue 3, Apr 2016.
[Bondhugula17] U Bondhugula, V Bandishti, I. Pananilath *Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations*, IEEE Transactions on Parallel and Distributed Systems (TPDS), pg 1285-1298, Vol 28, Issue 5, May 2017.
[Chang10] D. Chang, C. Kimmer, M. Ouyang, *Accelerating the Nussinov RNA folding algorithm with CUDA/GPU*. In: The 10th IEEE International Symposium on Signal Processing and Information Technology, pp. 120–125, doi:10.1109/ISSPIT.2010.5711746, 2010.
[Cormen09] T. H. Cormen, C. Stein, R. L. Rivest, C. E. Leiserson, Introduction to Algorithms, 3rd ed. The MIT Press, ISBN 0262033844, 9780262033848, 2009..
[Darte94] A. Darte, Y. Robert, C*onstructive methods for scheduling uniform loop nests*, IEEE Trans. Parallel Distrib. Syst. 5, 814–822, 1994.
[Darte96] A. Darte, F. Vivien, *Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs*, in: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques, PACT '96, IEEE Computer Society, Washington, DC, USA, pp. 281–291, 1996.
[Darte00] A. Darte, Y. Robert, F. Vivien, *Scheduling and Automatic Parallelization*, Birkhäuser Boston, 2000.
[Feautrier92_1]P. Feautrier, *Some efficient solutions to the affine scheduling problem: I. one-dimensional time*, Int. J. Parallel Prog. 21 (5) (1992) 313–348.
[Feautrier92_2] P. Feautrier, *Some efficient solutions to the affine scheduling problem: II. multi-dimensional time*, Int. J. Parallel Prog. 21 (5) (1992) 389–420.
[Feautrier12] P. Feautrier. Approximating the transitive closure of a boolean-affine relation. In U. Bondhugula and V. Loechner, editors, IMPACT 2012, 2012.
[Feautrier15] P. Feautrier. The power of polynomials, In U. Bondhugula and V. Loechner, editors, IMPACT 2013, 2015.
[Fisch74] M. J. Fischer, M. O. Rabin, *Super-exponential complexity of Presburger arithmetic,* Proceedings of the SIAM-AMS Symposium in Applied Mathematics Vol. 7: 27–41, 1974.
[Griebl00] M. Griebl, P. Feautrier, and C. Lengauer. *Index set splitting*. International Journal of Parallel Programming, 28(6):607–631, 2000.
[Griebl04] M. Griebl, *Automatic Parallelization of Loop Programs for Distributed Memory Architectures,* D.Sc. thesis, University of Passau, Passau, 2004
[Grosser14] T. Grosser, S. Verdoolaege, A. Cohen, P. Sadayappan, *The relation between diamond tiling and hexagonal tiling,* Parallel Processing Letters 24(03): 1441,002, 2014.
[Grosser15] T. Grosser, S. Verdoolaege, A. Cohen, *Polyhedral ast generation is more than scanning polyhedra*, ACM Trans Program Lang Syst 37(4):12:1–12:50, 2015.
[Irigoin88] F. Irigoin, R. Triolet, *Supernode partitioning,* Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'88, San Diego, CA, USA, pp. 319–329, 1988.

Palli

[Kelly95] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, D. Wonnacott, *The omega library interface guide,* Tech. rep., College Park, MD, USA, 1995.
[Kelly96] W. Kelly, W. Pugh, E. Rosser, T. Shpeisman, *Transitive closure of infinite graphs and its applications,* Languages and Compilers for Parallel Computing , LNCS, vol. 1033, pp. 126-140, 1996.
[Kim09] D. Kim, S.V. Rajopadhye, *Parameterized tiling for imperfectly nested loops,* Technical Report CS-09-101, Colorado State University, Fort Collins, CO, 2009.
[Klöckner15] A. Klöckner *islpy, a Python wrapper around Sven Verdoolaege's isl,* https://documen.tician.de/islpy, 2015.
[Li14] J. Li, S. Ranka, S. Sahni, *Multicore and GPU algorithms for Nussinov RNA folding,* BMC Bioinformatics 15(8), 1,. doi:10.1186/1471-2105-15-S8-S1, 2014.
[Lim94] A.W. Lim, M.S. Lam, M.S. *Communication-free parallelization via affine transformations*, in K. Pingali etal. (Eds.), 24th ACM Symposium on Principles of Programming Languages, Springer-Verlag, Berlin/Heidelberg, pp. 92–106, 1994.
[Liv10] Livermore Loops Benchmark, http://www.netlib.org/benchmark/livermorec, 2010.
[Mullapudi14] R. T. Mullapudi, U. Bondhugula. *Tiling for dynamic scheduling*. In IMPACT 2014: 4rd International Workshop on Polyhedral Compilation Techniques, 2014.
[NPB15] *NAS Parallel Benchmarks suite,* http://www.nas.nasa.gov, 2015.
[NCBI17] National Center for Biotechnology Information, *https://www.ncbi.nlm.nih.gov.*
[Nussinov78] R. Nussinov, G. Pieczenik G, J.R. Griggs, D.J. Kleitman, *Algorithms for loop matchings*. SIAM J Appl Math.;35(1):68–82, 1978.
[OpenACC17] *The OpenACC 2.5 Application Programming Interface,* https://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5.pdf, 2017.
[OpenMP17] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.5, http://www.openmp.org/wp-content/ uploads/openmp-4.5.pdf, 2017.
[Palkowski15] M. Pałkowski, T. Klimek, W. Bielecki, *TRACO: An automatic loop nest parallelizer for numerical applications*, Annals of Computer Science and Information Systems, IEEE Xplore® Digital Library., Vol. 7, str. 681-686 (FedCsis Łódź), 2015.
[Palkowski17] M. Palkowski, W. Bielecki:, *A Practical Approach to Tiling Zuker's RNA Folding Using the Transitive Closure of Loop Dependence Graphs*. Advanced in Inteligent Systems, vol 656, ISAT (2), pp. 200-209, 2017.
[Park11] E. Park, L.N. Pouchet, J. Cavazos, P. Sadayappan. *Predictive Modeling in a Polyhedral Optimization Space*. In 9th IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11), Chamonix, France, April 2011.
[Pouchet15] L.N. Pouchet, T*he polyhedral benchmark suite v.4.1*, http://web.cse.ohio-state.edu/~pouchet/ software/polybench, 2015.
[Pugh93] W. Pugh, D. Wonnacott, A*n exact method for analysis of value-based array data dependences*, in: Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing, Springer-Verlag, 1993.
[Pugh97] W. Pugh, E. Rosser, *Iteration space slicing and its application to communication optimization*, in: International Conference on Supercomputing, pp. 221–228, 1997.
[Sean99] P. Sean Hsien-en , *UTDSP: A VLIW Programmable DSP Processor*, 1999.
[Strout04] M.M. Strout, L. Carter, J. Ferrante, B. Kreaseck, *Sparse tiling for stationary iterative methods, Internaional Journal of High Performance Computing Applications* 18(1): 2004.
[Tang11] Y. Tang, R. A. Chowdhury, B.C. Kuszmaul, C. Luk, and C. E. Leiserson, *The Pochoir Stencil Compiler*. page 117-128, 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'2011), 2011.
[TC17] *TC Optimizing Compiler* http://tc-optimizer.sourceforge.net, 2017.
[Traco17] *TRACO Compiler*, traco.sourceforge.net, 2017.
[Verdoolaege10] S. Verdoolaege *ISL: an integer set library for the polyhedral model.* In: Mathematical software— 810 ICMS 2010, Lecture notes in computer science. vol 6327. Springer, Berlin, pp 299–302, 2010.
[**Verdoolaege11**] Sven Verdoolaege, Albert Cohen, and Anna Beletska. Transitive closures of affine integer tuple relations and their overapproximations. In SAS, pages 216–232, 2011.
[Verdoolaege12] S. Verdoolaege, T. Grosser, *Polyhedral extraction tool.* In: In Proceedings of the 2nd international 819 workshop on polyhedral compilation techniques. Paris, France, 2012.

[Verdoolaege13] S. Verdoolaege, J.C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, F. Catthoor, *Polyhedral Parallel Code Generation for CUDA*, Journal of ACM Transactions on Architecture and Code Optimization (TACO), Volume 9 Issue 4, 2013, 54:1-54:23, 2013.
[Verdoolaege15] S. Verdoolaege, *Integer set library—manual,* http://www.kotnet.org/~skimo//isl/manual.pdf, 2015.
[Verdoolaege16] S. Verdoolaege, *Presburger formulas and polyhedral compilation,* v0.02. Polly Labs and KU 814 Leuven., 2016.
[Weiser84] M. Weiser, *Program slicing*, in: IEEE Transactions on Software Engineering, pp. 352–357, 1984.
[Wolf91] M.E. Wolf, M.S. Lam, *A data locality optimizing algorithm,* Proceedings of the ACM SIGPLAN, Conference on Programming Language Design and Implementation, Toronto, Canada, pp. 30–44., 1991.
[Wolfe95] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, pages 570, 1995.
[Wonnacott13] D. Wonnacott, M. Strout, *On the Scalability of Loop Tiling Techniques*, IMPACT 2013, http://impact.gforge.inria.fr/impact2013/papers/impact2013_on_the_scalability_of_loop_tiling_techniques.pdf
[Wonnacott15] D. Wonnacott D, T. Jin T, A. Lake, *Automatic tiling of "mostly-tileable" loop nests*. In: IMPACT 2015: 5th International Workshop on Polyhedral Compilation Techniques. Amsterdam; 2015. http://impact.gforge.inria.fr/ impact2015/papers/impact2015-wonnacott.pdf.
[Xue97] J. Xue, *On tiling as a loop transformation*, Parallel Processing Letters 7(4): 409–424, 1997.
[Xue12] J. Xue, *Loop Tiling for Parallelism*, Springer Science & Business Media, Springer-Verlag, New York, NY, USA, 2012.

Patkoli Mach