A Taxonomy of Methods and Models Used in Program Transformation and Parallelization

Sesha Kalyur(✉) and G. S. Nagaraja

Department of Computer Science and Engineering, R. V. College of Engineering, VTU, Bangalore, India
Sesha.Kalyur@Gmail.Com, nagarajags@rvce.edu.in

Abstract. Developing Application and System Software in a High level programming language, has greatly improved programmer productivity, by reducing the total time and effort spent. The higher level abstractions provided by these languages, enable users to seamlessly translate ideas into design and structure data and code effectively. However these structures have to be efficiently translated, to generate code that can optimally exploit the target architecture. The translation pass normally generates code, that is sub optimal from an execution perspective. Subsequent passes are needed to clean up generated code, that is optimal or near optimal in running time. Generated code can be optimized by Transformation, which involves changing or removing inefficient code. Parallelization is another optimization technique, that involves finding threads of execution, which can be run concurrently on multiple processors to improve the running time. The topic of code optimization and parallelization is quite vast and replete with complex problems and interesting solutions. Hence it becomes necessary to classify the various available techniques, to reduce the complexity and to get a grasp of the subject domain. However our search for good survey papers in the subject area, did not yield interesting outcomes. This work is an attempt to fill this void and help scholars in the field, by providing a comprehensive survey and taxonomy of the various optimization and parallelization methods and the models used to generate solutions.

Keywords: Taxonomy · Method · Model · Optimization · Transformation · Parallelization

1 Introduction

Software development in higher level languages, greatly reduces the burden on the programmer, to seek solutions to problems in the system and application domains. However, translation of programs from source languages to object code, generates inferior, inefficient code due to the inherent nature, of the structure

N. Kumar and R. Venkatesha Prasad (Eds.): UBICNET 2019, LNICST 276, pp. 233–249, 2019.
https://doi.org/10.1007/978-3-030-20615-4_18

of the programming languages. To generate code that is optimal from an execution perspective, cleanup of the translated code is necessary, an activity that is usually referred to as Code Optimization. Code optimization is possible from several perspectives namely, reduction of execution time, reduction of storage requirement or reduction of energy requirement. In the present context, we use the term code optimization to mean code transformation, to improve the running characteristics of the program. We sometimes use the term Optimization to mean either transformation or parallelization. Program transformation includes code changes, that affect a particular aspect of code, such as the instruction count. Program parallelization involves finding concurrent threads of execution, that can be run on separate processing elements.

The Program transformation landscape is quite fertile, and myriad solutions exist. Although the primary goal of all transformation methods, is to improve the running time of the program, the methodology followed by each technique, in reaching the goal is unique. However it is possible to categorize these individual techniques, based on one or more of the following criteria, namely those that target the Instruction Count, Memory Latency, Locality and those that enable other transformations and parallelization.

Program parallelization is a related problem, that is interesting as well, and can be classified along multiple axes. At a very high level the parallelism that is inherent in a program, can be visualized from a code or data perspective, and accordingly we have parallelism that is code centric or data centric. Based on the Programmer Involvement required or Ease of Use, parallelization can be categorized as Manual, Semi-Automated or Explicit and Automated or Implicit. Considering the Granularity of the Parallel Tasks, parallelization can be categorized as Fine Grained or Coarse Grained. Parallelism in a program is contained, in different regions and structures and accordingly can be classified as Loop Level parallelism, Thread Level parallelism or Process Level parallelism. From a performance criteria, parallelization can be classified as Task Level, Instruction Level and Pipeline parallelism. Depending on the Architectural characteristics of the target machine and the resulting Scalability, parallelization can be classified as Shared Memory Parallelization, or Distributed Memory Parallelization. Parallelization could also be categorized, based on the latest emerging trends in the field, as Parallelization by Speculation [1] and Parallelization by Comprehension. Parallelization presents some interesting sub-problems, such as Sub-Program Creation, Orchestration and Distribution which could also serve as criteria for classification.

Since by nature the problems and their solutions, in the field of program transformation and parallelization are non trivial, precise mathematical models are required, to represent the problems and subsequently derive solutions. The range of models used and reported, in the literature is quite vast. We have models based on Trees, Graphs, Machine Learning, Algebra, Statistics, Enumeration, Heuristics among others [2]. From the above presented arguments, it should clear to the reader that the transformation and parallelization domain is

exhaustive, and diverse and requires classification and categorization, to simplify and comprehend.

This research work is an effort in this direction and we attempt to fill the void by providing comprehensive taxonomies of the methods and models used in the domain of program transformation and parallelization. Section 2 provides the taxonomy of methods used in program transformation. Section 3 contains the taxonomic details of the methods used in program parallelization. Section 4 is dedicated to a discussion of the taxonomies of various models used in program transformation and parallelization. Section 5 presents the various taxonomies discussed earlier in graphical form, for easy comprehension.

2 Taxonomy of Methods Used in Program Transformation

The domain of Program transformation contains several techniques or methods that can be classified along the following characteristics,

- Instruction Count Reduction
- Locality Improvement
- Memory Latency Reduction
- Parallelization Enablement
- Transformation Enablement.

2.1 Instruction Count Reduction

Instruction count for a given program, is the number of instructions executed, for a certain run of the program. This metric is usually obtained, with the help of dedicated hardware counters present in the architecture. This metric has a direct bearing, on the execution time of the program. So one way to reduce the running time of the program, is to reduce the instruction count.

Table 1 lists several popular transformation methods along with their descriptions, which can be categorized as techniques that aim for the reduction of Instruction Execution Count.

2.2 Locality Improvement

Program Locality is the term used to refer to the program behavior, wherein recently used code and data are once again accessed, in a short time span. Hardware caches are used, to store a subset of the recently used code and datum. Accessing these items once again, can result in a cache hit. Since accessing an item from the cache takes fewer execution cycles, than getting them from memory, this can induce a substantial savings in the run time of a program.

The following table, Table 2 provides a listing of techniques and their explanations, whose primary goal is to improve the Locality behavior of the program, through caching of both code and data.

Table 1. Instruction count reducing transformations

	Method	Description
1	Dead Code Elimination	Removal of code controlled by an expression that always evaluates to false
2	Flow of Control Optimization	Removal of redundant jumps to jump instructions
3	Algebraic Simplification	Replacement of algebraic expressions with simpler ones
4	Reduction in Strength	Replacement of expressions with those that take fewer run cycles
5	Machine Idioms	Replacement of operations by more efficient ones
6	Common Sub-expression Elimination	Eliminate the redundant expressions by saving the result and using the result instead
7	Code Motion	Move an expression that produces a constant value in every loop iteration out of the body or header
8	Induction Variable Strength Reduction	Replace operations that involve induction variables with more efficient ones
9	Partial Redundancy Elimination	Replace redundant expressions by storing results and then using them subsequently
10	Bounds Check Elimination	Costly array access checks are substituted by similar checks at compile time
11	Leaf Routine Optimization	Eliminate or reduce the function prologue and epilogue overheads
12	Shrink Wrapping	A prologue and epilogue overhead elimination technique for non leaf routines

Table 2. Locality improving transformations

	Method	Description
1	Blocking	Split a matrix in to sub-blocks and process a sub-block in its entirety before processing another
2	Changing Data Layout	Rearrange data structures in memory to exploit locality
3	Fusion	Merge two adjacent loops
4	Reindexing	Shift iterations by a constant term
5	Scaling	Shift iterations by a constant factor
6	Reversal	Process the loop in reverse order
7	Permutation	Loops of a loop-nest are processed in the reverse order
8	Skewing	Process iterations at an angle
9	Array Contraction	An array variable in a loop is replaced by a scalar to improve cache locality
10	Strip Mining	Similar to Blocking but targets only a subset of loops in a loop-nest
11	Procedure Sorting	Rearrange procedure code based on its calling relationship and frequency

2.3 Memory Latency Reduction

For programs that do not exhibit good locality, caches cannot improve the running time. Techniques such as Prefetching are employed, whereby an item in memory is fetched in anticipation, before it is actually needed. Such techniques work, by hiding the memory latency from the user.

Redundant Load Store Elimination is a transformation, that falls under the category and involves removal of back-to-back Store followed by Load or vice-versa. Prefetching is another transformation of this kind, which attempts to fetch code and data to the cache in anticipation, before their actual reference. Cache Block Alignment is aimed at eliminating multiple fetch requests to objects, that span two cache lines by alignment, so that the request can be fulfilled in a single request.

2.4 Parallalization Enablement

Techniques such as Loop Unrolling, that prepare code and data to effectively enable the parallelization, which follows this step, can be referred to as Parallelization Enablers. They basically transform code so that they are more parallelization friendly even though they may not produce results right away.

Table 3 lists some transformations, that are parallelization enablers.

Table 3. Parallelization enabling transformations

	Method	Description
1	Loop Unrolling	Replace loop with straight-line code by duplicating the body the required number of times
2	Function Inlining	Replace function calls by the code constituting the function body
3	Fission	Split a loop into two or more resulting loops
4	Tail Recursion Removal	Replace recursive function calls by loop with calls to the function in its body
5	Predicated Execution	Replace the condition and controlled code with speculation and conditional moves
6	Software Pipelining	A compact loop unrolling technique based on the hardware pipelining concept
7	Scalar Privatization	Replace a scalar in a loop with an array so that each iteration has a private copy of the variable
8	Pipelining	Perform parallel execution in pipeline fashion
9	Wave Fronting	Transform inner loops of loop-nest so that the data dependencies are eliminated
10	Successive Over Relaxation	A Parallelization technique for solving simultaneous linear equations
11	Vectorization	Replace operations on array elements with a single vector operation that operates on all array locations

2.5 Transformation Enablement

There are some transformations such as Copy Propagation, which don't offer benefits right away. However other transformations which follow, can benefit from these preparatory transformations. These transformation techniques, can be referred to as Transformation Enablers.

Transformations such as Copy Propagation, Constant Propagation and Pointer Alias Analysis constitute the category of transformations enabling other transformations. Copy Propagation aims to replace the assignee of an assignment by the assigned in subsequent operations. Constant Propagation is a related transformation, that propagates constant values among, a sequence of related variables.

3 Taxonomy of Methods Used in Program Parallelization

The domain of program parallelization offers an interesting ensemble of methods and techniques, which can be grouped as follows,

- Simplicity and Ease of Use
- Performance
- Granularity
- Program Structure and Module
- Scalability
- Novelty
- Orchestration and Management.

3.1 Simplicity and Ease of Use

A Parallelization technique can be viewed, on the basis of how simple or easy it is to implement and use. For instance, manually parallelizing a program is cumbersome to users, compared to the compiler technique which automatically parallelizes a program.

Table 4 categorizes parallelization, based on the criteria of their simplicity and ease of use.

Table 4. Parallelization methods based on simplicity and ease of use

	Method	Description
1	Manual Parallelization	Programmer manually identifies parallel parts of the program and implements parallel code
2	Semi-Automatic Parallelization	Programmer provides informative tags identifying the parallel pieces and the compiler constructs the parallel program
3	Automatic Parallelization	Compiler creates the parallel program after extensive analysis of the given program with out user assistance
4	Explicit Parallelization	It is just an other name for Manual or Semi-Automatic Parallelization [3]
5	Implicit Parallelization	It is a synonym for Automatic Parallelization [4]

3.2 Performance

Parallelization exists at various levels in a program, such as procedures or statements. How we unleash it depends on how much performance we are expecting and the effort we are willing to invest.

Parallelization carried out at the Instruction Level or the level of the Task and inside the hardware instruction Pipeline, fall under this group. Accordingly, Instruction Level Parallelization is the parallelization carried out the level of program statements or instructions. Task Parallelization is realized at the level of procedures or modules. Pipeline Parallelization is conducted at the level of machine instructions.

3.3 Granularity

Granularity is a term which is used to refer, to the size of the structure (abstraction) of a program, such as a module or a procedure. Normally, extracting parallelism from a structure that is coarse grained (large size), is easier than extracting from a structure, that is fine grained (small size).

Parallelization carried out at the task level, can be classified as Coarse Grain Parallelization and parallelization carried out at the statement or instruction level could be termed as Fine Grain Parallelization.

3.4 Program Structure or Module

Parallelism is inherent in program elements, such as Loops and Procedures both normal and recursive. Based on the available source and the statement grouping techniques such as multi-programming and threading we can categorize resulting parallelization.

In Process Parallelization, inherent parallelization is extracted through multiple invocations of the same program, each invocation acting as parallel component of the original program. In Thread Parallelization multiple threads are used, to achieve parallel run of the given program. Loop Parallelization refers to the parallelism that is present and subsequently extracted, by concurrent executions of different group of iterations of the loop.

3.5 Scalability

Certain large scale programs such as those that are numerically intensive, can rigorously test the limits of the executing hardware. Some of the hardware architectures can reach a bottleneck, after running the program of a certain size. To overcome such hardware limits, newer architectures have been proposed to scale the program.

Shared Memory Parallelization and Distributed Memory Parallelization, are two parallelization techniques that can be distinguished, based on the target architecture used to run parallel sub-programs. Shared Memory Parallelization involves a setup where the parallel programs communicate through shared memory. In the case of Distributed Memory Parallelization, the parallel components communicate with the help of explicit Sends and Receive calls.

3.6 Novelty

Published literature, periodically presents novel methodologies, to solve traditional problems. We have such examples in the parallelization domain also, which can serve as a basis for classification of parallelization methods.

This category includes Speculative Parallelization and Parallelization by Comprehension, two techniques that are beneficiaries of seen some recent research activity. Speculative Parallelization is initially carried out without dependence testing, but checks for collision and possible rollback are carried out at a later stage [5]. Parallelization by Comprehension is a parallelization process, based on the concept of algorithm inference.

3.7 Orchestration and Management

Parallelization process is not complete, with out concurrently executing the parallel components identified, during the parallelization phases. The final step in parallelization, is the control and management of the parallel pieces, of the original program which can also serve as a criteria for classifying a parallelization methodology.

Independent of the facilities offered by the modern programming languages, one could classify parallel sub-program generation techniques and the parallelization realized as a result. Program Slicing is a technique for creating sub-programs by splitting the given program. Multi-programming involves creating sub-programs that are replicas of the original program, but executing only a subset of instructions in each replica, which when executed together produce the same result, as executing the original program.

4 Taxonomy of Models Used in Transformation and Parallelization

The domain of transformation and parallelization is rich, in terms of the mathematical models employed, to solve a problem at hand. Here we look at the various models, provide a definition for each modeling technique, and present a comprehensive listing of the use cases for each, in the published literature.

The various Models that are employed in the transformation and parallelization activity, can be categorized along the following criteria,

- Models based on the Tree concept
- Models based on the Graph concept
- Models based on Machine Learning
- Models based on Algebra
- Models based on Statistics
- Models based on Enumeration
- Models based on Heuristics.

4.1 Tree Based Models

Tree is a very basic mathematical model, that is widely used in the optimization domain and Parse tree is a specialized model that falls under this category. It graphically shows how a string is derived in some language [6]. Parse tree has been employed by researchers in implementing optimizations such as the Common Sub-expression Elimination [6].

4.2 Graph Based Models

Graph is the most popular model to solve problems, in the program transformation and parallelization domain. Table 5, provides a description of several models based on the graph concept. The following table, Table 6, provides a listing of their use cases in published literature.

4.3 Machine Learning Based Models

Machine learning offers several opportunities to solve problems, in the domain of transformation and parallelization. Table 7, provides a description of models based on Machine Learning followed by table, Table 8, which lists out their uses.

4.4 Models Based on Algebra

Several models exist in the domain based on the Algebra model. Models based on Integer Linear Programming, Polyhedra, Linear Algebra and Symbolic Algebra fall under this category.

Integer Linear Programming is a system with a set of variables to be optimized, based on a function and a set of constraints. Integer Linear Programming has been used to solve problem such as, finding the Longest path length of a loop-nest, Locality optimization, Loop-nest parallelization, Register allocator optimization [25], Speculative instruction scheduler [1]. Polyhedral models use linear algebra abstractions such as matrices and their operations. [26,27]. Polyhedral models have been used in literature to implement, Code size reduction, Vectorization selection [26], Compute loop iteration counts [28], Identify fusionable loops [29], Improve cache misses [30] Linear Algebra models use matrices, determinants, linear equations and their transformations and vector spaces. Linear Algebra models find use in solving Data Layout Transformation problems. Symbolic Algebra models are a collection of techniques for symbolically manipulating mathematical expressions. Symbolic Algebra has been employed in Power optimization, Floating point to fixed point conversion, Model conditions, loops, and procedures in programs.

Table 5. Description of graph models

	Model	Description
1	Call Graph	Procedures and Modules are nodes and edges represent call information
2	Data Dependence Graph	Nodes are operations and edges are the data values
3	Data Flow Graph	Basic blocks are the nodes and the data paths form the edges
4	Control Dependence Graph	Nodes are executable statements and edges are the dependence on the control node
5	Program Dependence Graph	Operators and operands are the nodes and the dependence is captured in the edges
6	Control Flow Graph	Nodes represent instructions and edges the control transfer
7	Hammock Graph	Are sections of control flow graphs with one entry and one exit node
8	Partition Graph	A graph where nodes are execution sets and edges denote partition relations, between nodes
9	Register Interference Graph	A graph where nodes represent live variables and edges denote overlapping live ranges of variables
10	Bayesian Network	A graph where nodes represent random variables and edges capture the conditional dependencies [7,8]
11	Binary Decision Diagram	Are directed acyclic graphs that correspond to a function that returns a boolean result
12	Finite State Machine	A machine where nodes are states and edges represent the transitions

Table 6. Application of graph models

	Model	Application
1	Call Graph	Detect phase transitions, Program Comprehension, Discover program structure, Procedure call relationship, Loop and array transformations
2	Data Dependence Graph	Critical path reduction, Inter-module dependency detection, Optimum code layout [9]
3	Data Flow Graph	Common sub-expression elimination, Hot path prediction, Parallelization
4	Control Dependence Graph	Detect phase transitions, Program Comprehension, Discover program structure, Procedure call relationship, Loop and array transformations
5	Program Dependence Graph	Automatic program distribution [10]
6	Control Flow Graph	Region detection, Partial redundancy elimination, Edge profiling, Dynamic data dependencies [11]
7	Hammock Graph	Unstructured branch conversion
8	Partition Graph	Hyper-block scheduling
9	Register Interference Graph	Improve thread context switch performance
10	Bayesian Network	Iterative optimization sequence [8]
11	Binary Decision Diagram	Predict path occurrence in Hardware Definition Language [12]
12	Finite State Machine	Behavioural synthesis, Predictor generation

Table 7. Description of machine learning based models

	Model	Description
1	Genetic Algorithm	An Evolutionary method to find the individual of maximum fitness from a random population
2	Genetic Programming	Evolutionary technique to find the most optimized program
3	Simulated Annealing	Search techniques with a stochastic basis
4	Markov Model	A probability model where each event depends in a previous event
5	Hill Climbing	A search technique that starts with a random location, and in a local fashion advances towards goal
6	Artificial Neural Network	A system whose operational characteristics are stored in trained inter-unit connection weights
7	Nearest Neighbor	Labels are generated for unseen features from a set of stored trained features
8	Support Vector Machine	A technique where Kernel functions map features to corresponding classes
9	Linear Regression	A system where a function maps a predictor variable to a response variable [13]
10	Decision Tree	A mapping from a feature (non-leaf node) to a class (leaf-node) [14]

Table 8. Application of machine learning based models

	Model	Application
1	Genetic Algorithm	Reduce code size [15], Register allocation and Instruction scheduling, Instruction template selection [16], Optimize energy consumption [17]
2	Genetic Programming	Super-block scheduling [18], Loop unrolling [19]
3	Simulated Annealing	Compiler tuning [20]
4	Markov Model	Optimization space search
5	Hill Climbing	Numerical analysis
6	Artificial Neural Network	Power draw prediction [21], Graph coloring [22]
7	Nearest Neighbor	Loop unroll prediction
8	Support Vector Machine	Predict benefits of loop unrolling, Combining optimization options [23]
9	Linear Regression	Chip energy optimization [24]
10	Decision Tree	Convert program spatial features for Machine Learning [14]

4.5 Models Based on Statistics

There are several models employed in the domain that are stochastic in nature.

Orthogonal Arrays and Principal Component Analysis are models based on the Statistics model. Orthogonal Arrays measure the influence of a process of independent variables, on response or dependent variables. It has been used to influence Compiler Option Selection. Principal Component Analysis model can be used to prune the program feature space. It has been used to solve Iterative Optimization problems.

4.6 Models Based on Enumeration

When the problems involves search, Enumeration provides several opportunities.

Models based on Branch and Bound, Nelder-Mead Simplex and Enumeration fall under this category. Branch and Bound is a recursive search technique that uses trees [31]. It has been used in the past for Combining Optimization Options [31]. Nelder-Mead Simplex is a multi-dimensional search technique. It has been employed to solve Iterative Parameter Search problems [32]. Enumeration is a search space pruning technique that uses history data. Researchers have used enumeration to implement Optimal Scheduling of Super-blocks.

4.7 Models Based on Heuristics

When exhaustive methods do not provide solutions in a prescribed amount of time, Heuristic methods provide approximate solutions to fill the void.

Heuristics provide approximate solutions to NP-Hard problems and have been used to solve optimization problems, in the domain of translation and parallelization such as Combinatorial Optimization [33].

5 A Graphical Taxonomy of Methods and Models Used in Transformation and Parallelization

This section presents the taxonomy, of the various methods and models used in program translation and parallelization in graphical form.

5.1 A Graphical Taxonomy of Methods Used in Transformation

The various program transformation techniques, that are available today were chronicled earlier. Figure 1 provides a taxonomy of the transformation techniques in graphical form.

5.2 A Graphical Taxonomy of Methods Used in Parallelization

A classification of the various parallelization methods, cited in published literature are presented here in graphical form. See Fig. 2 for details.

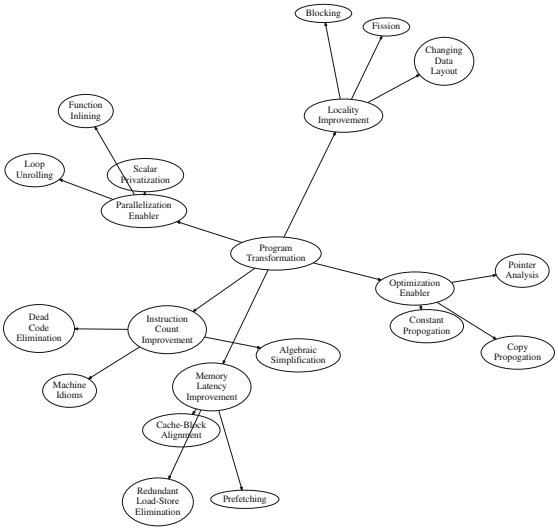

Fig. 1. Taxonomy of methods used in transformation

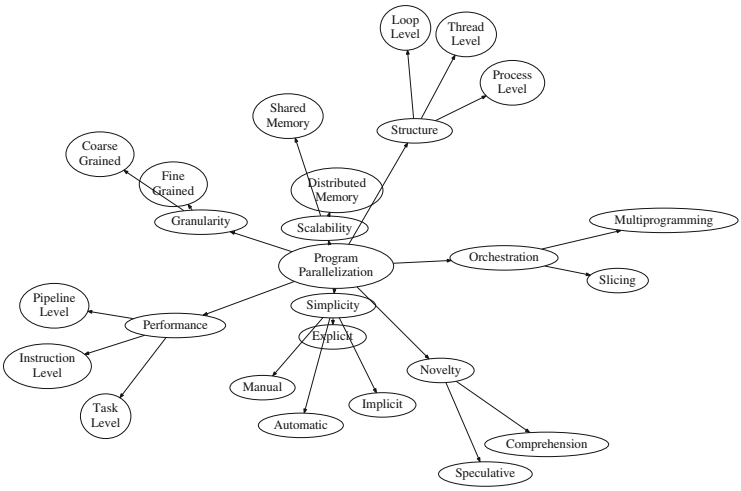

Fig. 2. Taxonomy of methods used in parallelization

Fig. 3. Taxonomy of models used in transformation and parallelization

5.3 A Graphical Taxonomy of Models Used in Transformation and Parallelization

We discussed the various transformation and parallelization models found in published literature earlier. Here we provide a taxonomy of the models in a hierarchical graph representation. See Fig. 3 for details.

6 Conclusion

We started this research work, with the goal of developing a detailed taxonomy of methods and models, used in the program transformation and parallelization domain, which would foster learning and mastering of the subject area. We classified the program transformation methods along the following axes namely, instruction count reduction, locality improvement, memory latency reduction, parallelization enablement and transformation enablement. In a similar vein, program parallelization methods were categorized based on the following criteria namely, simplicity and ease of use, performance, granularity, program structure and module, scalability, novelty, and orchestration and management. Models used in the transformation and parallelization domain, allow the problems to be represented mathematically and subsequently develop solutions. The various models used in the domain as reported in published literature were classified as follows: models based on the tree concept, models based on the graph concept, models based on machine learning, models based on algebra, models based on statistics, models based on enumeration and models based on heuristics. The outcomes of our research are organized in the form of tables and graphs for easy

comprehension. We hope the comprehensive taxonomy we have developed here, will benefit researchers and practitioners alike, and help them in their respective endeavors.

References

1. Winkel, S.: Optimal versus heuristic global code scheduling. In: 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2007, pp. 43–55, December 2007
2. Kalyur, S., Nagaraja, G.S.: A survey of modeling techniques used in compiler design and implementation. In: 2016 International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), pp. 355–358, October 2016
3. Ayguade, E., et al.: The design of OpenMP tasks. IEEE Trans. Parallel Distrib. Syst. **20**(3), 404–418 (2009)
4. Bondhugula, U., et al.: Towards effective automatic parallelization for multicore systems. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–5, April 2008
5. Hertzberg, B., Olukotun, K.: Runtime automatic speculative parallelization. In: 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 64–73, April 2011
6. Canedo, A., Sowa, M., Abderazek, B.A.: Quantitative evaluation of common subexpression elimination on queue machines. In: International Symposium on Parallel Architectures, Algorithms, and Networks, I-SPAN 2008, pp. 25–30, May 2008
7. Chin, G., Choudhury, S., Kangas, L., McFarlane, S., Marquez, A.: Evaluating in-clique and topological parallelism strategies for junction tree-based Bayesian network inference algorithm on the cray XMT. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), pp. 1710–1719, May 2011
8. Ashouri, A.H., Mariani, G., Palermo, G., Silvano, C.: A bayesian network approach for compiler auto-tuning for embedded processors. In: 2014 IEEE 12th Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia), pp. 90–97, October 2014
9. Li, P., Luo, H., Ding, C., Hu, Z., Ye, H.: Code layout optimization for defensiveness and politeness in shared cache. In: 2014 43rd International Conference on Parallel Processing (ICPP), pp. 151–161, September 2014
10. Kalyur, S., Nagaraja, G.S.: ParaCite: auto-parallelization of a sequential program using the program dependence graph. In: 2016 International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), pp. 7–12, October 2016
11. Tineo, A., Corbera, F., Navarro, A., Asenjo, R., Zapata, E.L.: A novel approach for detecting heap-based loop-carried dependences. In: International Conference on Parallel Processing, ICPP 2005, pp. 99–106, June 2005
12. Jayaraman, D., Tragoudas, S.: Occurrence probability analysis of a path at the architectural level. In: 2011 12th International Symposium on Quality Electronic Design (ISQED), pp. 1–5, March 2011
13. Vaswani, K., Thazhuthaveetil, M.J., Srikant, Y.N., Joseph, P.J.: Microarchitecture sensitive empirical models for compiler optimizations. In: International Symposium on Code Generation and Optimization, CGO 2007, pp. 131–143, March 2007

14. Malik, A.M.: Spatial based feature generation for machine learning based optimization compilation. In: 2010 Ninth International Conference on Machine Learning and Applications (ICMLA), pp. 925–930, December 2010
15. Zhou, Y.-Q., Lin, N.-W.: A study on optimizing execution time and code size in iterative compilation. In: 2012 Third International Conference on Innovations in Bio-Inspired Computing and Applications (IBICA), pp. 104–109, September 2012
16. Mahalingam, P.R.: Knowledge-augmented genetic algorithms for effective instruction template selection in compilers. In: 2013 Third International Conference on Advances in Computing and Communications (ICACC), pp. 21–24, August 2013
17. Azeemi, N.Z.: Multicriteria energy efficient source code compilation for dependable embedded applications. Innov. Inf. Technol. **2006**, 1–5 (2006)
18. Mahajan, A., Ali, M.S.: Superblock scheduling using genetic programming for embedded systems. In: 7th IEEE International Conference on Cognitive Informatics, ICCI 2008, pp. 261–266, August 2008
19. Leather, H., Bonilla, E., O'Boyle, M.: Automatic feature generation for machine learning based optimizing compilation. In: International Symposium on Code Generation and Optimization, CGO 2009, pp. 81–91, March 2009
20. Zhong, S., Shen, Y., Hao, F.: Tuning compiler optimization options via simulated annealing. In: Second International Conference on Future Information Technology and Management Engineering, FITME 2009, pp. 305–308, December 2009
21. Tiwari, A., Laurenzano, M.A., Carrington, L., Snavely, A.: Modeling power and energy usage of HPC kernels. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), pp. 990–998, May 2012
22. Wang, X., Qiao, Q.: Solving graph coloring problems based on a chaos neural network with non-monotonous activation function. In: Fifth International Conference on Natural Computation, ICNC 2009, vol. 1, pp. 414–417, August 2009
23. Li, F., Tang, F., Shen, Y.: Feature mining for machine learning based compilation optimization. In: 2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), pp. 207–214, July 2014
24. Gschwandtner, P., Knobloch, M., Mohr, B., Pleiter, D., Fahringer, T.: Modeling CPU energy consumption of HPC applications on the IBM POWER7. In: 2014 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 536–543, February 2014
25. Falk, H., Schmitz, N., Schmoll, F.: WCET-aware register allocation based on integer-linear programming. In: 2011 23rd Euromicro Conference on Real-Time Systems (ECRTS), pp. 13–22, July 2011
26. Trifunovic, K., Nuzman, D., Cohen, A., Zaks, A., Rosen, I.: Polyhedral-model guided loop-nest auto-vectorization. In: 18th International Conference on Parallel Architectures and Compilation Techniques, PACT 2009, pp. 327–337, September 2009
27. Pouchet, L., Bastoul, C., Cohen, A., Vasilache, N.: Iterative optimization in the polyhedral model: Part i, one-dimensional time. In: International Symposium on Code Generation and Optimization, CGO 2007, pp. 144–156, March 2007
28. Lokuciejewski, P., Cordes, D., Falk, H., Marwedel, P.: A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In: International Symposium on Code Generation and Optimization, CGO 2009, pp. 136–146, March 2009

29. Pouchet, L., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P.: Combined iterative and model-driven optimization in an automatic parallelization framework. In: 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–11, November 2010
30. Xue, Y., Zhao, C.: Automated phase-ordering of loop optimizations based on polyhedron model. In: 10th IEEE International Conference on High Performance Computing and Communications, HPCC 2008, pp. 672–677, September 2008
31. Desai, N.P.: A novel technique for orchestration of compiler optimization functions using branch and bound strategy. In: IEEE International Advance Computing Conference, IACC 2009, pp. 467–472, March 2009
32. Lu, P., Che, Y., Wang, Z.: An effective iterative compilation search algorithm for high performance computing applications. In: 10th IEEE International Conference on High Performance Computing and Communications, HPCC 2008, pp. 368–373, September 2008
33. Martí, R., Reinelt, G.: The Linear Ordering Problem: Exact and Heuristic Methods in Combinatorial Optimization, 1st edn. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-16729-4