DOI: 10.14529/jsfi260206

Optimizing Synthesizer of Parallel-pipelined Programs for Reconfigurable Computer Systems

Andrey A. Gulenok[1]***, Evgeniy A. Semernikov***[1]

Existing synthesizers (software tools that convert models implemented in hardware description languages into a list of logic gate connections) for field-programmable gate arrays (FPGAs), such as Vivado (Xilinx), Quartus II (Intel), and Libero (Microsemi), are designed to implement applications within a single chip. These tools perform optimization of the computational components of algorithms only at the level of logical primitives. Such optimization is inherently local and, therefore, cannot substantially reduce the hardware resources utilized on an FPGA. Moreover, conventional FPGA synthesizers do not analyze the implemented algorithms with respect to potential computational redundancies or inefficient design realizations. To date, the only synthesizer for multichip reconfigurable computer systems, developed at the Scientific Research Center of Supercomputers and Neurocomputers, provides automated mapping of the algorithms information graph onto multiple FPGA devices, as well as inter-chip routing and synchronization of control and data flows within the reconfigurable computing system. The novel version of the optimizing multichip synthesizer for parallel-pipelined programs for reconfigurable computer systems, presented in this paper, incorporates a library and a general algorithm for information-equivalent transformations, along with an original method of nested auto-substitutions for recursive expressions. These features enable a substantial and automated improvement in the performance of generally specified algorithms when implemented on reconfigurable computing systems.

Keywords: synthesis of parallel-pipelined programs, information graph, optimizing synthesizer, reconfigurable computing system.

Introduction

Reconfigurable computer systems (RCS), whose primary hardware resources are field-programmable gate arrays (FPGAs) integrated into a unified multichip computational device, are currently being effectively applied to solve problems across various fields of science and engineering [1, 2]. The high achieved performance and energy efficiency of RCS, in comparison with traditional cluster systems, demonstrate their competitiveness in the domain of high-performance computing [3].

Existing FPGA synthesis tools, such as Vivado (Xilinx), Quartus II (Intel), and Libero (Microsemi), are designed to implement applications within a single-chip or across a set of independent FPGAs. If efficient problem implementation requires a large number of interconnected FPGAs, the use of single-chip synthesis tools becomes labor-intensive, as developers must manually partition program fragments across individual devices and ensure proper alignment of data flows both within and between FPGAs. It should be noted that automatic synthesis tools exhibit low efficiency with a high density of chip filling, as exemplified by the Vivado synthesis tool [4]. It leads to reducing the effective performance of RCS or increasing the development time for FPGA-based solutions.

The only synthesis tool for multichip RCS, developed at the Scientific Research Center of Supercomputers and Neurocomputers, provides automatic mapping of the algorithms information graph (IG) across multiple FPGA devices, inter-chip routing, and synchronization of control and data flows in RCS [5]. The information graph of an algorithm for solving the problem is

[1] "Scientific Research Center of Supercomputers and Neurocomputers" Co Ltd ("SRC SC & NC" Co Ltd), Taganrog, Russian Federation

defined as a directed graph in which vertices correspond to operations, data sources, and data sinks, while edges represent data dependencies between them [6].

This paper presents a new version of the multichip synthesis tool and describes the implemented methods of information-equivalent transformations of the algorithms IG. These methods take into account the properties of associative and distributive operations, as well as the regularity of graph structures. Such transformations make it possible to reduce the number of operations in many times for some problems and automatically obtain efficient parallel-pipeline solutions that are often not obvious or even known to the vendor at application development.

Section 1 provides a brief overview of existing synthesis tools, high-level synthesis (HLS) compilers, and the multichip synthesis tool, as well as their application in the development of programs for multichip RCS. Section 2 presents a general algorithm for information-equivalent graph transformations developed for the new version of the optimizing synthesizer, which has been verified for problems from three distinct application domains. Section 3 introduces a method of nested auto-substitution for pipelining calculations of recursive expressions. Section 4 presents the results obtained using the optimizing synthesizer for a number of applied problems. Finally, the obtained results are analyzed in the conclusion.

1. Development of Efficient Programs for Reconfigurable Computer Systems

Currently, standard synthesis tools, such as Xilinx Vivado or Intel Quartus Prime, are used to obtain highly efficient implementations of RCS solutions. Due to the use of low-level digital circuit description languages such as VHDL or Verilog, the vendor can detail the design of the computing structure of a parallel-pipeline program and achieve a high device utilization taking into account the FPGA architectural features.

However, the development time for an efficient solution on a multichip RCS using this approach typically ranges from several months to a year, even for highly qualified hardware engineers. This is due both to the use of low-level design tools and to the necessity of developing multiple projects simultaneously, each of which implements a fragment of the overall program mapped onto a separate FPGA device.

The use of high-level synthesis (HLS) compilers [7], such as Vivado HLS or Vitis, reduces development time by automatically generating complex functional units (IP-cores) from C-language program fragments. However, this approach does not guarantee efficient implementation on RCS for arbitrary C programs. Moreover, issues related to scaling and integration of IP-cores across a multichip RCS remain within the vendors responsibility.

At the Scientific Research Center of Supercomputers and Neurocomputers (Taganrog), a Fire!Constructor multichip synthesis tool is being developed and improved to relieve vendors of a number of routine problems, associated with designing solutions across multiple FPGAs. The inputs to this synthesizer are the IG of a parallel-pipelined program and a description of the RCS architecture. The synthesizer automatically performs automatic layout of the IG into non-overlapping subgraphs, each of which is subsequently located on a separate FPGA device.

The objective of the partition and placement stages is to create favorable conditions for subsequent inter-chip routing. However, even “optimal” solutions of partition and placement problems according to selected criteria cannot guarantee successful routing. Therefore, these

tasks are solved repeatedly using different algorithms, which increaces the probability of successful routing in Fire!Constructor.

As a rule, the operation time of graph partition algorithms has a polynomial or even exponential dependence on the number of vertices in the graph. Therefore, a multilevel partition scheme is employed for large-scale graphs (from several thousand and more) in the synthesizer Fire!Constructor [8], which iteratively contracts pairs of adjacent vertices (Coarsening Phase). Afterward, partition and placement tasks are performed by a graph refinement stage in which the contraction process is iteratively reversed (Uncoarsening Phase). During inter-chip routing, the synthesizer assigns external connections of subgraphs to the available physical communication links between FPGAs in the RCS.

Additionally, Fire!Constructor performs synchronization of control and data flows by inserting delays to align input streams at each pipeline stage of the computational structure. The synchronization problem is solved on the IG by assigning each vertex a latency value corresponding to its pipeline stage. Then, for each incoming edge, the cumulative signal latency from data sources is calculated. Based on these values, flow alignment is achieved by inserting delays into leading signals.

The resulting synchronization subsystem can consume FPGA resources comparable to those required for the computational logic itself. Three types of FPGA resources are used to implement delay elements: flip-flops, programmable delay elements (MLUTs), and block memory (BRAM). Therefore, Fire!Constructor incorporates a set of procedures aimed at reducing both the total number of delays and the utilization of critical hardware resources allocated for synchronization [9].

Using single-chip synthesis tools, all of the above tasks must be handled manually by the vendor. Thus, the use of a multichip synthesizer such as Fire!Constructor significantly reduces development time for RCS-based solutions.

At the final stage of Fire!Constructor synthesis, a single-chip project for the Xilinx Vivado environment is generated for each FPGA involved in the multichip RCS. Each single-chip project contains the corresponding fragment of the circuit-level implementation of the target algorithm, which is subsequently compiled into FPGA configuration files.

Existing FPGA synthesis tools perform optimizations primarily at the level of logical primitives. However, such transformations are local in nature, including the removal of unused elements or simplification of constant logic. The computing structure of the implemented algorithm largely remains as specified by the vendor and may exhibit significant redundancy or inefficiency. Therefore, the efficiency of the final implementation depends heavily on the vendors expertise in parallel-pipelined computing.

A new version of the multichip synthesizer is being developed. In addition to the previously implemented functions for displaying a parallel-pipelined program on the RCS, it automatically identifies fragments of the IG program that are ineffective from the point of view of their parallel-pipelined implementation on the RCS. Analyzing and upgrading the identified fragments through information-equivalent transformations, the synthesizer can significantly increase the real or specific performance of the RCS compared to the execution of the original program.

2. General Algorithm of Information-Equivalent Transformations on a Graph

The structure of a parallel-pipelined program is transmitted to a multichip synthesizer in the form of IG. It is independent of the source programming language and, consequently, is not overloaded by unique syntactic constructions (such as switch-case, union, etc.). Note that the same IG can correspond to multiple program texts (including written in different languages), differing both in the order of independent operators and syntactic constructions, which determine the sequence of operators. This contrasts with program text, where related calculations can be placed in different lines of code, procedures, or even program modules. These IG features described above enable algorithmic simplification of structural analysis procedures, identifying redundant and inefficient structures and procedures for information-equivalent transformations, unlike the program text.

The term "optimizing compiler" refers to the implementation of various algorithms to generate a more optimal program code while preserving its functionality. Similarly, the new developed version of the multichip synthesizer is referred to as an "optimizing synthesizer", since it performs transformations analogous to those of optimizing compilers, but applied to configuration files for multiple FPGAs.

A library of information-equivalent transformations is realized in the optimizing synthesizer. It includes such transformations on individual vertices as calculating expressions for which all operators are constants, exclusion of multiplications and logical "AND" operations with the constant "1" and additions and logical "OR" with a constant "0", and replacing the multiplication or division of an integer by a constant equal to 2, to an integer extent, by shifts (since due to the FPGA architectural features, the shift is implemented in it by bit recomputation, which does not take up hardware resources), etc.

The library also includes transformations for groups of operations which take into account the associativity and distributivity properties. Based on the transformation method of homogeneous IG from sequential to parallel-pipelined form [10], the optimizing synthesizer implements a procedure for graph fragment transformation consisting of sequential associative operations of the same type (e.g., addition, multiplication, logical AND) into a pyramidal form (Fig. 1) or a series of intermediate forms.

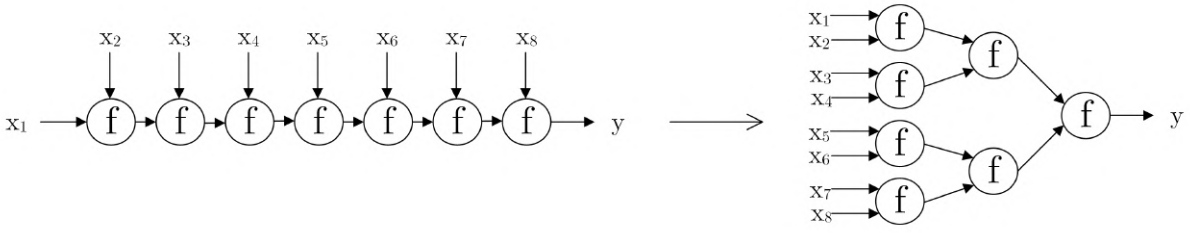


Figure 1. Transformation of a homogeneous IG from sequential to parallel-pipelined form

Note that such transformation does not reduce the number of operations, but can be performed to decrease the latency of the IG fragment. The optimizing synthesizer can more easily analyze input operands in the pyramidal form. If they are constants, then identify functional dependencies between them. Two smaller pyramids can be obtained, which can be further transmitted for analysis, excluding the closing vertex from such a fragment.

Based on the transformation method of heterogeneous IG from the sequential to parallel-pipelined form [10], the optimizing synthesizer also implements transformations of graph fragments the analogues of which are putting the common multiplier in parentheses and opening parentheses in mathematics (Fig. 2).

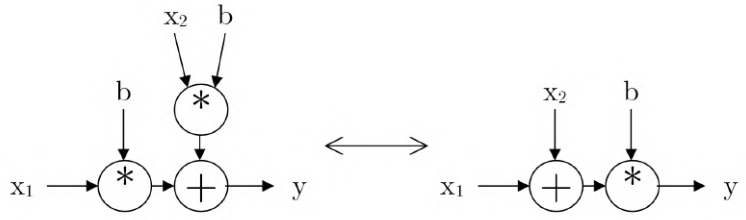


Figure 2. Transformations involving distributive operations

The IG transformation, similar to opening parentheses, is performed to reduce either the latency of the IG fragment or the number of operations. The transformation analogous to expansion may reduce the number of operations when expressions contain identical operands or constants. After opening parentheses, similar terms are combined, and part of the expression with constants can be calculated in advance.

Note that all these transformations, as well as transformations in single-chip synthesizers, are local in nature. Their implementation violates the regularity of the IG structure of algorithm to solve the problem, although it reduces the hardware resource. This complicates its analysis for redundancy and inefficiency. Therefore, in the optimizing synthesizer, the first stage involves the search and reduction of calculation redundancy in initial graph structure and the transformation of inefficient fragments to more efficient ones from the point of view of the procedural pipeline organization of calculations.

A general algorithm has been developed to perform information-equivalent transformations on graphs, representing solutions to various problems, including those from different application domains:

1. The initial graph G is placed into the array Mas of processed graphs.

2. All graphs in Mas are transformed into structures that can be decomposed into pairs of equivalent subgraphs solving identical subtasks of the original problem, along with "merge" subgraphs that combine the results of these pairs.

3. If such decomposition is not possible, proceed to step 9.

4. All processed graphs are partitioned into subgraphs.

5. Isomorphic subgraphs (with respect to operations) are analyzed, and computational redundancy is eliminated.

6. Information-equivalent transformations are performed both in computational subgraphs and in "merge" subgraphs.

7. The array Mas is cleared and filled with the resulting subgraphs.

8. Return to step 2.

9. End.

This algorithm is based on a divide-and-conquer approach, in which the original IG of the problem is divided into two subgraphs, each of which implements the processing of its own half of the original input data. These subgraphs are checked for redundancy and transformed in order to reduce it. Each of resulting subgraphs becomes a new graph for analysis, and so on until not

obtained subgraphs of minimal dimension that cannot be separated. The information-equivalent transformations described in step 6 are specific for each type of initial IG. The transformations described in step 2, as well as the methods for partitioning graphs into subgraphs in step 4, also depend on the original graph type.

Let us consider the application of a general algorithm for functionally equivalent transformations using the example of IG algorithms for problems in various subject areas.

2.1. Transformation of an Information Graph via Vandermonde Matrix Multiplication by a Vector of Unknowns

We demonstrate how an optimizing synthesizer automatically performs an IG transformation of the discrete Fourier transform (DFT) into an IG similar to the fast Fourier transform (FFT) using a general algorithm.

The DFT problem is solved through the multiplication of the complex Vandermonde matrix W of dimension $N \times N$ by a vector X. Figure 3 shows a fragment of the IG of an 8-point DFT algorithm for calculation of the elements of the output vector Y with indices 0 and 4.

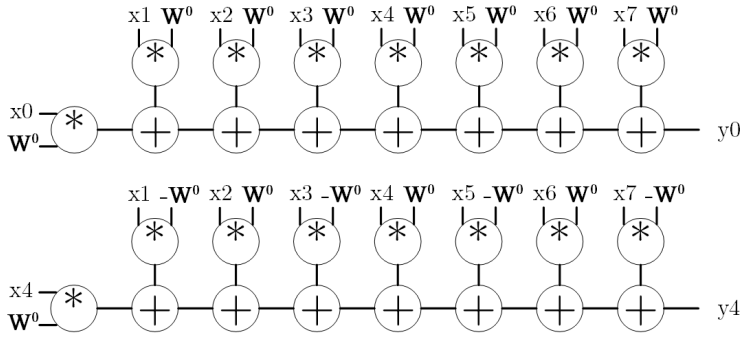


Figure 3. Fragment of the DFT IG

The elements of the complex Vandermonde matrix for the DFT have a functional dependence, using which it is possible to reduce the number of operations using information-equivalent transformations on IG. According to step 1 of the general algorithm, the entire DFT IG is loaded into an array of processed graphs. If the matrix-vector multiplication algorithm uses a sequential for-loop, the adders on the graph are connected sequentially (Fig. 3). Further, in accordance with step 2, the original graph is transformed so that the structure of the adders takes on a pyramidal form (Fig. 4).

According to step 4, in this graph fragment, the two original pyramids of adders are split into four equivalent subgraphs $G1$, $G2$, $G3$ and $G4$, as shown in Fig. 4. Each of them is a pyramid of three adders and four multipliers, and two "merge" subgraphs $G5$ and $G6$ containing one adder each.

Figure 4 shows that the sequence of multipliers differs from the original (Fig. 3). Since the sequence of addition of the multiplication results can be any, due to the associativity of the addition operation, the input arcs of the pyramid of adders can be reversed. It is performed in the implementation of step 5 in the analysis of isomorphic operations of subgraphs.

Subgraph $G1$ is isomorphic to $G3$ by vertices, with identical input data. Subgraph $G2$ is isomorphic to $G4$ by vertices, but the input data (constants) differ. Their difference can be expressed as $W_{G2} = (-1) \cdot W_{G4}$. Since they have a common functional dependency, using the

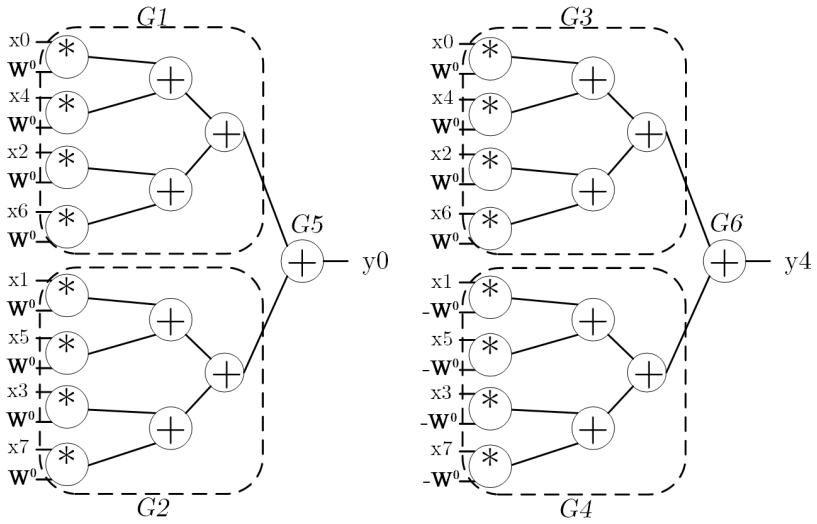


Figure 4. Fragment of the transformed DFT graph

corresponding information-equivalent transformation in the step 6, the graph $G4$ can be made such that the input data will coincide with graph $G2$, namely, by making a transformation similar to taking the common denominator out of brackets. To do this, multiply the coefficients W_{G4} by "-1" and add the multiplication by "-1" after the last adder of the $G4$ subgraph (we will add this multiplication to the $G6$ merge subgraph).

By combining the subgraphs $G1$ with $G3$ and $G2$ with $G4$, we obtain the graph structure shown in Fig. 5.

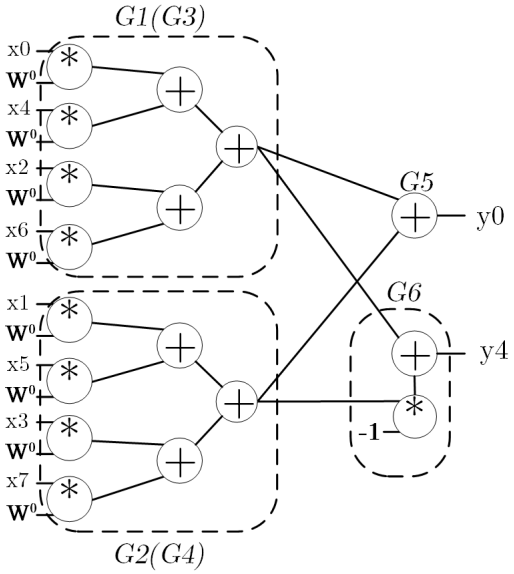


Figure 5. Fragment of the transformed DFT graph

As given in Fig. 5, the addition and multiplication by "-1" can be replaced by subtraction. However, such transformation can generally disrupt the regularity of the IG structure. Therefore, such transformations are performed in the optimizing synthesizer after executing the general information-equivalent transformation algorithm.

For subgraphs that calculate other values of the output vector, similar subgraphs are generated, differing only in the constants of the multipliers, corresponding to their functional dependencies. After step 7, the array of processed graphs includes subgraphs $G1(G3)$, $G2(G4)$, and similar subgraphs for computing other output values.

By iteratively dividing processed graphs into pairs of smaller subgraphs, the algorithm terminates once the minimal subgraphs containing a single addition are processed. Consequently, the original DFT IG is automatically transformed into an IG analogous to the FFT graph with computational complexity $O(N) = \frac{3}{2}N \cdot \log_2 N$. The original DFT IG has the complexity $O(N) = N \cdot (2N - 1)$.

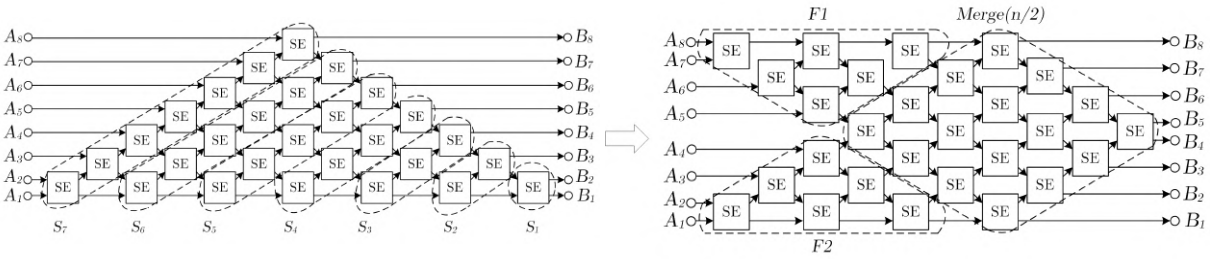
The general algorithm of information-equivalent transformations for matrix-vector multiplication can be supplemented with additional transformations to ensure that the resulting graph represents the FFT algorithm itself, rather than a merely equivalent structure. These transformations rely on the uniqueness and functional dependencies of the complex Vandermonde coefficients W and are not generally applicable to arbitrary matrix-vector multiplication tasks.

The proposed general algorithm was also verified on another problem, the Walsh-Hadamard transform [11], which, like the DFT, is a matrix-vector multiplication problem. Similar computational complexity estimates were obtained: $O(N^2)$ for the original graph and $O(N \cdot \log_2 N)$ for the "fast" version [12].

Due to the developed general algorithm of information-equivalent graph transformations, the optimizing synthesizer automatically produces a solution with higher computational efficiency than the original, general implementation.

2.2. Transformation of an Information Graph for a Sorting Network

The developed general algorithm of information-equivalent transformations was also used for graphs solving problems from a different area, the sorting of an input sequence. Let us consider the operation of this algorithm using the transformation of the bubble sorting algorithm into an initial array A containing eight elements. The original IG (Fig. 6a), developed on the "head-tail" combining steps principle described in [13], is transformed, according to step 2 of the algorithm, into a sorting network, developed using the "divide-in-half" principle (Fig. 6b). According to step 4, pairs of equivalent subgraphs $F1$ and $F2$ are formed, each of which implements subtask of sorting half of the array, along with a merge subgraph $Merge(n/2)$ that combines the results of the subgraph pairs.



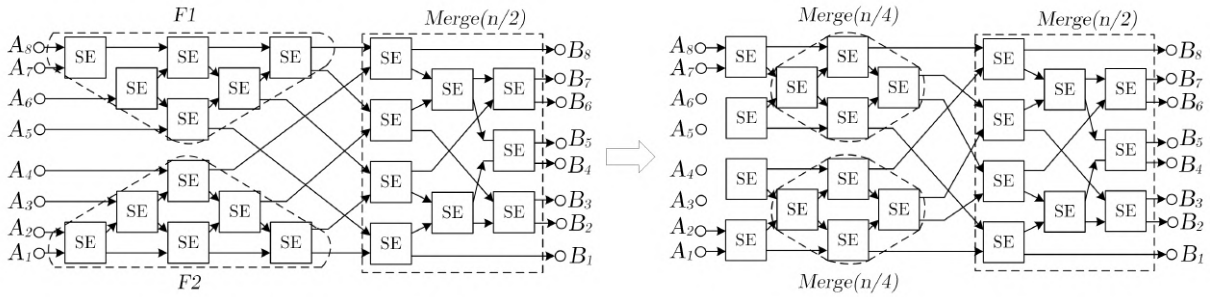
(a) Bubble sorting IG

(b) IG after combinig steps

Figure 6. Transformations of sorting networks

Since the graph structure of the sorting algorithm does not contain isomorphic subgraphs with identical input data, step 5 is skipped. Further, due to the functional redundancy shown

in [13], the described transformation methods and changing the order of operations in accordance with step 6, information-equivalent transformations of the $Merge(n/2)$ subgraph are performed (Fig. 7a).



(a) Transformations of the $Merge(n/2)$ (b) Subgraphs $F1$ and $F2$ transformation

Figure 7. Transformations of sorting networks

According to step 7, subgraphs $F1$ and $F2$ are placed into the array of processed graphs. At the next iteration, in accordance with step 2, transformations are performed to subgraphs $F1$ and $F2$. Each of subgraphs is further divided into two subgraphs that sort one-quarter of the original array, along with two merge networks $Merge(n/4)$ (Fig. 7b). After performing step 6 transformations on the $Merge(n/4)$ networks, the resulting sorting algorithm structure corresponds to the Batchers odd-even sorting network (Fig. 8).

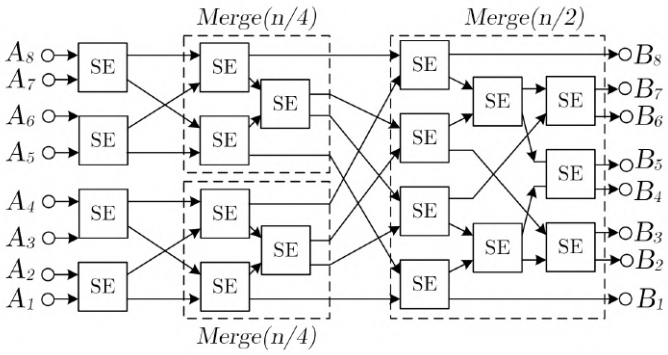


Figure 8. Batchers odd-even sorting network

The resulting four sorting subgraphs contain a single sorting element and cannot be further transformed, so the algorithm completes its work. Therefore, the original IG of the bubble sorting algorithm with computational complexity $O(N) = \frac{1}{2}N^2$, is automatically transformed into the IG of Batchers odd-even sorting network with computational complexity $O(N) = \frac{1}{4}(N \cdot \log_2^2 N + 2 \cdot \log_2 N)$. The sorting networks automatically obtained through information-equivalent transformations have been simulated and analyzed in [13, 14], which also verify the correctness of such transformations.

2.3. Transformation of an Information Graph for Solving Tridiagonal SLAEs Using the Jacobi Method

The general algorithm of information-equivalent transformations was verified on problems of solving tridiagonal systems of linear algebraic equations (SLAEs) by the Jacobi method. In

this method, the vector of unknowns is calculated as:

$$Y_i^v = \frac{1}{2}(Y_{i+1}^{v-1} + Y_{i-1}^{v-1}) + \frac{\Delta^2 C_i}{2}, \tag{1}$$

where v is the current iteration number; i is the iteration layer from which the data is taken; Δ is the grid step; C_i are the free terms. The computational complexity of the classical Jacobi method for tridiagonal SLAEs is $O(N) = N \cdot K$ in a sequential implementation, where N is the matrix dimension and K is the number of iterations required to reach a specified accuracy. Figure 9 shows the IG for SLAE of dimension $N = 9$ and the number of iterations K=4, where the vertex f implements the calculation of the function $f = \frac{1}{2}(Y_{i+1}^{v-1} + Y_{i-1}^{v-1})$. Calculations related to the free terms C_i vector (1) have been omitted so as not to overload the illustrative material, but the transformations described below take into account all operations of the original formula for calculating the vector of unknowns.

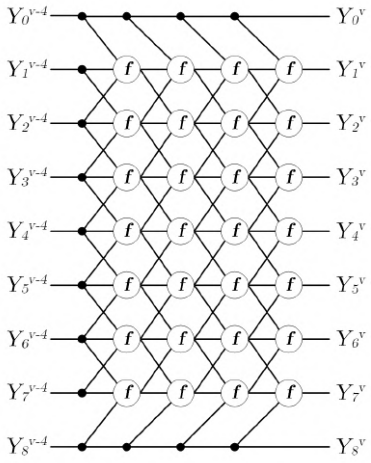
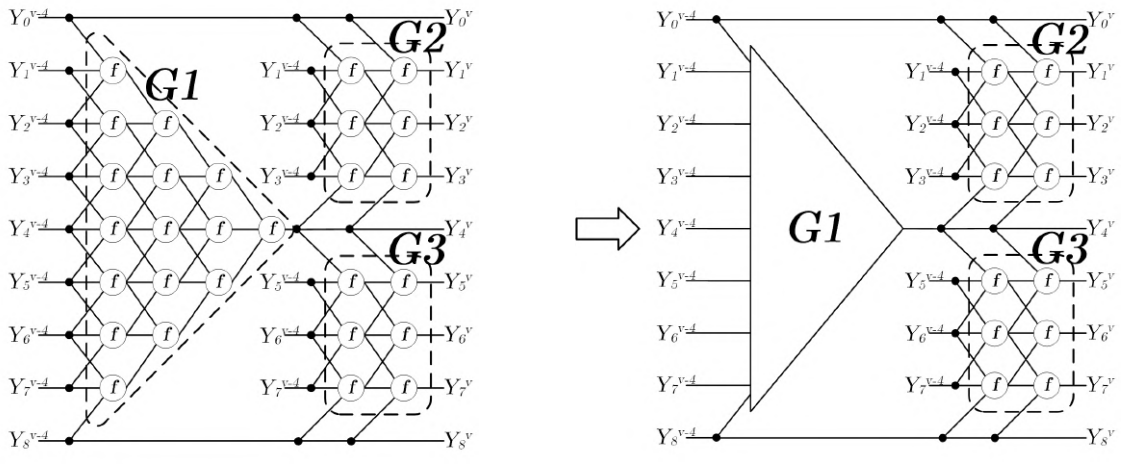


Figure 9. IG of the Jacobi method for tridiagonal SLAEs

According to step 1 of the general algorithm, the initial IG of the algorithm for solving tridiagonal SLAEs by the Jacobi method is placed in an array of processed graphs. According to paragraphs 2-4 of the general algorithm, the IG is transformed to the following form (Fig. 10a), where a subgraph $G1$ is formed to calculate the Y_4 central element of the vector of unknowns with a given depth of iterations and two subgraphs $G2$ and $G3$, each of which implements a half-size SLAE calculation.

Since the resulting graph structure (Fig. 10a) does not contain isomorphic subgraphs with identical input data, step 5 is skipped. According to step 6, the subgraph $G1$ is processed: the vertices f are expanded into arithmetic addition and multiplication operations, and information-equivalent transformations corresponding to the expansion of brackets and combination of like terms are applied. As a result, the subgraph $G1$ acquires a pyramid structure of adders with multiplications at the base, where the total number of computational operations scales logarithmically with the input array size (Fig. 10b).

According to step 7, the array of processed subgraphs is cleared, and subgraphs $G2$ and $G3$ are added with the general algorithm continuing from step 2. Each of $G2$ and $G3$, is further subdivided, according to steps 2-4, into subgraphs $G4$, $G5$, $G6$ and $G7$, $G8$, $G9$ respectively (Fig. 11a). Subgraphs $G4$ and $G7$ are transformed into pyramid structures as in $G1$ after applying

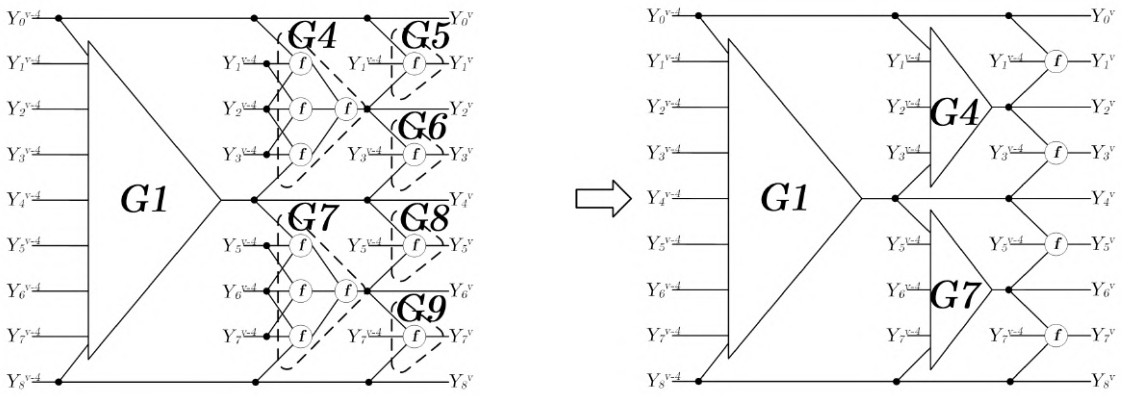


(a) Divide Jacobi IG by two

(b) Transformation subgraph $G1$

Figure 10. Transformations of the Jacobi IG

step 6 (Fig. 11b). The remaining subgraphs $G5$, $G6$, $G8$ and $G9$ contain only a single function f and cannot be further transformed, marking the completion of the algorithm.



(a) Divide subgraphs $G2$ and $G3$ by two

(b) Transformation subgraphs $G4$ and $G7$

Figure 11. Transformations of the Jacobi IG

The resulting "fast" Jacobi method represents an iterative calculation of the midpoint of the computational grid, where the next step processes two halves of the original range. The number of operations for the classical Jacobi method is given by $S_{classic} = (4 \cdot N - 2) \cdot K \approx 4 \cdot N \cdot K$, while for the fast Jacobi method it is $S_{fast} = 67 \cdot K \cdot (\log_2(N-1) - 1) \approx 67 \cdot K \cdot \log_2 N$, where N is the row length of the SLAE, and K is the number of iterations required to achieve the required accuracy. Due to information-equivalent transformations, the optimizing synthesizer automatically generates a new computational method that can be structurally implemented on a RCS, achieving higher specific performance than the original method. Such transformations for tridiagonal SLAEs using the Jacobi method have been simulated and analyzed in [15], which also verify their correctness.

3. Nested Auto-Substitution Method

In addition to the general algorithm and the library of information-equivalent transformations, the optimizing synthesizer implements an original nested auto-substitution method

described in [16], which increases the efficiency of parallel-pipelined implementation of recursive expressions.

The input data rate to the structure implementing a recursive expression depends on the execution time of all operations along the critical path of the feedback loop at pipelined computing. In other words, you can submit the next input count only after processing the previous count is completed. It is known that the velocity of such calculations can be increased using the well-known method of recursive expression pipelining-sequential auto-substitution.

The principle of this method is that any recursive expression of the form:

$$y_i = f(y_{i-1}, a_i), \tag{2}$$

where y_i is the current calculated value; y_{i-1} is the previous value; a_i are certain coefficients of the function f, can be expanded recursively. Expressing y_{i-1} as

$$y_{i-1} = f(y_{i-2}, a_{i-1}), \tag{3}$$

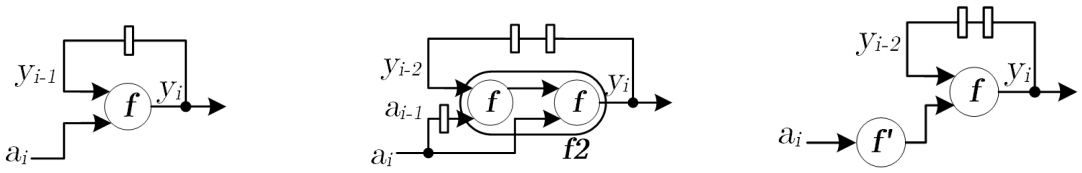
and substituting (3) into (2), we obtain

$$y_i = f(f(y_{i-2}, a_{i-1}), a_i) = f2(y_{i-2}, a_i, a_{i-1}) \tag{4}$$

Continuing this procedure, y_i can be expressed recursively:

$$y_i = fn(y_{i-n}, a_i, a_{i-1}, ..., a_{i-n+1}) \tag{5}$$

Sequential auto-substitution can be implemented on a graph using information-equivalent transformations. The initial graph corresponding to (2) is given in Fig. 12a, where the vertex corresponding to the delay of the operand stream by one sample is indicated as empty rectangle in the feedback.



(a) Recursive expression IG

(b) Auto-Substitution IG

(c) Transformed auto-Substitution IG

Figure 12. Sequential auto-substitution on a graph

Graph transformation corresponding to (2) is performed by cascading the node representing f and adding delays to its input arcs (Fig. 12b). If f is a linear function, further transformations, corresponding to bracket expansion and combining like terms, can be performed resulting in the graph shown in Fig. 12c. In this resulting graph, the critical path length remains the same as in the original graph, while the number of registers in the feedback loop increases, allowing a higher input data rate for pipeline processing [16]. Repeating such transformations allows continuous data flow; however, computing fn requires more operations than the original expression, with the number of additional operations scaling linearly with the recursion depth.

The new nested auto-substitution method improves upon sequential auto-substitution. After obtaining formula (4), instead of expressing y_{i-2} from the original formula (2), it is expressed

from formula (4) itself:

$$y_{i-2} = f2(y_{i-4}, a_{i-2}, a_{i-3}) \tag{6}$$

Substituting (6) into (4) we obtain:

$$y_i = f2(f2(y_{i-4}, a_{i-2}, a_{i-3}), a_i, a_{i-1}) = f4'(y_{i-4}, a_i, a_{i-1}, a_{i-2}, a_{i-3}) \tag{7}$$

Note that $f4$ from sequential auto-substitution differs from $f4$ in the nested method. Subsequent steps follow the same principle, applying the previously obtained formula at each iteration:

$$y_i = f4'(f4'(y_{i-8}, a_{i-6}, ...), a_i, ...) = f8'(y_{i-8}, a_i, ...),$$

$$y_i = f8'(f8'(y_{i-16}, a_{i-14}, ...), a_i, ...) = f16'(y_{i-16}, a_i, ...),$$

$$y_i = fN'(y_{i-N}, a_i, ...),$$

where N is a power of two corresponding to the number of recursion-unfolding steps.

On a graph, transformations in accordance with the nested auto-substitution method, unlike sequential, consist in the fact that for cascading, not the subgraph describing the original function f is selected, but the subgraph, obtained at the previous step. Therefore, for linear function f (Fig. 12c) the second iteration of graph transformations is given in Fig. 13.

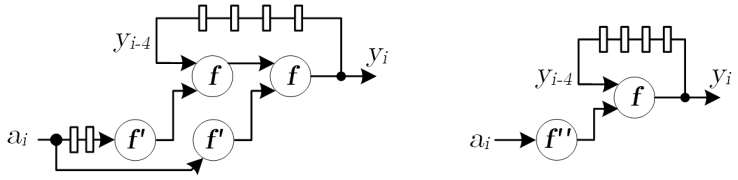


Figure 13. Nested auto-substitution on a graph

In sequential auto-substitution, the growth of additional hardware from pipelining is estimated as $O_{seq}(N) = K_1 \cdot N$, where N is the degree of pipelining and K_1 depends on function f. In the new nested method, the additional hardware growth is reduced to $O_{nested}(N) = K_2 \cdot \log_2 N$, where N is always a power of two corresponding to the recursion-unfolding steps. Synthesized solutions were simulated on RCS of various configurations, confirming the correctness of the information-equivalent transformations.

The nested auto-substitution method can also be used for the simple iteration method for solving SLAEs [17], described by $\overrightarrow{x}_i = T \times \overrightarrow{x}_{i-1} + \overrightarrow{\phi}$. In this case, the function on the right-hand side involves matrix and vector operations. The computational complexity of the original algorithm is $O(n) \approx (2 \cdot M^2 + M) \cdot n$, where n is the number of iterations, M is the SLAE dimension. Using nested auto-substitution, the "fast" version reduces the complexity to $O(n) \approx (2 \cdot M^2 + M) \cdot \log_2 n$.

4. Results of Application of the Optimizing Synthesizer

The effectiveness of the new version of the multichip optimizing synthesizer for parallel-pipelined programs for RCS was investigated on a number of benchmark tasks on the RCS Tertius architecture [18] with a peak performance of 2.5 TFLOPS. The system includes four Xilinx Kintex UltraScale XCKU095 FPGAs, each with 100 million equivalent logic gates, connected in a ring using LVDS and GTY/GTH channels.

Note that the solutions generated by the developed optimizing synthesizer can also be directly implemented by a vendor in the program code of a parallel-pipelined program for RCS. However, in this case, the program size, debugging time and the vendors skill requirements increase, often reducing the program readability. In addition, the source C programs almost never contain an effective structural and procedural organization of calculations for parallel-pipelined RBC programs obtained during the automatic conversion of an input sequential program in C.

The performance of the optimizing synthesizer was tested on graphs for solving discrete Fourier transform problems, sorting networks, solving tridiagonal SLAE using the Jacobi method, and for a second-order recursive filter with constant coefficients. The results of the optimizing synthesizer implementation were the structures of "fast" algorithms that were automatically transformed in accordance with the general algorithm and nested auto-substitution, where the effectiveness of the solutions obtained was evaluated.

The result of the optimizing synthesizer for DFT problem with dimension 8 is given in Fig. 14. The resulting transformed graph is not isomorphic to the graph of the well-known FFT method, but contains the same number of computational elements. The original DFT IG includes 56 adders and 64 multipliers for complex numbers. The resulting IG contains 24 adders and subtractors and 6 multipliers (25% from the original). The synthesizer was also verified on larger DFT problems, producing graphs with computational complexity equivalent to FFT graphs.

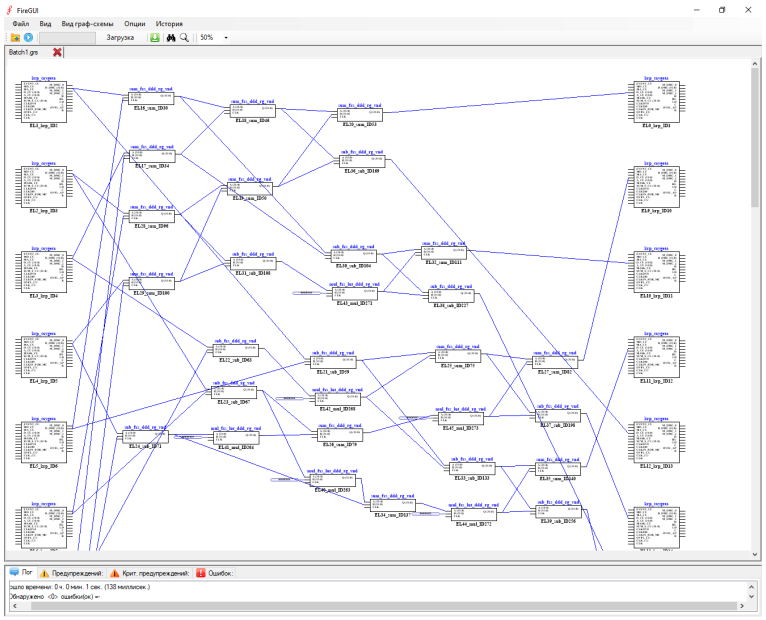


Figure 14. Result of the optimizing synthesizer for DFT problem

An odd-even Batcher network for sorting 16 elements, automatically generated by the synthesizer from a bubble-sort network, is given in Fig. 15. The synthesizer was tested on networks with dimensions 16, 32, 128 and 256.

For solving a tridiagonal SLAE using the Jacobi method with dimension 128, the synthesizer automatically produced a "fast" Jacobi IG, containing approximately five times fewer operations than the classical Jacobi method. As the matrix dimension increases, this indicator increases, since for the classical Jacobi method the number of operations depends linearly on the size of the matrix, and in the "fast" Jacobi method this dependence is logarithmic.

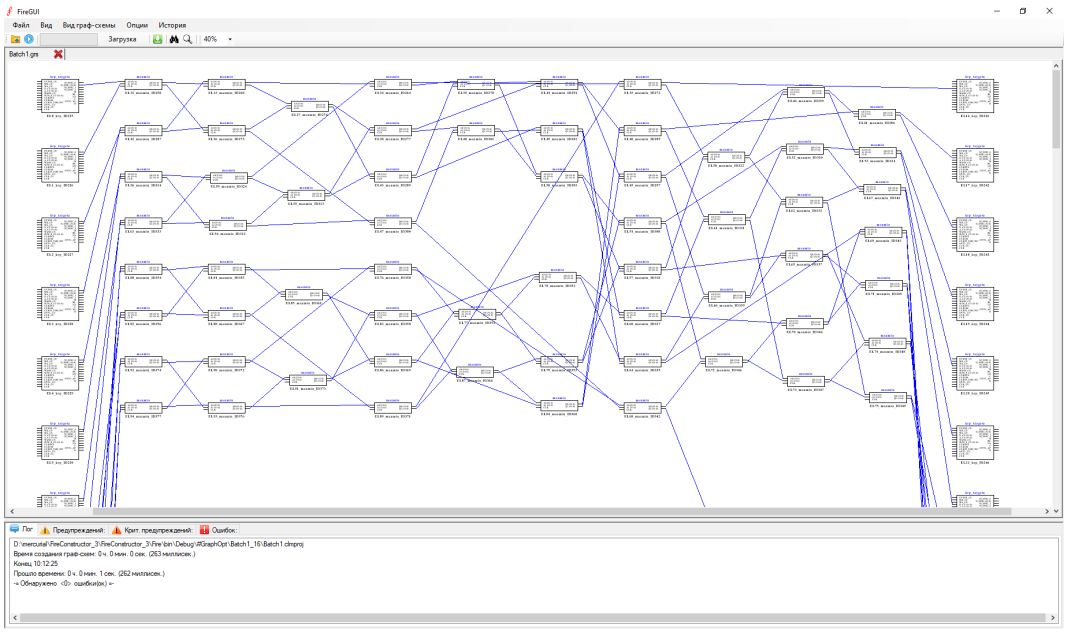


Figure 15. Result of the optimizing synthesizer by sorting problem

Figure 16 presents the result of the synthesizer on a second-order recursive filter problem with constant coefficients.

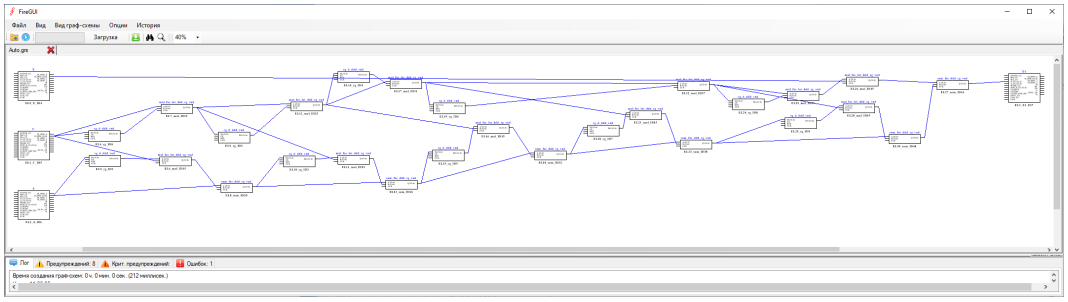


Figure 16. Result of the optimizing synthesizer by recursive filter problem

The additional hardware required for pipelining the second-order recursive filter using sequential auto-substitution is equal to $O_{seq}(N) = 2 \cdot N$, where N is the pipeline depth, while for nested auto-substitution it is equal to $O_{nested}(N) = 6 \cdot \log_2 N$. Based on these formulas, it is obvious that for this filter, using the nested auto-substitution becomes more effective than sequential auto-substitution at $N > 8$.

To ensure that adders and multipliers operate at the maximum RCS Tertius frequency of 500 MHz, they were implemented in a pipelined configuration. The adder and multiplier contains 12 and 8 pipeline registers, respectively. In the case of development a recursive part of the filter on this element base, the feedback circuits contain two adders connected in series and one multiplier. Therefore, the result is formed after 32 cycles in feedback circuits. It determines the choice of the pipelining degree. The efficiency of the second-order recursive filter solution automatically created in the synthesizer with constant coefficients on the Tertius RCS implementation was approximately twice that of the previously known sequential auto-substitution method. Similar results were obtained for adaptive filters with variable coefficients.

Conclusion

The new version of the optimizing synthesizer for parallel-pipelined programs described in this paper not only maps the IG algorithm of the solved problem on multichip RCS but also significantly reduces the number of operations in programs created by an unqualified user. The library implemented in the optimizing synthesizer and the general algorithm of information-equivalent transformations, as well as the original method of nested auto-substitutions, allow to automatically obtain the "fast" algorithms, the effectiveness of which is not inferior to previously known methods or new ones based on original methods developed by the research team at Scientific Research Center of Supercomputers and Neurocomputers (Taganrog).

The number of computational operations in traditional algorithms for solving the considered model problems is determined by the formula $O(N) = C_1 \cdot N^m$, where C_1 and m depend on the specific problem. In all resulting "fast" algorithms, the number of operations is determined by the formula $O(N) = C_2 \cdot N^{m-1} \cdot \log_2 N$. Therefore, the efficiency of the optimizing synthesizer only increases with increasing problem dimension. So, for the problem dimension $N = 1024$, the number of operations in the algorithm can be reduced by two decimal orders of magnitude, and the specific productivity will increase a hundredfold, which is unattainable for similar synthesis tools.

Future research directions include the development of optimization methods for computing structures in problems with variable operand stream intensity and IGs containing structural analogs of unconditional jumps.

Acknowledgements

The study was supported by the Russian Science Foundation grant No. 26-11-00209, https://rscf.ru/project/26-11-00209/.

References

1. Levin, I.I., Dordopulo, A.I., Fedorov, A.M., Kalyaev, I.A.: Reconfigurable computer systems: from the first FPGAs towards liquid cooling systems. Supercomputing Frontiers and Innovations 3(1), 22–40 (2016). https://doi.org/10.14529/jsfi160102

2. Kalyaev, I.A., Levin, I.I.: Reconfigurable Computing Systems Based on FPGAs. Publishing House of SSC RAN, Rostov-on-Don (2022). 506 p.

3. Antonov, A.S., Afanasyev, I.V., Voevodin, Vl.V.: High-performance computing platforms: current status and development trends. Computational Methods and Programming 22(2), 135–177 (2021). https://doi.org/10.26089/NumMet.v22r210

4. Dichenko, A.A., Levin, I.I., Sorokin, D.A.: Method for generating topological constraints of computational structures for reconfigurable computing system. Izvestiya SFedU. Engineering Sciences, 33–46 (2022). https://doi.org/10.18522/2311-3103-2025-6-33-46

5. Gulenok, A.A.: Synthesizer of Structural Parallel Application Programs for Multichip Reconfigurable Computing Systems: Candidate of Technical Sciences Dissertation. Taganrog, 2011.

6. Kotlyarov, D.V., Kutepov, V.P., Osipov, M.A.: Graph-based stream parallel programming and its implementation on cluster systems. RAS Publishing, Theory and Control Systems 1, Moscow (2005).

7. Nane, R., *et al.*: A survey and evaluation of FPGA high-level synthesis tools. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35(10), 1591–1604 (2016). https://doi.org/10.1109/TCAD.2015.2513673

8. Karypis, G., Kumar, V.: Multilevel algorithms for multi-constraint graph partitioning. Technical Report TR 98-019, Department of Computer Science, University of Minnesota (1998).

9. Gulenok, A.A., Dudko, S.A.: Hardware resource management during automatic placement of synchronization triggers. 12th Multiconference on Control Problems, MKPU-2019, Divnomorskoye, Russia, vol. 3, pp. 86–88 (2019).

10. Mikhaylov, D.V.: Methods for creating pipeline-parallel programs for tasks with sequential information graphs: Candidate of Technical Sciences dissertation, specialty 05.13.11 "Mathematical and software support for computing machines, complexes, and networks", advisor: Prof. I.I. Levin. Taganrog (2022). 213 p.

11. Golubov, B.I., Efimov, A.V., Skvortsov, V.A.: Walsh Series and Transforms: Theory and Applications. Nauka, Moscow, (1987). 360 p.

12. Semernikov, E.A., Gulenok, A.A.: Structural optimization of the information graph of matrix-vector multiplication. Proceedings of the XIV All-Russian Meeting on Control Problems VSPU-2024, pp. 3024–3028. IPU RAS, Moscow (2024).

13. Levin, I.I., Alekseev, K.N.: Transformation of sorting networks for different degrees of parallelism. Izvestiya SFedU. Technical Sciences 5(235), 104–118 (2023). https://doi.org/10.18522/2311-3103-2023-5-104-118

14. Alekseev, K.N., Levin, I.I., Gulenok, A.A.: Transformation of simple sorting networks into Batchers odd-even networks. Izvestiya SFedU. Technical Sciences 4(240), 50–63 (2024). https://doi.org/10.18522/2311-3103-2024-4-50-63

15. Levin, I.I., Chekina, M.D.: Fast Jacobi method for solving SLAEs on reconfigurable computing systems. Science in the South Russia, vol. 2, pp. 145–165. Nauka Publisher, Rostov-on-Don (2026).

16. Levin, I.I., Gulenok, A.A., Semernikov, E.A.: Nested auto-substitution method for pipelining computations in recursive filters. PAO Radiofizika. Radar and Communication 2(40), 17–28 (2025). https://doi.org/10.18127/j00338486-202510-02

17. Samokhin, A.B.: Integral Equations and Iterative Methods in Electromagnetic Scattering. Moscow: Radio i Svyaz (1998).

18. Computing units of the NIT supercomputers and neurocomputers. http://superevm.ru/index.php?page=modern-developments, accessed: 2026-04-01